Mathematical Approach to an Optimal Algorithmic *Snake* Game Agent

Malav Mehta

April 10, 2021

Contents

1 Introduction

One of the primary focuses of mathematics is the modelling and optimization of real-life scenarios to automate processes or maximize desired results while minimizing use of resources. This field has countless practical applications in the realms of scientific, engineering, finance and algorithmic problems.

As an aspiring computer scientist, I have developed a particular interest in the intersection between math and programming. Alongside my passion for applied math, I also enjoy playing video games, being introduced to them after I first played the popular *Snake* game on an old Nokia 3310 during my childhood. After taking the HL Math course, I started thinking about the things I regularly do from a mathematical point of view, so naturally, I decided to find a mathematically suitable way to create an efficient and autonomous agent for the *Snake* game.

Hence, this investigation aims to find various **mathematical approaches to creating an optimal algorithmic *Snake* game agent** and comparing them to conclusively determine the most efficient solution. For this investigation the most efficient algorithm will be the one which beats the game in the minimum amount of time. All approaches will be transformed into software agents for testing. This exploration will look at graph theory as a way to model the game and arrive at raw base approaches which will then be evaluated and combined with an analytical approach using probability density functions and calculus.

2 *Snake* Game Mechanics

The *Snake* game is a very popular game, being first conceptualized in the 1976 arcade game *Blockade*. Since then, there have been hundreds of implementations of the game. It is thus important to define the rules of the version of the *Snake* game that will be used in this exploration.

In the implementation that I have recreated in `C++` for this investigation (Figure 1), the autonomous agent controls a snake on the game plane. As the snake moves forward, left or right, it leaves a trail of fixed size behind it. The snake grows longer when its head runs into the apples and dies when it runs into itself or the borders. The objective is to grow long enough to cover the grid entirely, at which point the game is won.

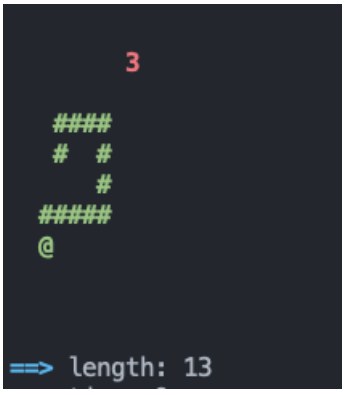


Figure 1: The graphical representation of my implementation of the *Snake* game. The `@`, `#` and red digit represent the snake's head, the snake's body and the apple, respectively

3 Modelling The Game

While researching suitable models for representing the game, I came across graph theory, which is a branch in discrete mathematics. This was option topic 10 in last year's HL Math curriculum. In this section, I will explain what graph theory is, the notation that will be used in this exploration and how *Snake* can be modelled by a graph.

3.1 Graph Theory

Graph theory refers to a way in which data can be represented. A graph is a pair $G = (V, E)$ where V is a set of vertices (also called nodes) and E is a set of unordered pairs of vertices. The edge e between vertices v_1 and v_2 is written as $e = v_1v_2 = v_2v_1$. v_1 and v_2 are thus the endpoints of edge e and are said to be adjacent to each other, written $v_1 \leftrightarrow v_2$. A pair of vertices can be joined by at most one edge and no vertex can be joined to itself by an edge. For example, $G_1 = (V_1, E_1)$ where $V_1 = \{v_1, v_2, v_3\}$ and $E_1 = \{v_1v_2, v_2v_3\}$ is drawn:

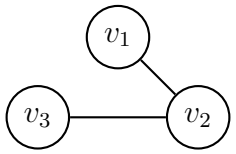


Figure 2: Graph G_1

In graphs, there sometimes exist paths between two vertices and cycles containing multiple vertices. Formally, a graph $G_p = (V_p, E_p)$ of n vertices is a path from vertex v_1 to v_n if $V_p = \{v_1, \ldots, v_n\}$ and $E_p = \{v_iv_{i+1} : i = 1, \ldots, n-1\}$. G_p is a cycle under the condition that $E_p = \{v_iv_{i+1} : i = 1, \ldots, n-1\} \cup \{v_nv_1\}$.

3.2 Representing *Snake* As A Graph

To represent *Snake* as a graph, it must first be visualized as a grid of width m and height n with mn cells where each cell c has coordinates $(x, y) : \{1 \leq x \leq m, 1 \leq y \leq n\}$ and the cell $c = (1, 1)$ is on the top left of the grid (see Figure 3).

$(1,1)$	$\ldots$	$(m,1)$
$\ldots$		$\ldots$
$(1,n)$	$\ldots$	(m,n)

Figure 3: The *Snake* game plane in grid format

Under this representation, the position of the snake and the apple can be represented as a set of cells and an individual cell, respectively. The snake can travel to cells that are adjacent to its head and are not already occupied by itself. To express this model as graph $G = (V, E)$ (see Figure 4), we can make the following considerations:

1. Each cell c can be represented as some 1vertex $v = (c_x, c_y) : \{1 \leq c_x \leq m, 1 \leq c_y \leq n\}$. The set of vertices V of G can thus be represented as $V = \{v_{i,j} : 1 \leq i \leq m, 1 \leq j \leq n\}$ where vertex $v_{i,j}$ refers to the cell at (i, j).
2. In the grid, the snake can travel to each cell non-diagonally adjacent to its head. We can express the ability to travel from one cell to another as an edge between the corresponding two vertices. Therefore, for all $v_{i,j} \in V$, there exists an edge between $v_{i,j}$ and $v_{i+1,j}$, $v_{i-1,j}$, $v_{i,j+1}$ and $v_{i,j-1}$ in E of G if these adjacent vertices are in V.
3. The snake's body can be represented as a set of vertices S containing the vertices occupied by the snake's body such that $S \subseteq V$.
4. The snake head can be represented as some vertex $v_{i,j} \in V$, denoted v_h for presentation.
5. The apple can be represented as some vertex $v_{i,j} \in V$, denoted v_a for presentation.

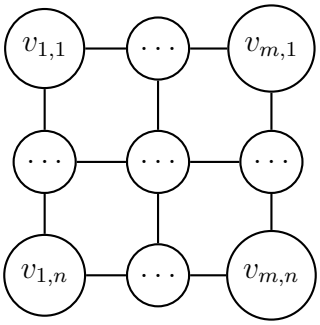


Figure 4: Graph G

4 Base Approaches

The graph representation of the *Snake* game represents the current state and models the positions of the snake and the apple on the game plane. Since the end goal is to be able to program the mathematical approach, a common way to implement mathematical logic is to have it isolated to a function where the information about the game state after each move is given as input and a suitable return value is outputted.

In the case of this exploration, the purpose of the function will be to find the best path that the snake should follow from its head in order to win the game. With this path generated, the first edge in the set of edges of the path contains the next vertex of where the snake head needs to go. Knowing this, the program can calculate the direction in which the snake must turn. Therefore, the return value should be the first edge of the best path.

4.1 Breadth-First Search

The most intuitive method for generating the best path with the goal of producing a time efficient mathematical approach is to compute the shortest path between the snake head and the apple. Returning the first edge of this path for each game state should be the quickest way for the snake to get to the apple, increase in size and eventually win the game.

A systematic approach for finding the shortest path between two vertices in a graph is to use the Breadth-First Search (BFS) algorithm. In the real-world, alongside other path-finding algorithms such as A-star and Dijkstra's, BFS is used in problems where exploring a graph is needed, such as for finding optimal flight routes between two cities. The BFS traversal of a graph terminates when every vertex of the graph has been visited. The concept is to visit all adjacent vertices of the starting vertex before visiting the vertices adjacent to the adjacent vertices. The algorithm takes the following steps:

1. Keep a list of all vertices seen so far. Add the starting vertex to this list.
2. Take the first vertex from the list, visit it and add unvisited adjacent vertices to the end of the list.
3. Repeat the previous step until the list is empty.

Consider Figure 5 with graph $G_2 = (V_2, E_2)$:

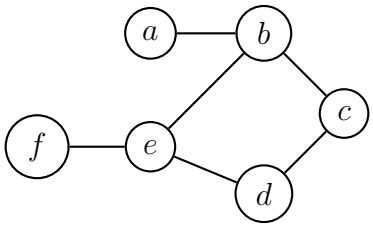


Figure 5: Graph G_2

Directly applying the BFS algorithm with starting vertex a, the traversal for the graph G_2 would be: $a \to b \to c \to e \to d \to f$. Rather than traversing all vertices, the application of BFS for the *Snake* game problem requires finding the shortest path between the snake head and the apple. To accomplish this goal, we can modify the initial algorithm as such:

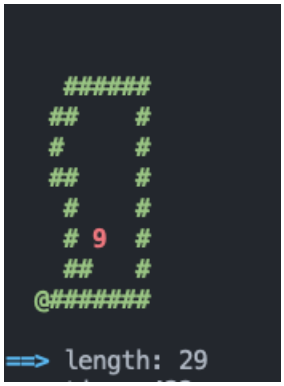
1. Keep a list of vertices seen so far and the path required to get to the vertex as a pair. Add the pair of the starting vertex and an empty path to this list.
2. Take the first pair from the list. This pair will have vertex v and path P. Visit v and for each unvisited vertex u adjacent to v, duplicate path P but also add edge $e = vu$ to it and finally add the pair (u, modified path P) to the end of the list.
3. Repeat the previous step until the end vertex is added to the list. If so, the associated path will be the shortest path from the starting vertex to the end vertex and return this path.
4. If the list is empty before the end vertex is reached, return an empty path.

Applying this modified BFS algorithm to G_2 with starting vertex a and end vertex e, we see that the edges of the generated path are correctly $\{ab, be\}$ rather than $\{ab, bc, cd, de\}$. Therefore, this algorithm is suitable for finding the aforementioned best path for the snake to take under the shortest path approach.

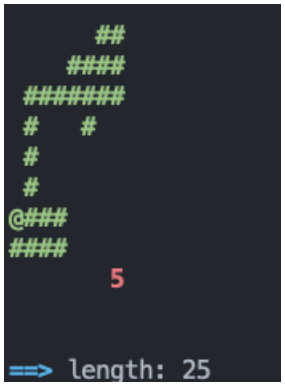
To apply BFS to the *Snake* game, we set v_h as the starting vertex and v_a as the end vertex. One final modification to make, to ensure that the generated path does not involved

the snake head running into its body, is to add the following rule to step 2 from the modified algorithm: only adjacent vertices u that are not already occupied by the snake's body should be added to the list. The final code for this approach is shown in Figure 10 of Appendix A.

Despite being a very quick solution and seemingly sound at first glance, a more comprehensive examination of the shortest path approach reveals a major flaw. Since the BFS algorithm is a greedy solution to the problem, in that it finds the best move for the current state and performs it without thinking of the consequences, it is not able to foresee the position of the snake after it has moved along the shortest path and thus traps itself. After running 100 000 simulations with this approach, it was never successful at beating the game. Each game ends in the ways shown in Figure 6.



(a) The apple gets enclosed by the snake



(b) The snake traps itself along the border

Figure 6: Lost scenarios with the BFS approach

4.2 Hamiltonian Cycle

To remedy the flaw in the BFS approach, I decided to focus on a solution that would guarantee a win even if it meant that it might not be efficient time-wise. In my research, I watched videos of players winning *Snake*. Although at first these players would take a greedy shortest path approach, they changed their strategy once the snake got longer, instead sticking to one fixed path and following it endlessly. Essentially, they traverse the game plane such that they visit every point exactly once before returning to the starting point and then repeating this loop. This can be recognized as a Hamiltonian cycle (also called a Hamiltonian circuit), formally defined to be a closed loop on a graph where each vertex is visited exactly once. Consider Figure 7 with graph $G_3 = (V_3, E_3)$:

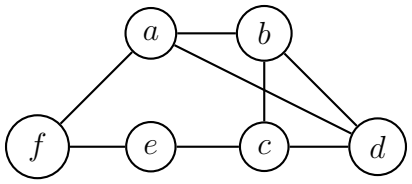


Figure 7: Graph G_3

G_3 is said to be Hamiltonian since there is a Hamiltonian cycle starting from vertex a: $a \to b \to d \to c \to e \to f \to a$. The rationale behind having the snake follow a Hamiltonian cycle is that no matter where it goes, it will never be able to trap itself since it will not be able to run into itself until the entire graph consists only of the snake. Thus, it is guaranteed that an agent based on this approach will win the game.

However, in order for the graph G which represents the *Snake* game plane to be Hamiltonian, the width of m and height n of this vertex grid must exist such that:

- $m > 1$, $n > 1$ and either m or n is even. These constraints can be proved to work with a proof by construction. First, if $m = 2$ or $n = 2$ then a Hamiltonian cycle can be constructed by forming a ring that reaches all vertices.

  If this is not the case, then there must exist a side in the grid with an even number of vertices greater than 2. We will call this direction *vertical* (meaning that the graph will have an even number of rows) and the other direction *horizontal*. Start from the first column of the first row and begin the path by extending it to the last column of this first row. Go one row down and return to the second column. If this is the last row of the graph, then further extend the path to the first column and connect it back to the vertex on the first column of the first row. If this is not the last row, then stay on the second column, drop one column down and repeat the initial process. Since there are an even number of rows, there will be exactly $\frac{n}{2}$ repetitions. Note that as long as $m > 2$ ($m = 2$ case considered above), the length of m does not matter.

  Therefore, it is always possible to construct a Hamiltonian cycle when $m > 1$, $n > 1$ and either m or n is even. See Figure 8 for a visual example of the above proof.

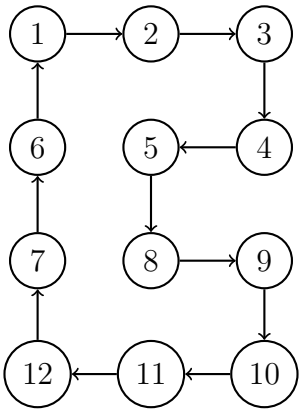


Figure 8: Numbered Hamiltonian cycle of grid graph for $m = 3$ and $n = 4$

- **Or** $mn = 1$. Under this scenario, the graph contains only one vertex and is trivially Hamiltonian with a zero length path that visits the vertex.

If only one of m and n are equal to 1, then the graph is a straight line. In this case, it is impossible to form a Hamiltonian cycle. If both m and n are odd and mn is not equal to 1, then we have a bipartite graph with unbalanced vertex parity, which, by Grinberg's Theorem, is known to not be Hamiltonian.

To implement this idea in code as an autonomous agent, the most efficient solution would be to generate a Hamiltonian cycle for the game graph when the game is first initialized. This way, the function into which the game state would be the input already has the best path for the snake in memory and the code can check where the snake head is to determine where it should turn next.

Note that generating a Hamiltonian cycle is an NP (non-polynomial time) hard problem in computation, since there are $\frac{1}{2}(n-1)!$ candidate cycles in any graph with n vertices. Instead, I used a heuristic proposed by Chuyang Liu (see code on Figure 11 of Appendix A). After making this implementation, as can be seen in Figure 9 the game can now be solved.

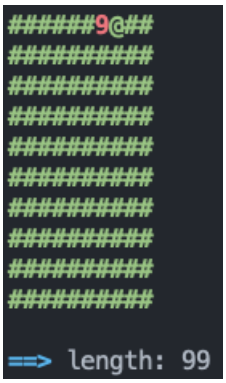


Figure 9: Completed game for a 10×10 grid

While this is a valid approach to solving the game, it does not address the needs outlined in the aim of this investigation of developing an **optimal** approach: one requiring a minimum amount of time. At the beginning of the game, when the length of the snake is only one, the snake head may need to move up to $mn-1$ turns before reaching the apple in the worst case scenario. On average, at length l, the snake needs to move for $\frac{1}{2}(mn-l)$ turns before reaching the apple. Therefore, this is an extremely slow approach and is thus not the ideal way to mathematically solve the game.

5 Analytical Approach

Despite their shortcomings, both the BFS and Hamiltonian cycle approaches had their strengths. For a grid where $m = n = 10$, the agent using the BFS approach experimentally survived until the length of the snake was over 20. Notably, it got to this length very quickly using its greedy shortest path approach. The Hamiltonian cycle on the other hand, despite being a slower approach, always guaranteed that the game would be won.

Referring back to the method that human players took to beat the game, we can observe that a combination of the two aforementioned base approaches will be the most optimal. A final approach, in which both of these solutions are combined, will be enhanced by both of their benefits. To model the human players, the third approach will first begin with BFS followed by a conversion to the Hamiltonian cycle after some length. The most critical aspect with this approach is to determine what length maximizes the benefits of the speed of BFS

in order to minimize the time but also assures survival of the snake. To find this length, I can optimize using differential calculus a function T where $T(l)$ represents the total amount of moves (or turns including going straight) that will be taken if the models switches from BFS to the Hamiltonian cycle at length l.

5.1 Probabilistic Modelling

5.2 Optimization

6 Evaluation

6.1 Comparison

6.2 Reflection

7 Conclusion

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891–921, 1905.

[3] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`

A C++ Code

```
vector<Node> shortest_path(Node des) {
    queue<vector<Node> > q{{snake.back()}};
    vis.clear();
    for (Node u : snake)
      vis[u] = 1;
    vis[des] = 0;

    while (!q.empty()) {
      vector<Node> path = q.front();
      q.pop();

      if (path.back() == des)
        return path;

      for (Node u : path.back().adj()) {
        if (is_valid(u) && !vis[u]) {
          vis[u] = 1;
          vector<Node> new_path(path);
          new_path.push_back(u);
          q.push(new_path);
        }
      }
    }
    return {};
}
```

Figure 10: Code for performing a **Breadth-First Search**.

```
void build_cycle() {
    cycle.clear();
    vector<Node> path_dir = longest_path(snake[0]), snake_path = dir_path(snake);
    for (Node u : snake_path)
      path_dir.push_back(u);
    Node cur = snake.back();
    for (Node u : path_dir) {
      cycle[cur] = u;
      cur = cur + u;
    }
    cycle_size = path_dir.size();
}
```

Figure 11: Code for generating a **Hamiltonian Cycle** (heuristic).