Mathematical Approach to an Optimal Algorithmic *Snake* Game Agent

1 Introduction

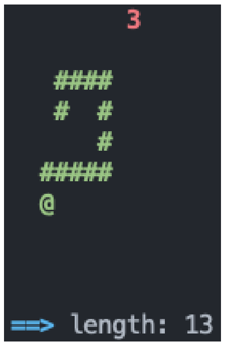
A primary focus of mathematics is modelling and optimizing real-life scenarios to automate processes or maximize desired results while minimizing use of resources. This field has countless practical applications in the realms of scientific and algorithmic problems.

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 perspective, 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

In my implementation of the *Snake* game (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.



```
      3

 ####
 #  #
    #
#####
@



==> length: 13
```

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. Although this was not taught in class, I found many explanatory articles online.

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 and E is a set of pairs of vertices. The edge e between vertices v_1 and v_2 is written as $e = v_1v_2$. v_1 and v_2 are thus the endpoints of edge e and are said to be adjacent to each other [4]. 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:

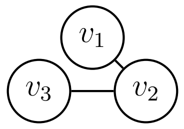


Figure 2: Graph G_1

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 $E_p = \{v_iv_{i+1} | i = 1, \ldots, n-1\}$ [4]. G_p is a cycle under the condition that $E_p = \{v_iv_{i+1} | i = 1, \ldots, n-1\} \cup \{v_nv_1\}$ [4].

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 at the first row of the first column 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 model, 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 make the following considerations:

1. Each cell c can be represented as some vertex $v = \{(c_x, c_y) | 1 \leq c_x \leq m, 1 \leq c_y \leq n\}$. The set of vertices V 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 coordinates (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 to $v_{i+1,j}$, $v_{i-1,j}$, $v_{i,j+1}$ and $v_{i,j-1}$ in E 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.

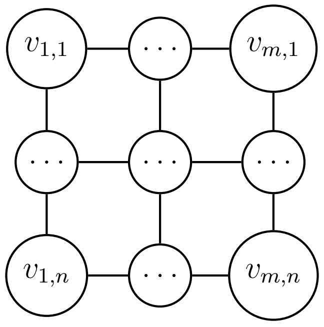


Figure 4: Graph G

4 Base Approaches

This model represents the current state and the positions of the snake and the apple on the game plane. The end goal is to be able to program a mathematical approach; a common way to implement mathematical logic into computer programs is to describe it in a function where the current state is passed as input and a useful value is returned. For this exploration, this function must find the best path that the snake should follow 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 be. Knowing this, the program can calculate the direction in which the snake must turn. Thus, the return value should be the first edge of the best path.

4.1 Breadth-First Search

An intuitive method for generating the best path with the goal of producing a time efficient solution is to compute the shortest path between the snake head and the apple since it is the quickest way for the snake to reach the apple and increase in size. 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 [3]:

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 $G_2 = (V_2, E_2)$:

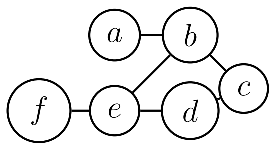


Figure 5: Graph G_2

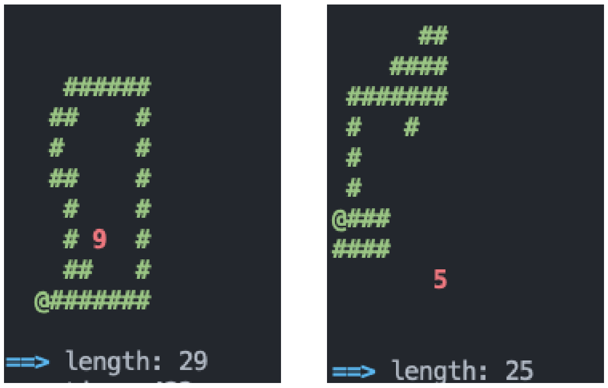
Directly applying the BFS algorithm with starting vertex a, the traversal for the graph G_2 would be: $a \rightarrow b \rightarrow c \rightarrow e \rightarrow d \rightarrow f$. Rather than traversing all vertices, the application of BFS for the *Snake* game problem requires finding the shortest path between the snake head and the apple. To accomplish this goal, we can modify the initial algorithm as such [1]:

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 $(u, \text{modified path } P)$ to the end of the list.
3. Repeat the previous step until the end vertex is added to the list, in which case the associated path will be the shortest path from the starting vertex to the end vertex and return this path. In the case that the list is empty, 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 involve 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 15 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 10 000 simulations with this approach, it was never successful at beating the game. Each game ends in the ways shown in Figure 6.



```
  ######
 ##    #
 #     #
 ##    #
  #    #
  # 9  #
   ##  #
@#######
==> length: 29
```

(a) The apple gets enclosed by the snake

```
      ##
    ####
 #######
 #   #
 #
 #
@###
####
       5

==> length: 25
```

(b) The snake traps itself along the border

Figure 6: Losing 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*. At first these players would take a greedy shortest path approach but changed their strategy once the snake got longer, instead looping one fixed cycle [2]. They traverse the game plane such that they visit each point exactly once before returning to the starting point. This is recognized as a Hamiltonian cycle, formally defined to be a closed loop on a graph where each vertex is visited exactly once [7]. Consider Figure 7 with graph $G_3 = (V_3, E_3)$:

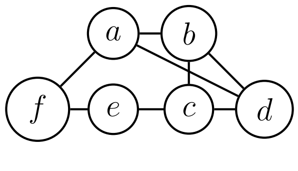


Figure 7: Graph G_3

G_3 is said to be Hamiltonian since there is a Hamiltonian cycle: $a \to b \to d \to c \to e \to f \to a$. The rationale behind having the snake follow this cycle is that it will never be trapped 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.

For graph G to be Hamiltonian, its width m and height n must follow these constraints:

- $m > 1$, $n > 1$ and either m or n is even. **Proof by construction** [9]: If m or n is 2, a Hamiltonian cycle can be constructed by forming a ring around the vertices. Otherwise, there must be 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 below and return to the second column. If this is the last row of the graph, then 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 if $m > 2$, 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.

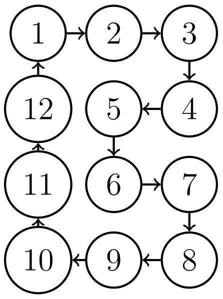


Figure 8: Numbered Hamiltonian cycle of grid graph for $m = 3$ and $n = 4$

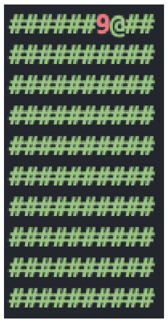
- **Or** $mn = 1$. Under this scenario, the graph contains only one vertex and is trivially Hamiltonian with a zero length cycle that visits the vertex.

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 [9].

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 [6]. Instead, I used a heuristic proposed by Chuyang Liu [5] (see code in Figure 16 of Appendix A). After making this implementation, the game can now be won as can be seen in Figure 9.

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.



```
######9@##
##########
##########
##########
##########
##########
##########
##########
##########
##########
```

Figure 9: Completed game for a 10×10 grid

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 usually 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 $T(l_s)$ represents the total amount of moves (which are to be understood as turns including going straight) that will be taken if the model switches from BFS to the Hamiltonian cycle at length l_s. Note that much of this analysis requires knowing the size of the grid beforehand. Therefore, $m = n = 10$ will be used as the grid size for the game plane.

5.1 Modelling Each Approach

The first step to take is to model each of the two base approaches with respect to the average amount of moves required for increasing the length to $l + 1$ when the snake is at length l.

For the BFS approach, I found this relationship experimentally by letting a BFS agent play the game 10 000 times. For each game, I added a data point of how many moves it took for a snake of length l to get to $l + 1$, and then averaged this data. The results are summarized in the graph below (see Figure 10).

The limited domain of the data points collected is a result of the fact that with the BFS agent, the snake begins dying a lot more often as the snake length increases (since it is more likely to trap itself), so only this domain was deemed applicable for analysis. In order to express this relationship as a function, I considered the use of several different trend line models for the scatter plot. However, I noted that a polynomial function could not be used given that it must extrapolate the number of moves required over the entire available domain with respect to the maximum snake length of 99. In a polynomial function, the end behaviours would govern the extrapolation for larger snake lengths beyond $l = 27$.

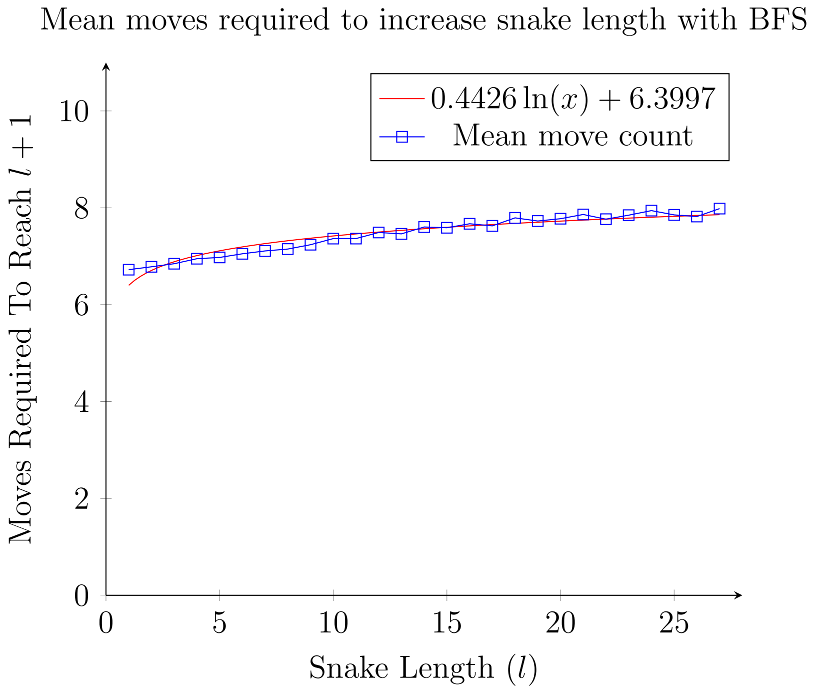


Figure 10

Therefore, observing that the graphed data points on the scatter plot appeared to be shaped concave down, I decided to use a logarithmic function to model the relationship. The corresponding trend line equation was computed in Microsoft Excel. Calculating Pearson's coefficient value to determine the validity of this model, the result is $R^2 = 0.9358$, indicating that this equation is a good fit. This is a logically suitable model, since the length of the shortest path should increase when more vertices are blocked by the snake body. Let $b(l)$ be the function representing the amount of moves required to get from length l to length $l + 1$ with the BFS approach.

$$b(l) = 0.4426\ln(l) + 6.3997 \tag{1}$$

Modelling the Hamiltonian cycle approach is a lot more simple. For any length l, the apple can be one of $mn - l = (10)(10) - l = 100 - l$ vertices. Therefore, assuming a uniform random distribution for the position generation of the apple, on average it should take $\frac{100-l}{2}$ moves to get to the apple. Using $h(l)$ as the function representing the amount of moves required to get from length l to length $l + 1$ with the Hamiltonian cycle approach:

$$h(l) = 50 - \frac{1}{2}l \tag{2}$$

Recall that function $T(l_s)$ models the **total** moves required for winning the game if we switch to the Hamiltonian cycle agent from the BFS agent at length l_s. Knowing functions $b(l)$ and $h(l)$, we can find the equation for $T(l)$ for grid size 100:

$$\begin{aligned} T(l_s) &= \int_0^{l_s} b(l)dl + \int_{l_s}^{100} h(l)dl \\ &= \int_0^{l_s} (0.4426\ln(l) + 6.3997)dl + \int_{l_s}^{100} (50 - \frac{1}{2}l)dl \end{aligned} \tag{3}$$

Doing integration by parts on the first term of the first integral and directly integrating the remaining terms of this equation, we have:

$$\begin{aligned} T(l_s) &= [0.4426l\ln(l) + 5.9571l]\Big|_0^{l_s} + [50l - \frac{1}{4}l^2]\Big|_{l_s}^{100} \\ &= 0.4426l_s\ln(l_s) + 5.9571l_s + 2500 - 50l_s + \frac{1}{4}l_s^2 \\ &= 0.4426l_s\ln(l_s) + \frac{1}{4}l_s^2 - 44.0429l_s + 2500 \end{aligned} \tag{4}$$

It should follow that l_s at the local minimum of $T(l_s)$ is the optimal length of snake at which the Hamiltonian cycle agent should be used instead of the BFS agent. However, looking at the graph of the function (see Figure 11), we observe that this local minimum occurs when $l_s \sim 80$. Since we know that the BFS approach cannot survive to that length, we must find a way to factor in the probability of the snake's death prior to changing to the Hamiltonian cycle approach so that the final approach may win the game.

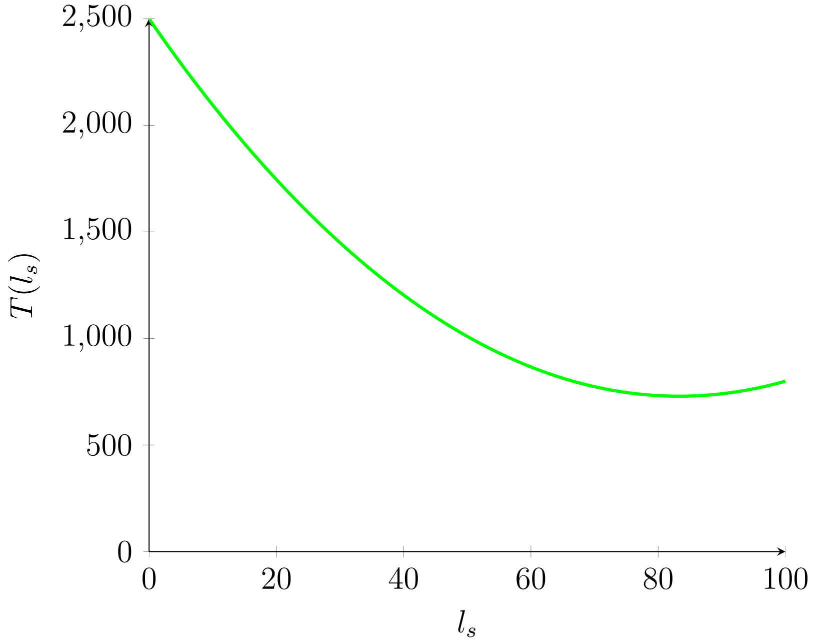


Figure 11: Graphing $T(l_s)$

5.2 Probabilistic Modelling

To add consideration of the snake's death we must model the probability of the snake dying at any given length. This relationship can be found by letting a BFS agent play 10 000 times and recording the dying length of each game. Dividing the count of deaths at each length by the total deaths, we find the probability of the snake dying at a given length. The results of this simulation are summarized below (see Figure 12).

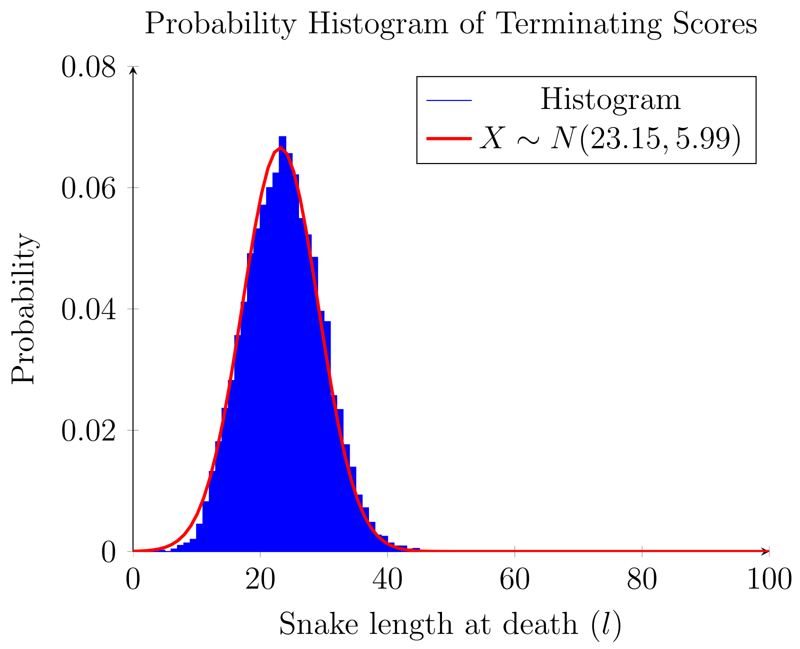


Figure 12

As can be seen from this graph, the snake's dying lengths fit a normal distribution with mean $\mu = 23.15$ and standard deviation $\sigma = 5.99$. Note that because the dying length of a snake is a discrete variable being approximated by normal distribution, which is continuous, a continuity correction is needed. Rather the the height of the curve, it is the area under a continuous distribution that gives the probability of an event, so for an integer value x from a set of discrete values including $\{\ldots, x-1, x, x+1, \ldots\}$, $P(X = x)$ can be approximated to $P(x - 0.5 < X < x + 0.5)$ [8]. The halfway points are the boundaries of the bars that represent the area under the curve for the discrete value x.

For adding this probabilistic model to $T(l_s)$, we will find the probability of the snake being dead by a certain length. If this probability is very high, then it can be concluded that it is not favourable to continue using the BFS approach beyond this length, and the autonomous agent should switch to the Hamiltonian cycle approach. Using $e(l)$ to represent the probability that the snake will be dead (and the game will end) at length l, we have:

$$e(l) = P(0 < X < l + 0.5) \tag{5}$$

5.3 Optimization

Finally, we can incorporate $e(l)$ into $T(l_s)$. For a constant k, we can now write $T(l_s)$ as:

$$T(l_s) = ke(l_s) + 0.4426 l_s \ln(l_s) + \frac{1}{4} l_s^2 - 44.0429 l_s + 2500 \tag{6}$$

The concept behind this equation is to penalize waiting longer to switch between approaches. The purpose of $T(l_s)$ was to model the number of total moves taken to beat the game by switching between the approaches at l_s, but since $e(l)$ grows at longer lengths as the probability of the snake dying by the BFS agent will increase the longer it gets, a "penalty" move count is added to $T(l_s)$. This will shift the local minimum of the graph towards a shorter switch length and will thus ensure that the switch length is not computed such that the snake will already be dead due to the BFS agent.

Since $0 \leq e(l) \leq 1$ whereas the total move counts are in the thousands, $e(l)$ is multiplied by a constant k. The bigger the value of k, the more the weight of the snake death penalty and the shorter the switch length of the snake. Referring back to experienced players who develop quick strategies to beat the game, their switch between the shortest path and the Hamiltonian cycle approach comes at a trade-off between finishing the game in a lower amount of time and dying early on. Essentially, k is arbitrarily based on how "risky" a player wants to be in pursuit of a quicker win time. The graph shown in Figure 13 below uses $k = 700$. Using a GDC to compute the first local minimum of the final version of $T(l_s)$, we solve for the optimal length $l_s = 17.779$. Note this result is a real number, not an integer length. Thus, in practice, $\lfloor l_s \rfloor$ will be used as the switching point in the final agent as the safer option between $\lfloor l_s \rfloor$ and $\lceil l_s \rceil$ in terms of ensuring the snake's survival.

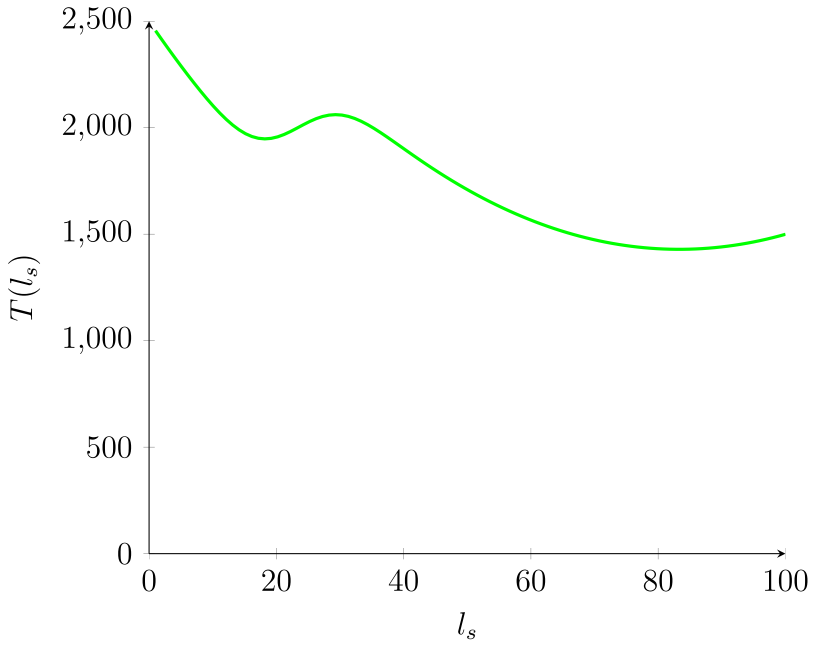


Figure 13: Graphing $T(l_s)$

6 Evaluation

Approach	Dying Length	Total Moves	Win %	Total Moves (Wins)
Breadth-First Search	22.89	164.77	0.00%	—
Hamiltonian Cycle	100.00	2488.57	100.00%	2488.57
Analytical ($l_s = 17$)	80.25	1454.93	77.61%	1871.94

Figure 14: Average dying length, move count and win % for each approach, 1000 runs

To determine the best approach, they can all be compared with respect to the data in Figure 14 when they are run for 1000 trials. As expected from previous reflections, if Breadth-First Search was a flawless approach where the snake would never be trapped, it would have the least moves. The pure Hamiltonian cycle approach guaranteed a win, whereas the analytical approach, despite only winning 77.61% of the time, takes 24.78% less moves than the Hamiltonian cycle approach. The win rate and move count can be tuned arbitrarily by changing k.

The process of this investigation was successful at comprehensively mathematically modelling the *Snake* game as well as generating a methodology for calculating the switch length for any m and n in the analytical approach. Transferring the logic to a computer program was also achieved. By contrast, one significant limitation was that experimental simulations were required to be conducted for finding a) coefficients for the logarithmic model of the move count function for the BFS approach and b) finding the correct μ and σ values for the normal distribution of the probability function $e(l)$. Thus, while applying the process undertaken to find l_s for $m = n = 10$ is robust, the dependence on large quantities of experimental data can become difficult and resource-heavy to compute for much larger sizes of m and n, making the overall approach not practical for the general case. Hence, a relevant extension could be to find a process not requiring experimental data. Additionally, m and n being constrained by the restriction of the Hamiltonian cycle limit the scope of the approaches used in the investigation. Another limitation is that by only exploring two base approaches, it cannot be conclusively determined that the overall approach is the most efficient way to win the game. For example, alternative path-finding algorithms other than BFS may be less prone to being trapped.

7 Conclusion

In conclusion, the most optimal algorithmic *Snake* game agent was the one using both base approaches. The analytical approach was more efficient than the pure Hamiltonian cycle and more closely modelled how human players won the game in short times. The only consideration to make when choosing this approach is that for lower values of k, the win rate may not be very high. Through this exploration I learned many new concepts — namely graph theory — and applied concepts I was already familiar with to a real-life problem. This experience taught me that math can significantly help optimize and analyse real-world problems and the theory I used in this investigation are key for many applications I use daily, such as Google Maps.

References

[1] Bose, Prosenjit, and John Howat. *Depth-First Search and Breadth-First Search.* Discrete Mathematics Study Center. July 12, 2015.
`https://cglab.ca/ discmath/graphs-applications.html`

[2] EasySpeezy. *He Thought I Couldn't Speedrun the Google Snake Game...* YouTube. November 15, 2020. `https://www.youtube.com/watch?v=O3bDZXOSAfk`

[3] Gadagkar, Shweta. *Traversing Graphs in Discrete Math.* Study.com. September 16, 2018. `https://study.com/academy/lesson/coloring-traversing-graphs-in-discrete-math.html`

[4] Galvin, David. *Discrete Mathematics, Spring 2009 Graph Theory Notation.* March 4, 2009. `https://www3.nd.edu/~dgalvin1/60610/60610_S09/60610graphnotation.pdf`

[5] Liu, Chuyang. *Algorithms.* GitHub, November 18, 2019.
`https://github.com/chuyangliu/snake/blob/master/docs/algorithms.md`

[6] Mitchell, John. *Hamiltonian Cycle Is NP-Complete.* Rensselaer Polytechnic Institute. February 2, 2019.
`http://eaton.math.rpi.edu/faculty/Mitchell/courses/matp6620/notesMATP6620/lecture06/06A_hamiltoniancycle.pdf`

[7] M. Sohel Rahman, and M. Kaykobad. *On Hamiltonian cycles and Hamiltonian paths.* Information Processing Letters 94, no.1 (2005): 37-41.

[8] Wilson, Steven. *Almost Normal Distributions with Discrete Data.* Milefoot.com. September 4, 2011. `http://www.milefoot.com/math/stat/pdfc-normaldisc.htm`

[9] Zagaran. *Hamilton Paths/Cycles in Grid Graphs.* Mathematics Stack Exchange. August 31, 2018.
`https://math.stackexchange.com/questions/1699203/hamilton-paths-cycles-in-grid-graphs`

Appendices

A C++ Code

```cpp
vector<Node> shortest_path(Node des) {
   // A "vector" represents a list and a "Node" is a data structure with
   // variables "i" and "j", defining a vertex's position on the grid.
   // The queue "q" is the list of pairs of vertices and associated paths.
   // In this structure, an ordered list of vertices is stored in the queue
   // to save memory. Taking these vertices two at a time, we have the path
   // and the last vertex of the list is the vertex associated with the path.
    queue<vector<Node> > q{{snake.back()}};

    // "vis" is an array to mark whether a node (vertex) has been visited. For
    // a node u, if vis[u] equals 1 then the node has been visited.
    vis.clear();
    for (Node u : snake)
      vis[u] = 1;
    vis[des] = 0;

    // Perform the BFS algorithm while "q" is not empty.
    while (!q.empty()) {
      vector<Node> path = q.front();
      q.pop();

      // If the target ("des") node is reached, return this path.
      if (path.back() == des)
        return path;

      // Add nodes adjacent to this node that haven't already been visited to "q".
      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 15: Code for performing a **Breadth-First Search**.

```
void build_cycle() {
  // Clear the list "cycle" where the Hamiltonian cycle representation will
  // be stored.
   cycle.clear();

   // The Hamiltonian cycle heuristic proposed by Chuyang Li consists of
   // finding the longest path between a vertex v and adjacent vertex u. This
   // longest path should visit each vertex exactly once getting from v to u
   // (although this may not always be possible, which is why this is considered
   // a heuristic). Finally, adding the edge vu to the end of the path will
   // complete a proper Hamiltonian cycle. Note that the implementation of the
   // "longest_path" method is also a heuristic proposed by Chuyang Liu,
   // available for viewing at: https://github.com/chuyangliu/snake/blob/master/.

   // "path_dir" contains this longest path.
   vector<Node> path_dir = longest_path(snake[0]), snake_path = dir_path(snake);

   // Since we are constructing a Hamiltonian cycle when the snake will also have
   // a body, "path_dir" contains the longest path between the snake's head and
   // its tail. To complete the cycle, rather than adding edge vu as specified
   // above, the edges connecting each body part to the next, starting with the
   // tail to the head, will complete the Hamiltonian cycle.
   for (Node u : snake_path)
     path_dir.push_back(u);

   // Store this path in a computer friendly manner in the "cycle" list.
   Node cur = snake.back();
   for (Node u : path_dir) {
     cycle[cur] = u;
     cur = cur + u;
   }

   // Store the size of this cycle to check from a different function if this
   // cycle size is mn and thus verify that the heuristic worked.
   cycle_size = path_dir.size();
}
```

Figure 16: Code for generating a **Hamiltonian Cycle** (heuristic) [5]