Solving Snake Using Hamiltonian Cycles

Advanced Algorithms: Assignment 4

Leo Shchurov, Ajani James Bilby

November 2022

Repository: https://github.com/arduano/snake-solver

1 Introduction

Snake is a simple game from the late 70's where you play as the snake that moves around a 2d grid with a very long tail and needs to eat apples to continue growing its tail. The tail starts out short, and the more apples you eat the longer it grows. If you crash into your own tail, you lose. The only way to win is to fill every cell of the world.

"Solving" snake should be algorithmically possible, and we propose multiple solutions to "finish" snake in a provably unkillable way.

2 The Problem

The snake needs to follow a path where it can guarantee that it won't die until it grows to fill up the entire board. We approached this problem from several different angles, iteratively building up on previous solutions. All of our solutions involve Hamiltonian cycles, as it's the best way to guarantee a path throughout the grid that touches every single cell once.

2.1 Assumptions

- World width and height must both be even numbers.
- World width/height are equal, e.g. a size 60 world is 60x60 cells (Although not strictly necessary, this helped work with some of the implementations easier).
- The snake world isn't cyclical unlike some snake implementations.

3 Zig-Zag

This method simply zig-zags down the entire domain leaving a single column remaining for the snakes return back to the beginning. This method is very simple to compute, however it means the snake needs to travel the entire domain of the board to revisit any single point, meaning since the food cannot spawn within the snake it must travel the length of the board to consume any food. This method effectively creates a Hamiltonian cycle without having to store any extra metadata to generate it.

3.1 Computing Complexity

The time/space complexity is very simple as the snake can simply either generate the path one section at a time if it knows it's position and the board size, or compute the entire circuit once and reuse this value. Below we take n to be the number of total grid cells, and compute the entire circuit the snake travels.

Time complex $O(n)$
Space complexity $O(n)$

However this is the complexity over the entire life time of the game, as the path only needs to be calculated once, then can be reused for the result of the game until completion. Using this metric to compare to the later Dynamic Hamiltonian algorithm will create too much complexity to meaningfully compare the two, so instead we will consider the computation required after each successive consumption of food, which would be.

Time complex $O(1)$
Space complexity $O(n)$

As the snake can simply reuse it's path generated during the first iteration.

3.2 Observations

The Zig-Zag algorithm is very computationally efficient, however it results in a path which is the least optimal option without revisiting any point in a single path. While the computation time is good our goal is instead to get the snake to outlive the board (grow to the length of the board) as fast as possible.

4 Static Hamiltonian

The goal of this method is to generate a tree which spans over the entire grid space spanning a minimum of two cells. It must always span at least two cells because then there is always enough room for the snake to go down a path and come back. Once the tree is generated the snake can simply follow one wall of the tree tracing the whole space, allowing it to cover the entire domain as the tree touches all cells and contains no loops which could create unreachable zones.

This simple version only generates the tree once, and then simply follows the same path for it's entire life, while this is

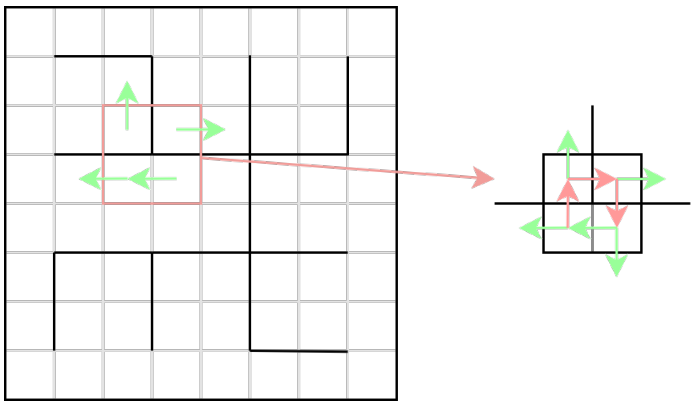

Figure 1: Demonstration of how the minimum spanning tree is laid out in the snake grid, and how the snake paths around it.

not much of an improvement to zig-zag, it does provide novel solutions, and has the potential for further optimisations we will discuss later.

4.1 Implementation

4.1.1 Tree Generation

To create the tree first we start of creating a sparse connected graph between the boarders of every second grid square. This creates a grid where every node is connected to it's vertical and horizontal neighbours - but not it's diagonal neighbours. These connections are also given random weights to aid in creating a novel tree.

A minimum spanning tree is then generated from the graph using the node to the top left of the food as the seed point, generating the tree using Kruskal's algorithm.

4.1.2 Path Tracing

Updated to reflect the new Zig-Zag complexity

After the minimum spanning tree is generated, creating a Hamiltonian cycle path around the tree is fairly trivial by following a simple set of rules.

The snake grid is divided into 2x2 cells, and each 2x2 cell group would contain a spanning tree node in the center (as seen in Figure 1). Then each cell gets 2 possible directions: going clockwise along the 2x2 cells, or going outwards (along the tree edge, left of the clockwise direction). First, try to go clockwise, if going clockwise is blocked by a tree edge then go outwards. If following this pattern, the snake will move around the minimum spanning tree while hugging the right edge, reaching every single cell in the grid.

4.2 Computing Complexity

Minimum Spanning:
The MST is generated using Kruskal's algorithm so the n nodes with $2n$ edges is put into it's complexity functions to derive:
Time complex $\theta(\frac{n}{2}\log\frac{n}{4})$
Space complexity $\theta(\frac{n}{4}+\frac{n}{2})$

Path Tracing:
The snake follows the tree based on simple deterministic rules to traverse over all grid spaces:
Time complex $\theta(n)$
Space complexity $\theta(n)$

Overall:
Time complex $\theta(\frac{n}{2}\log\frac{n}{4}+n)$
Space complexity $\theta(\frac{7n}{4})$

However this overall calculation is the total amount of calculation needed over the entire lifetime of the game, and again like Zig-zag we will convert this to the food-to-food computation complexity to make it more easily comparable to the 'dynamic hamiltonian method'. As we can cache and reuse the results for the entire game we end up with a complexity of
Time complex $\theta(1)$
Space complexity $\theta(n)$

4.3 Observations

As this method also follows the same path for it's entire lifetime traversing the whole board, it's overall behaviour is exactly the same as the Zig-Zag method. However it takes greater computational complexity to achieve it's results. The only benefit of this exact algorithm is if novelty is desired. Otherwise Zig-zag is a more efficient algorithm to use.

5 Dynamic Hamiltonian

This approach is an extension from the previous Hamiltonian cycle approach, however it dynamically grows the tree and involves A* pathfinding to attempt to find the shortest path to the food.

The algorithm involves the following steps:

- The snake is already in a state where its path is stenciling out a section of the previous minimum spanning tree. The first step involves going through the snake one cell at a time, and determining the tree edges that should exist to continue stenciling out the current snake location. This is done by checking each cell and the direction from that cell, if the direction is clockwise along the 2x2 path tracing grid then there is no edge, and if it's outwards then there's an edge, if it's clockwise then we mark the edge as "overlapped by current snake".
- The next step is to perform A* pathfinding from the food location. However, there are certain rules that this follows. For example, it can only path outwards in valid directions that the snake can move from (clockwise and out), and it can't move in a direction that could create a new connecting edge to an existing edge if an edge exists (as seen in Figure 2). This ensures that only valid directions are generated along the A* grid, however it also means that there can be large gaps with unreachable

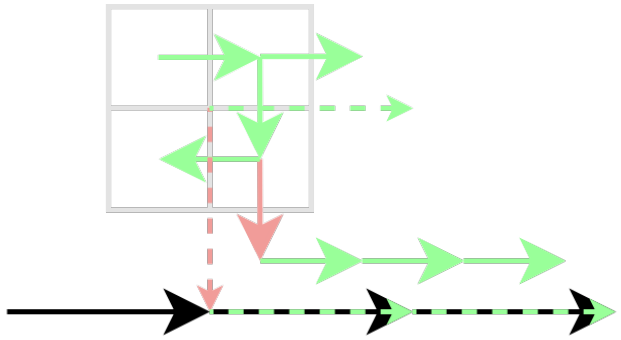

Figure 2: Demonstration of valid steps that the pathfinding algorithm considers. It checks if an edge may be created when taking a step, and if the edge creates a new connection then it's an invalid step as that may create loops. But if it doesn't make a new connection or the edge already exists then it's valid.

values in the grid too. However, it is impossible for a snake's head to end up in those graps as the snake is always hugging an edge (the edge is always regenerated in the previous step) and the pathfinding rules always allow continuing to hug an edge.

- After the pathfinding grid is generated, it is then traversed from the snake's head towards the food following the valid stepping rules. A snake path isn't generated here, instead new tree edges are placed whenever the traversal chooses an outwards direction. These edges will then guide future tree generation. Also, edges that overlap with the future snake path are marked on the grid too.
- Now, all we have is a partial tree and also a set of edges that overlap with the current snake path and the future snake path. From here, we look for any empty edge that's connected to the tree that we can seed the tree growth from. Instead of using Kruskal's algorithm, it was easier to use a depth first search algorithm so that the seeded tree only fills up the space that's available to it. After all possible edges that aren't covered by the current and future snake are filled from, we check if there are any edges that are covered by the future snake can be seeded from. If none are left, then we know the direct route to the food. If there are some hole still left over and we need to modify the path by branching off of future snake edges, then we do that but also mark the current route as "indirect".
- After the entire minimum spanning tree is generated, we then path trace around the spanning tree to form the new Hamiltonian cycle.
- Before outputting the final path, we check if the path is indirect (as decided in step 5). If the path is indirect and jittering is enabled, then we only output the next X steps where X is a constant (e.g. 10 or 1). Otherwise, we output the entire path. Jittering ensures that the snake will re-path towards the food as soon as an opening is found again, e.g. if the tail moves out of the way.

5.1 Computing Complexity

The time complexity of each component in the algorithm is as follows:

- Tracing the snake to lock the existing tree edges requires stepping down the snake 1 cell at a time, making the complexity be $O(n)$ where n is the length of the snake.
- A* Pathfinding from the food, visiting each cell, making the time complexity be $O(n)$ where n is the number of cells in the snake world.
- Tracing the path towards the food along the A* pathfinding grid, stepping through each cell towards the food, with the time complexity being no more than $O(n)$ although each step usually requires deciding between 2 directions and picking the best one until the minimum value is reached, so the average case time complexity is $\theta(log(n))$ unless the snake is locked into having only 1 direction choice at each cell that forces it to travel around the whole grid.
- After the path's tree branch is traced, the rest of the tree needs to be grown. Valid seed locations are chosen, one at a time, and the tree is seeded from them. Although the seed function may be called multiple times, each node in the tree is only visited once when growing, so the complexity is $O(n)$.
- And lastly, a path is traced around the tree, the function of which has the time complexity of $O(n)$ where n is the number of cells in the snake world.

Overall the time complexity is directly proportional to the number of cells in the snake world.

5.2 Space Complexity

All of the grid graphs (grids with edges between the cells) and the pathfinding grid are stored in 2D arrays that are proportional to the size of the snake world. The space complexity is $O(n)$ where n is the number of cells in the snake world.

5.3 Effectiveness

Unlike the previous two algorithms, this algorithm "finishes" snake much more optimally due to the guided pathfinding aspect. However, it becomes significantly inefficient if it has to seed edges from the future snake path, as usually that results in a tree that spans the entire grid which the snake has to trace before it reaches the food, creating virtually the least optimal path from the snake head to the food. However, this can be solved by occasionally re-pathing to check if a new more optimal path has opened up.

5.4 Observations

Although the pathfinding algorithm finds the most optimal path, sometimes it leaves no room for the tree to grow without

overlapping the future snake, which causes inefficiencies. It would be more efficient in some circumstances to leave some space around other snake parts when pathing around them to ensure that the tree has a place to seed from without overlapping the future snake path, however algorithmically deciding that would be very complex.

Also, when a jitter of 1 is used, the snake appears to choose an interesting space filling pattern. Although it was entirely unintentional, it's interesting that such a repeatable pattern is created even with the randomness involved (Figure 3).

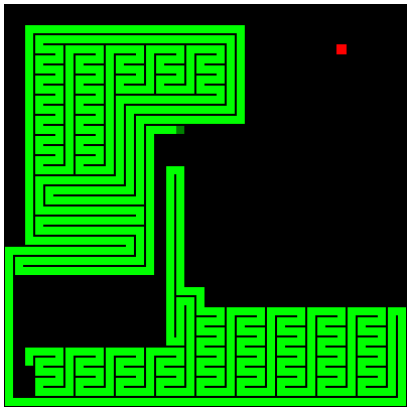

Figure 3: Repetitive patterns emerging from a jitter setting of 1 even though the rest of the generated tree is always random.

6 Results

Running each algorithm 100 times on a size 60 board shows the clear difference between the different algorithms (Figure 4).

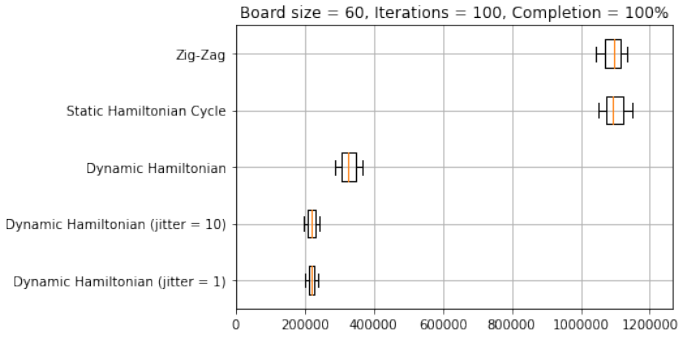

Figure 4: Average number of steps to fill 100% of the board using each algorithm, 100 runs

We observed that in the later stages of the algorithms, there is less space left on the board so pathfinding doesn't have as much of a benefit, so we ran the benchmarks again to see how many iterations it takes to fill 10% of the board and it showed a much larger difference between the algorithms (Figure 5).

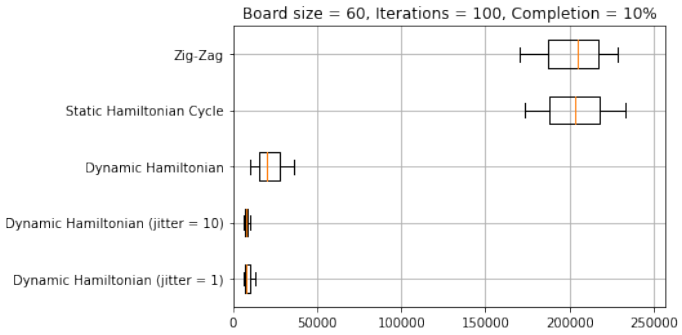

Figure 5: Average number of steps to fill 10% of the board using each algorithm, 100 runs

We have no included the benchmarks on other sizes of boards, because the trends were almost identical to the 60x60 sized board, however the deltas between algorithms were proportionally smaller.

We also ran performance timing benchmarks on the functions, getting the average execution time of the "get_next_path" function for each algorithm, for different world sizes. We theorized that the execution time for each algorithm to generate a new path is $O(n)$ where n is the cell count in the world, and the benchmark shown in Figure 6 reflects that. The benchmark was run by completing a snake world from start to end for each world size, timing the average "get_next_path" execution time, and repeating this process 20 times. Then dividing the final times by the maximum time in each algorithm to relatively represent them. The graphs converge to a generally straight line, which shows that our assumptions are likely accurate.

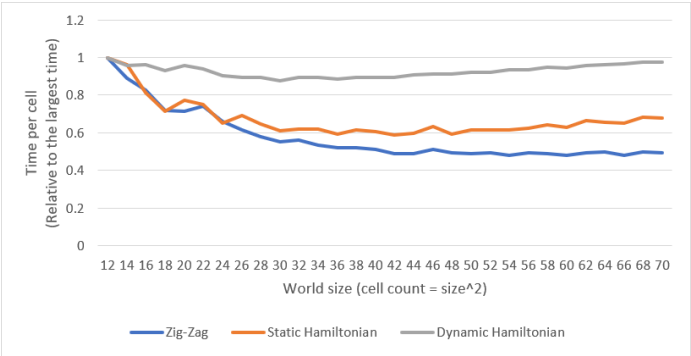

Figure 6: Benchmark of time taken to calculate a new path divided by cell count, for different world sizes.

7 Conclusion

The 'Zig-Zag' and 'Static Hamiltonian' algorithms both perform almost identically as they both always follow the same path for the entirety of the game, so the snake will always have to travel an average of $\frac{n-length}{2}$ distance away from the food resulting in identical behaviour.

However the improved dynamic version is on average ≈ 3.4x better than the static circuit following algorithms. However when you only look at the first 10% of each game the dynamic approach shows it's strength as it has a larger domain to potentially optimise yielding a $\approx$ 10x improvement with a

further ≈ 2.63x on top of that when we restrict re-calculate (jitter) the dynamic approach every 10 steps when there's no direct path.

Overall, our most efficient results show a ≈ 5.5x improvement over brute force approaches to complete snake, and a ≈ 31x improvement to fill 10% of the board.

8 Discussion

We chose not to include a brute force algorithm to compare how effective our 'Dynamic Hamiltonian' algorithm's path was due to the uncomputability of the problem space. At any point in time the snake can travel in 1 of 3 directions. The food on average will spawn $\frac{n^2}{2}$ meaning in the best case $3 \times \frac{n^2}{2}$ choices will be made. If you account for the fact that the choices of previous cycles (from food to next food) impact future cycles, then the number of possible choices expands massively to:

$$\begin{aligned} \text{Choices}(\text{Starting Length}, n) &= \prod_{l=s}^{n-1} \frac{3}{2} \times n^2 \\ &= \frac{3n^2}{2}^{n-s-1} \\ \text{Choices}(60, 5) &= 3.096 \times 10^{163} \end{aligned}$$

Hence we can determine the problem itself is NP-Complete as the problem space is more than polynomial time on a deterministic machine, and verifying a solution to the problem can be solved in polynomial worse case time:

$$\begin{aligned} O(n) &= \sum_{l=s}^{n} n^2 - 1 \\ &= (n-s)(n^2-1) \\ &= n^3 - sn^2 - n - s \end{aligned}$$