Genius Snake

Eliška Svobodová
ČVUT–FIT
svoboel5@fit.cvut.cz

17. května 2020

1 Introduction

This project studies the game Snake and applies different approaches to reach the highest score possible. Game represents a path-finding problem in dynamically changing environment. To win the game, snake also has to come up with some strategy to not become stuck in a dead end.

2 Methods

Snake has two main goals:
1) Avoid running into an obstacle (wall or body)
2) Find and eat apples

2.1 Hamiltonian cycle

First problem is solved by using Hamiltonian cycle. Game field can be interpreted as graph with each cell representing a node with four edges leading to its direct horizontal and vertical neighbours. Hamiltonian cycle connects all nodes in the graph while visiting each node exactly once. Snake can circle round and never bump into its body, unless its body already covers the whole maze. By following this strategy, snake completely ignores apples, but, because cycle always covers the whole filed, it will also visit a cell with the apple. This method has two flaws. Firstly, Hamiltonian cycle doesn't have to exist. It is granted, that the game field is rectangular, but if it has odd number of cells, there is no way snake can visit each node and do it exactly once. There will be always one node that will remain not visited. Secondly, it is extremely ineffective. Apple is placed randomly in an empty cell and snake doesn't care where.

2.2 A* Algorithm

Second goal, finding the apple, can be solved by any of the path-finding algorithms. A* algorithm was chosen, because of its time efficiency and because it guarantees an optimal path. Snake calculates the shortest path to the apple and executes it, then again and so on. When path doesn't exist, snake moves randomly. This approach fails when apple appears in the dead end (surrounded by other body parts and walls). The problem is that snake doesn't take into account that it has to have an escaping route in order to survive.

2.3 Genetic Programming

To find balance between the two main goals, genetic programming was used, breeding the optimal driver function. It is composed from terminals (direction, in which snake is supposed to continue) and functions (giving it information about its surroundings). Some examples of functions are: if_wall_forward (next square in the direction snake is moving is a wall), if_food_forward (if an apple is in line with snake's head), if_moving_down (snake is heading down). Driver function is a tree that is executed before each step of the snake and returns the next move (forward, left, right). In the beginning, driver functions are randomly generated and tested in the game. Those with the highest scores are bred with crossover and mutation operators and produce offspring which then compete with its parents. Only fittest driver functions are passed to the next generation and the cycle repeats. Driver functions with "good parts"usually reach higher score and they have better chance at passing its genes into next generation. That way, successive populations are supposed to be closer to the optimal solution.
One of the biggest challenges in this approach is, that we have no way telling which parts of the driver functions are "good"(= avoid obstacles, move toward food), like: [if_wall_forward: SNAKE_MOVE_LEFT, ...] and which are "bad"(= prevents snake from eating apples, makes it bump into obstacles), like: [if_food_forward: SNAKE_MOVE_RIGHT, ...]
All breeding is random, we pick random sub-trees that will swap and random nodes that will be mutated. Only by selection of the fittest we converge to solution. Another problem is that we might get stuck in the local maximum. That means, population will consist of quite successful and similar driver functions but they won't be optimal.

3 Results

Methods were tested on the game field of size 22x19 squares (max score = 413) and compared according to reached scores.

3.1 Hamiltonian cycle

If Hamiltonian cycle exists in the graph, this method always reaches maximal score 413. As mentioned above, this comes for the price of very low efficiency. Every apple is put approximately

$$\frac{number\_of\_free\_cells}{2}$$

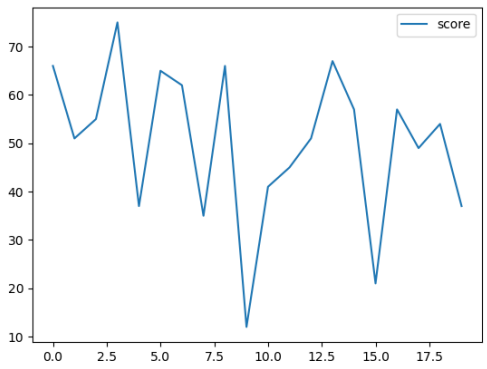
squares away. So, in the field 22x19 and snake 5 squares long in the beginning, that gives

$$\sum_{n=1}^{413} \frac{n}{2} = 42745.5$$

for snake to visit. If has graph odd number of cells and Hamiltonian cycle cannot be created, there is an equal chance, each time snake eats an apple, that apple will appear in one cell, that isn't covered in the cycle and snake will starve to death.

3.2 A* Algorithm

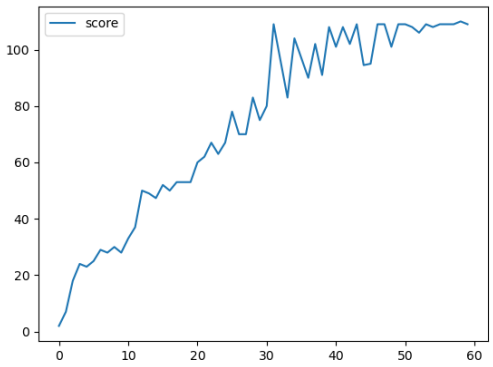
This algorithm doesn't prevent snake from running into dead ends which, in the end, proved to be the fatal flaw. Since snake is long enough to create a loop, apple will (sooner or later) appear in the area from where there is no escape. Graph 1 shows what scores were reached in 20 runs of the A* algorithm.



Obrázek 1: A* algorithm scores

3.3 Genetic Programming

Population from the most successful run was improving in the first 50 generations. Strategy was evolved during that time. Snake ran near walls counterclockwise, until food on the left appeared. Then it headed left, until there was a body in front of it. Then it continued up, to the wall and resumed its cycle near the wall. This pattern scored around 110 points in the end. None of the runs of the genetic programming was capable of breeding the optimal driver function (giving the highest score possible). Possibly, that was caused by premature convergence to the local maximum. Then, most of the individuals have very similar driver function and further improvement would require an extremely good mutation. Graph 2 shows highest scores from each generation.



Obrázek 2: Highest scores

There is a steep climb in the beginning, where driver functions learn, what effect do functions (like if_food_left) have and they develop strategy. The fluctuation near the middle is caused by randomness. The best individual was lucky, that food appeared longer on non-deadly positions.

4 Conclusion

Half-term work implements three different approaches to solving the game Snake. It studies their strong and weak aspects and compares the results. Project can be further evolved, especially the genetic programming part. For example, evolve more separate populations at once and breed them together in the end, so different strategies are developed and combined. It would be also useful to be able to safe the best driving functions from the run, so they can be applied to future runs of genetic programming method.

Reference

[1] Tobin Ehlis. Application of genetic programming to the snake game. online, 2000. [cit. 2020–17–5] `https://www.gamedev.net/tutorials/programming/artificial-intelligence/application-of-genetic-programming-to-the-snak`

[2] Jason Brownlee PhD. Clever algorithms: Nature-inspired programming recipes. book, 2011. [cit. 2020–17–5] http://www.cleveralgorithms.com/nature-inspired/evolution/genetic_programming.html.

[3] Frank Klawonn Christian Moewes Matthias Steinbrecher Pascal Held Rudolf Kruse, Christian Borgelt. Computational intelligence: A methodological introduction.

[4] Jan Žižka. Some interesting source. online, 1415. [cit. 2020–02–09] http://zizka.trocnov/husiti/conspiration.pdf.