A* algorithm for Snake game

Marek Provazník

Faculty of Mechanical Engineering, Brno University of Technology
Institute of Automation and Computer Science
Technicka 2896/2, Brno 616 69, Czech Republic
200640@vutbr.cz
github.com/AddictionLord

Abstract: *This paper is focused on A* artificial intelligence algorithm and used heuristics for game Snake implemented in Python programming language using object oriented programming.*

Keywords: *Artificial intelligence, AI, algorithm, A*, game, Snake, Python, pygame, OOP, object oriented programming*

1 Introduction

This paper is a technical report for semestral project to artificial intelligence algorithms subject. The goal of this project is to choose programming language, artificial intelligence algorithm and make an application. Python programming language was chosen to implement A* path finding algorithm to simple snake game made from scratch.

2 A* algorithm

A* algorithm is one of many known pathfinding algorithms like Djikstra's algorithm, bread first search algorithm, depth first search algorithm. It belongs to the family of informed search algorithms because it uses additional knowledge about search space. There are also many various modifications of A* search algorithm such as Anytime A*, D*, Theta*, New Bidirectional A* (NBA*). These are widely used in computer games industry due to it's good features and performance.

2.1 Evaluating function

Algorithm repeats several steps in a loop over and over until it meets certain condition. Most promising state is expanded in each iteration. The state is chosen according to evaluating function which assign a value to every expanded node. In the beginning the expanded state is, obviously, starting node [1]. The evaluating function is given by formula (1).

$$f(i) = g(i) + h(i) \tag{1}$$

Where i stands for current node, $f(i)$ is resulting value of evaluating function (the less the better), $h(i)$ is value of heuristic function and $g(i)$ is value given by function (2).

$$g(i) = g(j) + c(i, j) \tag{2}$$

Here j stands for ancestor node to i node. Value $g(i)$ is distance of node i from the starting node calculated as ancestors distance from starting point $g(j)$ plus distance between nodes i and j - $c(i, j)$ [2]. Note that snake game state space in a discrete space is made out of cells/nodes of same sizes and distances so the value $c(i, j)$ of between neighboring nodes in the source code is always set to 1. This could vary if the algorithm is used for path search in graphs, where distances (edges) between nodes could acquire any value.

2.2 Heuristic function

Heuristic function $h(i)$ is used by evaluating function to estimate distance between node (i) and target node. Very important part of heuristic function is metrics which has a huge impact on value of evaluation function. In [2] several metrics system are analyzed and explained. In this paper only two basic ones are mentioned below and only one of them is used in the project.

2.2.1 Euclidean metric

Euclidean metric is simple to calculate. If coordinates of two points are known, euclidean distance can be calculated using Pythagoras theorem. Let p_1 be a starting point 2D in space with cartesian coordinates (x_1, y_1) and target point $p_2 = (x_2, y_2)$, then euclidean distance between those points corresponds to (3).

$$h(i)_{euclidean} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3}$$

In the figure 1 euclidean distance from point p_1 to point p_2 is shown [2].

2.2.2 Manhattan metric

Manhattan metric is used in the project because it was considered the best form due to snake movement in discrete space. According to (4) absolute values of distances in x axis and y axis are summed up [2].

$$h(i)_{manhattan} = |x_2 - x_1| + |y_2 - y_1| \tag{4}$$

In the figure 2 it is possible to see how the calculated distance looks like. Without any obstacle in the way, snake would follow exactly this path.

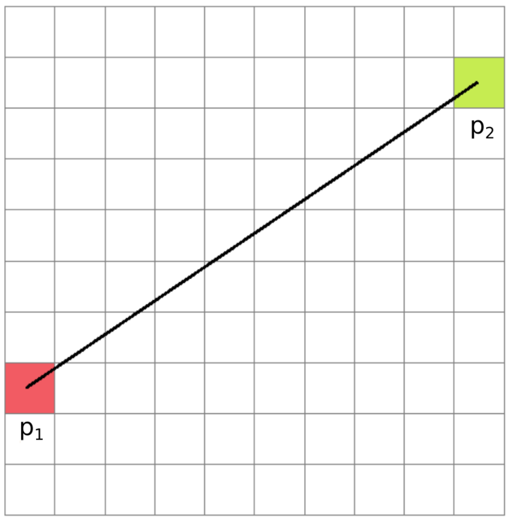

Figure 1: Euclidean metric [2]

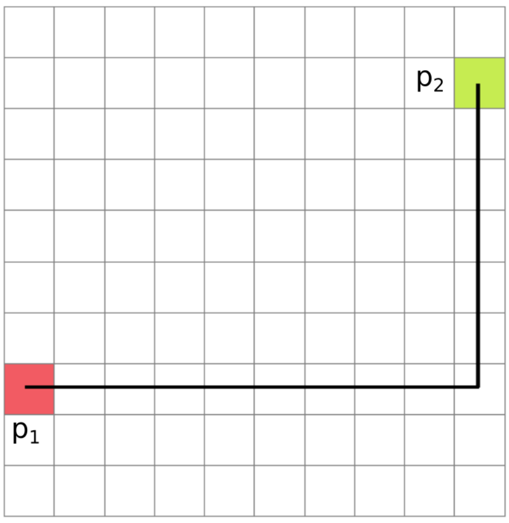

Figure 2: Manhattan metric [2]

2.3 Algorithm

In the picture 3 algorithm is written in pseudocode. First lines defines two queues - OPEN and CLOSE both empty. Starting node is inserted into the OPEN queue and $g(i)$ value for start point is initialized to 0 (considered distance from itself). On the fourth line code enters while loop with condition. The condition will make the code run until the OPEN queue is empty. If the OPEN queue is empty, while loop breaks and path is not found. Next step is to choose node u from OPEN with the least $f(i)$ value and check it with target node. If it's coordinates are equal, while loop breaks and the path is found. In next line for each neighbour node to u

node $g(i)$, $h(i)$ and $f(i)$ values are calculated and the node is inserted to OPEN queue if not already there. If it is, $g(i)$ values are compared and node with less value belongs to OPEN queue. Finally chosen node u is replaced from OPEN to CLOSED queue and while loop runs again [2].

```
1: $open \longleftarrow start$
2: $closed \longleftarrow \emptyset$
3: $g(start) = 0$
4: while $open \neg \emptyset$ do
5:     $u \leftarrow Extract - Min(open)$
6:     if $u == t$ break then
7:     end if
8:     for all edge $e = (u, v)$ do
9:         $distance \leftarrow g[u] + weight[e]$
10:        if $v \notin open$ then
11:            $g[v] \leftarrow distance$
12:            $h[v] \leftarrow Distance - To - Target(v, t)$
13:            $f[y] \leftarrow g[v] + h[v]$
14:            $open \leftarrow v$
15:        else
16:            if $distance < g[v]$ then
17:                $g[v] \leftarrow distance$
18:                $h[v] \leftarrow Distance - To - Target(v, t)$
19:                $f[y] \leftarrow g[v] + h[v]$
20:            end if
21:        end if
22:    end for
23:    $closed \leftarrow u$
24: end while
```

Figure 3: A* algorithm pseudocode [2]

2.4 Implementation

For implementation of A* algorithm object-oriented programming style was chosen. Two classes were used. First class *State* which instances represents nodes in path search. Constructor parameters represents actual node coordinates, coordinates of his ancestor and target coordinates. The class has methods for counting $f(i)$ and $g(i)$ values, heuristic and two extra methods to return $f(i)$ and $g(i)$ values which are private in this case. Another class *AI* has constructor with parameter snake which represent reference to snake instance and is necessary to get additional information about snake like position of his head and body. Main method of this class is called *A_star* with parameter *start* and *target* (coordinates representation) and it runs above mentioned while loop. It gradually calls another methods like *sort_and_pick* to sort OPEN queue and pick node with correct value, *expand* and *check_expanded* to expand chosen node and check on this new expanded nodes, *count* to instantiate them and calculate needed values and finally $find\_path$ to find and return final path from start to target.

3 Code

For this project object-oriented style of programming and Python's *pygame* library were used. Each class is structured to different file to keep the code structured and readable. Game is launched by instantiating *Game* class which instantiate another classes *Snake*, *Apple* and *AI*. Then *main_menu* method is called where four instances of class *Button* are made and then while loop of main menu is opened. In the figure 4 main menu is shown. Code runs in while

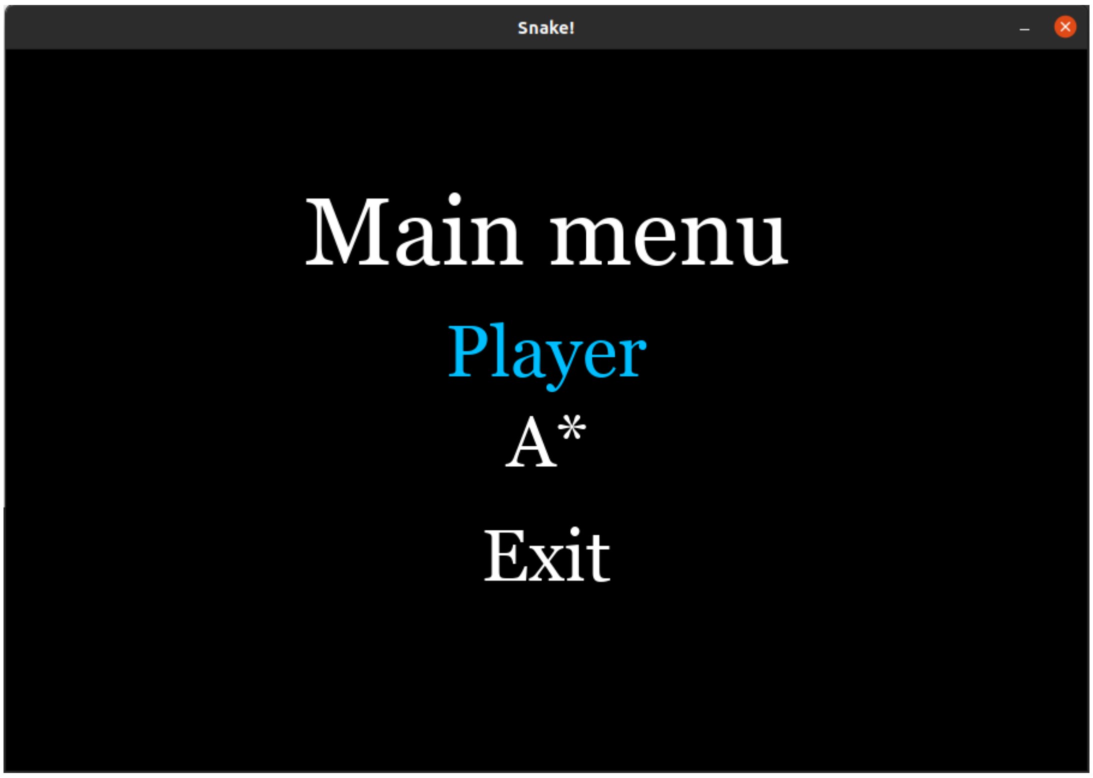

Figure 4: Main menu

loop and every time checks for several events. Cursor position is tracked. *Button* method *is_over* returns true if cursor is over the exact button and in this case button's text colour turns blue. There is option for play game yourself (button *Player*), let artificial intelligence play the game (button *A*∗) - in those two cases *main* method is called. Last option is exit the game (button *Exit*). At the end of the while loop method *graphics* is called which purpose is to to draw buttons and background. Optionally it can draw the game itself, it depends on entered arguments.

3.1 Game loop

If button *Play* or *A*∗ is pressed, code is redirected to *main* method which contains while loop the so-called game loop. This loop is the heart of whole game. It is set at 60 frames per second and performs three basic tasks shown on figure 5. First it handles game events like clicking, key pressing et cetera. Then it updates game state based on game events. This means it turns the snake left or right according to player's or AI's decision, checks for collision and if the apple was eaten. Methods doing this kind of tasks are called *set_movement*, *artificial_intelligence*, *check_eating* and *check_collision*. If called, game state is updated - snake's position is incremented/decremented by one (this causes movement) or snake's body grow. Game crashes in case of snake's collision with his own body or game window border. Finally these changes are drawn to screen by already mentioned *graphics* method [3].

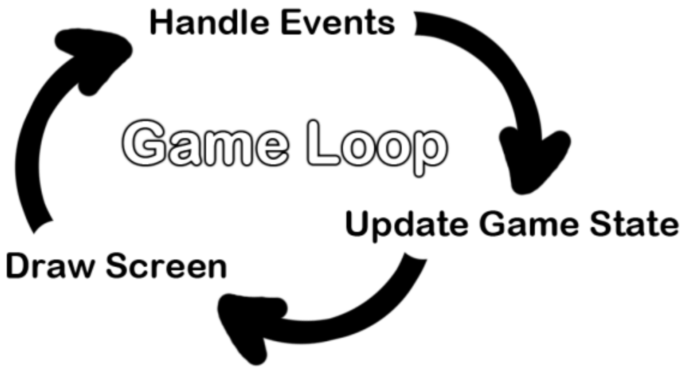

Figure 5: Game loop [3]

4 Conclusion

The game was successfully built with Python's *pygame* and *sys* library. No other libraries were used. There obviously is space for code improvement but the core of the game runs just fine. Author decided to build the game for scratch although A* algorithm could be implemented to already existing open source snake game. The decision was made due to learning reasons in Python programming, game development and *pygame* library. First thing to improve could be in game loop connection with A* algorithm. Specifically game crashes if snake blocks his way to apple by his own body. This happens because A* algorithm is not able to find the path. Next thing is new apple generating. It may slow the game down in case of snake occupying majority of cells because the coordinates of new apple are randomly generated and then checked for collision with snake. Some other improvements could be done in way of minimizing/maximizing the game, adding score and obstacles in snake path et cetera. Although the code is not perfect, A* algorithm implemented to game and the game itself works so the project is considered successful.

References

[1] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[2] Lenka Maňáková. Pokročilé metody plánování cesty mobilního robotu. *Vysoké učení technické v Brně, Fakulta strojního inženýrství*, 2020.

[3] Al Sweigart. *Making Games with Python & Pygame*. 2012.