Snake Game using A* Algorithm

CS254: Introduction to Artificial Intelligence

December 5th, 2022

Yasmin Kamal-Deen and Bernd Opoku-Boadu

Dr. Ayorkor Korsah, Prof. Michael Opoku Agyeman

Cohort A

Snake Game using A* Algorithm

Introduction

Since the programming of the snake game in 1997, it has become trendy and has taken many forms. In the game, the snake searches for an apple to swallow to grow. Search is an increasingly important concept in Artificial Intelligence that can be applied in real life. In computer science, search is the process of going from a start state to a goal state through a series of actions and states. Search is popularly used in game theory. In an attempt to expand our knowledge of search algorithms, this project aims at using an A* search algorithm to replicate the popular snake game.

The A* algorithm is a search algorithm that finds the shortest path from an initial state to a goal state. This way, not only does the snake reach its goal (an apple that it has to swallow to grow), but it reaches the goal using the shortest path. To visualize the algorithm and enhance the observers' experience, we will attempt to build a simple Graphical User Interface using Python's popular library, Turtle.

Background

The A* algorithm was first published in 1968 by three professors at Stanford Research Institute. It is often seen as an improvement of Dijkstra's algorithm. Another algorithm similar to the A* algorithm is the Uniform Cost search. The difference between the A* algorithm and Dijkstra's algorithm and uniform cost search is that the A* algorithm is informed. This means that the algorithm is aided by an informed guess of how close it is to reaching its goal. A heuristic (estimate) and actual distance is used to calculate an f-value which is the sum of a g-

value (the distance from a parent to the current state) and an h-value which is the distance from the current state to the goal (Barnouti et al., 2016).

$$f = g + h$$

The f-value is used to determine the best path needed to be taken to reach a goal from a starting point. The general approach for the A* algorithm is given below.

- Initialise an open list to keep track of unexplored states. The open list has states with a corresponding f-value
- Initialise a closed list to keep track of explored states.
- While the open list is not empty, find the state with the least f-value and pop it of the open list.
- Generate the successor states of the state with the least f-value and set the new states' parent to the previous state from which the new states were generated.
- For each successor state generated, if the successor state is the goal, stop the search and return the path taken to get there. Otherwise, calculate the g and h-value for the successor state. The f-value for the successor is derived from the g and h-values.
- If a node with the same position as successor state is in the open list and has a lower f-value than the successor state, skip the successor state. The same thing applies to the closed list. Otherwise, add the state to the open list.
- Add the state whose successor states were generated to the closed list since it has been explored now (Sharma et al., 2012).

Methodology

A user-played snake game was implemented first to understand how an AI can be made to play it using the algorithm. The implementation was done using the Object-oriented programming (OOP) approach with a class in each file that was used. The classes used were

- snake, which was used to create the snake that can move in four directions (up, down left or right),
- the food class, which was used to create the apple that the snake has to find and eat,
- the scoreboard class, which was used to display the points given to the snake when it eats an apple,
- node class, which was used to help integrate the search with the snake

The first three classes are implemented using the snake module. The node class in the image below was used to represent each possible movement.

```
class Node:

    def __init__(self, current_state, parent, g_value, h_value, orientation):
        self.current_state = current_state
        self.parent = parent
        self.g_value = g_value
        self.h_value = h_value
        self.orientation = orientation

    def __hash__(self):
        return self.current_state.__hash__()

    def f_value(self):
        return self.g_value + self.h_value

    def __lt__(self, other):
        return self.f_value() < other.f_value()
```

self.current stores the (x, y) coordinates of the snake and self.parent points to the parent node. The distance from the initial position of the 'snake head' to the current position of the 'snake head' was the g-value, and the distance from the 'snake head' current position to the goal was the h-value. The h-value was calculated using the Manhattan distance since we dealt with four directions, and the positions were represented as coordinates (Kong & Mayans, 2014). The f_value() function returns the f_value of the node. The _ _lt_ _ function is used to compare instances of the node class. This is useful for addition to a priority queue.

```
def a_star_search(start_node):
    open_list = PriorityQueue()
    closed_list = set()
    positions_map_open_list = dict()
    positions_map_closed_list = dict()
    open_list.put(start_node)
    positions_map_open_list[start_node.current_state] = start_node

    while open_list.qsize() > 0:
        min_node = open_list.get()
        positions_map_open_list.pop(min_node.current_state)

        if goal_test(min_node.current_state):
            return solution_path(min_node)
        successor_nodes = get_neighbouring_states(min_node)

        for successor_node in successor_nodes:
            if goal_test(successor_node.current_state):
                return solution_path(successor_node)
            if successor_node.current_state in positions_map_closed_list:
                continue
            if successor_node.current_state not in positions_map_open_list and successor_node.current_state not in positions_map_closed_list:
                open_list.put(successor_node)
                positions_map_open_list[successor_node.current_state] = successor_node
            elif successor_node.current_state in positions_map_open_list:
                holder_node = positions_map_open_list.get(successor_node.current_state)
                holder_node_fvalue = holder_node.f_value()
                successor_node_fvalue = successor_node.f_value()
                if successor_node_fvalue < holder_node_fvalue:
                    open_list.put(successor_node)
                    open_list.queue.remove(holder_node)
                    positions_map_open_list.pop(holder_node.current_state)
                    positions_map_open_list[successor_node.current_state] = successor_node
        closed_list.add(min_node)
        positions_map_closed_list[min_node.current_state] = min_node
    return None
```

The search algorithm is given a start node which is the 'snake's head' and it is put on an open list (priority queue) with its calculated f-value. A dictionary for the open list, which takes

states and nodes as key-value pairs, takes the start node's current state and state (coordinate) as the key and value respectively. While the open list is not empty, the node with the least f-value is gotten from it, and its current state is checked to see if it is where the food is (goal). It is then put in a closed list which contains the nodes that have been explored. A dictionary is also made for the closed list, which contains states and nodes as key-value pairs. If the condition is true, the path taken to get to it is returned. Otherwise, the successor nodes are generated. If one of the states generated is the goal, the path to the goal is returned.

If the nodes current state is in the closed list dictionary, the state is skipped. For nodes whose states are not in the open list or closed list dictionary, the node is put in the open list and the open list dictionary takes the nodes current state and node as a key-value pair. If the nodes current state is in the open list dictionary, the f-values are compared and the node with the lower f-value is put on the open list and the node with the higher f-value is removed. The nodes current state is also updated in the open list dictionary.

To test our progress, we implemented the search algorithm using arbitrary points to ensure that it was functioning correctly. After that, we integrated the path found from the algorithm to the snake so that it uses it to find the apple by applying the algorithm to the snake's head which was represented as a node.

Results

The snake was able to use the path generated by the A* algorithm to find its way to the food and each time the food respawns at a new random location, the snake is able to keep up and find the path to it. This process was done using a for loop which ran 10 times. The number of times is arbitrary.

Conclusion

We present the implementation of the A* algorithm using a snake game where the snake uses the path generated by the algorithm to find the food and eat it. Although the A* algorithm finds the shortest path, sometimes it can lead to the snake running into the screen barrier when the food is close to it. For future work, we plan to provide obstacles to make it more challenging and also allow the user to determine where the food should spawn.

References

Barnouti, N. H., Mahmood Al-Dabbagh, S. S., Sahib Naser, M. A., & Publishing, S. R. (2016, September 8). *Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm*. Pathfinding in Strategy Games and Maze Solving Using a* Search Algorithm. Retrieved from https://www.scirp.org/journal/paperinformation.aspx?paperid=70460\

Kong, S., & Mayans, J. (2014). *COMPSCI 271 INTRO ARTIFCL INTEL 1 Automated Snake Game Solvers via AI Search Algorithms*. Retrieved from https://cpb-us-e2.wpmucdn.com/sites.uci.edu/dist/5/1894/files/2016/12/AutomatedSnakeGameSolvers.pdf

Sharma, H., Alekseychuk, A., Leskovsky, P., Hellwich, O., Anand, R. s, Zerbe, N., & Hufnagl, P. (2012). Determining similarity in histological images using graph-theoretic description and matching methods for content-based image retrieval in medical diagnostics. *Diagnostic Pathology*, *7*, 134. https://doi.org/10.1186/1746-1596-7-134