# Solve Snake Game with Artificial Intelligence

**Abstract-** There are many computer action games and the Snake game is one of them, whose goal is to control a snake to move and collect food on a map. They have created a controller based on development rating capacities considering space, food, and smoothness. The computer with the help of methods and algorithms to play the famous Snake game automatically there includes 3 searching algorithms related to A∗ Search with forward checking, Best First Search, artificial intelligence, and two pattern strategies, The Almighty Move and common Random Move. These strategies can be the core strategy in a robotized Snake Game with AI Solver, but they have quite a bit changes in their performances. With a large number of experiments to compare their performance and get to their differences. Furthermore, how the distinctive strategies have abilities to make snakes lead at a dead-end.

**Index Terms—** Snake Game, Artificial Intelligence with Best First Search, A∗ Search, Forward Checking.

# Introduction:

The famous Snake game which appeared in 1997 on Nokia 6110, whose goal is to control a snake to move and collect food within boundaries. In this applying, artificial intelligence to achieve food with minimum path and snake's head don't touch with the snake's body, and for calculating the number of paths that reached the food with the minimum, optimal and longest distance with A* Search algorithm, Breadth-First Search algorithm, and Hamiltonian Path, for the graphic user interface where an environment created with PyGame and snake created with three blocks lengthy at the start of the game and every time a snake eat the apple, Apple will spawn at the random location all happened within the environment.

The game environment is 20x20 and it can be changed, The snake starts with the length of 3 blocks. At each step, the snake moves one step, and movements are straight, left, right. The food in this game is an apple with red color. Eating an apple increases the

length of the snake by one block. There is always one apple in the environment, Human can play the original snake game with a simple strategy: move the snake up and down from the eight to the left without touching the last row (the row at the bottom); when the snake is close to the leftmost boundary, go back to the rightmost boundary along the last row. In this way, the snake keeps alive until it occupies the whole environment.

The game ends when any of the following conditions are satisfied.

(1) The snake hits its own body or the wall.

(2) The snake size reached the size of the environment.

(3) Press the “Esc” button from the keyboard manually.

This makes it troublesome to realize a tall score. The ultimate objective of the amusement is to control the snake to eat as numerous apples as conceivable without running out of the board or leading the snake to chomp its own body. According to this situation, the game closed.

# Implementation:

**Apple-** appears anywhere in the environment except the body of the snake which is containing blocks at a single time only one apple appears with red color until the snake eats it and the appearance of the apple is totally randomized.

**Snake-** have a few essential characteristics around execution. Within the most common way, usage consists of the snake moving on a square board with a 20x20 grid, attempting to eat as numerous apples as conceivable without biting itself. Once the snake eats an apple, a new apple is set in a free position on the board and the snake's length develops by one spet. When the snake has no choice other than to chomp itself, the diversion is over and the last score is returned. In this usage, essentially calculate the score as the number of apples the snake has eaten or comparably, the length expanded by the snake. The taking after is a few fundamental rules taken after by the execution:

(1) **Goal -** the snake has a goal to kill as numerous apples, as he can, inside limited steps. In spite of the fact that it is conceivable to keep the snake ‘Alive’ with boundless steps, these sorts of arrangements are not considered to realize the most extreme score. The primary need for the snake is to not nibble itself whereas the moment is to extend the result.

(2) There are four conceivable headings the snake can move: east, west, north, and south. Be that as it may, since of the situation of its tail, a few bearings may not be accessible. The clearest case is that the snake can never swap to an inverse heading i.e. to the south from the north and to the west from the east etc.

(3) The snake develops by one unit when eating an apple. The development is promptly reflected by the picked-up length of the tail, i.e. the apple was the tip of the tail that possesses the square.

(4) The size of the board is settled to a square and 20x20 grid. It can be changeable, black in color, and implemented with PyGame which is one of Python's famous libraries.
(5) After an apple is eaten by the snake, another apple is put haphazardly with uniform likelihood on one accessible square, the square's accessibility is indicated by the reality that it isn't possessed by the snake.

**Algorithms-** to move the snake towards coming to the apple which presents three AI calculations and two pattern strategies to play Snake. The three AI calculations have a place to inform heuristic search whereas the two standard strategies are created naturally based on the game's character.

**Best First Search-** This Greedy Best-First Search algorithm encompasses a one-move skyline and as it was considered moving the snake to the position on the board that shows up to be closest to the objective, i.e. apple. We utilize Manhattan remove as a heuristic work to characterize how near the snakehead is to the apple. This strategy has nearly ensured that the snake will be able to eat in an ideal (most limited) way at least the primary ten to twelve apples. In last explanation leads that the snake's body can't build a circle in an environment with a length of five blocks, there may exist a few cases in which the calculation does not utilize an ideal way or it indeed gets into a dead conclusion. These cases may be considered extraordinary.

In any case, the one-step skyline moreover makes it simple to push on nearby minima and levels. The natural clarification is that the snake as it were looks for the following step that's expected best or nearest to the apple, but without in view of its tail. This makes it simple for the snake to chomp itself when it gets longer. Inevitably, this strategy will halt being ideal after the snake has eaten more than four apples. There will outline how this strategy can get stuck in a dead-end.

**A∗ Search-** A∗ joins a heuristic in a different move skyline. Sometimes recently taking action, it considers not as it were where the objective is and how distant it is, but also the current state it has looked distant.

This A∗ algorithm employments the Manhattan distance to the apples from the head as we know heuristic and Number of steps as "previous cost". Each cycle of the calculation keeps going until a way is found Which causes the snake to eat the apple. Stepping to the Finest Finding the full path to the apple and starting the search calculation by not stopping at a major move has the advantage of being free from deadlock conclusions on the way to the apple. Without memory or time limitations, the calculation is ensured to discover an ideal way to the apple on the off chance that one exists. There are two minor adjustments compared to a standard A∗ algorithm. First, ties between hubs with the same evaluated add up to fetch (that's, the whole heuristic and taken a toll to reach the hub) are not broken arbitrarily. Instep, one of the hubs with the least heuris

One of the inadequacies incorporates the fact that, once the apple is eaten, the snake can reach a dead-end which can be maintained a strategic distance from other

ways. In other words, the algorithm does not take into consideration the impacts of the chosen way once the apple is eaten. To maintain a strategic distance from these dead ends, the A∗ calculation is additionally prepared with a Breadth-First search calculation that's utilized to compute on the off chance that a way to an objective also leads to a dead conclusion. Once an emphasis of the A∗ calculation closes, it'll at that point call the Breadth-First Search calculation starting at the objective state found by the A∗ calculation. From here, it'll investigate the complete tree up to a certain number of hubs. On the off chance that the tree is contained interior this hub bound, i.e. it may be a dead-end, the path to the apple is rendered as not great, and the objective hub from the A∗ calculation will be disposed of (the A∗ cycle will proceed in spite of the fact that). This dead-end check is additionally utilized when the A∗ calculation cannot discover a way to the objective. It'll at that point select the Finest To begin with course

**Random Move-** In addition to the 3 artificial intelligence strategies already described so there, we provide two standard methods. The first is an arbitrary move.

In this, Irregular Move takes the following steps to move as randomly. We enforce the condition, so to speak: the selected movement may not complete the entertainment if possible. As we can imagine, this strategy doesn't take into account the location of these apples, so snakes search for apples for a long time. In addition, this strategy does not take into account the perfect position of the snake on the board, which can effectively lead to dead ends. This strategy doesn't seem to work well, but it can be a good pattern for exploratory comparisons with other AI techniques.

**Almighty Move**-We, at last, presents the moment of the two pattern algorithms: Almighty Move.

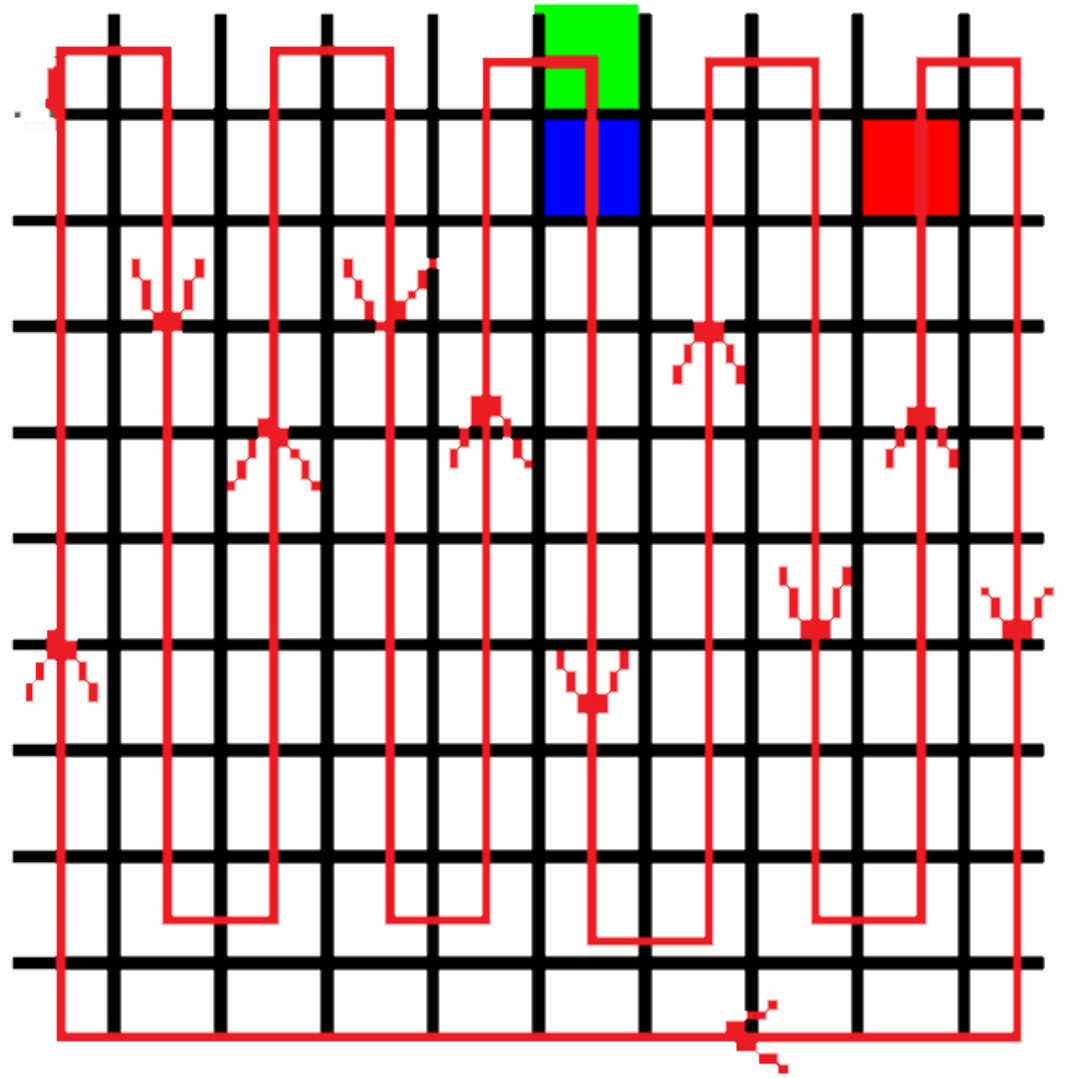


**Fig. 1**. On a board, Almighty Move runs preferably with even-numbered width or top. As lengthy, because the period of width or top is a certain number, Almighty Move can make certain that the snake eats the foremost apples.

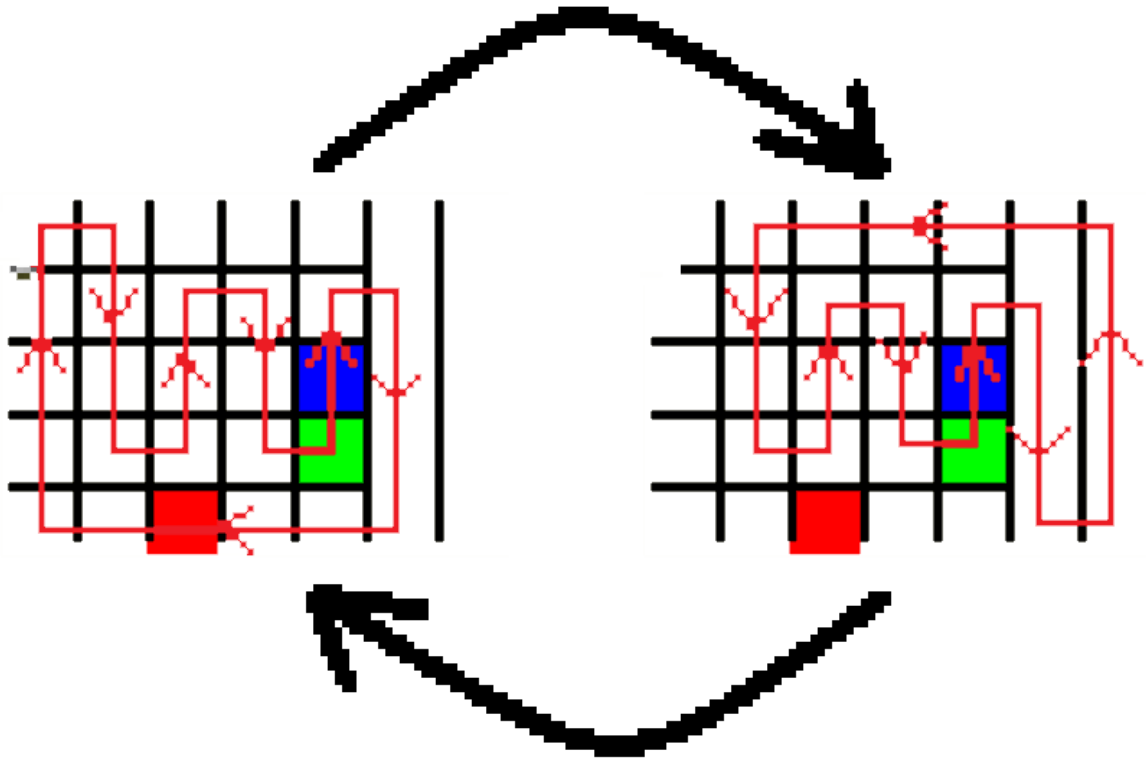


**Fig. 2.** Almighty moves can also be performed on boards of odd widths and heights. To increase the score, the omnipotent movement is repeated in two circular patterns.

Basically, Almighty Move fairs the queuing in the board, as shown in Figures 1 and 2. By actually counting the width or height of the board, it's easy to understand why this guarantees the most extreme scoring strategies. See Figure 1. However, if you have an inner board with an odd width and height, you may need to make some adjustments to your circuit design. See Figure 2. It was a unit of the tail, at this point the snake is eating an (hw-2) apple. Usually, after eating the last apple, the snake takes the entire board with a square head and becomes body length (hw -1). Snakes have a body length of 1, so in the unlikely event that h and w are even heights and widths on the board, they will eventually eat (hw – 2) apples. Anyway, with a wild boar.

# Challenges:

To properly compare differentiated algorithms and strategies, each strategy is run 100 times and the average performance is found as standard deviation. We ran the algorithm on several papers of different sizes, but in this report, we looked at the 20 × 20 board because it is the largest square board that can do all the calculations in a

reasonable amount of time. Each run is laid out as a vector, where each component is a score for that step. The unique results of the experiment are explored in the following passages.

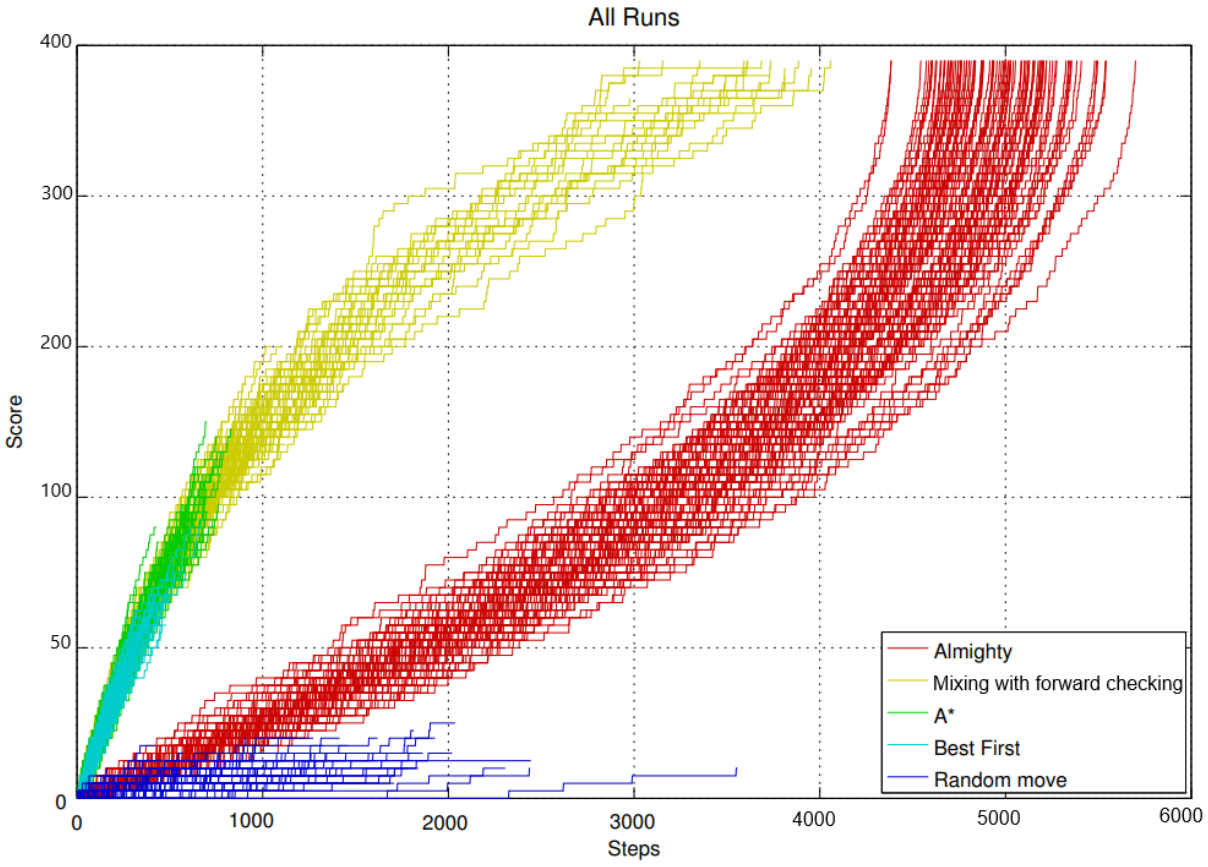



**Fig. 3.** All run on a 20x20 board for various calculations. This illustration is best viewed in color and enlarged size. It runs for all algorithms. From this figure we can get the data as if it were a rough final result or the number of steps he took, but also roughly how effective each calculation is, how reliable it is, or how fast he eats an apple. Obviously, two calculations give the most extreme results. A * Confirm delivery and move the Almighty. Be that as it may, the way they accomplish this greatest score is clearly diverse. Whereas A∗ with dead-end checking is much speedier all the way to half the most extreme score it loses productivity from there onwards. It is the inverse case for Almighty Move. Whereas at the starting eats apples way slower than any educated search calculation, the way the snake is organized at the conclusion makes it exceptionally effective, making it greatly quick at the conclusion diversions. A∗ and Best First Search, coordinate the proficiency of A∗ with forward checking at the introductory stages.

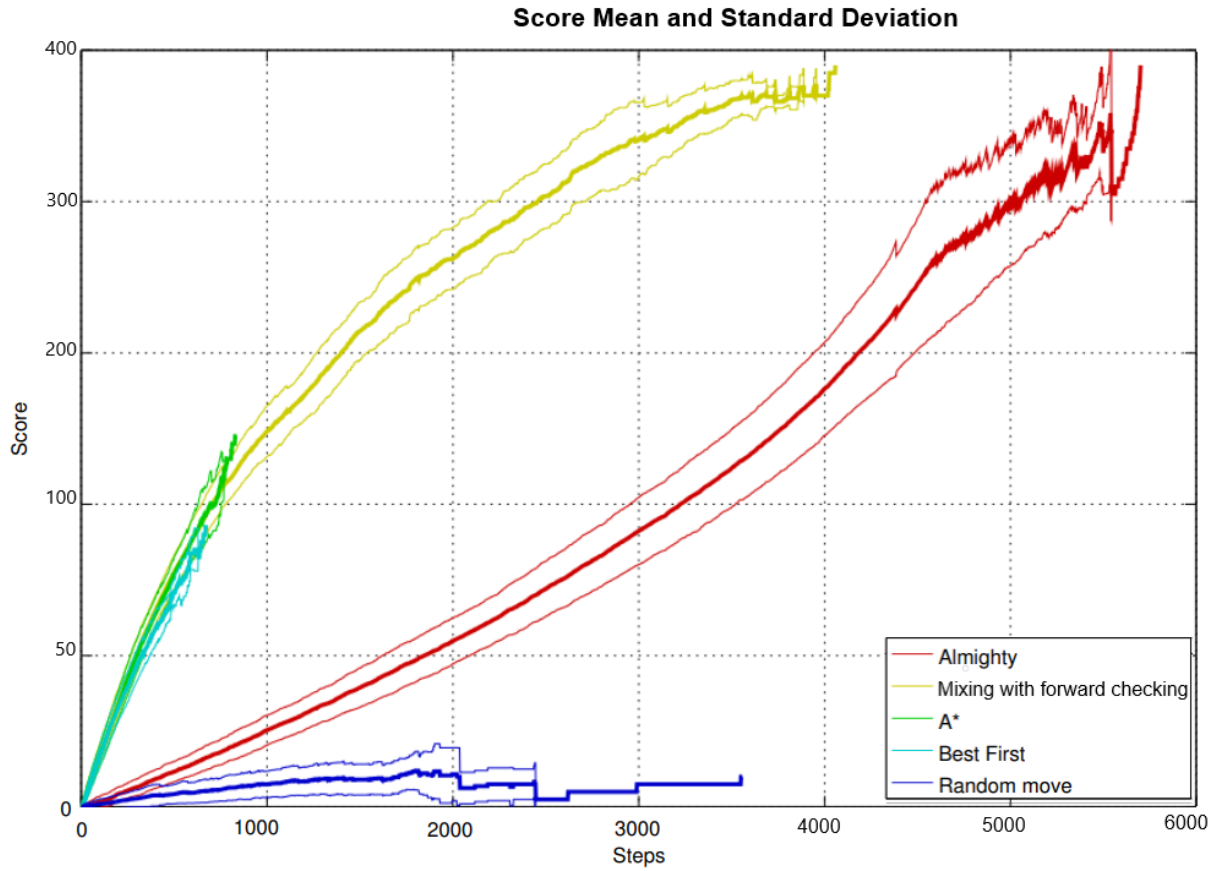



**Fig. 4.** The thick line is compared to the normal score of all runs that reached this stage. Thin lines indicate standard deviations higher or lower than normal. This picture is best seen and enlarged in color. Brutal assessment that each calculation has reached a certain stage. If the cycle ended in the previous step, it is discarded, which causes the picture to swell. The thin line on the chart shows the values of the brutal up and down standard deviations. Based on these numbers, we see that Better Start Search, A*, and A* lookahead performed similarly early in the game, while Almighty moves much lower. More importantly, you can see that erratic moves are fighting to increase your score, and how likely it is that your free moves scores are exceptionally low at any step number.

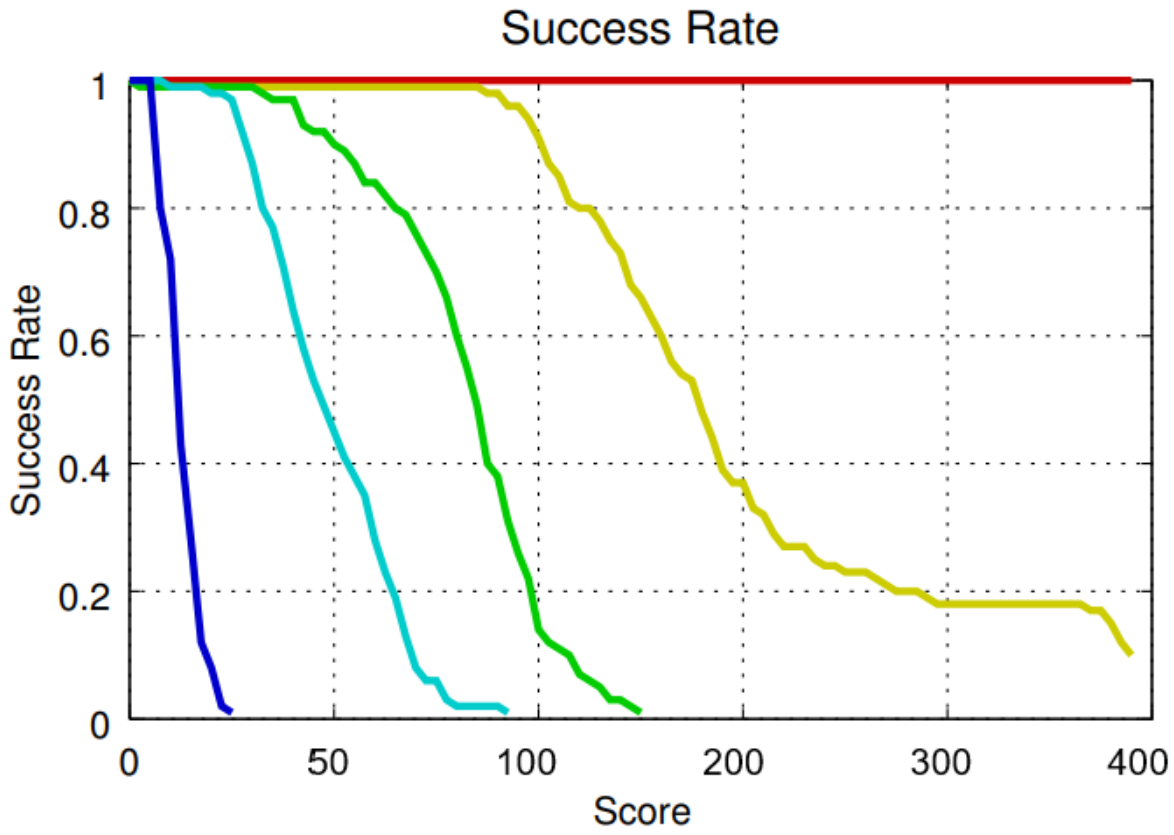



**Fig. 5.** Win percentage is reported as the number of runs that reached that score divided from the total number of runs. Calculations - from smallest to most notable - random moves, best-first search, predict A*, A*, and BFS, omnipotent moves. The ratio of the number of executions of each calculation represents a specific score, the reliability of each calculation. Obviously, the Almighty Movement maintains a win rate of 1 regardless of the score. That is, each execution of the almighty movement reaches its maximum score. The Almighty Movement is an exception, and the win rate for other calculations seems to be below. Looking at the drawings, it is easy to classify calculations into an array of unshakable elegance, from spontaneous to omnipotent movements.

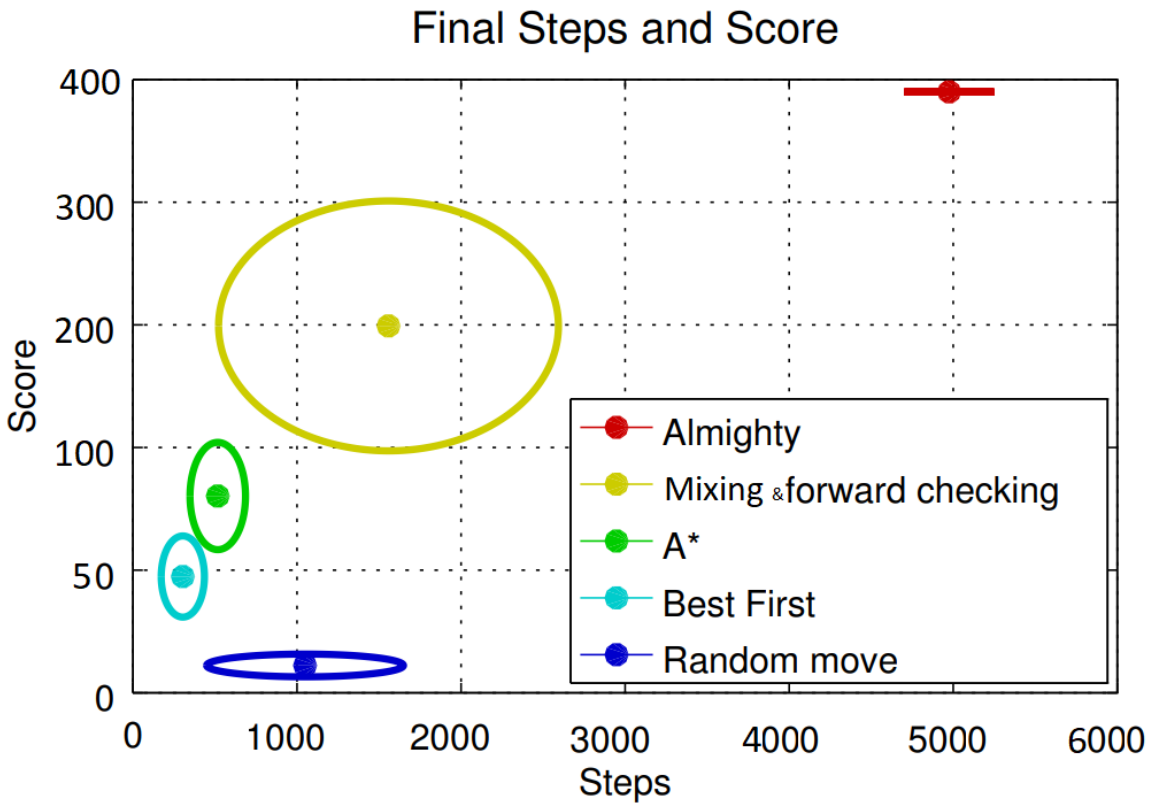
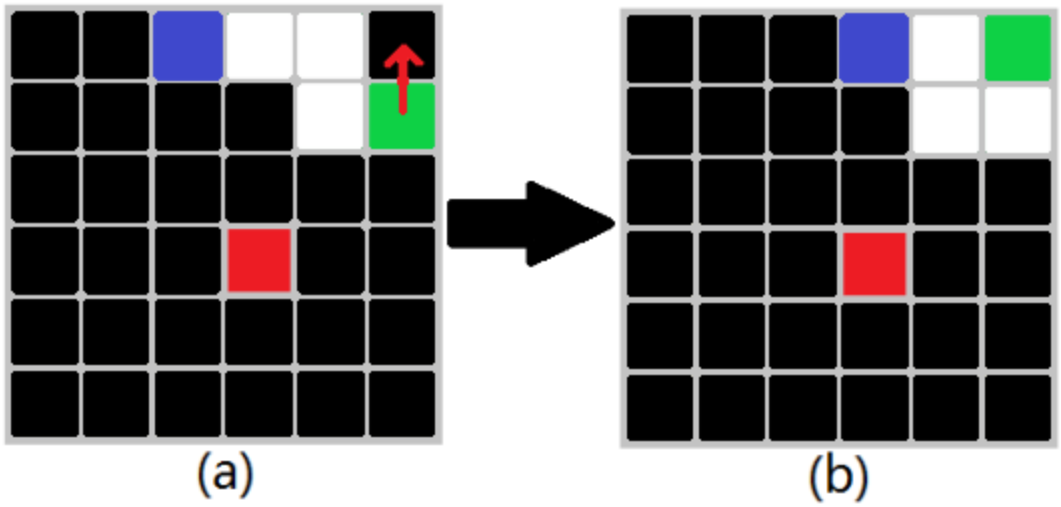



**Fig. 6.** The center point of the ellipse is compared to the typical stride length and final score for each run. The length of the tomahawk is compared to two standard deviations of the specified measurements. Advanced steps can be used to achieve the most surprising results. Finally, Figure 6 has a brutal last count. the cruel number of steps required to attain it for each calculation. The ovals tomahawks are 2 standard deviations. The figure appears in which locale of the score steps plane the calculation is most likely to conclude up. Neglecting the Irregular Move calculation, we see that the higher the anticipated score is the more steps are moreover anticipated. Once more, the figure appears how Almighty Move is able to attain the most noteworthy scores at the cost of utilizing the foremost steps.

# Issues and Limitations:

This part analyzes each strategy in detail and explains how and why each strategy can be stopped and terminated prematurely. With this perspective in mind, we'll offer you a great way to carefully combine different strategies to develop overall.

**Dead End-** Irregular movements can actually lead to dead-hatching as they move forward indiscriminately. The case for the current state is shown in Figure 1. 7 (a) In case of accidental movement.




Irregular movements advance indiscriminately, which can easily lead to dead conclusions. Current state(s) in this figure. If the Arbitrary Move decides to move up, the conclusion is dead at that moment. Select Move Up as shown. 7(b), at this point a dead conclusion will be drawn.

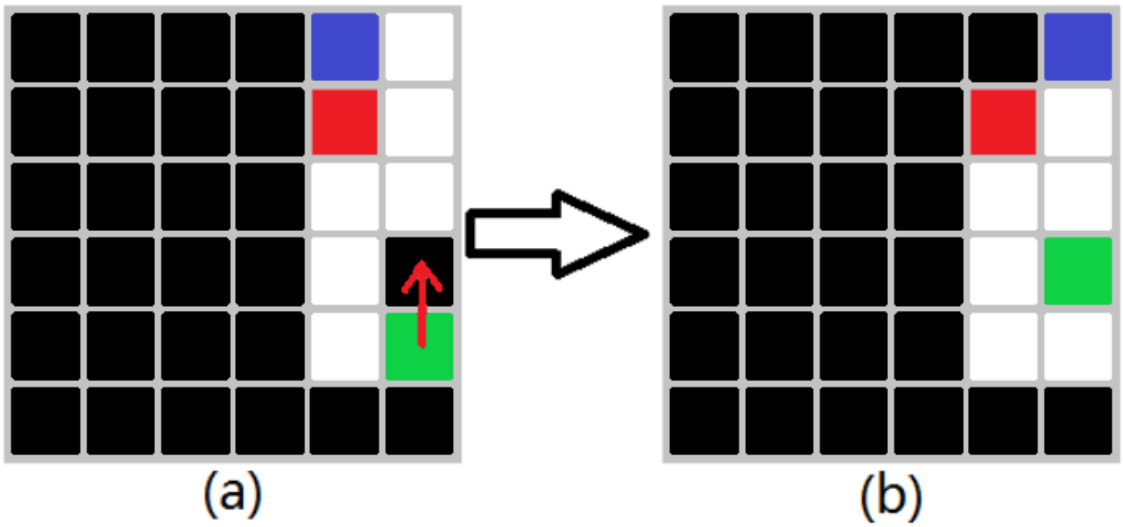



**Fig. 8.** The best-first search chooses to move up based on its current state (a). As you ascend, you reach a dead end as in (b). A* will avoid this deadlock.

**Best First Search-** Best First Search marks a horizontal line of one move and selects the next move based on how far Manhattan is to the destination. For a few key apples discussed in Zone 3.1, it is expected to be done very quickly. In any case, an elongated snake could actually lead to a dead hatch. For example, in Figure 8 Best First Search chooses the upward motion because the square of the snakehead strike is closer

to the apple than the square to the bottom of the apple. Anyway, as soon as it moved, the snake came to the conclusion that it was dead, and the run was over.

**A∗ Search-** A ∗ will certainly find an ideal path if it exists. In Snake, A ∗ Search uses the Manhattan distance as a test. To some extent, A ∗ is comparable to BreadthFirst Search, since both can find the perfect match. Either way, A ∗ Search takes heuristic data into account and expands the hubs so that it can lead to an ideal path, thus increasing fewer hubs than BreadthFirst Search. Indeed, despite the fact that it can beat BreadthFirst Search or Profundity, for starters, Search, Search A ∗ can still lead to a dead-end if the snake is long enough. Figure 9 appears in such a case. In this case, A ∗ Search causes the snake to successfully eat the apple but then the snake can do nothing but move forward until no more movement is accessible. This often happens because of the ownership of A ∗ Search which it thinks is so to speak to the current situation and how to achieve the goal more effectively without considering the conceivable consequences afterward.

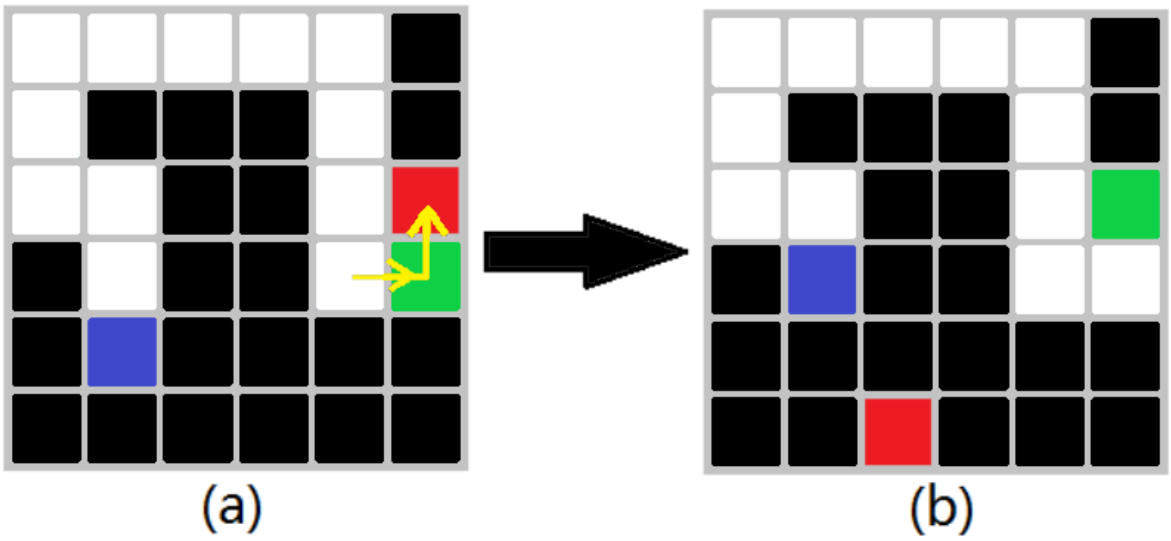



**Fig. 9.** A∗ Research can continuously find the narrowest path as long as there is an important path leading to the goal. Either way, he didn't take into account the effects of the snake eating the apple. In this figure, A ∗ Search-select moves right and up. Either way, after eating the apple, the snake had no choice but to keep climbing until it couldn't move anymore. By distinguishing from Fig. 8 in this case the Best First Search and A ∗ will be stuck in the final conclusion.

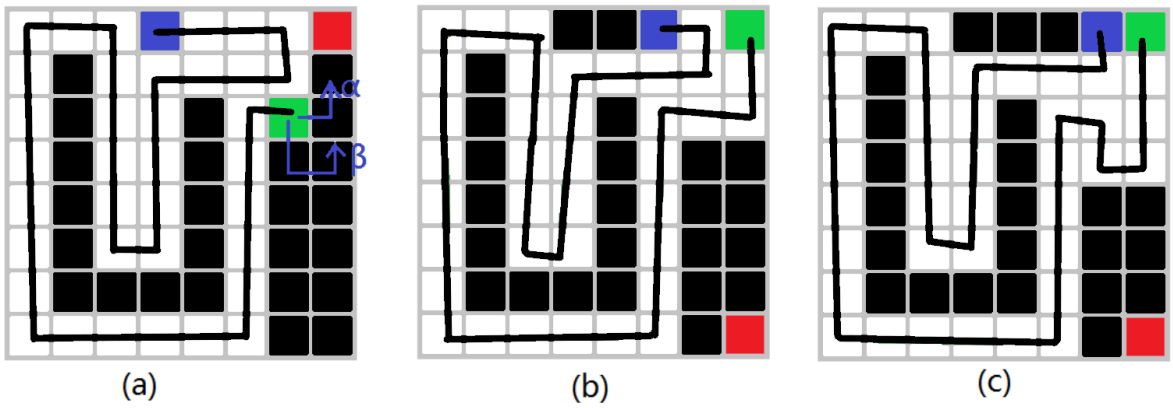



**Fig. 10.** A ∗ Searching with shipment verification can still be a dead end. To clearly see how the snake moves, we have included a gray line along the body of the snake to clarify the snake's shape. With the current state in (a), A ∗ Search will choose to move up and right. This will lead to a dead-end, as shown in (b). In all cases, A ∗ Search with submission verification looks at possible impacts by predicting certain steps. For this case, depth-testing ahead of three movements will move the snake downwards, to the right, and up. This happens in (c). This is because although the snake always avoids the fatal conclusion, to begin with, it will always end up with a deadly conclusion. achieve the goal. In the next subsection, we will look at how the A ∗ advances to aid in avoiding this situation.

**A∗ Search or Mixed with Forward Checking-** Once we see how and why searching for A* can always lead to an end, we tweak it by including some forward-checking capabilities as discussed in section 3.3. This prospective review checks several steps in advance to see if the decision is conclusive. This update should make some performance improvements over a simple A∗ search. Either way, A∗ Search with Submit Verification could lead to an avoidable dead end. Figure 10 illustrates such an illustration. Given the current state of Fig. 10 (a), A∗ Search will choose to move right and up. This easily leads to a dead-end, as shown in (b). Either way, A ∗ Search with three steps forward check, that will take the snake back one block, sometimes as recently to the right and upwards. This will lead to (c), not an end.

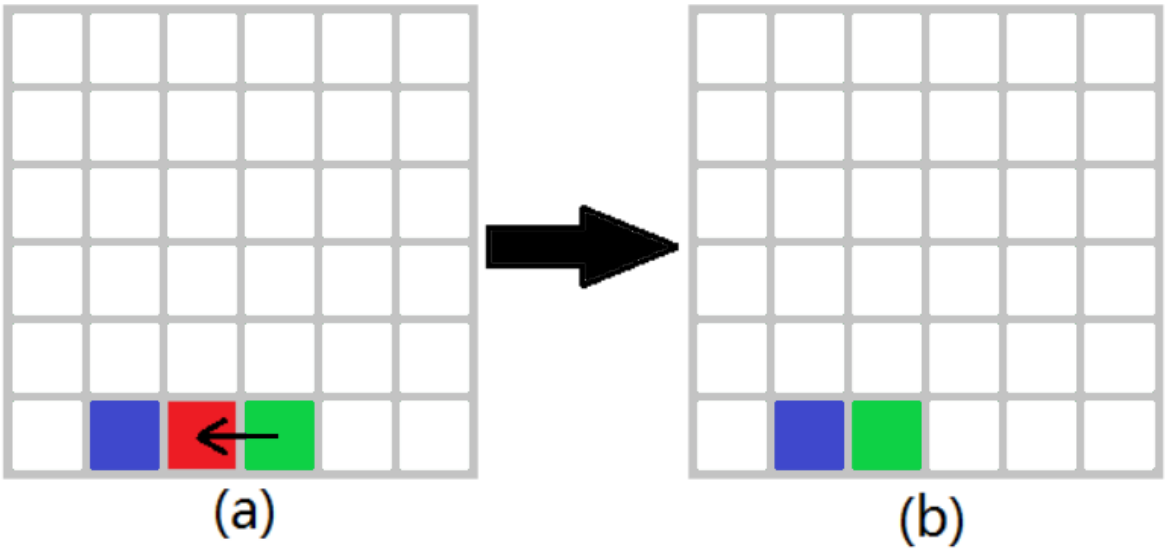



**Fig. 11.** Almighty pull creates a winding circle in the board, as shown in Figure 1. (A) and (b) are the last two states of passing an Almighty train on a 6x6 board. Note that snakes reach the most extreme lengths you can imagine.

It shows the snake moving freely and the tip of its tail moving down to make a way to the head. This is often determined by the snake's developmental equipment. But as the line moves a few more steps, the deadlock becomes permanent. However, you can avoid this deadlock by moving the snake down 5 steps in (a). This can create space around the tip of the tail. In other words, by checking more steps, you can eliminate more dead closing than anything else with fewer forward check steps. Another validity is that A ∗ can lead to a complete conclusion in the transmission test. When using breadth-first search to look for possible dead closes, Apple's placement is not taken into account. Placing apples can change what was initially thought to be a dead-end into one. Imagine a snake's head and primary colors being moved to a specific position on the board. Placing an apple in this position will evolve the cue by one unit. Perhaps the calculation of breadth-first search calculations is meaningless. Even if the forward check is not deep enough, an A ∗ search using an outbound check can be a dead end. Checking more steps with more dead closing slows down the overall method and makes the computerized snake game solver strange, despite the fact that the time elapsed for the check is avoided. Become.

**Almighty Move-** As previously analyzed, by submitting a check, A ∗ Search forces the dead close by pre-checking the specified number of steps. This indicates that queuing is a bit limited to maintain a strategic distance from the dead close. Head to move. Therefore, it makes sense to plan Snake's advanced strategy for creating snake circuits in the board, as shown in Section 3.5. Figure 11 shows the last two steps of a snake using omnipotent movement on a 6x6 board. You can see that the length of the

line will eventually increase depending on the number of units on the board. This dead-end is inevitable, meaning that the detour is completed with the highest score.

**CONCLUSION-** This report introduced five algorithms and methods for creating an automated snake game solver. We analyzed various algorithms and conducted experiments to investigate their performance. With the exception of the random movements, which are only used as a base, the rest seem to have advantages and disadvantages depending on whether speed or reliability is a top priority. The informed search algorithm is fairly reliable and very efficient at the beginning of the run, but these properties disappear at the end of the game. In contrast, Almighty Movement is a slow algorithm at the beginning of the run, but it guarantees the highest score and its efficiency is highest at the end of the game. By intuition, the author believes that a combination of different algorithms can be combined to achieve full reliability while keeping the initial efficiency of the game-high. However, determining the threshold for the score to switch algorithms is an important issue and remains an issue for future tasks. The score threshold is assumed to depend on the size of the board and the various parameters of each algorithm. You can further improve performance by implementing other AI algorithms, such as those introduced in.

Main.py  code – python

```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec 12 19:17:28 2021

"""

from snakeGame import snakeGame

if __name__ == '__main__':
    snakeGame().launch()
```

Game_run.py  code – python

```
from Apple import Apple
from snake_info import Snake
from display import display_base


class Player(display_base):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(**kwargs)
        self.snake = snake
        self.apple = apple

    def explore_neighbors(self, node):
        """
        fetch and yield the four neighbours of a node
        :param node: (node_x, node_y)
        """
        for diff in ((0, 1), (0, -1), (1, 0), (-1, 0)):
```

```
            yield self.node_add(node, diff)

    @staticmethod
    def is_node_in_queue(node: tuple, queue: iter):
        """
        Check if element is in a nested list
        """
        return any(node in sublist for sublist in queue)

    def is_invalid_move(self, node: tuple, snake: Snake):
        """
        Similar to dead_checking, this method checks if a given node is a valid move
        :return: Boolean
        """
        x, y = node
            if not 0 <= x < self.cell_width or not 0 <= y < self.cell_height or node in
snake.body:
            return True
        return False
```

foward_Check_algo.py  code – python

```
from game_run import Player
from Apple import Apple
from long_path_algo import LongestPath
from BFS_algo import BFS
from snake_info import Snake

class Fowardcheck(Player):
    longgest_path_cache = []

    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
        self.kwargs = kwargs

    def run_forwardcheck(self):
```

```
        bfs = BFS(snake=self.snake, apple=self.apple, **self.kwargs)

        path = bfs.run_bfs()

        print("trying BFS")

        if path is None:
            snake_tail = Apple()
            snake_tail.location = self.snake.body[0]
            snake = Snake(body=self.snake.body[1:])
                                longest_path = LongestPath(snake=snake, apple=snake_tail,
**self.kwargs).run_longest()
            next_node = longest_path[0]
            # print("BFS not reachable, trying head to tail")
            # print(next_node)
            return next_node

        length = len(self.snake.body)
        virtual_snake_body = (self.snake.body + path[1:])[-length:]
        virtual_snake_tail = Apple()
        virtual_snake_tail.location = (self.snake.body + path[1:])[-length - 1]
        virtual_snake = Snake(body=virtual_snake_body)
                                virtual_snake_longest = LongestPath(snake=virtual_snake,
apple=virtual_snake_tail, **self.kwargs)
        virtual_snake_longest_path = virtual_snake_longest.run_longest()
        if virtual_snake_longest_path is None:
            snake_tail = Apple()
            snake_tail.location = self.snake.body[0]
            snake = Snake(body=self.snake.body[1:])
                                longest_path = LongestPath(snake=snake, apple=snake_tail,
**self.kwargs).run_longest()
            next_node = longest_path[0]
            # print("virtual snake not reachable, trying head to tail")
            # print(next_node)
            return next_node
        else:
            # print("BFS accepted")
            return path[1]
```

Human_controls.py  code –python

```
from game_run import Player
from Apple import Apple
from snake_info import Snake
import pygame
from heapq import *

class Human(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)

    def run(self):
        for event in pygame.event.get():  # event handling loop
            if event.type == KEYDOWN:
                if (event.key == K_LEFT or event.key == K_a) and self.snake.last_direction != (1, 0):
                    diff = (-1, 0)  # left
                                elif (event.key == K_RIGHT or event.key == K_d) and self.snake.last_direction != (-1, 0):
                    diff = (1, 0)  # right
                elif (event.key == K_UP or event.key == K_w) and self.snake.last_direction != (0, 1):
                    diff = (0, -1)  # up
                elif (event.key == K_DOWN or event.key == K_s) and self.snake.last_direction != (0, -1):
                    diff = (0, 1)  # down
                else:
                    break
                return self.node_add(self.snake.get_head(), diff)
        # If no button is pressed down, follow previou direction
        return self.node_add(self.snake.get_head(), self.snake.last_direction)
```

Long_path_algo.py  code – python

```
from Apple import Apple
from snake_info import Snake
from BFS_algo import BFS

class LongestPath(BFS):
    """
    Given shortest path, change it to the longest path
    """

    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
        self.kwargs = kwargs

    def run_longest(self):
        """
        For every move, check if it could be replace with three equivalent moves.
          For example, for snake moving one step left, check if moving up, left, and down is valid. If yes, replace the
        move with equivalent longer move. Start this over until no move can be replaced.
        """
        path = self.run_bfs()

        # print(f'longest path initial result: {path}')

        if path is None:
            # print(f"Has no Longest path")
            return

        i = 0
        while True:
            try:
                direction = self.node_sub(path[i], path[i + 1])
            except IndexError:
                break

                # Build a dummy snake with body and longest path for checking if node replacement is valid
```

```
        snake_path = Snake(body=self.snake.body + path[1:], **self.kwargs)

        # up -> left, up, right
        # down -> right, down, left
        # left -> up, left, down
        # right -> down, right, up
        for neibhour in ((0, 1), (0, -1), (1, 0), (-1, 0)):
            if direction == neibhour:
                x, y = neibhour
                diff = (y, x) if x != 0 else (-y, x)

                extra_node_1 = self.node_add(path[i], diff)
                extra_node_2 = self.node_add(path[i + 1], diff)

                                        if  snake_path.dead_checking(head=extra_node_1)  or
snake_path.dead_checking(head=extra_node_2):
                    i += 1
                else:
                    # Add replacement nodes
                    path[i + 1:i + 1] = [extra_node_1, extra_node_2]
                break

        # Exclude the first node, which is same to snake's head
        return path[1:]
```

Mixed_algo.py  code – python

```
from Apple import Apple
from BFS_algo import BFS
from snake_info import Snake
from game_run import Player


class Mixed(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
```

```
        self.kwargs = kwargs

    def escape(self):
        head = self.snake.get_head()
        largest_neibhour_apple_distance = 0
        newhead = None
        for diff in ((0, 1), (0, -1), (1, 0), (-1, 0)):
            neibhour = self.node_add(head, diff)

            if self.snake.dead_checking(head=neibhour, check=True):
                continue

            neibhour_apple_distance = (
                                abs(neibhour[0] - self.apple.location[0]) + abs(neibhour[1] -
self.apple.location[1])
            )
            # Find the neibhour which has greatest Manhattan distance to apple and has path
to tail
            if largest_neibhour_apple_distance < neibhour_apple_distance:
                snake_tail = Apple()
                snake_tail.location = self.snake.body[1]
                # Create a virtual snake with a neibhour as head, to see if it has a way to its tail,
                  # thus remove two nodes from body: one for moving one step forward, one for
avoiding dead checking
                snake = Snake(body=self.snake.body[2:] + [neibhour])
                bfs = BFS(snake=snake, apple=snake_tail, **self.kwargs)
                path = bfs.run_bfs()
                if path is None:
                    continue
                largest_neibhour_apple_distance = neibhour_apple_distance
                newhead = neibhour
        return newhead

    def run_mixed(self):
        """
        Mixed strategy
        """
        bfs = BFS(snake=self.snake, apple=self.apple, **self.kwargs)

        path = bfs.run_bfs()
```

```
        # If the snake does not have the path to apple, try to follow its tail to escape
        if path is None:
            return self.escape()

        # Send a virtual snake to see when it reaches the apple, does it still have a path to its
own tail, to keep it
        # alive
        length = len(self.snake.body)
        virtual_snake_body = (self.snake.body + path[1:])[-length:]
        virtual_snake_tail = Apple()
        virtual_snake_tail.location = (self.snake.body + path[1:])[-length - 1]
        virtual_snake = Snake(body=virtual_snake_body)
            virtual_snake_longest = BFS(snake=virtual_snake, apple=virtual_snake_tail,
**self.kwargs)
        virtual_snake_longest_path = virtual_snake_longest.run_bfs()
        if virtual_snake_longest_path is None:
            return self.escape()
        else:
            return path[1]
```

snakeGame.py code – python

```
import contextlib
import time
import sys

from display import display_base
from snake_info import Snake
from Apple import Apple
from mixed_algo import Mixed
from heapq import *

with contextlib.redirect_stdout(None):
    import pygame
    from pygame.locals import *



WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
```

```
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
DARKGRAY = (40, 40, 40)

#@dataclass
class snakeGame(display_base):

    fps: int = 60

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.kwargs = kwargs

        pygame.init()
        self.clock = pygame.time.Clock()
        self.display = pygame.display.set_mode((self.window_width, self.window_height))
        pygame.display.set_caption('AI Snake Game')

    def launch(self):
        while True:
            self.game()
            # self.showGameOverScreen()
            self.pause_game()

    def game(self):
        snake = Snake(**self.kwargs)

        apple = Apple(**self.kwargs)
        apple.refresh(snake=snake)

        step_time = []


        while True:

            # AI Game Player
            for event in pygame.event.get():  # event handling loop
                    if event.type == QUIT or (event.type == KEYDOWN and event.key ==
K_ESCAPE):
                self.terminate()
```

```
        start_time = time.time()

        # BFS Solver
        # new_head = BFS(snake=snake, apple=apple, **self.kwargs).next_node()

        # Longest Path Solver
        # this solver is calculated per apple, not per move
        # if not longgest_path_cache:
                #         longgest_path_cache = LongestPath(snake=snake, apple=apple,
**self.kwargs).run_longest()
        # new_head = longgest_path_cache.pop(0)

        # A star Solver
        # new_head = Astar(snake=snake, apple=apple, **self.kwargs).run_astar()

        # FORWARD CHECKING
                                # new_head = Fowardcheck(snake=snake, apple=apple,
**self.kwargs).run_forwardcheck()
        new_head = Mixed(snake=snake, apple=apple, **self.kwargs).run_mixed()
        print(new_head)

        end_time = time.time()
        move_time = end_time - start_time
        # print(move_time)
        step_time.append(move_time)

        snake.move(new_head=new_head, apple=apple)

        if snake.is_dead:
            print(snake.body)
            print("Dead")
            break
        elif snake.eaten:
            apple.refresh(snake=snake)

        if snake.score + snake.initial_length >= self.cell_width * self.cell_height:
            break

        self.display.fill(BLACK)
        self.draw_panel()
        self.draw_snake(snake.body)
```

```
            self.draw_apple(apple.location)
            pygame.display.update()
            self.clock.tick(self.fps)

        print(f"Score: {snake.score}")
        print(f"Mean step time: {self.mean(step_time)}")

    @staticmethod
    def terminate():
        pygame.quit()
        sys.exit()

    def pause_game(self):
        while True:
            time.sleep(0.2)
            for event in pygame.event.get():  # event handling loop
                if event.type == QUIT:
                    self.terminate()
                if event.type == KEYUP:
                    if event.key == K_ESCAPE:
                        self.terminate()
                    else:
                        return

    def draw_snake(self, snake_body):
        for snake_block_x, snake_block_y in snake_body:
            x = snake_block_x * self.cell_size
            y = snake_block_y * self.cell_size
            snake_block = pygame.Rect(x, y, self.cell_size - 1, self.cell_size - 1)
            pygame.draw.rect(self.display, WHITE, snake_block)

        # Draw snake's head
        x = snake_body[-1][0] * self.cell_size
        y = snake_body[-1][1] * self.cell_size
        snake_block = pygame.Rect(x, y, self.cell_size - 1, self.cell_size - 1)
        pygame.draw.rect(self.display, GREEN, snake_block)

        # Draw snake's tail
        x = snake_body[0][0] * self.cell_size
        y = snake_body[0][1] * self.cell_size
```

```
        snake_block = pygame.Rect(x, y, self.cell_size - 1, self.cell_size - 1)
        pygame.draw.rect(self.display, BLUE, snake_block)

    def draw_apple(self, apple_location):
        apple_x, apple_y = apple_location
            apple_block = pygame.Rect(apple_x * self.cell_size, apple_y * self.cell_size,
self.cell_size, self.cell_size)
        pygame.draw.rect(self.display, RED, apple_block)

    def draw_panel(self):
        for x in range(0, self.window_width, self.cell_size):  # draw vertical lines
            pygame.draw.line(self.display, DARKGRAY, (x, 0), (x, self.window_height))
        for y in range(0, self.window_height, self.cell_size):  # draw horizontal lines
            pygame.draw.line(self.display, DARKGRAY, (0, y), (self.window_width, y))
```

Snake_info.py  code – python

```
from Apple import Apple
from display import display_base

class Snake(display_base):
    def __init__(self, initial_length: int = 3, body: list = None, **kwargs):
        """
        :param initial_length: The initial length of the snake
        :param body: Optional. Specifying an initial snake body
        """
        super().__init__(**kwargs)
        self.initial_length = initial_length
        self.score = 0
        self.is_dead = False
        self.eaten = False

            # last_direction is only used for human player, giving it a default direction when
game starts
        self.last_direction = (-1, 0)

        if body:
            self.body = body
        else:
            if not 0 < initial_length < self.cell_width:
```

```
            raise ValueError(f"Initial_length should fall in (0, {self.cell_width})")

        start_x = self.cell_width // 2
        start_y = self.cell_height // 2

        start_body_x = [start_x] * initial_length
        start_body_y = range(start_y, start_y - initial_length, -1)

        self.body = list(zip(start_body_x, start_body_y))

    def get_head(self):
        return self.body[-1]

    def dead_checking(self, head, check=False):
        """
        Check if the snake is dead
            :param check: if check is True, only return the checking result without updating
snake.is_dead
        :return: Boolean
        """
        x, y = head
            if not 0 <= x < self.cell_width or not 0 <= y < self.cell_height or head in
self.body[1:]:
            if not check:
                self.is_dead = True
            return True
        return False

    def cut_tail(self):
        self.body.pop(0)

    def move(self, new_head: tuple, apple: Apple):
        """
        Given the location of apple, decide if the apple is eaten (same location as the snake's
head)
        :param new_head: (new_head_x, new_head_y)
        :param apple: Apple instance
        :return: Boolean. Whether the apple is eaten.
        """
        if new_head is None:
            self.is_dead = True
```

```
            return

        if self.dead_checking(head=new_head):
            return

        self.last_direction = self.node_sub(new_head, self.get_head())

        # make the move
        self.body.append(new_head)

        # if the snake eats the apple, score adds 1
        if self.get_head() == apple.location:
            self.eaten = True
            self.score += 1
        # Otherwise, cut the tail so that snake moves forward without growing
        else:
            self.eaten = False
            self.cut_tail()
```

Apple.py  code – python

```
import random
from display import display_base
from itertools import product

class Apple(display_base):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        self.location = None

    def refresh(self, snake):
        """
        Generate a new apple
        """
        available_positions = set(product(range(self.cell_width - 1), range(self.cell_height - 1))) - set(snake.body)

            # If there's no available node for new apple, it reaches the perfect solution. Don't draw the apple then.
```

```
        location = random.sample(available_positions, 1)[0] if available_positions else (-1,
-1)

        self.location = location
```

A_star_algo.py  code – python

```
from game_run import Player
from Apple import Apple
from snake_info import Snake


def heuristic(start, goal):
    return (start[0] - goal[0])**2 + (start[1] - goal[1])**2

class Astar(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
        self.kwargs = kwargs

    def run_astar(self):
        came_from = {}
        close_list = set()
        goal = self.apple.location
        start = self.snake.get_head()
        dummy_snake = Snake(body=self.snake.body)
        neighbors = [(1, 0), (-1, 0), (0, 1), (0, -1), (-1, -1), (-1, 1), (1, 1), (1, -1)]
        gscore = {start: 0}
        fscore = {start: heuristic(start, goal)}
        open_list = [(fscore[start], start)]
        print(start, goal, open_list)
        while open_list:
            current = min(open_list, key=lambda x: x[0])[1]
            open_list.pop(0)
            print(current)
```

```
        if current == goal:
            data = []
            while current in came_from:
                data.append(current)
                current = came_from[current]
                print(data)
            return data[-1]

        close_list.add(current)

        for neighbor in neighbors:
            neighbor_node = self.node_add(current, neighbor)

                if dummy_snake.dead_checking(head=neighbor_node) or neighbor_node in
close_list:
                continue
            if sum(map(abs, self.node_sub(current, neighbor_node))) == 2:
                diff = self.node_sub(current, neighbor_node)
                    if dummy_snake.dead_checking(head=self.node_add(neighbor_node, (0,
diff[1]))
                                        ) or self.node_add(neighbor_node, (0, diff[1])) in close_list:
                    continue
                        elif dummy_snake.dead_checking(head=self.node_add(neighbor_node,
(diff[0], 0))
                                                ) or self.node_add(neighbor_node, (diff[0], 0)) in
close_list:
                    continue
            tentative_gscore = gscore[current] + heuristic(current, neighbor_node)
                if tentative_gscore < gscore.get(neighbor_node, 0) or neighbor_node not in
[i[1] for i in open_list]:
                gscore[neighbor_node] = tentative_gscore
                fscore[neighbor_node] = tentative_gscore + heuristic(neighbor_node, goal)
                open_list.append((fscore[neighbor_node], neighbor_node))
                came_from[neighbor_node] = current
```

Display.py  code – python

```
from operator import add, sub
from typing import Tuple
```

```
from dataclasses import dataclass

@dataclass
class display_base:
    cell_size: int = 20
    cell_width: int = 10
    cell_height: int = 10
    window_width = cell_size * cell_width
    window_height = cell_size * cell_height

    @staticmethod
    def node_add(node_a: Tuple[int, int], node_b: Tuple[int, int]):
        result: Tuple[int, int] = tuple(map(add, node_a, node_b))
        return result

    @staticmethod
    def node_sub(node_a: Tuple[int, int], node_b: Tuple[int, int]):
        result: Tuple[int, int] = tuple(map(sub, node_a, node_b))
        return result

    @staticmethod
    def mean(l):
        return round(sum(l) / len(l), 4)
```

BFS_algo.py  code – python

```
from game_run import Player
from Apple import Apple
from snake_info import Snake


class BFS(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)

    def run_bfs(self):
```

```
    """
    Run BFS searching and return the full path of best way to apple from BFS searching
    """
    queue = [[self.snake.get_head()]]

    while queue:
        path = queue[0]
        future_head = path[-1]

        # If snake eats the apple, return the next move after snake's head
        if future_head == self.apple.location:
            return path

        for next_node in self.explore_neighbors(future_head):
            if (
                self.is_invalid_move(node=next_node, snake=self.snake)
                or self.is_node_in_queue(node=next_node, queue=queue)
            ):
                continue
            new_path = list(path)
            new_path.append(next_node)
            queue.append(new_path)

        queue.pop(0)

def next_node(self):
    """
    Run the BFS searching and return the next move in this path
    """
    path = self.run_bfs()
    return path[1]
```