Bachelors of Science in Software Engineering
June 2020

Comparison of Searching Algorithms in AI Against Human Agent in Snake Game.

Naga Sai Dattu Appaji

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

Contact Information:
Author(s):
Naga Sai Dattu Appaji
E-mail: naap19@student.bth.se

University advisor:
Dr. Prashant Goswami
Department of Computer Science

Faculty of Computing	Internet	:	www.bth.se
Blekinge Institute of Technology	Phone	:	+46 455 38 50 00
SE–371 79 Karlskrona, Sweden	Fax	:	+46 455 38 50 57

Abstract

Context: Artificial Intelligence(AI) is one of the core branches of computer science. The use of AI has been increasing rapidly. The primary implementation source of AI is in games. By using AI, the non-player characters(NPC's) are developed in the game. They are many ways to implement AI in the game, the most common method is the search method. There are different search algorithms used in the implementation of AI in games.
A**Objectives:** The main objective of this thesis to compare the search algorithms ( ) in AI and contrast with the Human Agent using the snake game.
Methods: Literature review was to be conducted for the search algorithms. Based on the results from the literature review some algorithms are implemented in the snake game.
Conclusion: From the literature review, we have concluded some algorithms which suit best for the searching algorithms. After conducting the experiment, we have concluded that the A* Search algorithm works better than Breadth-First Search, Depth-First Search, Best First Search, Hamiltonian Search algorithms.

Keywords: Artificial intelligence -AI, Behaviour Tree -BT, Finite State machines-FSM, Reinforcement Learning -RL.

Acknowledgments

I would like to thank my parents, teachers, and friends who were behind me to complete my bachelor's degree. I would also thanks to my sister who shares her value knowledge each time.

I am very much thankful to Professor Prashant Goswami sir for supervising my thesis project and also helped me to create interest in the AI field with his teaching.

Contents

Chapter 1

Introduction

AI is one of the most important subjects in the field of Computer Science. Nowadays AI is used to solve complex task with ease. AI consists of many algorithms which try to mimic the human behaviour [3]. With the AI, the machines behave like a computer-controlled robot to accomplish tasks commonly associated with intelligent beings. The term AI is most commonly applied to computer programmed robots because they posses the smart characteristic of humans, such as the ability to learn, discover new things, learn from past, generalize. As this is the era of new technologies, the AI is developed in such a way that has attained more performance level of the human experts and becomes professional in performing tasks. So the AI is found in many applications such as voice reorganization, handwriting recognition, self-driving cars, etc [20] [21].

AI in the real-world and AI in games is quite different. Computer gaming is one of the AI research area. If anybody has played a video game at any time in their life, that means they have interacted with AI. In games, AI is the reason behind of programming an opposition player who calls it as the Non-player characters(NPC). AI in games provide a good playing experiences. AI in games does not require a great amount of knowledge, the only thing is to apply specific rules and conditions to appear the character as an intelligent [27].

Today the software industry provides much research on the gaming field but the companies are mainly focusing on the graphics but not on the AI of the agents. There are a lot of games that were successfully completed by Human Agents. So in order to increase interest among the Human Agents more hard levels should be provided to play. The design of the levels in the games are designed by using the performance of the algorithms only. If the performance of the algorithm is good it is used to implement the hard levels of the game. One of the most important AI implementations in the games was the searching algorithms because to move an NPC's from one place to another place, searching algorithms are required. So searching helps the AI agents to reach a specific location in the game. Implementing the agent by using each searching algorithm in a game takes a lot of time and difficulty. So the best searching algorithm is to be implemented in the game to get efficient results.

In this thesis, the implementation of some of the searching algorithms is to done to find their performance in a snake game and compared the performance of these algorithms against the Human Agent and each other. Breadth-First Search, Depth First Search, A* Search, Best First Search, Hamiltonian path are some of the selected searching algorithms. The reason for selecting these algorithms is these are the commonly used searching algorithms and are path finding type algorithms.

1.1 Background

The are many methods used in the AI. Here some of the popular methods which are mainly used to develop AI are

- Finite State Machines (FSM)
- Behaviour Trees (BT)
- Tree Search
- Evolutionary Computation
- Reinforcement Learning (RL)
- Supervised Learning
- Unsupervised Learning

1.1.1 Finite State Machines

A finite state machine is one of the Game AI methods which is mainly used to develop the control and decision making for the NPCs. The FSMs consists of a set of input events, output events, and transition functions. Based on the provided inputs events the state of the NPC changes using the transition function and result in an output event in the game. FSMs are simple to design and implement them in games. The representation of the FSMs is done by using the graphs. The FSM graph is an abstract representation of the actions, states, transitions. The developed NPCs in games using FSMs can only be one state at a time, Based on provided input action the state transforms to another state if the game condition satisfies against input action. The use of the FSMs in games works well in past few years. It is too difficult to develop FSMs in games on a large scale because of too much computational size. There is not too much adaptivity and evolution of FSMs in games. Thus the FSMs become very much predictable gaming behavior. The drawbacks of the FSMs are done by using probabilities and fuzzy logic and fuzzy rules [14].

Example of FSM in games

In this game, the states are the patrolling, shooting enemy, dive for cover. The transition states are the No enemy in sight, Enemy insight, Grenade insight, and Grenade detonated. If the FSM controller in the patrolling states is lookout for the enemy if the enemy is the insight the state of the FSMs changes to the Shooting enemy state where the controller tries to kill the opponent. If any Grenard nearby distance of the controller, the controller is used to run away from it and look for cover and health. If there is sufficient health and enemy insight, the FSM controller activates the Grenade detonated and again lookout for enemies. This is how the FSMs is worked in the games.

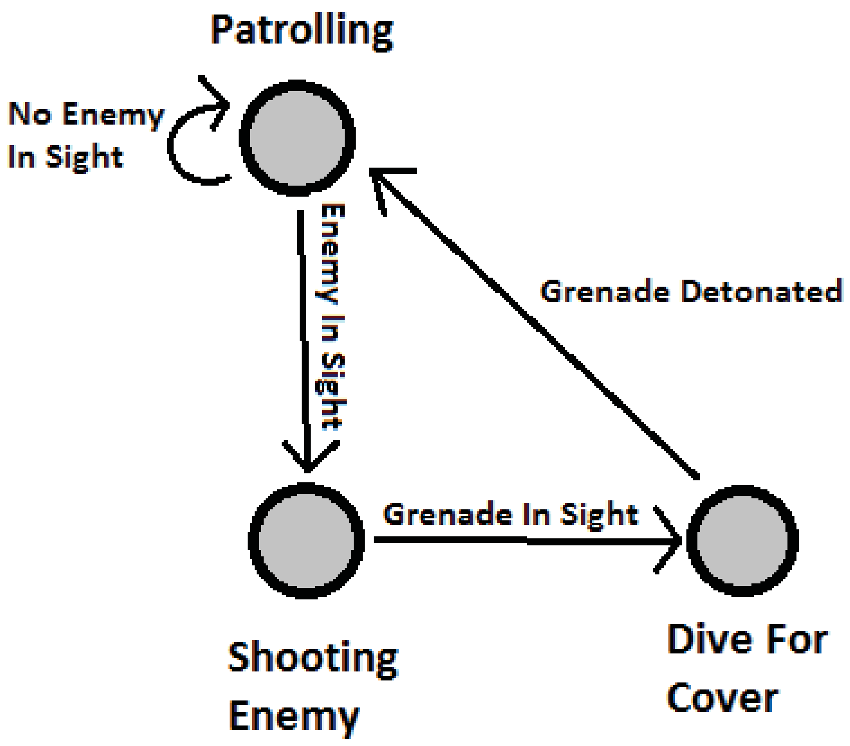


Figure 1.1: FSM in the shooting game [1].

1.1.2 Behaviour Trees

The behavior trees are also one of the executions of the NPCs character in the games. It is a model, which is to develop an expert knowledge structure about the transition states in the game. The behaviors of the NPCs are defined as the transition states and these states are represented using the tree structure. A behavior tree is a decision making of transition states for the NPCs, these transition states are performed based on the hierarchical structure of behaviors [22]. The main difference between Behaviour trees and FSMs is that they consist of behaviors about the transition states instead of states. Behavior trees are easy to implement and debug. The behavior trees are very successful in games like Halo 2 and Bioshock. A child node can return the values below with the parent node in fixed time of steps (ticks):

1. Execute if the behaviour is still active
2. Success if the behaviour is still active
3. The behaviour is completed
4. Failure if the behaviour failed.

BT is composed of three node types :

1. Sequence
2. Selector
3. Decorator

The basic functionality of these three are as follows:
Sequence:
If the child's node behaviour succeeds, the sequence continues and eventually the parent node succeeds if all child node behaviours succeed, otherwise the sequence fails.
Selector:
There are two main types of selector nodes: the probability and the priority selectors. When a probability selector is used by the child node, behaviours are selected based on parent-child probabilities set by the BT designer. On the other hand, if priority selectors are used, child node behaviours are ordered in a list and tried one after the other. Regardless of the selector type used, if the child's node behaviour succeeds the selector succeeds. If the child's node behaviour fails, the next child node in the order is selected (in priority selectors) or the selector fails (in probability selectors).
Decorator:
The decorator node adds complexity to and enhances the capacity of single child node behaviour. Decorator examples include the number of times a child's node behaviour runs or the time given to a child's node behaviour to complete the task [27].

1.1.3 Tree Search

Generally, most of the implementation of AI is done by using search algorithms. The search algorithms are usually known as the Tree search as they typically perform the search operation like a tree which explores the branches. There are different tree search techniques used to implement AI in games. They are informed searching, Uninformed searching, Problem Reduction, mean and analysis, Hill climbing, Generate and test, constrain satisfaction problem, and interleaving search. The tree search is most used in two-player games where the best move in a game is searched. The Minimax and Monte Carlo Tree search are most used in two-player games. For simple games, the size of the tree is less. But for the large games like chess, the tree size will huge to store [5].

1.1.4 Evolutionary Computation

Evolutionary computation is one of the optimizing solutions which are inspired by biological evolution. The key objective for the evolutionary computation is utility function, fitness function, or evolution function which returns the integer that optimizes the solution. Evolutionary algorithms are referred to as the subset of evolutionary computation. Evolutionary algorithms use the mechanism of reproduction, mutation, recombination, and selection for optimization of the solution [6]. Evolutionary computation in games

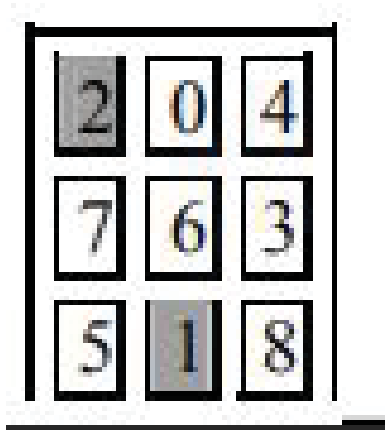


Figure 1.2: 8 puzzle in the game.

By using the evolutionary computation the 8 puzzle game or sliding tile game is easily optimized. In this example, the initial population of the game is p[2 0 4 7 6 3 5 1 8]. It first generates different populations based on the misplaced tiles to the goal state p1 [2 0 4 7 1 3 5 6 8] and p2 [7 6 4 5 0 3 1 8 2]. For each population, the fitness function values are calculated. The cross over of the populations was done. The resulting population is p[0 2 4 7 6 5 3 1 8.] Finally, the mutation of the population was done by changing the random position of the population p[0 2 4 7 6 5 3 1 8]to p[0 2 4 1 6 5 3 7 8]. For the new population again fitness function estimates if its maximum value it stops or it again tries for selection of the population and run the steps.

1.1.5 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning approach in which the agent gets the reward for each right action and gets a penalty for the wrong action. Reinforcement Learning is inspired by behaviorist psychology. The agents learn from the environment by interacting with it each time. In reinforcement learning, the main objective of the agent is to find the policy such that it maximizes the rewards. It is a feedback-based learning method, the feedback provides the learning method for an agent and improves its performance [26]. RL has been studied from a variety of disciplinary perspectives including research on operations, game theory, information theory, and genetic algorithms, and has been successfully applied to problems involving a balance between long-term and short-term rewards such as robot control and games.

1.1.6 Supervised Learning

Supervised learning is a method where the data is trained by a lot of examples, features, attributes. By using the training data the machine or game characters can predict the results of the outcome. A common example of supervised learning is a system that is expected to differentiate between two things based on providing a set of characteristics or data attributes such as the objects. Initially, the computer learns to distinguish by seeing several examples of available data of the objects. The machine should ideally be able to predict the kind of the object. supervised learning is also used in a wide variety of applications including financial services, medical care, fraud detection, categorization of web pages, identification of images and expression, and user modelling [10].

1.1.7 Unsupervised Learning

The AI Algorithm class is determined by the utility type (or training signal). The training signal is provided as data labels (target outputs) in supervised learning and is derived from the environment as a reward in reinforcement learning. Instead, unsupervised learning aims to discover input associations by looking for patterns in all input data attributes and without reference to a target output a method of machine learning that is typically guided by Hebbian learning and self-organization principles [25]. With unsupervised learning, instead of trying to mimic or predict target values, we concentrate on the intrinsic structure of and associations in data.

1.1.8 Snake Game

Snake game is a simple video game in which there are two elements which are a snake and the fruit. The game is limited to a confined space. The confined space can be decided by the developer or the width of the screen. The e snake is controlled by the human or an AI agent. The goal of the game is to capture as many fruits as possible without touching the boundaries and the snake self. The coordinates of the fruit remain until the snake-head captures it. If the snakes head captures with the fruit coordinates the length of the snake increases by one unit and the score was recorded each time. Every time the snake eats the fruit then the fruit coordinates move randomly in the box. The computer uses different algorithms in the implementation of the snake in the game. Human uses his or her intelligence to navigates the snake by using the direction keys to reach the fruit coordinates. If the head of the snake collides with the boundary points of the box, then the game ends. If the head of the snake collides with itself, then the game ends [2].

1.2 Aim and Objectives

1.2.1 Aim

This project mainly deals with the comparison and performance analysis of the searching algorithms in AI against Human-agent in the Snake Game.

1.2.2 Objectives

1. To identify some of the different searching algorithms used in the AI in games.

2. To develop the snake game by using the some of the selected searching algorithms.
3. To compare the results of the algorithms against Human Agent.

1.3 Research Questions

RQ1: What are the different searching algorithms used in the AI in games?
Motivation: The motivation behind selecting this research question is to find the some of the different searching algorithms in the AI through literature review.
RQ2: Which search algorithms perform better than the Human Agent?
Motivation: The motivation behind selecting this research question is to know about the performance of some of the search algorithms against the Human Agent and each other.

1.4 Research Methodology

Firstly a thorough literature review is conducted for understanding some of the different search algorithms and its implementation and its efficiency in a theoretical way.
Based on the theoretical study, few algorithms are selected based on the practical implementation of the AI in the snake game. The practical performance of these algorithms are compared with each other and also with the Human-agent.

Chapter 2

Related Work

Electronic Arts is one of the topmost leading companies in developing games. They also have shown very great success in developing AI in games. Their one of the most successful application AI is Ms. Pacman. Their most important paradigm in the development of AI is to optimize the parameters of the controller in the game. Ms. Pacman is a game has three or more ghosts those are at the closing position of the door when the game starts the Ms. Pacman needs to collect the points as the time goes on, the ghosts choose different paths to kill Ms. Pacman. Gallager and Ryan [8] developed a rule-based controller for the Ms. Pacman game. The current state of Ms. Pacman is determined by the controller based on the distance between the ghosts. So Ms. Pacman can retreat or explore in the game. The rule-based controller consists of a set of rules in which help to determine the new direction of movement for Ms.Pacman. with the distance threshold and direction probabilities, different weight parameters are evolved for the controller which helps in the implementation of the machine learning at a minimum level in the game. This related work helps in the development of the snake game by applying some of the rules which makes the algorithms more efficient than normal. For example, if the snakes head collides with the box then the game should stop by using the set of the rules we developed the snake in such a way that it moves away if any danger occurs in the games. So, this work helps to apply some of the constraints and rules that should apply to the snake in the game.

Eight puzzles are one of the classical board game, where it consists of eight distinct movable tiles, plus blank tile. The tiles are numbered from one to eight and are arranged randomly on the board. The game contains the two types of configurations initial and final. The blank tile is used to change the positions of the tile by swapping it with any tile horizontally or vertically adjacent to it to achieve the initial configuration to the final configuration of the tiles in the game. Daniel R. Kunkle [18] developed different Hurestic search functions by using the Manhattan distance and Manhattan distance plus Reversal Penalty Search between the original tile position to the goal tile position to achieve the final configuration. The Heuristic Search function methods find the optimal solutions which take a different amount of time to do so. So by this solution, AI can produce the minimum number of moves in the game. This paper provides more information about the Heuristic Search function which is used in the development of the A* Search algorithm and the Best First Search algorithm in the snake game.

Mario AI is a benchmark that aims to develop the best controller for playing games.

The game consists of stages where enemies and destructible blocks are present. The controller manipulates the Mario agent to reach the goal by dodging the enemies. To enhance an agent, the controller should operate according to the appropriate situation. Shunsuke Shinohara, Toshiaki Takano, Haruhiko Takase, Hiroharu Kawanaka [24] proposed the Mario AI agent by using the combination of the A* algorithm and Q-Learning in the game. The Q-Learning is used in the game to memorize the target location to the present location of the Mario. With the location nodes between the target and the present, the A* algorithm is used and to find the optimal path for the Mario to reach the goal. So the controller moves the Mario agent according to the path and becomes effective to reach the goal quickly in the game. This work provides the way for developing the heuristic function in the game by using the Q-learning method. The heuristic function is developed by calculating the Manhattan distance between the fruit coordinates to the adjacent coordinates of the snakes head in the game.

Kalaha is a simple two-player board game which six holes of rows on each side and a store on players right side. Each hole on both sides contains a specific number of stones or marbles or pebbles. The game starts by picking off any of his or her hole of all stones and dropping by the in the anti-clock by direction. If the dropping of the stones ends with the hole of his or her store then the player gains another chance in the game. The game ends with all the completion of his or her stones in all the holes. The AI agent by using the Minimax algorithm with the alpha-beta pruning has been developed with the utility function as the knowledge-based which is used to create a tree and searched the deeper of the tree at the depth of the 12 [12]. Thus it makes the AI agent which performs better than different algorithms like Minimax algorithm with utility function as simple and Minimax algorithm with the utility as the knowledge-based and Minimax with knowledge-based alpha-beta pruning with utility function as simple.Related work gave us understanding of the Minimax algorithm with different utility functions and their performance results. This work helps in understanding how the depth of the tree is developed in the game and why the game trees are more helpful in the two-player games.

Robo code is a game that plays with tank bullets of the player on one side and opponents on the other side is called as armored tanks. In this game, the player can collect the points by shooting to the opponent tank and their space. Yehonatan Sichel[13] developed the controller for the game by using genetic programming. The controller is used to move the NPCs, to change the direction of the NPCs, to attack the opponent, and to armor itself. With the Genetic programming, the chromosomes are the trees in the game and used to represent the attributes of the game and numeric constants. The genetic programming is used to generate different operations based on the situations for the controller. So the controller is effective in the game playing.

This study helps in understanding the effectiveness of the controller in the game and also the implementation of the genetic algorithms in the games. This study is helpful and motivated us for the further development of the games by using the optimization algorithms.

Chapter 3

Method

In this chapter, the literature review of the searching algorithms and the implementation of the searching algorithms in the snake game was performed. From sections 3.1 to 3.5 the theoretical study of the algorithms was presented and in section 3.6 the implementation of the algorithms was done.

3.1 UnInformed Searching Techniques

These are the techniques which blindly performs the search operation to reach the goal node.

3.1.1 Depth-First Search

The Depth-First Search explores a vertex to its deepest level and then it backtracks until unexplored nodes of the vertices reached. It is also called an edge-based method. Depth-First Search works on the stack data structure, which uses a last-in-first-out manner. Depth-First Search works recursively until the goal node reaches. The Depth-First Search traverses the edges twice and vertex only once along the path. Its space requirement is linear, which cannot increase exponentially like Breadth-First Search when a search of a node in depth. The time complexity of the Depth-First Search algorithm is $O(b^d)$ and space complexity is O(b*n) where b is the branching factor and n is the number of levels [9].

Advantages:

1. It uses the minimum time when the target is nearer, i.e. stops the searching when the goal node reaches.
2. In the worst cases also, there is a chance of reaching the target using many paths to find goal state.
3. The main reason to use the Depth-First Search because of its space requirement in a linear manner while searching.

3.1.2 Breadth-First Search

The technique work by traversing or exploring the deepest node before the proceeding to the adjacent node. That is, it explores or traverses the search tree by expands all the nodes in the first-level and then expands in the second level and reaches the goal on its way. So it is called as the level by level traversal technique. In this technique, all solutions for each node are found out. This guarantees the optimal solution. The Breadth-First Search uses a queue data structure to store the values. Though there are cycles in the graph, it uses an array to store the visited vertices in the search tree. It works in the principle of the FIFO(first in first out). The time and space complexity of the Breadth-First Search is $O(b^d+1)$ and $O(b^d+1)$ respectively where b is the branching factor and n is the number of levels.

Advantages

1. The shortest number of steps is required to reach the goal node in the search tree.
2. Guarantees a path to reach the goal node in the search tree.

3.1.3 Uniformed Cost Search

Uniform Cost Search is used to find a minimum cost traversal search for a graph tree to reach a path. It is one of the state space search algorithms. It uses a priority queue to find the minimum cost of adjacent nodes. It backtracks and finds a new solution for all possible ways to reach the destination. Then it chooses a minimum cost to traverse from root to destination. It always compares the paths minimum costs for a graph/tree and chooses the optimal cost when the path that is explored cannot find good results. The Uniform Cost Search algorithm is also called Dijkstra's single-source shortest path algorithm. The successors with higher costs in the queue are removed when minimum cost nodes are found. It breaks the finding a path when minimum cost nodes are found if it cannot produce the optimal state it backtracks to the previous unexplored path. Path. For the calculation of each node cost, it uses the formula of the c(m) = c(n) + c(n,m) where c(m) is the cost of the current node and c(m) is the cost of the previous node and c(n,m) is the distance of the cost from the node n to m. The time and space complexity of the algorithm is O(b1+C*/e) and O(b1+C*/e) where C is the optimal solution cost, and each activity costs of least.

Advantages

1. It finds the optimal solution by considering the minimum cost for each node.
2. It minimizes the space by finding minimum cost nodes.
3. It reduces the time complexity by choosing minimum nodes to explore.

Disadvantages

1. It finds many ways to get the minimum cost traversal from root to destination hence time complexity increases.
2. Its space complexity also increases when one path cannot find the optimal solution.

3. If the path chosen cannot find the optimal solution it compares with the previous solution and takes a decision which makes space complexity and execution time more.

3.1.4 Iterative-Deepening Search

Iterative-Deepening search is the combination of both the Depth-First Search and Breadth-First Search. It performs a Depth-First Search level by level until the target node found in the state space search tree. Here each level of depth search is considered as the iteration. therefore it is also called as the Iterative Deepening Depth First Search (IDDFS)[4]. It can guarantee to find an optimal solution along with the first generation of the path. It terminates the search when the solution at depth d is found. Since it uses both features it can traverse the graph with or without cycles. If the graph contains cycles it uses a Breadth-First Search to optimize the space to store nodes. It uses a limit L to perform search until that depth. If the target is found that depth is less than the limit L. It also uses the stack data structure in the implementation[15]. The time and space complexity this technique is the $O(b^d)$ and O(bd) respectively where b is branching factor and d is the depth of the search tree.

Advantages

1. Iterative Deepening Depth First Search uses both a Depth-First Search and Breadth-First Search to find the minimum cost weighted edge path.
2. It uses a new threshold to find the minimum cost of all nodes generated by cutting off the previous iterations.
3. It uses the Depth-First Search to optimize the space.

Disadvantages

1. Iterative Deepening Depth First Search recursively performs the previous phases hence it requires more space.
2. It uses a lot of time to perform iterations before the one to find a solution.
3. Time complexity increases in performing several iterations that cannot produce the goal state path

3.1.5 Bidirectional Search

Bidirectional search points forward from root and backward from the target state. It is a brute force algorithm that requires an initial state and a clear description of each state, goal state. It terminates the searching when both forward pointer node and backward pointer node pointers meet. Forward pointer moves from source and explores the nodes and the backward pointer moves from goal state until the forward pointer meets. It concatenates both the pointers to find the optimal path. Either forward or backward pointer can use Breadth-First Search algorithm to meet at a particular node. Each pointer explore only to half depth of the tree [27].

Advantages

1. It reduces the time complexity because it only searches half of the tree.
2. It requires less space.
3. It guaranteed to find an optimal solution.
4. It reduces traversing the unexplored nodes using breadth first search.

Disadvantages

1. At least one pointer uses a Breadth-First Search algorithm to traverse half of the tree hence it sometimes requires space.
2. If the breadth first search on either forward pointer or backward pointer fails the space complexity increases.
3. Time complexity increases while breadth first search using with greater depths.

3.2 Informed Search Techniques

These are the techniques which uses the Heuristic function for the search operation.

Heuristic Search

A Heuristic Search is a technique that is used to find an optimal solution in an accurate amount of time and evaluates the available information each time while exploring to other nodes when classical methods do not work. Heuristic Search algorithms work faster than the uninformed search algorithms.

Heuristic Evaluating Function is defined as the evaluation of the problem desirability usually, represented as the cost between the nodes in the states space search tree. Heuristic Evaluating Function estimates an optimal cost between a pair of nodes in the states space tree. It iteratively calculates the distances for each node cost optimally until the goal state reaches. The key things for the heuristic evaluation function are the problem domain, cost metrics, heuristic information in the problem. Lower bounds are chosen by heuristic functions between two pairs of nodes than actual cost hence it is referred to as admissibility [16]. This evaluating function used in playing games to evaluate the next move favorable to win. It evaluates the probability to win, lose, or draw in the games. The heuristic function used graphs with cycles, trees. These evaluating functions are in linear form [7].

3.2.1 Best First Search

The Best First Search algorithm is one of the simple Heuristic Search algorithm. The Best First Search algorithm works under the principle of exploring the goal node according to a specific rule. The Best First Search algorithm uses the heuristic function for the estimation of the specific rule in the problem domain. In the state-space search tree, the heuristic function is used for the determination of the distance between the nodes. The algorithm uses the two lists which are the open list and the closed list. The open list maintains the nodes that are to be explored in the state space tree and the closed list is used to maintain the nodes that are visited in the state-space search tree. When an open node found the shortest path then it is saved and longer one discarded. When the closed node found the shortest path then it is moved to an open node associated with it. The Best First Search uses the best node in all the unvisited nodes in the state-space search tree. It is the combination of Depth-First Search and Breadth-First Search algorithms by using the heuristic function for the cost estimation of the nodes. The Best First Search algorithm uses the priority queue data structure to maintain the distance between the nodes in the ascending order. The time complexity of the algorithm is O(bd+1) and space complexity is O(bd) [27].

Advantages

1. It uses both the breadth first search and depth first search techniques.

Disadvantages

1. It is not optimal.
2. It get struck in the loop by using depth first search.

3.2.2 A* Search

A* search algorithm combines both the best features of Uniform Cost Search and pure Heuristic Search to find an optimal solution and completeness of the path and optimal efficiency in the state space search tree. A* Search algorithm aims to find the path from starting node to the goal node by maintaining the tree of nodes and extending them to reach the goal state until. A* Search algorithm iterates each time and extends the path from by using the cost estimation, which helps in reaching to the goal node. The A* Search algorithm uses the formulae for the cost estimation which selects the minimize path in the state space search tree is f(n) = g(n) + h(n) where g(n) cost of the path from the source node to n, h(n) is the estimated heuristic cost from the target node to n, f(n) is the total optimal cost of a path going through node n [7]. If the heuristic function in the algorithm is admissible, the A* Search algorithm always chooses the least cost from the start to the goal node. The A* Search uses the priority queue data structure. At each step, the lower f values are removed, the adjacent nodes of f(n) and g(n) values are updated accordingly the iteration stops until the least f(n) value than any node in the queue. The time complexity of the algorithm is $O(b^d)$ and space complexity is $O(b^d)$ [19].

Advantages

1. It finds the best path in minimum time and reduces time complexity.
2. It uses a linear data structure to store the f(n) values.

3. Time complexity minimizes by searching the nodes by using heuristic evaluating functions.

4. It can explore only fewer nodes.

Disadvantages

1. The space complexity increases when the optimal solution cannot be found in a Best-First Search.

2. This algorithm overhead the managing of open and closed lists.

Example In the 9*9 matrix of a Sudoku game the values are fixed in 1 to 9 numbers any number cannot be in the same row and the same column then each 3*3 box occur 1 to 9 numbers which makes a sum 45 on each 3*3 then the player thinks to fit the values in boxes. firstly the player checks for the auto-generated present numbers and then for every single number checks at row and column so that player finishes in time. If one number put wrong in a single block then overall boxes cannot suit.

3.2.3 Iterative deepening-A* Search

Iterative deepening-A* Search algorithm uses the graph into the decision tree. Iterative deepening-A* resolves space complexity of Breadth-First Search and performs Depth-First Search for each iteration that completely tracks the cost of each node generated. Like A* Search algorithm, it also uses the heuristic function the f(n) = g(n) + h(n) where g(n) cost of the path from the source node to n, h(n) is the estimated heuristic cost from the target node to n, f(n) is the total optimal cost of a path going through node n. It terminates the iteration of the path when the cost of the heuristic function reaches threshold value and the search continues before exploring that path. The threshold of depth value is defined as the heuristic value which is the cost between the source to goal node in the search tree. The threshold value changes by every iteration by selecting the least f(n) value which is greater than the previous threshold value. This algorithm ends when the goal state is reached and the total cost is less than the threshold. The algorithm uses the threshold value as the search boundary. It uses a linear data structure to store the cost values.

Advantages

1. This algorithm produces an optimal solution in the first iteration

2. memory space is less for maximum depth search.

3. It requires less execution time than A*.

Disadvantages

1. It does not guarantee if many solutions are found.

3.3 Constrain Satisfaction Problems

In constraint satisfaction problems is for variables, there are a set of values, and constraints are assigned to the variables such that it allows the valid assignments to the variables. A unary constraint applicable to a single variable and binary constraints applicable to two variables such that the assignments of one variable cannot violate restrictions of both variables. In the graph coloring technique, there is a binary constraint on both nodes so that no adjacent node colors are the same [8].

3.3.1 Brute Force Backtracking

Constraint satisfaction with the brute force approach is called backtracking. It selects the order for the variables and starts assigning values to all variables one at a time. For each assignment, it should satisfy all the constraints that are previously assigned. If the assignment of one variable constraint is violated then it should not possible to re-satisfy the constraints that are assigned before. This algorithm results in success when a complete, or a consistent assignment is found. If any constraint is violated then the inconsistent state is found the result shows as a failure.

3.3.2 Limited Discrepancy Search

It is a tree search algorithm. It is useful when the whole tree is too large to search. In that case, this algorithm works as on searching a subset of a tree rather than a strict left-root-right search. Assuming the tree has heuristic order then the left branch finds the solution in less time than the right branch. In limited discrepancy search it follows Depth-First Search iterations repeatedly as a series. In the first iteration, it explores the leftmost sub-tree and in the second iteration it explores root-leaf sub-tree with exact one right branch [17]. In the limited discrepancy search algorithm for each iteration, it explores the paths with k discrepancies ranges from zero to depth of the tree.

3.3.3 Intelligent Backtracking

The performance of brute force backtracking can be improved by value ordering, variable ordering, back jumping, forward checking. The variable instantiation order can affect the size of the tree. In Variable ordering the order of assignment of the variables from most constrained ones to least constrained ones. If the variable has only one value remaining if the variable is consistent with the previously Instantiated assigned variable then it should be assigned immediately. The size of the instantiated variable can increase either statically, dynamically, or by reordering the remaining variables each time when a new variable is assigned. The order of given variables determines to choose the search techniques for a tree [11]. It doesn't affect the size of the tree and if all solutions are found then conflicts are not araised. In value ordering the values are from least to most constraint ones. It finds the best solution in minimum time. In Backjumping undoing the last constraint that is made which leads to failure. The last violated constraint is removed so that the problem reaches a consistent state. In forward checking if one variable assignment is made it priorly checks all the uninstantiated variables associated with it are satisfied that is consistent with previously assigned variables. If not the variable is assigned with its next value.

3.3.4 Constraint Recording

In constraint satisfaction problems there are two types of constraints implicit and explicit constraints. Implicit constraints discovered at the time of backtracking whereas explicit constraints are imposed by others. In Constraint recording the implicit constraints need not rediscovered it can be saved on explicitly.

3.4 Problem Reduction

The Problem Reduction is a method that divides the problems into sub problems and the solution of each sub-problem is represented by the AND-OR trees or graphs. AO* search algorithm is a Heuristic Search algorithm that solves the Problem Reduction problems in AI. AO* search algorithm does not explore all the solutions once it got a solution in the AND-OR trees or graphs. AO* uses the open list for the nodes which are that are to be traversed and closed list that are already processed. If a solvable node is visited in the graph it traverses again to reach the goal node, if an unsolvable node is reached it returns as the failure [26].

3.5 Hill Climbing Search

Hill climbing is a Heuristic Search algorithm that finds the solution in a reasonable time. Heuristic Search allocates the ranks for all potential alternatives using the information available. It uses a heuristic function and large inputs to find the solution. It cannot be guaranteed on finding the solution may be globally optimal. It solves the problem by choosing a large set of inputs and analyzing the minimum or maximum points using heuristic functions. It only checks the immediate neighbor to know whether it is maximum or minimum from the present point. It searches locally in the increasing order of the elevation to find the peak value or optimal cost solution for a problem. Hill-Climbing Search algorithm used for optimizing the mathematical problems. Heuristic functions select the best route out of possible routes to find solution in optimal time. space complexity of Hill Climbing Search is O(b) [23].

Advantages

1. It can find the best solution in an optimal time.
2. It generates all possible solution for a problem such that an optimal solution can find easily.

Disadvantages

1. It can quit searching when the neighbor state has worse value than the current state.
2. The process terminates even it can find the best solution.

3.6 Implementation of the Algorithms in Snake Game

In this the implementation of the algorithms was done. The algorithms are Breadth-First Search, Depth First Search, Best First Search, Hamilton Search, A* Search, Best First Search.

3.6.1 Experimental setup

The algorithms are implemented in the programming language python. The module used in the python was the pygame. The programs do not need any specific laptop for the implementation. Python version with pygame module supported is required to run.

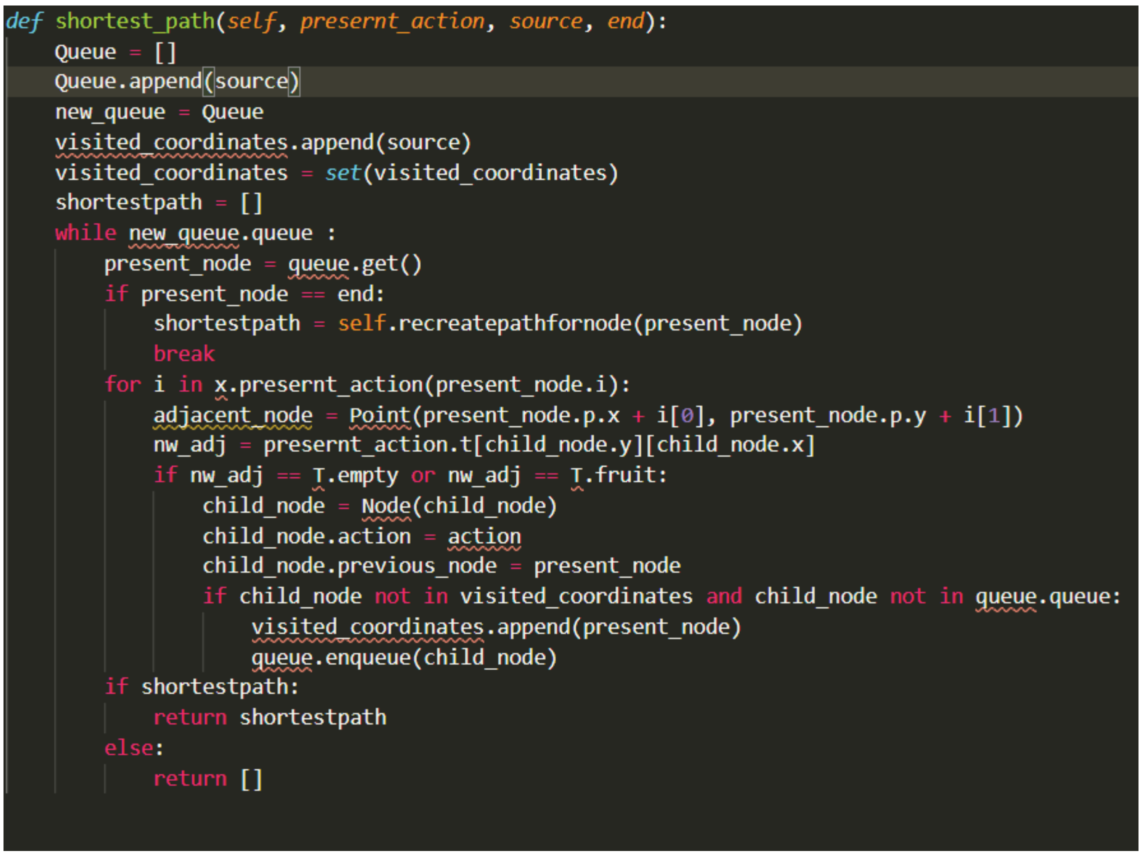
3.6.2 Using Breadth First Search

By using the Breadth-First Search technique in the snake game, the snake traverses or explores the adjacent coordinates rather than the deepest coordinates of the game. The algorithm uses the queue data structure and append each adjacent nodes recursively and find the path of the fruit coordinates in the snake game. By using the path, the snake reaches the fruit coordinates in the game. The snake does not visit the coordinates again until it has reached the fruit coordinates in the game.

Working

- Step 1: In this step, the initial coordinates of the snake are pushed on the queue and set the initial coordinates as visited coordinates.
- Step 2: In this step, Dequeue the coordinates in the queue one by one, and its all unvisited adjacent coordinates are pushed on the queue and set them as visited coordinates.
- Step 3 :In this step, the step2 will be repeated continuously until the fruit coordinates are visited.
- Step 4: In this step, if the fruit coordinates are visited it stop and traces the path and returns it.

In this, the pseudo-code and working steps of the Breadth-First Search algorithm was presented.



```
def shortest_path(self, presernt_action, source, end):
    Queue = []
    Queue.append(source)
    new_queue = Queue
    visited_coordinates.append(source)
    visited_coordinates = set(visited_coordinates)
    shortestpath = []
    while new_queue.queue :
        present_node = queue.get()
        if present_node == end:
            shortestpath = self.recreatepathfornode(present_node)
            break
        for i in x.presernt_action(present_node.i):
            adjacent_node = Point(present_node.p.x + i[0], present_node.p.y + i[1])
            nw_adj = presernt_action.t[child_node.y][child_node.x]
            if nw_adj == T.empty or nw_adj == T.fruit:
                child_node = Node(child_node)
                child_node.action = action
                child_node.previous_node = present_node
                if child_node not in visited_coordinates and child_node not in queue.queue:
                    visited_coordinates.append(present_node)
                    queue.enqueue(child_node)
        if shortestpath:
            return shortestpath
        else:
            return []
```

Figure 3.1: Pseudo code for the BFS algorithm in Snake game

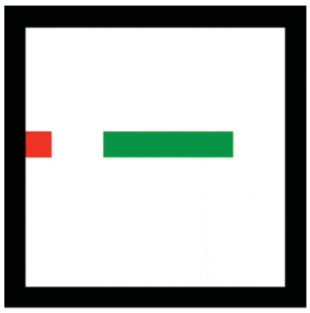


Figure 3.2: BFS algorithm in Snake game

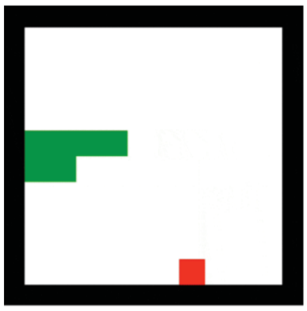


Figure 3.3: BFS algorithm in Snake game

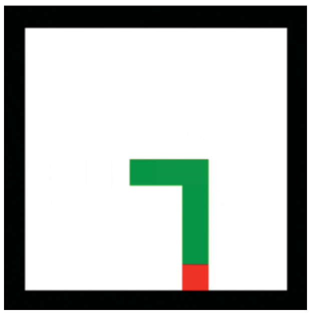


Figure 3.4: BFS algorithm in Snake game

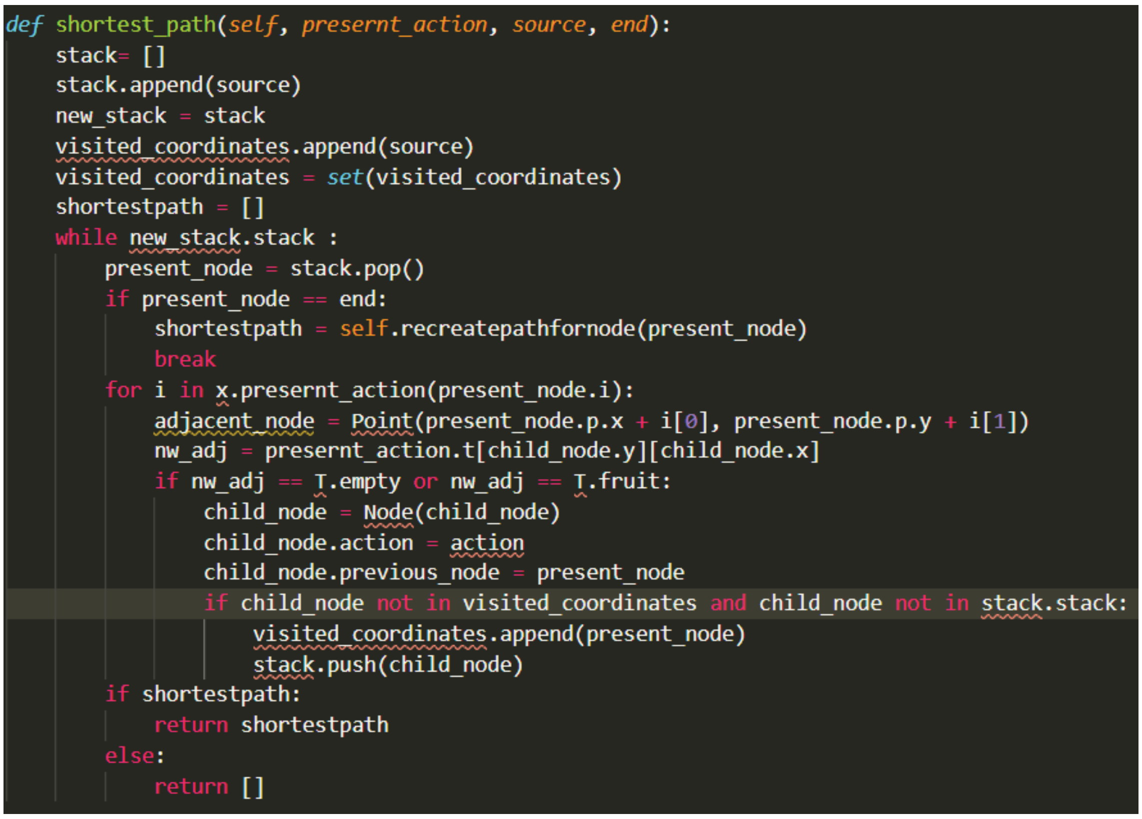
3.6.3 Using Depth First Search

By using the Depth-First Search in the game, the snake explores the coordinates to its deepest level, and then it backtracks until unexplored coordinates to reach the fruit coordinates in the game. The algorithm uses the stack data structure for the exploration of the coordinates and append each coordinates recursively and find the path for the snake game to reach the fruit coordinates in the game. By using the path, the snake reaches the fruit coordinates in the game. The snake does not visit the coordinates again until it has reached the fruit coordinates in the game. The snake uses more time to reach the fruit coordinates in the game in many cases. The snake traverses the long path by using this algorithm. The average number of visited coordinates in the game is very high by using this algorithm. There is a high chance of reaching the dead state in the game when the snake grows higher.

Working

- Step 1: The initial coordinates of the snake are pushed on the stack and set the initial coordinates as visited coordinates.
- Step 2: The next adjacent unvisited coordinates are pushed on to the stack recursively and set the coordinates as the visited coordinates.
- Step 3: The step2 will be repeated continuously until the fruit coordinates are visited.

- Step 4: If the fruit coordinates are visited it stop and traces the path and returns it.



```
def shortest_path(self, presernt_action, source, end):
    stack= []
    stack.append(source)
    new_stack = stack
    visited_coordinates.append(source)
    visited_coordinates = set(visited_coordinates)
    shortestpath = []
    while new_stack.stack :
        present_node = stack.pop()
        if present_node == end:
            shortestpath = self.recreatepathfornode(present_node)
            break
        for i in x.presernt_action(present_node.i):
            adjacent_node = Point(present_node.p.x + i[0], present_node.p.y + i[1])
            nw_adj = presernt_action.t[child_node.y][child_node.x]
            if nw_adj == T.empty or nw_adj == T.fruit:
                child_node = Node(child_node)
                child_node.action = action
                child_node.previous_node = present_node
                if child_node not in visited_coordinates and child_node not in stack.stack:
                    visited_coordinates.append(present_node)
                    stack.push(child_node)
        if shortestpath:
            return shortestpath
        else:
            return []
```

Figure 3.5: Pseudo code for the DFS algorithm in Snake game

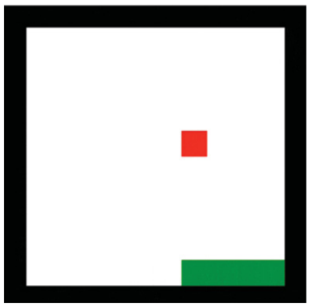


Figure 3.6: DFS algorithm in Snake game

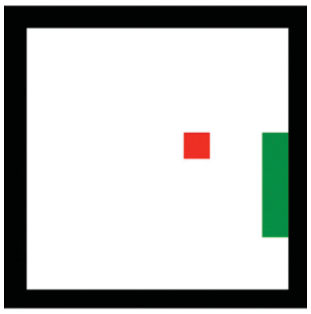


Figure 3.7: DFS algorithm in Snake game

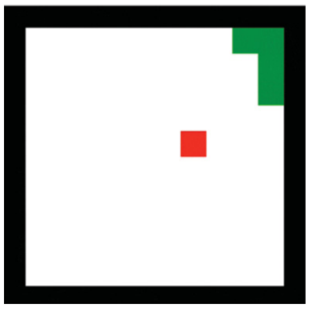


Figure 3.8: DFS algorithm in Snake game

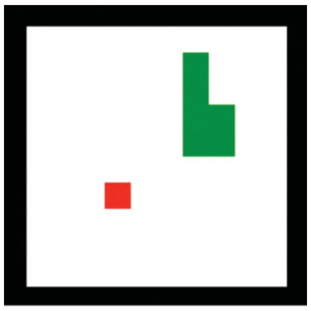


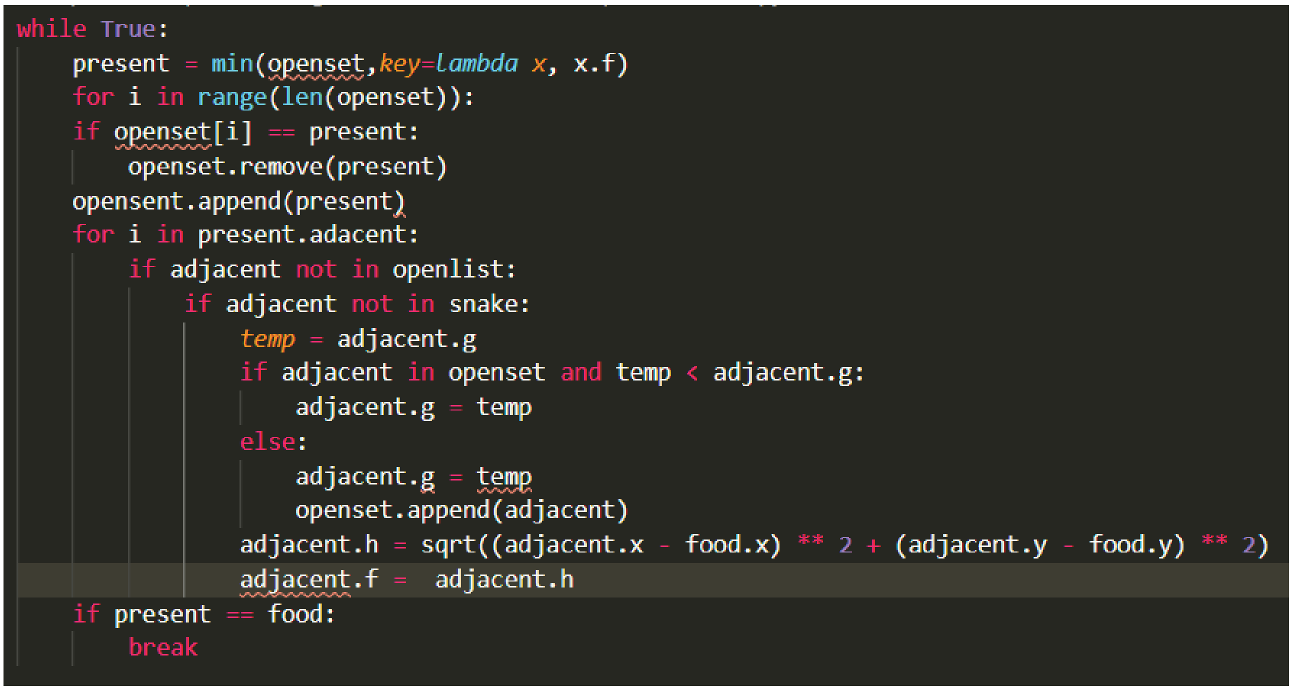
Figure 3.9: DFS algorithm in Snake game

3.6.4 Using Best First Search

The Best First Search algorithm uses the estimation function f(n) = h(n) where h(n) calculates the Manhattan distance between the adjacent coordinates to the fruit coordinates in the game. Based on the least f(n) value the snake moves along the coordinates and reaches the fruit coordinates in the game. The Best First Search algorithm finds the optimal path for the snake to reach the fruit coordinates in the game. Because of the greedy algorithm, The open list is used to explore the coordinates. The closed list is used to stored the visited coordinates.The average number of visited coordinates in the game is less. By using the Best First Search algorithm some of the dead ends in the game are resolved.

Working

- Step 1: The initial coordinates of the snake are appended and set it as the current node.
- Step 2: The Manhattan distance between the adjacent coordinates of the current node and fruit coordinates are calculated and appended in the open list.
- Step 3: The least Manhattan distance of the adjacent coordinates is selected from the open list and make it as the current node and current node is set to the closed list.
- Step 4: The step2 and step3 repeat continuously until the fruit coordinates were visited and the path was returned.



```
while True:
    present = min(openset,key=lambda x, x.f)
    for i in range(len(openset)):
    if openset[i] == present:
        openset.remove(present)
    opensent.append(present)
    for i in present.adacent:
        if adjacent not in openlist:
            if adjacent not in snake:
                temp = adjacent.g
                if adjacent in openset and temp < adjacent.g:
                    adjacent.g = temp
                else:
                    adjacent.g = temp
                    openset.append(adjacent)
                adjacent.h = sqrt((adjacent.x - food.x) ** 2 + (adjacent.y - food.y) ** 2)
                adjacent.f =  adjacent.h
    if present == food:
        break
```

Figure 3.10: Pseudo code for the Best first algorithm in Snake game

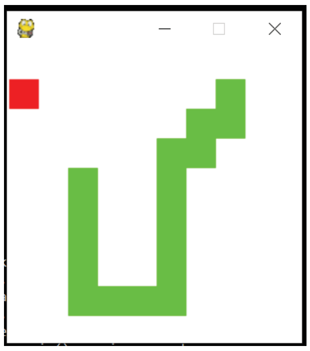


Figure 3.11: Best First Search algorithm step 1 in Snake game

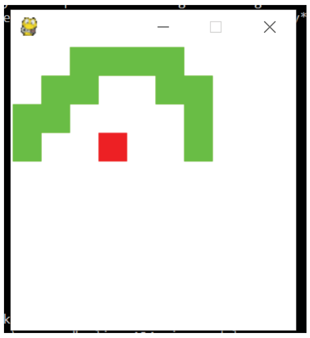


Figure 3.12: Best First Search algorithm step 2 in Snake game

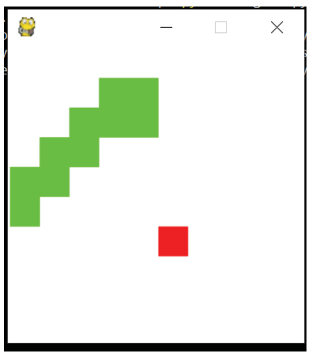


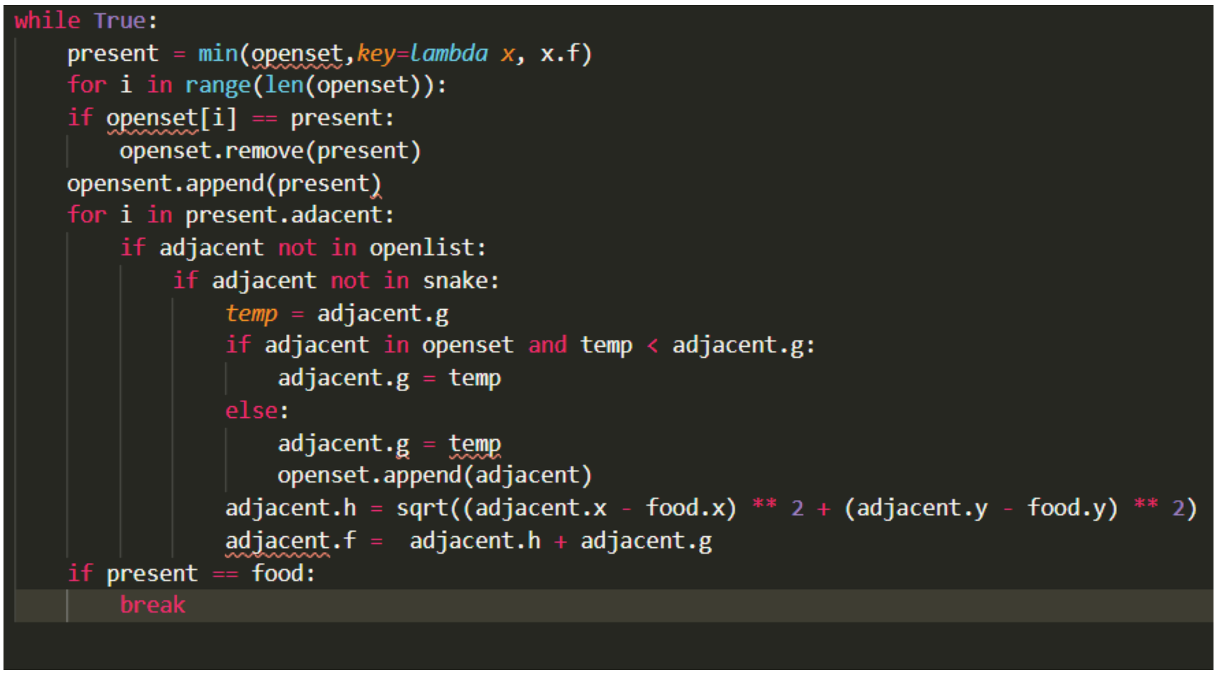
Figure 3.13: Best First Search algorithm step 3 in Snake game

3.6.5 Using A* Search

The A* Search algorithm uses the estimation function f(n) = g(n) + h(n) where h(n) calculates the Manhattan distance between the adjacent coordinates to the fruit coordinates in the game and g(n) is the heuristic cost of that coordinates. Based on the least f(n) value the snake moves along the coordinates and reaches the fruit coordinates in the game. This algorithm improves the path of the Best First Search by using the g(n) function. By using the A*search algorithm some of the dead ends in the game are resolved.

Working

- Step 1: The initial coordinates of the snake are appended and set it as the current node.
- Step 2: The neighbors of the current node were appended in the open list.
- Step 3: The Manhattan distance between the adjacent coordinates of the current node to the fruit coordinates is calculated and the estimated cost of the adjacent coordinates to the current node is added which is called the f(n) value.
- Step 4: The least f(n) value is from the open list coordinates are selected and makes it to the current node and the current node is set in the closed list.
- Step 5: The step 4,step 2, and step 3 repeat continuously until the fruit coordinates were visited and the path was returned.



```
while True:
    present = min(openset,key=lambda x, x.f)
    for i in range(len(openset)):
    if openset[i] == present:
        openset.remove(present)
    opensent.append(present)
    for i in present.adacent:
        if adjacent not in openlist:
            if adjacent not in snake:
                temp = adjacent.g
                if adjacent in openset and temp < adjacent.g:
                    adjacent.g = temp
                else:
                    adjacent.g = temp
                    openset.append(adjacent)
                adjacent.h = sqrt((adjacent.x - food.x) ** 2 + (adjacent.y - food.y) ** 2)
                adjacent.f =  adjacent.h + adjacent.g
    if present == food:
        break
```

Figure 3.14: Pseudo code for the A* Search algorithm in Snake game

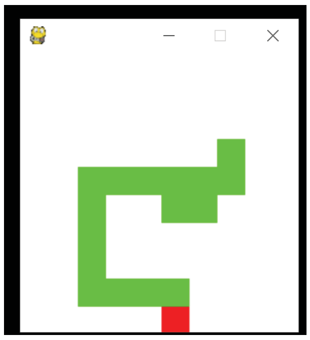


Figure 3.15: A* Search algorithm in Snake game

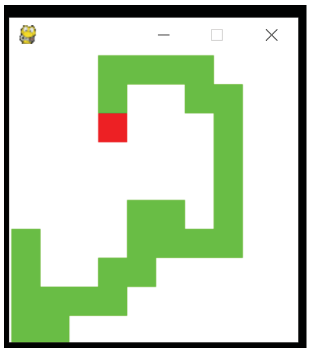


Figure 3.16: A* Search algorithm in Snake game

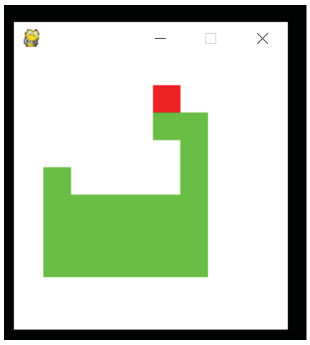


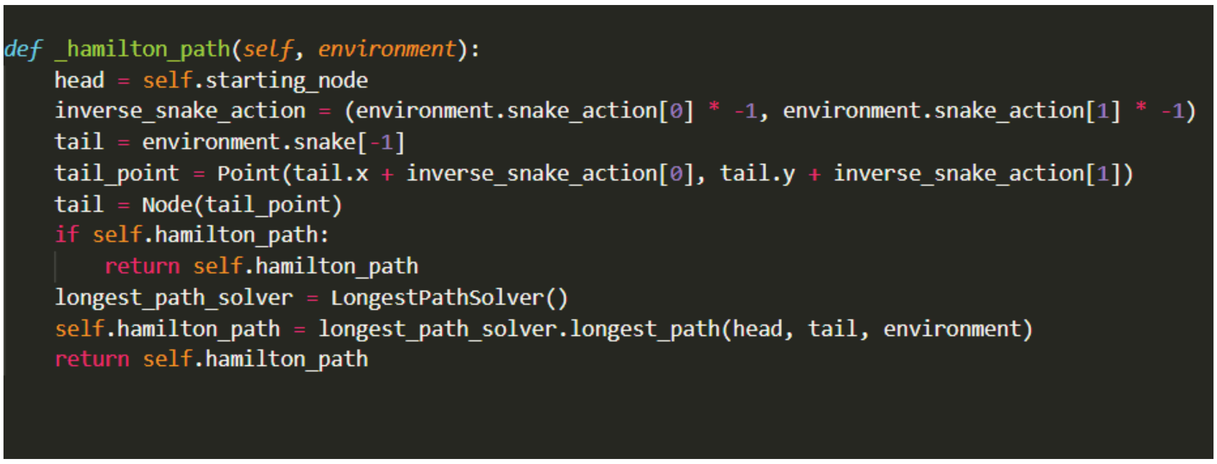
Figure 3.17: A* Search algorithm in Snake game

3.6.6 Using Hamilton Search

Hamilton path is the path which the node in the graph should visit exactly only once. It is one of the brute force search algorithm. This algorithm is very much similar to the depth-first algorithm. In the snake game, it explores all the possible paths and eventually reaches the fruit coordinates. Longest paths are explored by using this algorithm. There is a high chance of reaching the dead state in the game when the snake grows higher. The snake does not visit the coordinates again until it has reached the fruit coordinates in the game.

Working

- Step 1: The initial coordinates of the snake are pushed on the stack and set the initial coordinates as visited coordinates
- Step 2: The next adjacent unvisited coordinates are pushed on to the stack recursively and set the coordinates as the visited coordinates.
- Step 3: If any coordinates are visited already once it backtracks and changes the path.
- Step 4: The step 2, step 3 will be repeated continuously until the fruit coordinates are visited.
- Step 5: If the fruit coordinates are visited it stops and returns the traces path.



```
def _hamilton_path(self, environment):
    head = self.starting_node
    inverse_snake_action = (environment.snake_action[0] * -1, environment.snake_action[1] * -1)
    tail = environment.snake[-1]
    tail_point = Point(tail.x + inverse_snake_action[0], tail.y + inverse_snake_action[1])
    tail = Node(tail_point)
    if self.hamilton_path:
        return self.hamilton_path
    longest_path_solver = LongestPathSolver()
    self.hamilton_path = longest_path_solver.longest_path(head, tail, environment)
    return self.hamilton_path
```

Figure 3.18: Pseudo code for the Hamilton Seach algorithm in Snake game

3.6.7 Human Agent

In the Snake game, the Human Agent uses the keyboard arrow keys for the movement of the Snake and chooses the random path to reach the fruit coordinates in the game. The right direction key is used to move the snake along the right. The left direction key is used to move the snake along the left. The up direction key is used to move the snake along the upwards. The down direction key is used to move the snake along the down. The selection of the path depends upon the agent only.

Chapter 4

Results and Analysis

4.1 Experiment 1

The experiment of the algorithms was done by setting the timer of the 120 seconds and run each algorithm about three times and the height and width of the game are 300 and 300 respectively. The fruit coordinates of the game were generated randomly. The score of each algorithm was noted down in the below tables.

Algorithm	Food eaten	Time taken
Breadth First Search	73	120
Depth First Search	22	120
Best First Search	78	120
A* Search	92	120
Human Agent	32	120
Hamiltonian Search	26	120

Table 4.1: Results of the Algorithm in First run

In the second run the algorithms run about 90 seconds and the fruit coordinates of the game were generated randomly here also.

Algorithm	Food eaten	Time taken
Breadth First Search	56	90
Depth First Search	19	90
Best First Search	62	90
A* Search	74	90
Human Agent	26	90
Hamiltonian Search	19	90

Table 4.2: Results of the Algorithm in Second run

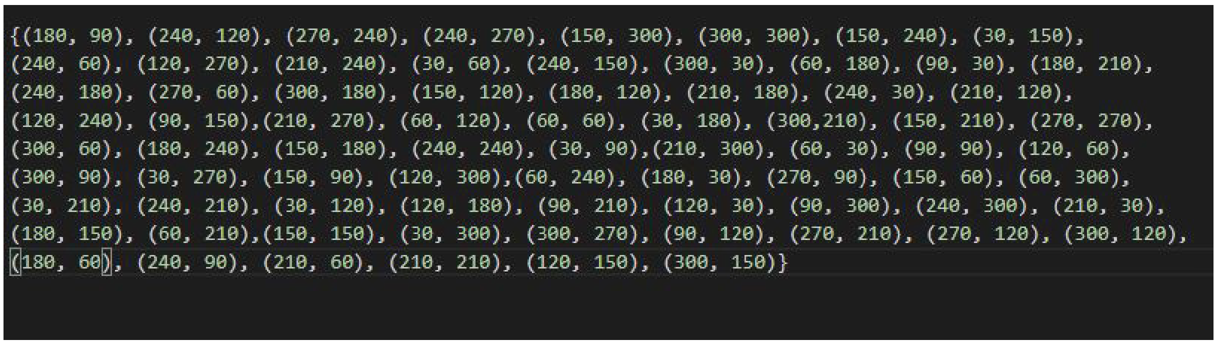
In the third run the algorithms run about 60 seconds and the fruit coordinates of the game were generated randomly here also.

Algorithm	Food eaten	Time taken
Breadth First Search	42	60
Depth First Search	15	60
Best First Search	43	90
A* Search	50	90
Human Agent	21	90
Hamiltonian Search	17	90

Table 4.3: Results of the Algorithm in Third run

4.2 Experiment 2

In the second experiment, we place the fruit coordinates at some fixed coordinates and run the algorithms each time. This experiment is used to calculate the time taken by each algorithm to reach a specified goal. These are specified positions of the fruit coordinates are placed in the game and run the algorithms. For example we place the fruit coordinates at coordinates (30,60) and run each algorithm at starting location at (0,0) coordinates of the snake head.



```
{(180, 90), (240, 120), (270, 240), (240, 270), (150, 300), (300, 300), (150, 240), (30, 150),
(240, 60), (120, 270), (210, 240), (30, 60), (240, 150), (300, 30), (60, 180), (90, 30), (180, 210),
(240, 180), (270, 60), (300, 180), (150, 120), (180, 120), (210, 180), (240, 30), (210, 120),
(120, 240), (90, 150),(210, 270), (60, 120), (60, 60), (30, 180), (300,210), (150, 210), (270, 270),
(300, 60), (180, 240), (150, 180), (240, 240), (30, 90),(210, 300), (60, 30), (90, 90), (120, 60),
(300, 90), (30, 270), (150, 90), (120, 300),(60, 240), (180, 30), (270, 90), (150, 60), (60, 300),
(30, 210), (240, 210), (30, 120), (120, 180), (90, 210), (120, 30), (90, 300), (240, 300), (210, 30),
(180, 150), (60, 210),(150, 150), (30, 300), (300, 270), (90, 120), (270, 210), (270, 120), (300, 120),
(180, 60), (240, 90), (210, 60), (210, 210), (120, 150), (300, 150)}
```

Figure 4.1: List of fruit coordinates to be fixed

Algorithm	Food eaten	Time taken
Breadth First Search	75	104 seconds
Depth First Search	75	203 seconds
Best First Search	75	94 seconds
A* Search	75	82 seconds
Human Agent	75	146 seconds
Hamiltonian Search	75	184 seconds

Table 4.4: Results of the Algorithms at fixed positions of the fruit coordinates

4.3 Final Results

In this section, we test the algorithms separately by making one algorithm as the selected and other algorithms as the opposing algorithms. Experiment 1 was conducted each between the selected algorithm and opposing algorithms and predicted the winner based on the score of the food eaten and experiment 2 between the algorithms. In Table 4.5 the results for Breadth-First Search Algorithm are Shown below where the opposing algorithms Depth First Search, Best First Search, A* Search, Human-Agent, Hamiltonian Search. Firstly the Breadth-First Search and Depth-First Search was run separately by using the experiment 1 and experiment 2 based on the score of these experiments the winner between the algorithms are decided. These experiments were done ten times and the win and loss of the algorithms were noted. Here we have done the experiments ten times the Breadth-First Search algorithm was won ten out of the ten times against the Depth First Search. The winner was decided based on the score and time taken to reach specific coordinates of the fruit. Similarly the entire all the results of the algorithms noted by using the win or loss method.

In the Table 4.5 the results for Breadth Search Algorithm are Shown below where the opposing algorithms are Depth First Search, Best First Search, A* Search, Human Agent, Hamiltonian Search and the selected algorithm is Depth First Search algorithm.

Opposing algorithm	Wins(BFS)	losses	Win Percentage
Depth First Search	10	0	100%
Best First Search	6	4	60 %
A* Search	2	8	20%
Human Agent	9	1	90%
Hamiltonian Search	10	0	100%

Table 4.5: Results of the Breadth First Search

In the Table 4.6 the results for Depth Search Algorithm are Shown below where the opposing algorithms are Breadth First Search, Best First Search, A* Search, Human Agent, Hamiltonian Search and the selected algorithm is Depth First Search algorithm.

Opposing algorithm	Wins (DFS)	losses	Win Percentage
Breadth First Search	0	10	0%
Best First Search	0	10	0%
A* Search	0	10	0%
Human Agent	1	9	10%
Hamiltonian Search	3	7	30%

Table 4.6: Results of the Depth First Search

In the Table 4.7 the results for Best Search Algorithm are Shown bellow where the opposing algorithms are Breadth First Search, Depth First Search, A* Search, Human Agent, Hamiltonian Search and the selected algorithm is Best First Search algorithm.

Opposing algorithm	Wins(Best First Search)	losses	Win Percentage
Depth First Search	10	0	100 %
Breadth First Search	5	5	50 %
A* Search	3	7	30 %
Human Agent	10	0	100%
Hamiltonian Search	10	0	100 %

Table 4.7: Results of the Best First Search

In the Table 4.8 the results for A* Search Algorithm are Shown bellow where the opposing algorithms are Breadth First Search, Depth First Search, Best first Search, Human Agent, Hamiltonian Search and the selected algorithm is A* Search algorithm.

Opposing algorithm	Wins(A*)	losses	Win Percentage
Depth First Search	10	0	100 %
Breadth First Search	9	1	90 %
Best First Search	8	2	80 %
Human Agent	10	0	100%
Hamiltonian Search	10	0	100 %

Table 4.8: Results of the A* Search Algorithm

In the Table 4.9 the results for Human Agent are Shown bellow where the opposing algorithms are Breadth First Search, Depth First Search, A* Search, Best First Search, Hamiltonian Search and the selected algorithm is Human Agent.

Opposing algorithm	Wins (Human)	losses	Win Percentage
Depth First Search	10	0	100 %
Breadth First Search	1	9	10 %
Best First Search	1	9	10 %
A* Search	0	10	0%
Hamiltonian Search	7	3	70 %

Table 4.9: Results of the Human Agent

In the Table 4.10 the results for Hamiltonian Search Algorithm are Shown bellow where the opposing algorithms are Breadth First Search, Depth First Search, A* Search, Human Agent, Best First Search the selected algorithm is the Hamiltonian Search algorithm.

Opposing algorithm	Wins (Hamiltonian)	losses	Win Percentage %
Depth First Search	6	4	60 %
Breadth First Search	0	10	0 %
Best First Search	0	10	0 %
A* Search	0	10	0 %
Human Agent	2	8	20 %

Table 4.10: Results of the Hamiltonian search path algorithm

4.4 Results of the Literature Review

The literature review provides more information and implementation details of other AI techniques and algorithms in different scenarios. From the Literature review of the algorithms, there are many more different search techniques and algorithms that are found that are not known to me. The literature review quiet helped the thesis to understand which searching algorithms suitable to implement in the snake game.

4.5 Analysis of the Outcomes

This section is divided into an analysis of the searching algorithms, implementation of the searching algorithms in the snake game, results of searching algorithms in the game.

4.5.1 Analysis of the searching algorithms:

The uninformed searching technique works on the blind search, There is no guarantee to reach the goal position at a particular time. By using this technique different algorithms work. These are the algorithms Depth-First Search, Breadth-First Search, Bidirectional search, Iterative Deeping search, Uniform Cost Search etc Depth-First Search selects the node and traverses along the entire path of that node by using stack and recursively back it. So, this will take time consuming to reach the goal node which is nearer to the source node. Breadth-First Search selects the node and traverses level by level in the path of that node by using a queue. So, this will take time-consuming to reach the goal node which is farther to the source node. But using this the shortest path can be found. Uniform Cost Search is based on the selecting the least cost between the nodes and traverses along the level and backtracks it. This algorithm is similar to the Breadth-First Search but it is also founding all possible paths which are time-consuming. Iterative Deepening uses both the Breadth-First

Search and Depth-First Search which traverses level by level of the depth. It is also time consuming to reach the goal node. Bidirectional search uses to find the path in both directions from source to goal and goal to source. So exploring from both sides is unnecessary.
The informed searching technique uses an estimation function, which is used to reach the goal node in the tree. The selection of the estimation function depends on the problem domain or programmer selection, etc. A* Search, Best First Search, Iterative A* Search, etc are the different informed searching technique algorithms. Best-First Search uses the heuristic function for the cost estimation between the adjacent nodes to the goal node. So this makes the algorithm efficient in searching. A* Search uses the heuristic function for the cost estimation between the adjacent nodes to the goal node and its heuristic value. So this makes the algorithm more efficient in searching. It takes less time to reach a long path. Iterative A* Search uses the threshold value and cut off the greater estimation value and uses the minimum threshold value in the list for each step to reach the goal node in the search.

The Problem Reduction is used to solve the hard problem by diving into the tree and solve these divided problems using the AND-OR graphs. AO* algorithm estimates the heuristic value for each nodes and arcs, and changes the values heuristic values and finds the optimal path in solution. There will be a chance of not finding the optimal path in the solution also.

The main use of the constraint satisfaction problem is to satisfy the given set of variables. There are different methods used to solve the constraint satisfaction problems which are brute force backtracking, limited discrepancy search, intelligent backtracking, constraint recording, etc. The brute force backtracking search assigns all possible values to explicit constraints and verify that with all the implicit constraints. The limited discrepancy does not use the left root search in the tree and uses the heuristic function and explores the tree. The limited discrepancy is used to search the entire huge tree also. The intelligent backtracking is used to back jumping, forward checking, restoring values to make more efficient in solving the problems. Constraint recording is to rediscover the constraint in the time of backtracking.

The hill-climbing is one of the local search technique which continuously searches until it reaches the optimal solution which is the maximum point but if there are different optimal solutions it was unable to find out when it reaches one optimal solution.

4.5.2 Analysis of implementation of searching algorithms

In the snake game, the Breadth-First Search traverses the optimal path to reach the fruit coordinates. The time complexity of the algorithm is high because to find the optimal path. By using the Depth-First Search the snake traversers the longer distance even if the fruit coordinates nearer to its initial coordinates. The number of nodes processed is high in the game by using this algorithm. Hamilton search algorithm also similar to the Depth-First Search but it only visits the coordinates only once and also traverser the straight path for along time. By using this algorithm also the number of nodes processed is high. The Best first calculates the heuristic distance fruit coordinates and adjacent coordinates and traverses it. By using this algorithm the number of nodes processed is also less. The time complexity of the algorithm is low compared to others. The A* Search algorithm also uses cost estimation by calculating the heuristic value between the adjacent coordinates to fruit coordinates and adjacent coordinates heuristic value. The performance of the algorithm is effective compared to others and take the optimal path to reach the fruit coordinates.

4.5.3 Analysis of the results

From the Table 4.5 The Breadth-First Search Algorithm was won more against the Depth-First Search, Human-Agent, Hamiltonian Search and loss more against the A* Search algorithm and the Best First Search algorithm.
From Table 4.6 The Depth-First Search Algorithm was not won more against all the remaining algorithms.
From the Table 4.7 The Best-First Search Algorithm was won more against the Depth-First Search, Human-Agent, Hamiltonian Search and loss more against the A* Search algorithm and equally against the Breadth-First Search.
From the Table 4.8 The A* Search algorithm was won more against all the remaining algorithms.
From the Table 4.9 The Human Agent was won more against the Depth First Search and Hamiltonian Search and loss more against A* Search, Breadth First Search, Best First Search.
In the Table 4.10 The Hamiltonian Search was won more against the Depth First Search and loss more against A* Search, Breadth First Search, Best First Search, and Human Agent.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The purpose of the thesis is to identify a few searching algorithms used in AI by literature review and also to conduct the performance analysis of the human agent and a few algorithms in the snake game. The selected algorithms were compared with each other as well as against the human agent in terms of performance such as the score achieved by each algorithm in the game. From the Background work of the thesis, we concluded that the different AI methods are useful in the development of AI in games. The Literature review and related works provided enough knowledge to implement the algorithms in the snake game.We implemented the snake game by using some of the algorithms. The results of some algorithms and human agent in the snake game were done by using the experiments one and two. From the results of the algorithms, we concluded that the performance of the A* Search algorithm was relatively good when compared with other algorithms in the game. Because A* Search algorithm uses the Manhattan distance and heuristic cost that makes the AI take the shortest path and helps the AI to proceed less number of the nodes to reach the goal. A* Search algorithm also takes less time to execute. While many algorithms rather than A* Search algorithm travels the longer path that usually proceeds to takes more nodes and more time to execute in the game. While a Human Agent uses the random path, So the path to reach the goal is unpredictable in the game. We concluded that the performance of the algorithms was better than the human agent in the game.

5.2 Future Work

In this work, we implemented and compared a few of the search algorithms because of time constraints and work pressure we can implement more various number of search algorithms like Iterative A* Search, Uniform Cost Search, etc in this game. We can also combine different AI methods algorithms to make a better performance test on the games which might show an impact on new evolving algorithms. In the future, this work can be implemented with different constraint techniques like limited discrepancy search, backtracking, forward checking and ,genetic programming etc.

References

[1] Finite state machines (are boring). `https://martindevans.me/heist-game/2013/04/16/Finite-State-Machines-(Are-Boring)/`. (Accessed on 10/05/2020).

[2] Snake (video game genre) - wikipedia. `https://en.wikipedia.org/wiki/Snake_(video_game_genre)`. (Accessed on 09/13/2020).

[3] Rehman Butt. *Performance Comparison of AI Algorithms: Anytime Algorithms.* 2008.

[4] Rehman Butt and Stefan J Johansson. Where do we go now? anytime algorithms for path planning. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 248–255, 2009.

[5] Murray S Campbell and T. Anthony Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[6] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. Evolutionary algorithms for solving multi-objective problems. 5, 2007.

[7] 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.

[8] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68(2):211–241, 1994.

[9] R Gayathri. Comparative analysis of various uninformed searching algorithms in ai, 2019.

[10] Sally Goldman and Yan Zhou. Enhancing supervised learning with unlabeled data. In *ICML*, pages 327–334. Citeseer, 2000.

[11] Carla P Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. pages 208–213, 1998.

[12] Marcus Östergren Göransson. *Minimax Based Kalaha AI.* 2013.

[13] Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Evolving strategy for a probabilistic game of imperfect information using genetic programming. *Genetic Programming and Evolvable Machines*, 9(4):281–294, 2008.

[14] Daniel Johnson and Janet Wiles. Computer games with intelligence. In *10th IEEE International Conference on Fuzzy Systems.(Cat. No. 01CH37297)*, volume 3, pages 1355–1358. IEEE, 2001.

[15] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[16] Richard E Korf. *Real-time heuristic search*, volume 42. Elsevier, 1990.

[17] Richard E Korf. Improved limited discrepancy search. pages 286–291, 1996.

[18] Daniel R Kunkle. Solving the 8 puzzle in a minimum number of moves: An application of the a* algorithm. *Introduction to Artificial Intelligence*, 2001.

[19] Alberto Martelli. On the complexity of admissible search algorithms. *Artificial Intelligence*, 8(1):1–13, 1977.

[20] Carlos Ramos, Juan Carlos Augusto, and Daniel Shapiro. Ambient intelligence—the next step for artificial intelligence. *IEEE Intelligent Systems*, 23(2):15–18, 2008.

[21] Tom Schaul, Julian Togelius, and Jürgen Schmidhuber. Measuring intelligence through games. *arXiv preprint arXiv:1109.1314*, 2011.

[22] Yoones A Sekhavat. Behavior trees for computer games. *International Journal on Artificial Intelligence Tools*, 26(02):1730001, 2017.

[23] Bart Selman and Carla P Gomes. Hill-climbing search. volume 81, page 82. John Wiley & Sons, Ltd Chichester, 2006.

[24] Shunsuke Shinohara, Toshiaki Takano, Haruhiko Takase, Hiroharu Kawanaka, and Shinji Tsuruoka. Search algorithm with learning ability for mario ai–combination a* algorithm and q-learning. In *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 341–344. IEEE, 2012.

[25] Sen Song, Kenneth D Miller, and Larry F Abbott. *Competitive Hebbian learning through spike-timing-dependent synaptic plasticity*, volume 3. Nature Publishing Group, 2000.

[26] Csaba Szepesvári. *Algorithms for reinforcement learning*, volume 4. Morgan & Claypool Publishers, 2010.

[27] Georgios N Yannakakis and Julian Togelius. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335, 2014.

Appendix A
Supplemental Information