Advanced Artificial Intelligence Midterm Report:

Part 1: Implementation of Search Algorithms (75 Points)

Uninformed Search Algorithms (25 Points)

You will implement the following uninformed search strategies and analyze their performance in the Pac-Man game:
- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Iterative Deepening Search

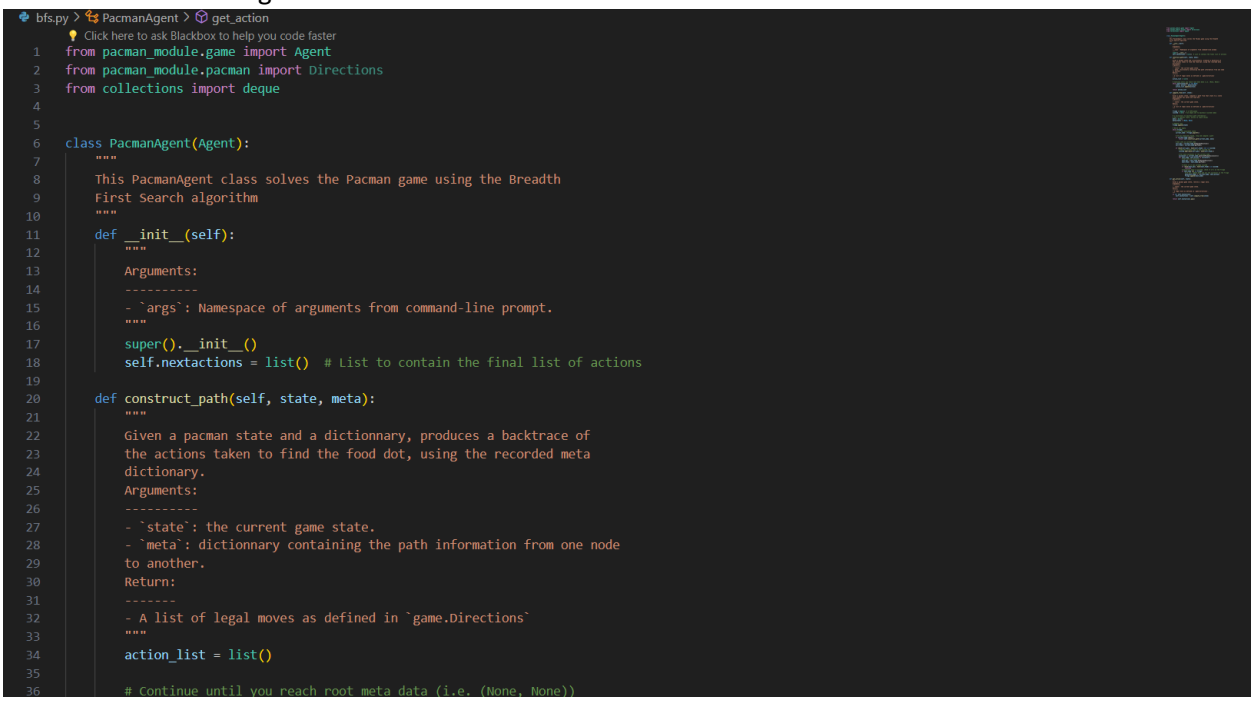
Informed Search Algorithms (35 Points)

You will enhance the Pac-Man agent by implementing the following informed search strategies:
- A* Search with a heuristic
- Greedy Best-First Search
- Develop a new or modify an existing heuristic to improve the performance of any of the above mentioned algorithms

Algorithm Analysis (15 Points)

Conduct a detailed comparison of all implemented algorithms based on execution time, nodes expanded, path cost, and optimality. Use graphical tools like matplotlib for visualization.

1) Breath First Search Algorithm:

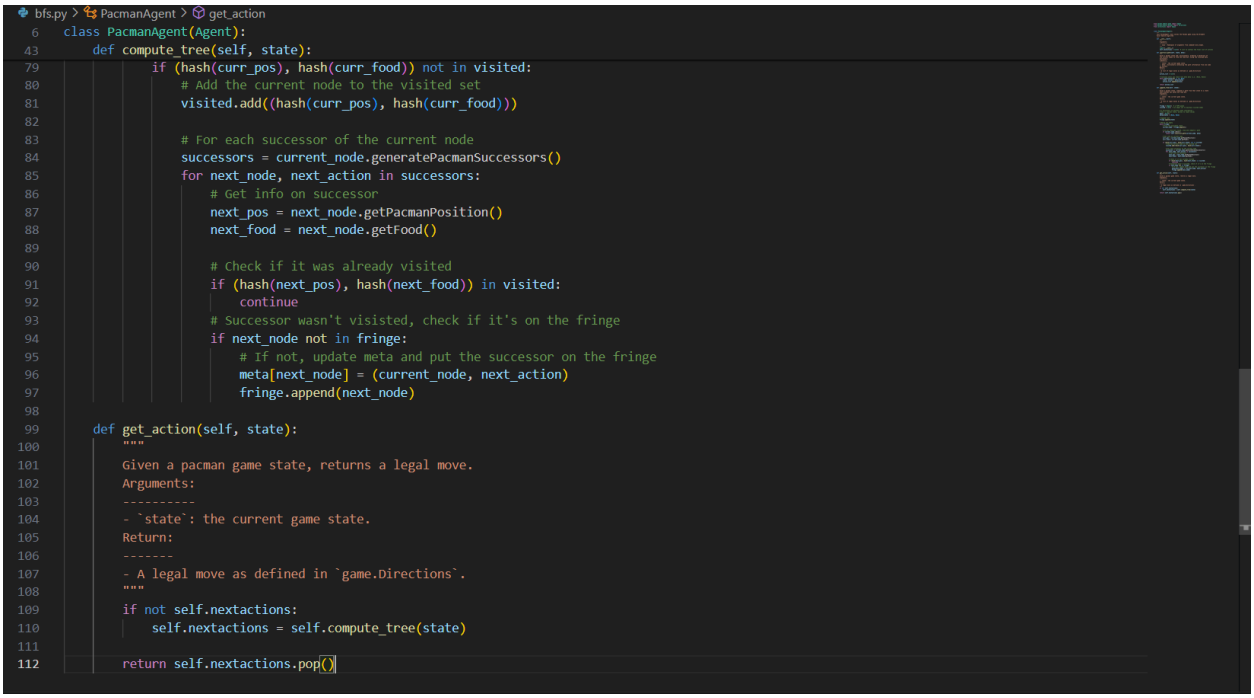


bfs.py > PacmanAgent > get_action

```
class PacmanAgent(Agent):
    def construct_path(self, state, meta):

        # Continue until you reach root meta data (i.e. (None, None))
        while meta[state][0] is not None:
            state, action = meta[state]
            action_list.append(action)

        return action_list

    def compute_tree(self, state):
        """
        Given a pacman state, computes a path from that state to a state
        where pacman has eaten one food dot.
        Arguments:
        ----------
        - `state`: the current game state.
        Return:
        -------
        - A list of legal moves as defined in `game.Directions`
        """

        fringe = deque()  # a FIFO queue
        visited = set()  # an empty set to maintain visited nodes

        # a dictionary to maintain path information :
        # key -> (parent state, action to reach child)
        meta = dict()
        meta[state] = (None, None)

        # Append root
        fringe.append(state)

        # While not empty
        while fringe:
            # Pick one available state
            current_node = fringe.popleft()
```

bfs.py > PacmanAgent > compute_tree

```
class PacmanAgent(Agent):
    def compute_tree(self, state):
            # Pick one available state
            current_node = fringe.popleft()

            # If all food dots found, stop and compute a path
            if current_node.isWin():
                return self.construct_path(current_node, meta)

            # Get info on current node
            curr_pos = current_node.getPacmanPosition()
            curr_food = current_node.getFood()

            if (hash(curr_pos), hash(curr_food)) not in visited:
                # Add the current node to the visited set
                visited.add((hash(curr_pos), hash(curr_food)))

                # For each successor of the current node
                successors = current_node.generatePacmanSuccessors()
                for next_node, next_action in successors:
                    # Get info on successor
                    next_pos = next_node.getPacmanPosition()
                    next_food = next_node.getFood()

                    # Check if it was already visited
                    if (hash(next_pos), hash(next_food)) in visited:
                        continue
                    # Successor wasn't visisted, check if it's on the fringe
                    if next_node not in fringe:
                        # If not, update meta and put the successor on the fringe
                        meta[next_node] = (current_node, next_action)
                        fringe.append(next_node)

    def get_action(self, state):
        """
        Given a pacman game state, returns a legal move.
        Arguments:
```



```
bfs.py > PacmanAgent > get_action
class PacmanAgent(Agent):
    def compute_tree(self, state):
            if (hash(curr_pos), hash(curr_food)) not in visited:
                # Add the current node to the visited set
                visited.add((hash(curr_pos), hash(curr_food)))

                # For each successor of the current node
                successors = current_node.generatePacmanSuccessors()
                for next_node, next_action in successors:
                    # Get info on successor
                    next_pos = next_node.getPacmanPosition()
                    next_food = next_node.getFood()

                    # Check if it was already visited
                    if (hash(next_pos), hash(next_food)) in visited:
                        continue
                    # Successor wasn't visisted, check if it's on the fringe
                    if next_node not in fringe:
                        # If not, update meta and put the successor on the fringe
                        meta[next_node] = (current_node, next_action)
                        fringe.append(next_node)

    def get_action(self, state):
        """
        Given a pacman game state, returns a legal move.
        Arguments:
        ----------
        - `state`: the current game state.
        Return:
        -------
        - A legal move as defined in `game.Directions`.
        """
        if not self.nextactions:
            self.nextactions = self.compute_tree(state)

        return self.nextactions.pop()
```

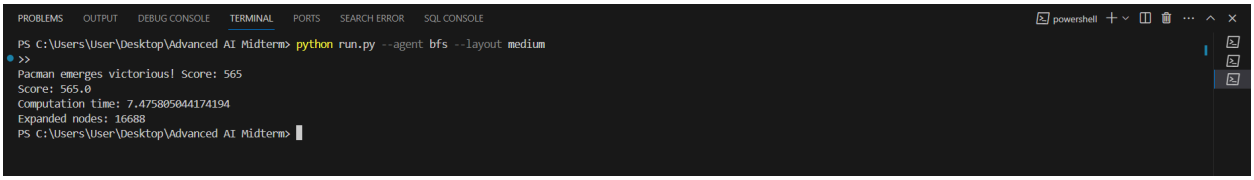
Explanation of the code for the Breadth First Search(BFS) Algorithm:

➔ This code is divided in 4 parts:

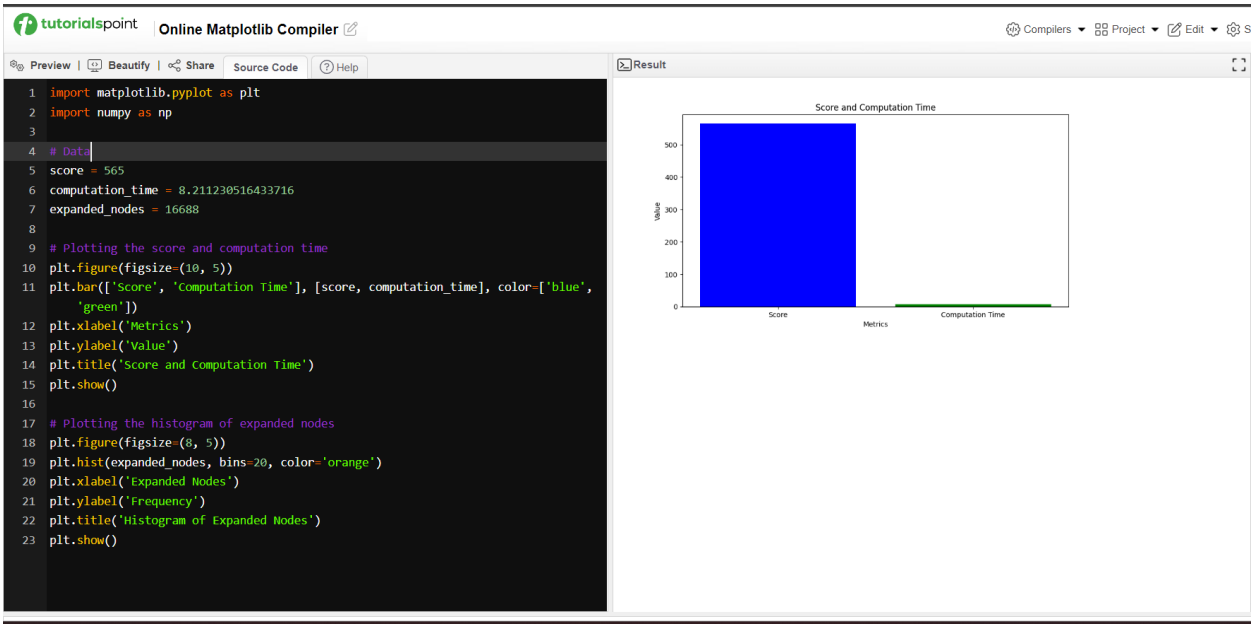
- The first part is the initialization where the PacmanAgent is defined, inheriting from the Agent class that is provided in the pacman_module, the file I downloaded from the assignment.
  - Inside this class, I have declared an __init__ method which initializes the agent. It initializes an empty list called nextactions which will contain the final list of actions.
- The second part is the construct_path method. This method takes a pacman state and a dictionary ‘meta’ as an input. Then it goes from the final state to the initial state using the information provided by the ‘meta’ dictionary. Lastly, it returns a list of legal moves that lead from the initial state to the final state.
- The third part which contains the logic for the bfs algorithm is explained this way:
  This method takes a pacman state as input and computes a path from that state to a state where Pacman has eaten one food dot. It uses a FIFO queue (fringe) to maintain the nodes to be explored and it initializes an empty set called visited to keep track of visited nodes. Then it initializes a dictionary meta to maintain path information. It starts with the initial state and adds it to the fringe.
  It iterates until the fringe is empty, exploring nodes one by one.
  For each node explored, it checks if it's a winning state (all food dots eaten). If so, it stops and computes the path using construct_path.
  If not a winning state, it generates successors of the current node, checks if they are visited, and adds them to the fringe if not visited.
  It updates the meta dictionary with the parent state and action to reach each successor node.

- Getting the Action (get_action method):
  - This method takes a pacman game state as input and returns a legal move for Pacman.
  - If nextactions list is empty, it computes the path using compute_tree method.
  - It returns and removes the last action from nextactions list, effectively following the computed path.

Score in this algorithm is:

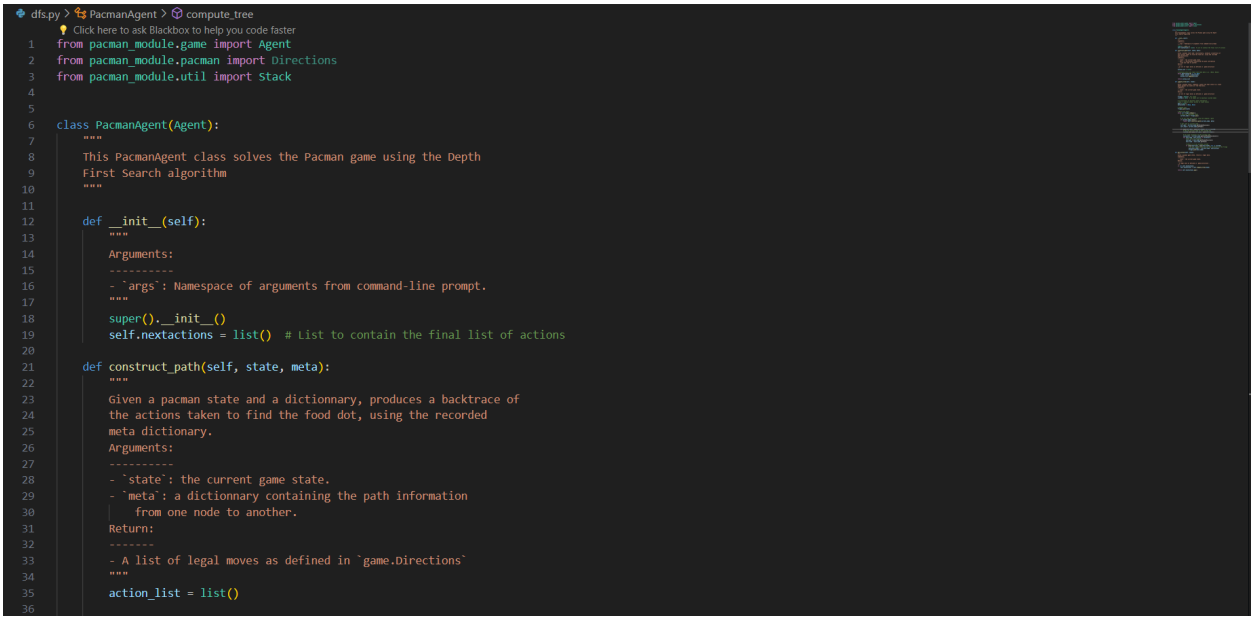


The matplotlib graph for this is shown below:



Overall, the bfs algorithm needs 8 seconds to compute and it searched over 16000 nodes to get the score of 565.

Depth First Search(DFS) Algorithm:



dfs.py > PacmanAgent > compute_tree

Click here to ask Blackbox to help you code faster

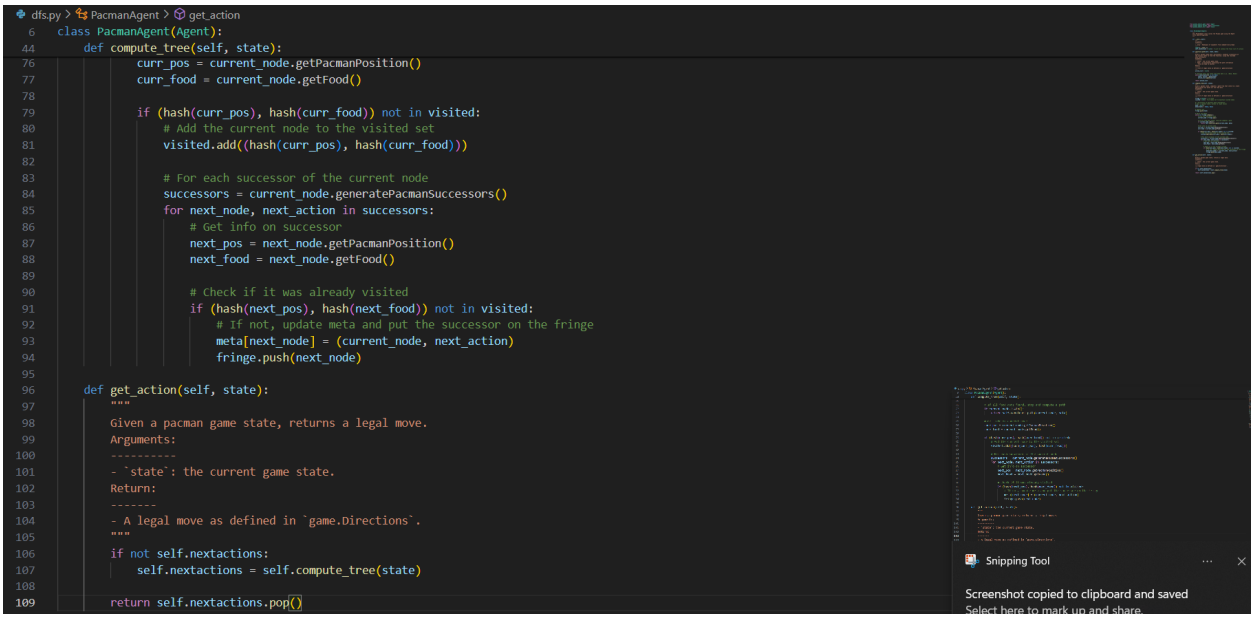
```python
from pacman_module.game import Agent
from pacman_module.pacman import Directions
from pacman_module.util import Stack


class PacmanAgent(Agent):
    """
    This PacmanAgent class solves the Pacman game using the Depth
    First Search algorithm
    """

    def __init__(self):
        """
        Arguments:
        ----------
        - `args`: Namespace of arguments from command-line prompt.
        """
        super().__init__()
        self.nextactions = list()  # List to contain the final list of actions

    def construct_path(self, state, meta):
        """
        Given a pacman state and a dictionnary, produces a backtrace of
        the actions taken to find the food dot, using the recorded
        meta dictionary.
        Arguments:
        ----------
        - `state`: the current game state.
        - `meta`: a dictionnary containing the path information
            from one node to another.
        Return:
        -------
        - A list of legal moves as defined in `game.Directions`
        """
        action_list = list()

```

dfs.py > PacmanAgent > compute_tree

```
6   class PacmanAgent(Agent):
21      def construct_path(self, state, meta):
35          action_list = list()
36
37          # Continue until you reach root meta data (i.e. (None, None))
38          while meta[state][0] is not None:
39              state, action = meta[state]
40              action_list.append(action)
41
42          return action_list
43
44      def compute_tree(self, state):
45          """
46          Given a pacman state, computes a path from that state to a state
47          where pacman has eaten all the food dots.
48          Arguments:
49          ----------
50          - `state`: the current game state.
51          Return:
52          -------
53          - A list of legal moves as defined in `game.Directions`
54          """
55          fringe = Stack()  # a stack
56          visited = set()  # an empty set to maintain visited nodes
57
58          # a dictionary to maintain path information :
59          # key -> (parent state, action to reach child)
60          meta = dict()
61          meta[state] = (None, None)
62
63          # Append root
64          fringe.push(state)
65
66          # While not empty
67          while not fringe.isEmpty():
68              # Pick one available state
69              current node = fringe.pop()
```

dfs.py > PacmanAgent > get_action

```
6   class PacmanAgent(Agent):
44      def compute_tree(self, state):
70
71              # If all food dots found, stop and compute a path
72              if current_node.isWin():
73                  return self.construct_path(current_node, meta)
74
75              # Get info on current node
76              curr_pos = current_node.getPacmanPosition()
77              curr_food = current_node.getFood()
78
79              if (hash(curr_pos), hash(curr_food)) not in visited:
80                  # Add the current node to the visited set
81                  visited.add((hash(curr_pos), hash(curr_food)))
82
83                  # For each successor of the current node
84                  successors = current_node.generatePacmanSuccessors()
85                  for next_node, next_action in successors:
86                      # Get info on successor
87                      next_pos = next_node.getPacmanPosition()
88                      next_food = next_node.getFood()
89
90                      # Check if it was already visited
91                      if (hash(next_pos), hash(next_food)) not in visited:
92                          # If not, update meta and put the successor on the fringe
93                          meta[next_node] = (current_node, next_action)
94                          fringe.push(next_node)
95
96      def get_action(self, state):
97          """
98          Given a pacman game state, returns a legal move.
99          Arguments:
100         ----------
101         - `state`: the current game state.
102         Return:
103         -------
104         - A legal move as defined in `game.Directions`.
```



- ➔ This code is divided in 4 parts:
  - The first part is the initialization where the PacmanAgent is defined, inheriting from the Agent class that is provided in the pacman_module, the file I downloaded from the assignment.
    - Inside this class, I have declared an __init__ method which initializes the agent. It initializes an empty list called nextactions which will contain the final list of actions.
- ➔ The second part is the construct_path method. This method takes a pacman state and a dictionary ‘meta’ as an input. Then it goes from the final state to the initial state using the information provided by the ‘meta’ dictionary. Lastly, it returns a list of legal moves that lead from the initial state to the final state.
- ➔ Computing the Tree (compute_tree method):
  - This method takes a pacman state as input and computes a path from that state to a state where Pacman has eaten all the food dots.
  - It uses a stack (fringe) to maintain the nodes to be explored. The key difference from BFS is that DFS uses a stack instead of a queue.
  - It initializes an empty set called visited to keep track of visited nodes.
  - It initializes a dictionary meta to maintain path information.
  - It starts with the initial state and pushes it onto the fringe.
  - It iterates until the fringe is empty, exploring nodes one by one in a depth-first manner.
  - For each node explored, it checks if it's a winning state (all food dots eaten). If so, it stops and computes the path using construct_path.
  - If not a winning state, it generates successors of the current node, checks if they are visited, and pushes them onto the fringe if not visited.
  - It updates the meta dictionary with the parent state and action to reach each successor node.
- ➔ Getting the Action (get_action method):

- This method takes a pacman game state as input and returns a legal move for Pacman.
- If nextactions list is empty, it computes the path using compute_tree method.
- It returns and removes the last action from nextactions list, effectively following the computed path.

Score of this algorithm is:

The matplotlib for the dfs algorithm is:

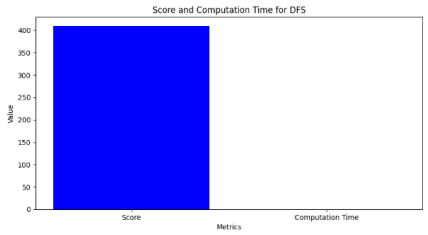
tutorialspoint | Online Matplotlib Compiler

Compilers | Project | Edit

Preview | Beautify | Share | Source Code | Help

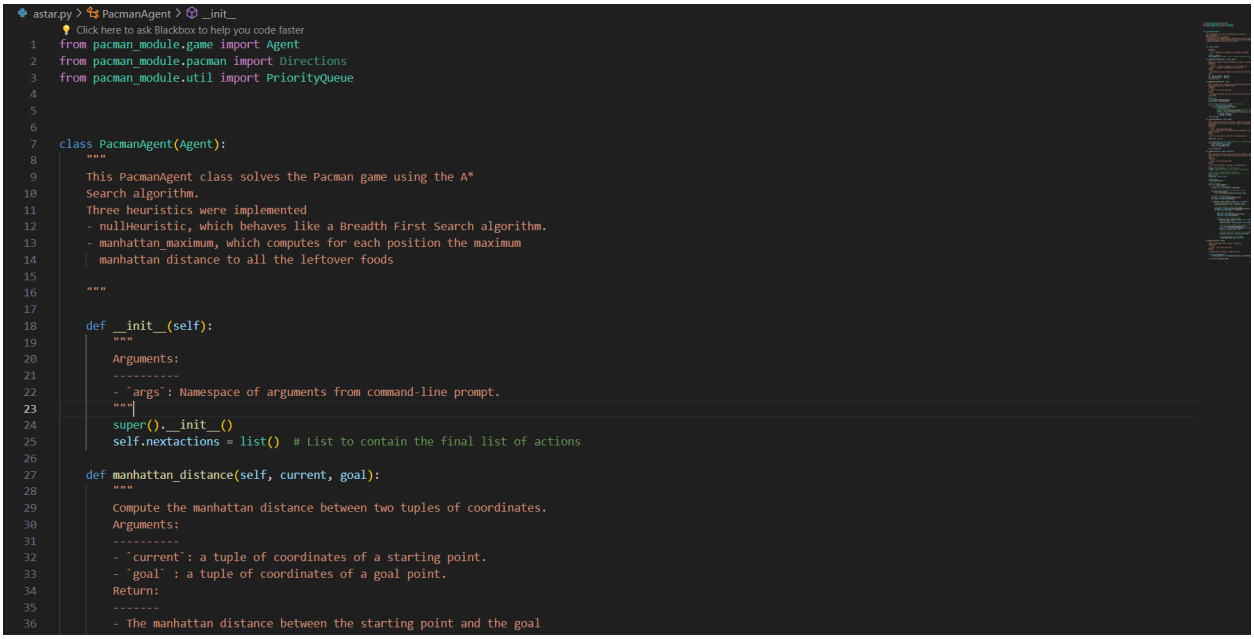
```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Data
5  score_dfs = 409
6  computation_time_dfs = 0.060864925384521484
7  expanded_nodes_dfs = 361
8
9  # Plotting the score and computation time for DFS
10 plt.figure(figsize=(10, 5))
11 plt.bar(['Score', 'Computation Time'], [score_dfs, computation_time_dfs], color
       =['blue', 'green'])
12 plt.xlabel('Metrics')
13 plt.ylabel('Value')
14 plt.title('Score and Computation Time for DFS')
15 plt.show()
16
17 # Plotting the histogram of expanded nodes for DFS
18 plt.figure(figsize=(8, 5))
19 plt.hist(expanded_nodes_dfs, bins=20, color='orange')
20 plt.xlabel('Expanded Nodes')
21 plt.ylabel('Frequency')
22 plt.title('Histogram of Expanded Nodes for DFS')
23 plt.show()
24
```

Result

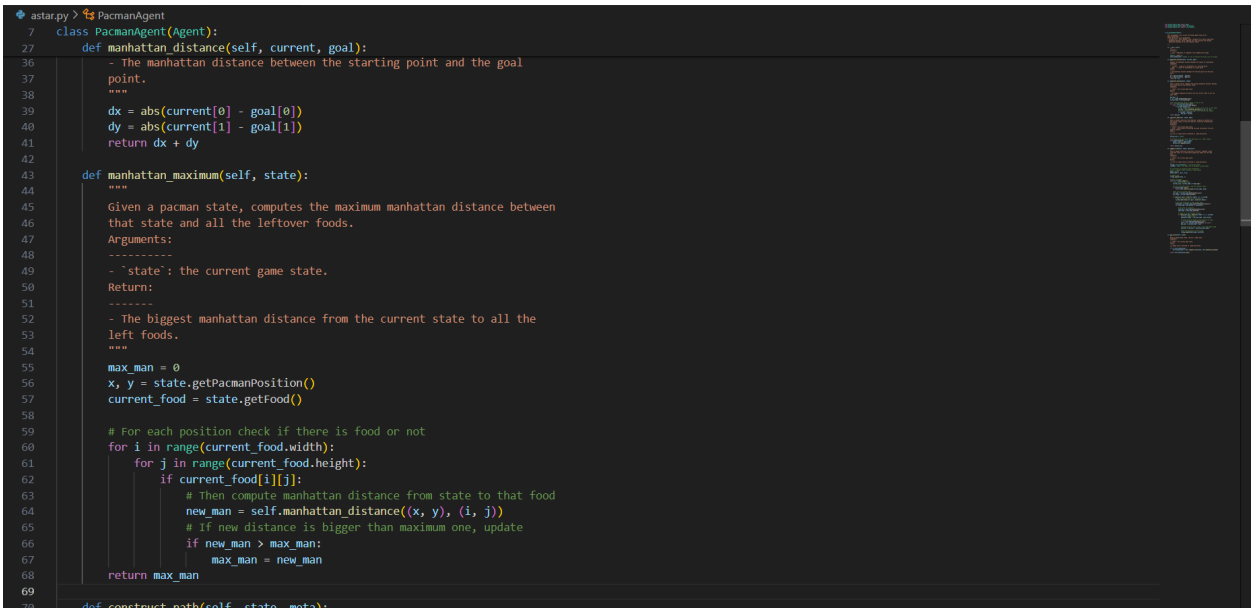


Overall the dfs algorithm is better than the bfs algorithm as it explores less nodes and the computation time is almost 0.06 seconds, meaning less than bfs but the score is less than the bfs algorithm by almost 100.

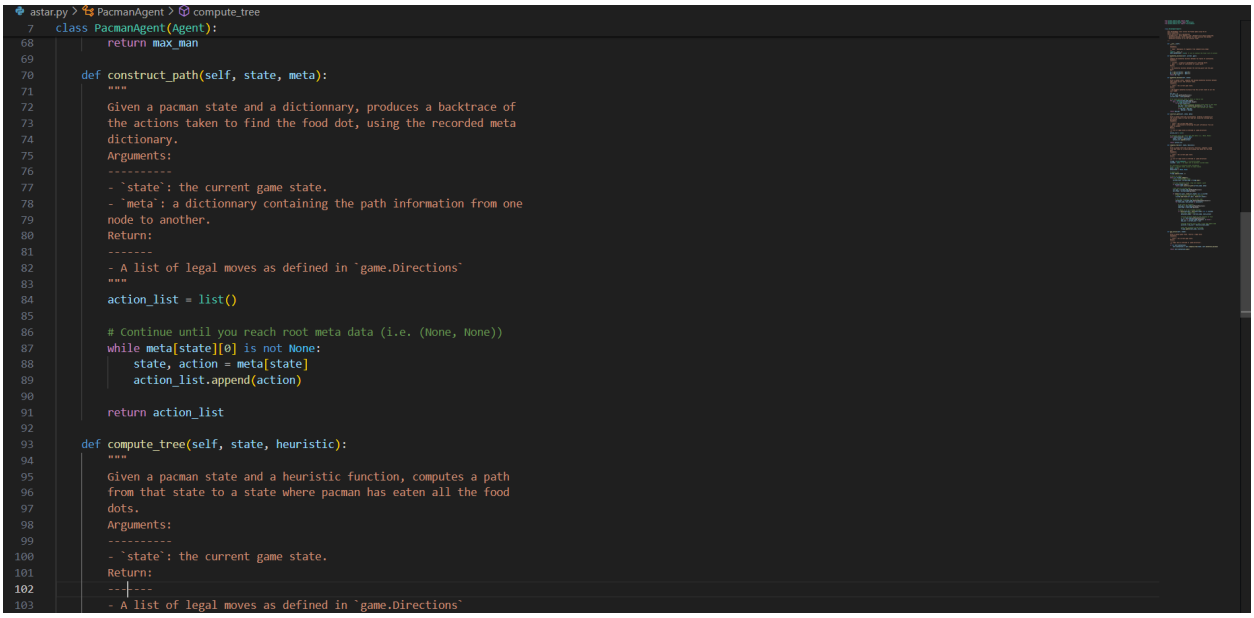
A* algorithm:



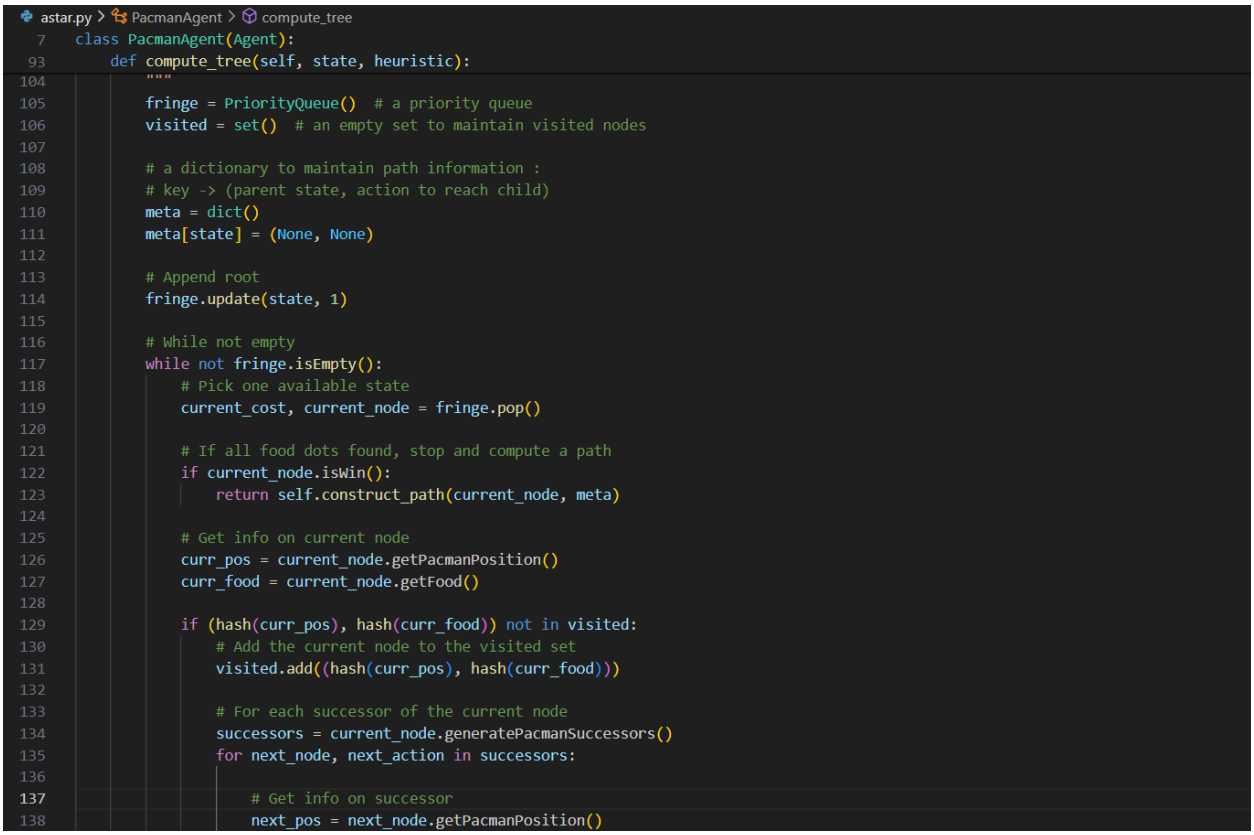
```
astar.py > PacmanAgent > __init__
Click here to ask Blackbox to help you code faster
from pacman_module.game import Agent
from pacman_module.pacman import Directions
from pacman_module.util import PriorityQueue



class PacmanAgent(Agent):
    """
    This PacmanAgent class solves the Pacman game using the A*
    Search algorithm.
    Three heuristics were implemented
    - nullHeuristic, which behaves like a Breadth First Search algorithm.
    - manhattan_maximum, which computes for each position the maximum
      manhattan distance to all the leftover foods

    """

    def __init__(self):
        """
        Arguments:
        ----------
        - `args`: Namespace of arguments from command-line prompt.
        """
        super().__init__()
        self.nextactions = list()  # List to contain the final list of actions

    def manhattan_distance(self, current, goal):
        """
        Compute the manhattan distance between two tuples of coordinates.
        Arguments:
        ----------
        - `current`: a tuple of coordinates of a starting point.
        - `goal` : a tuple of coordinates of a goal point.
        Return:
        -------
        - The manhattan distance between the starting point and the goal
```



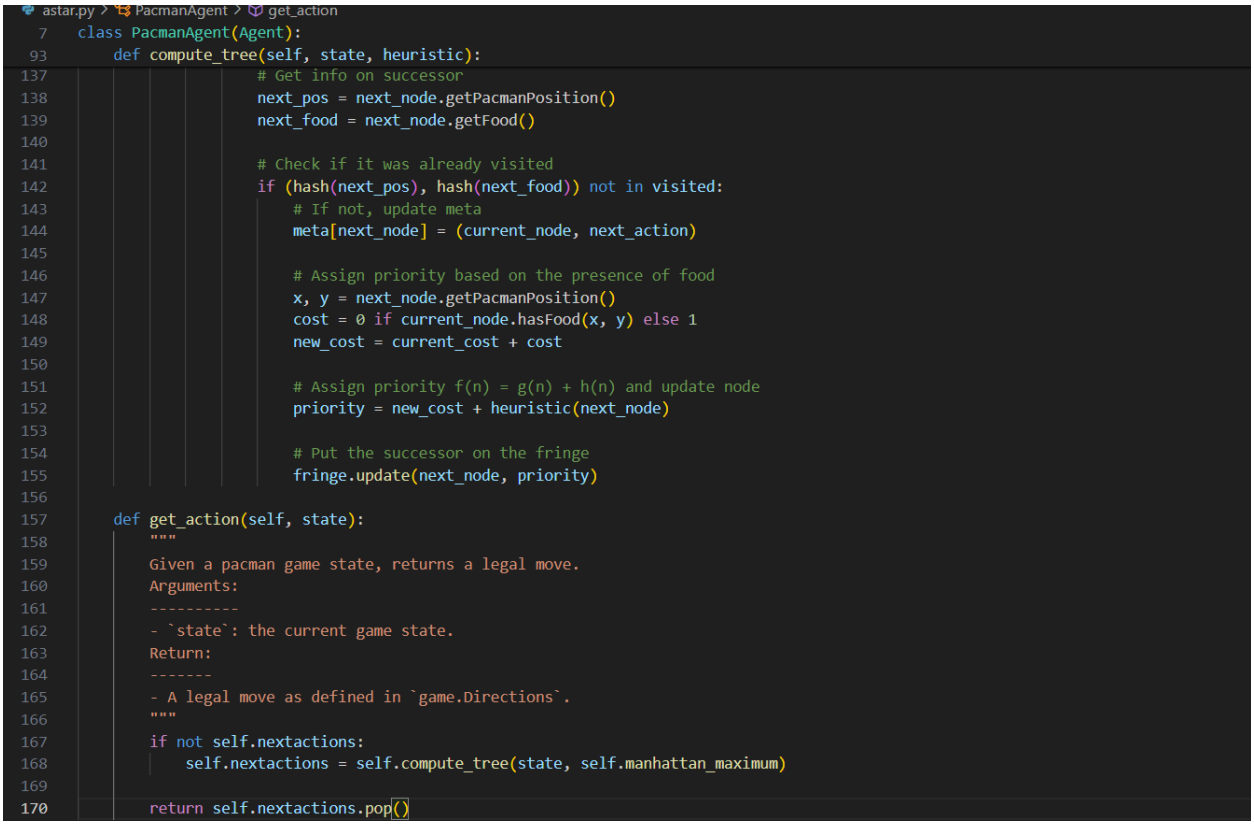
```
astar.py > PacmanAgent
class PacmanAgent(Agent):
    def manhattan_distance(self, current, goal):
        - The manhattan distance between the starting point and the goal
        point.
        """
        dx = abs(current[0] - goal[0])
        dy = abs(current[1] - goal[1])
        return dx + dy

    def manhattan_maximum(self, state):
        """
        Given a pacman state, computes the maximum manhattan distance between
        that state and all the leftover foods.
        Arguments:
        ----------
        - `state`: the current game state.
        Return:
        -------
        - The biggest manhattan distance from the current state to all the
        left foods.
        """
        max_man = 0
        x, y = state.getPacmanPosition()
        current_food = state.getFood()

        # For each position check if there is food or not
        for i in range(current_food.width):
            for j in range(current_food.height):
                if current_food[i][j]:
                    # Then compute manhattan distance from state to that food
                    new_man = self.manhattan_distance((x, y), (i, j))
                    # If new distance is bigger than maximum one, update
                    if new_man > max_man:
                        max_man = new_man
        return max_man

    def construct_path(self, state, meta):
```



```
astar.py > PacmanAgent > compute_tree
class PacmanAgent(Agent):
        return max_man

    def construct_path(self, state, meta):
        """
        Given a pacman state and a dictionnary, produces a backtrace of
        the actions taken to find the food dot, using the recorded meta
        dictionary.
        Arguments:
        ----------
        - `state`: the current game state.
        - `meta`: a dictionnary containing the path information from one
        node to another.
        Return:
        -------
        - A list of legal moves as defined in `game.Directions`
        """
        action_list = list()

        # Continue until you reach root meta data (i.e. (None, None))
        while meta[state][0] is not None:
            state, action = meta[state]
            action_list.append(action)

        return action_list

    def compute_tree(self, state, heuristic):
        """
        Given a pacman state and a heuristic function, computes a path
        from that state to a state where pacman has eaten all the food
        dots.
        Arguments:
        ----------
        - `state`: the current game state.
        Return:
        -------
        - A list of legal moves as defined in `game.Directions`
```



```
astar.py > PacmanAgent > compute_tree
class PacmanAgent(Agent):
    def compute_tree(self, state, heuristic):
        """
        fringe = PriorityQueue()  # a priority queue
        visited = set()  # an empty set to maintain visited nodes

        # a dictionary to maintain path information :
        # key -> (parent state, action to reach child)
        meta = dict()
        meta[state] = (None, None)

        # Append root
        fringe.update(state, 1)

        # While not empty
        while not fringe.isEmpty():
            # Pick one available state
            current_cost, current_node = fringe.pop()

            # If all food dots found, stop and compute a path
            if current_node.isWin():
                return self.construct_path(current_node, meta)

            # Get info on current node
            curr_pos = current_node.getPacmanPosition()
            curr_food = current_node.getFood()

            if (hash(curr_pos), hash(curr_food)) not in visited:
                # Add the current node to the visited set
                visited.add((hash(curr_pos), hash(curr_food)))

                # For each successor of the current node
                successors = current_node.generatePacmanSuccessors()
                for next_node, next_action in successors:

                    # Get info on successor
                    next_pos = next_node.getPacmanPosition()
```



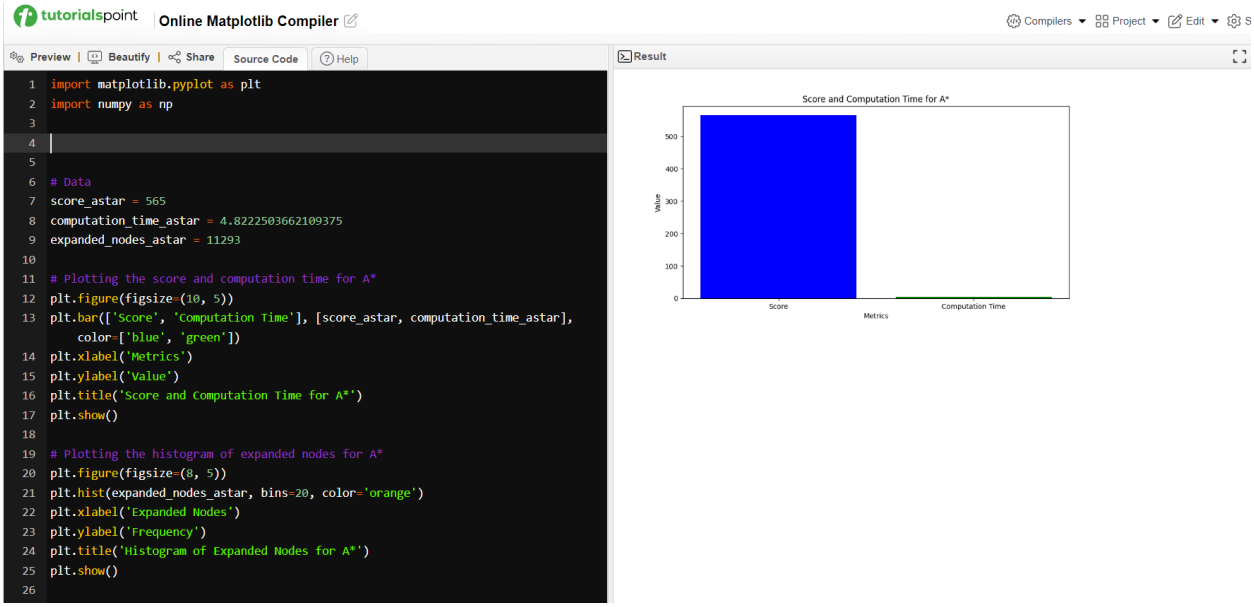
```
astar.py > PacmanAgent > get_action
  7  class PacmanAgent(Agent):
 93      def compute_tree(self, state, heuristic):
137                  # Get info on successor
138                  next_pos = next_node.getPacmanPosition()
139                  next_food = next_node.getFood()
140
141                  # Check if it was already visited
142                  if (hash(next_pos), hash(next_food)) not in visited:
143                      # If not, update meta
144                      meta[next_node] = (current_node, next_action)
145
146                      # Assign priority based on the presence of food
147                      x, y = next_node.getPacmanPosition()
148                      cost = 0 if current_node.hasFood(x, y) else 1
149                      new_cost = current_cost + cost
150
151                      # Assign priority f(n) = g(n) + h(n) and update node
152                      priority = new_cost + heuristic(next_node)
153
154                      # Put the successor on the fringe
155                      fringe.update(next_node, priority)
156
157      def get_action(self, state):
158          """
159          Given a pacman game state, returns a legal move.
160          Arguments:
161          ----------
162          - `state`: the current game state.
163          Return:
164          -------
165          - A legal move as defined in `game.Directions`.
166          """
167          if not self.nextactions:
168              self.nextactions = self.compute_tree(state, self.manhattan_maximum)
169
170          return self.nextactions.pop()
```

1. Initialization:
    - The PacmanAgent class is defined, inheriting from the Agent class provided by the pacman_module.
    - Inside the class, there's an __init__ method that initializes the agent. It initializes an empty list called nextactions, which will contain the final list of actions.
2. Heuristics:
    - Two heuristics are implemented:
        - nullHeuristic: This behaves like Breadth First Search (BFS).
        - manhattan_maximum: This computes, for each position, the maximum Manhattan distance to all the leftover foods.
3. Manhattan Distance Calculation (manhattan_distance method):
    - This method computes the Manhattan distance between two tuples of coordinates.
4. Maximum Manhattan Distance Calculation (manhattan_maximum method):
    - This method computes the maximum Manhattan distance between the current state and all the leftover foods.
    - It iterates through all the positions to check if there is food or not.

- For each position with food, it computes the Manhattan distance from the current state to that food.
- It updates the maximum distance if the new distance is greater than the current maximum.

5. Constructing the Path (construct_path method):
    - This method takes a Pacman state and a dictionary meta as input.
    - It backtracks from the final state to the initial state using the information stored in the meta dictionary.
    - It returns a list of legal moves that lead from the initial state to the final state.
6. Computing the Tree (compute_tree method):
    - This method takes a Pacman state and a heuristic function as input and computes a path from that state to a state where Pacman has eaten all the food dots.
    - It uses a priority queue (fringe) to maintain the nodes to be explored, prioritizing nodes with lower costs.
    - It initializes an empty set called visited to keep track of visited nodes.
    - It initializes a dictionary meta to maintain path information.
    - It starts with the initial state and adds it to the fringe.
    - It iterates until the fringe is empty, exploring nodes with the lowest estimated total cost.
    - For each node explored, it checks if it's a winning state (all food dots eaten). If so, it stops and computes the path using construct_path.
    - It generates successors of the current node, updates the meta dictionary, calculates the priority based on the presence of food, and updates the node's priority in the fringe.
7. Getting the Action (get_action method):
    - This method takes a Pacman game state as input and returns a legal move for Pacman.
    - If nextactions list is empty, it computes the path using the compute_tree method with the manhattan_maximum heuristic.
    - It returns and removes the last action from nextactions list, effectively following the computed path.

The matplotlib for this algorithm is this:



And the score is:

```
PS C:\Users\User\Desktop\Advanced AI Midterm> python run.py --agent astar --layout medium
>>
Pacman emerges victorious! Score: 565
Score: 565.0
Computation time: 4.8222503662109375
Expanded nodes: 11293
```

Overall, A* algorithm is better than the dfs and bfs algorithm as it explores less nodes and the computation time is less than the bfs even though their score is the same.

IDS Algorithm:

```
from pacman_module.game import Agent, Directions
from pacman_module.util import Stack


def key(state):
    """Returns a key that uniquely identifies a Pacman game state.

    Arguments:
        state: a game state. See API or class pacman.GameState.

    Returns:
        A hashable key tuple.
    """
    return state.getPacmanPosition(),


class PacmanAgent(Agent):
    """Pacman agent based on iterative deepening depth-first search (IDS-DFS)."""

    def _init_(self):
        super()._init_()
        self.moves = None

    def get_action(self, state):
        """Given a Pacman game state, returns a legal move.

        Arguments:
            state: a game state. See API or class pacman.GameState.

        Return:
            A legal move as defined in game.Directions.
        """
        if self.moves is None:
            self.moves = self.iterative_deepening_search(state)

        if self.moves:
            return self.moves.pop(0)
        else:
            return Directions.STOP

    def iterative_deepening_search(self, state):
```

```
        """Iterative Deepening Depth-First Search (IDS-DFS).

        Arguments:
            state: a game state. See API or class pacman.GameState.

        Returns:
            A list of legal moves.
        """
        depth_limit = 0

        while True:
            result = self.dfs(state, depth_limit)
            if result is not None:
                return result
            depth_limit += 1

    def dfs(self, state, depth_limit):
        """Depth-First Search (DFS) with depth limit.

        Arguments:
            state: a game state. See API or class pacman.GameState.
            depth_limit: maximum depth to explore.

        Returns:
            A list of legal moves.
        """
        fringe = Stack()
        fringe.push((state, []))  # Initialize an empty path
        closed = set()

        while not fringe.isEmpty():
            current, path = fringe.pop()

            if current.isWin():
                return path

            current_key = key(current)

            if current_key in closed:
                continue
            else:
                closed.add(current_key)

            if len(path) < depth_limit:
                for successor, action in current.generatePacmanSuccessors():
```

```
                    fringe.push((successor, path + [action]))

        return None


if __name__ == '_main_':
    # You can test the IDS-DFS agent here if needed
    pass
```

1. Initialization:
    - The PacmanAgent class is defined, inheriting from the Agent class provided by the pacman_module.
    - Inside the class, there's an __init__ method that initializes the agent. It sets moves to None, which will hold the sequence of moves determined by the IDS-DFS algorithm.
2. Key Function (key function):
    - This function generates a unique key for a Pacman game state. It's used for checking whether a state has been visited or not during the search.
3. Get Action (get_action method):
    - This method takes a Pacman game state as input and returns the next legal move for Pacman.
    - If moves is None, it initializes it by calling the iterative_deepening_search method.
    - It returns and removes the first move from moves, effectively determining Pacman's next action.
4. Iterative Deepening Search (iterative_deepening_search method):
    - This method performs Iterative Deepening Depth-First Search (IDS-DFS) until a solution is found.
    - It starts with a depth limit of 0 and increases it iteratively until a solution is found.
    - It calls the dfs method with the current depth limit.
5. Depth-First Search (dfs method):
    - This method performs Depth-First Search (DFS) with a depth limit.
    - It uses a stack (fringe) to maintain the nodes to be explored, along with the path from the initial state.
    - It initializes an empty set called closed to keep track of visited states.
    - It continues exploring nodes until the fringe is empty or the depth limit is reached.

- If a winning state is found, it returns the path leading to it.
- It generates successors of the current state and adds them to the fringe along with the extended path, as long as the depth limit has not been exceeded.
- It returns None if no solution is found within the depth limit.

GBFS Algorithm:

```
from pacman_module.game import Agent
from pacman_module.game import PacmanGame
from pacman_module.pacman import Directions
from pacman_module.util import PriorityQueue


class PacmanAgent(Agent):
    """
    This PacmanAgent class solves the Pacman game using the Greedy Best-First
Search algorithm
    """

    def __init__(self):
        """
        Arguments:
        ----------
        - `args`: Namespace of arguments from command-line prompt.
        """
        super().__init__()
        self.nextactions = list()  # List to contain the final list of actions

    def construct_path(self, state, meta):
        """
        Given a pacman state and a dictionnary, produces a backtrace of
        the actions taken to find the food dot, using the recorded
        meta dictionary.

        Arguments:
        ----------
        - `state`: the current game state.
        - `meta`: a dictionnary containing the path information
                  from one node to another.

        Return:
        -------
        - A list of legal moves as defined in `game.Directions`
        """
        action_list = list()

        # Continue until you reach root meta data (i.e. (None, None))
        while meta[state][0] is not None:
            state, action = meta[state]
```

```
            action_list.append(action)

        return action_list

    def getHeuristic(self, state):
        """
        Calculates a basic heuristic: manhattan distance to closest food
        Arguments:
            state: The current pacman game state.
        Returns:
            The heuristic score.
        """
        food_list = state.getFood().asList()
        pacman_pos = state.getPacmanPosition()
        min_distance = float('inf')
        for food in food_list:
            distance = abs(food[0] - pacman_pos[0]) + abs(food[1] -
pacman_pos[1])
            min_distance = min(min_distance, distance)
        return min_distance

    def greedy_best_first_search(self, state):
        """
        Implements the Greedy Best-First Search algorithm for Pacman.

        Arguments:
            state: The initial state of the Pacman game.

        Returns:
            A list of actions leading to a goal state or None if not found.
        """

        # Initialize frontier (priority queue based on heuristic)
        frontier = PriorityQueue()
        frontier.push((state, self.getHeuristic(state)), 0)

        # Set to store explored states
        explored = set()

        # Keep track of parent states for path reconstruction (similar to DFS)
        meta = dict()
        meta[state] = (None, None)

        # Loop until frontier is empty
        while not frontier.isEmpty():
```

```
            # Get state with highest heuristic from frontier (greedy selection)
            current_state, _ = frontier.pop()

            # Check if goal state reached (all food eaten)
            if state.isWin():
                return self.construct_path(current_state, meta)

            # Mark current state as explored
            explored.add((hash(current_state.getPacmanPosition()),
hash(current_state.getFood())))

            # Generate successors for the current state
            successors = state.generatePacmanSuccessors()

            for next_state, action in successors:
                # Skip explored states
                if (hash(next_state.getPacmanPosition()),
hash(next_state.getFood())) not in explored:
                    # Calculate heuristic for the successor
                    h_value = self.getHeuristic(next_state)
                    # Add successor to frontier with priority based on heuristic
                    frontier.push((next_state, h_value), h_value)
                    # Update meta data to track parent for path reconstruction
                    meta[next_state] = (current_state, action)

        # No solution found within explored states
        return None

    def get_action(self, state):
        """
        Given a pacman game state, returns a legal move using Greedy Best-First
Search.

        Arguments:
        ----------
        - `state`: the current game state.

        Return:
        -------
        - A legal move as defined in `game.Directions`.
        """
        if not self.nextactions:
            # Perform Greedy Best-First Search and store path (actions)
            path = self.greedy_best_first_search(state)
            if path:
```

```
            self.nextactions = path.copy()  # Avoid modifying original path

        # Return the next action from the stored path
        return self.nextactions.pop(0)
```

1. Initialization:
    - The PacmanAgent class is defined, inheriting from the Agent class provided by the pacman_module.
    - Inside the class, there's an __init__ method that initializes the agent. It sets nextactions to an empty list, which will contain the sequence of actions determined by the Greedy Best-First Search algorithm.
2. Constructing the Path (construct_path method):
    - This method takes a Pacman game state and a dictionary meta as input.
    - It backtracks from the final state to the initial state using the information stored in the meta dictionary.
    - It returns a list of legal moves that lead from the initial state to the final state.
3. Heuristic Calculation (getHeuristic method):
    - This method calculates a basic heuristic: the Manhattan distance to the closest food dot.
    - It iterates over all the food dots and calculates the Manhattan distance between Pacman's position and each food dot.
    - It returns the minimum distance found.
4. Greedy Best-First Search (greedy_best_first_search method):
    - This method implements the Greedy Best-First Search algorithm for Pacman.
    - It initializes a priority queue (frontier) based on the heuristic values.
    - It initializes an empty set (explored) to store explored states.
    - It initializes a dictionary meta to maintain path information.
    - It iteratively explores states until the goal state (all food eaten) is reached or no solution is found.
    - It expands the state with the highest heuristic value from the frontier and checks if it's a goal state.
    - It marks the current state as explored and generates successors for it.
    - It adds successors to the frontier if they have not been explored and updates the meta data to track the path.

5. Get Action (get_action method):
    - This method takes a Pacman game state as input and returns the next legal move for Pacman using Greedy Best-First Search.
    - If nextactions is empty, it performs Greedy Best-First Search to determine the path and stores it.
    - It returns and removes the first action from nextactions, effectively determining Pacman's next move.

Part 2: Advanced Application and Analysis (20 Points)

Integration of Advanced Pathfinding Algorithms (20 Points)

Investigate two advanced pathfinding algorithms and propose how they could be adapted for this project. Include a detailed theoretical discussion on potential benefits and limitations, supported by pseudocode.

First algorithm is:

Dijkstra's Algorithm:

Theory: Dijkstra's algorithm is a versatile and efficient algorithm for finding the shortest paths between nodes in a graph. It operates by iteratively selecting the node with the lowest cost from a set of unvisited nodes, updating the costs of its neighbors, and marking it as visited. This process continues until all nodes have been visited or the destination node is reached.

Benefits:

Guaranteed to find the shortest path in weighted graphs with non-negative edge weights.

Can be adapted to find paths to multiple destinations simultaneously.

Limitations:

Inefficient for graphs with negative edge weights.

Requires additional data structures to efficiently update node costs.

Pseudocode:

```
function dijkstra(graph, start):
  Initialize:
    - Set of unvisited nodes
    - Map of distances with infinite initial values
    - Set of visited nodes
  Add start node to unvisited set with distance 0
  Loop until unvisited set is empty:
    - Select node with lowest distance from unvisited set
    - Mark node as visited
    - Update distances of neighbors
  Return distances map
```

Second Algorithm:

*A Algorithm with Custom Heuristic:**

- **Theory:** A* is an informed search algorithm that combines the advantages of uniform cost search (like Dijkstra's) with the efficiency of heuristic-based approaches. It evaluates nodes by combining the cost to reach the node from the start with an estimate of the cost to reach the goal from the node, often denoted as f(n)=g(n)+h(n, where $g(n)$ is the cost from the start to the node and $h(n)$ is the heuristic estimate of the cost from the node to the goal.
- **Benefits:**
    - Efficient and optimal if the heuristic is admissible (never overestimates the true cost).
    - Can be tailored with different heuristics for specific problem domains.
- **Limitations:**
    - Requires designing and implementing a suitable heuristic for the problem.
    - Heuristic quality directly impacts the algorithm's performance and optimality.

Pseudocode for this algorithm:

```
function a*(graph, start, goal, heuristic):
  Initialize:
    - Set of open nodes (priority queue)
    - Map of costs with infinite initial values
    - Set of closed nodes
  Add start node to open set with cost 0
  Loop until open set is empty:
    - Select node with lowest cost from open set
    - If node is goal, reconstruct path and return
    - Mark node as closed
    - Update costs of neighbors
  Return failure (if goal not found)
```