Take home End semester LaTeX Report

May 24, 2024

ROHIT.B
ME23B064

Introduction

This is a latex report on take home end semester examination. I am going to give a brief report and the problems given to me and how I approached them.

Path planning (RRT algorithm):

Path planning is a crucial aspect of autonomous robotics systems, enabling robots to navigate complex environments efficiently and safely. RRT is a popular algorithm used for path planning when the map of the environment is known. In this report, we present the implementation and analysis of the RRT algorithm, along with enhancements to address its limitations.

0.0.1 Tasks to be done:

- we have to write a program that plots a rectangle of some arbitrary width and height, which acts as the walls of the environment, and place 3-4 obstacles, which are rectangles of around one-tenth the size of the bounding wall.
- with that we have to perform RRT algorithm, perform smoothing on it, find a greedy approach from start to the goal point.
- after this we have to run the stimulation 1000 times for each variant and plot average number of nodes needed to reach the goal.

Environment representation:

We begin by defining an `Environment` class to represent the workspace. This class includes methods to add obstacles and visualize the environment using matplotlib (this module is used for plotting and graphs).

```
class Environment:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.obstacles = []
    def add_obstacle(self, obstacle):
        self.obstacles.append(obstacle)

    def plot():
```

- The Environment class represents the workspace in which the robot operates. It initializes with the width and height of the environment.The add obstacle method adds obstacles to the environment. The plot method visualizes the environment, including obstacles, start and end points, RRT nodes, and paths.

0.1 RRT Algorithm

The RRT algorithm is implemented in the `RRT` class. It iteratively grows a tree from the start point towards the goal by randomly sampling points and expanding towards them. The nearest node in the tree is connected to the new point, and collision checking ensures that the path avoids obstacles.

```
class RRT:
def __init__(self, start, goal, environment, step_size=0.25,
max_iter=2000, **kwargs):
...
def get_random_point(self, spacing=0):
...
def get_nearest_node(self, point):
...
def steer(self, from_node, to_point):
...
def path_to_goal(self, node):
...
def run(self)
```

- The RRT class implements the Rapidly-exploring Random Trees algorithm for path planning. It initializes with the start and goal points, environment, step size, and maximum number of iterations. The get_random_point method generates a random point within the environment, optionally considering spacing away from obstacles. The get_nearest_node method finds the nearest node in the RRT tree to a given point. The steer method moves from one node towards another, ensuring that the step size is not exceeded. The path_to_goal method traces back from a node to the root to construct the path to the goal. The run method executes the RRT algorithm by iteratively expanding the tree towards the goal while avoiding obstacles.

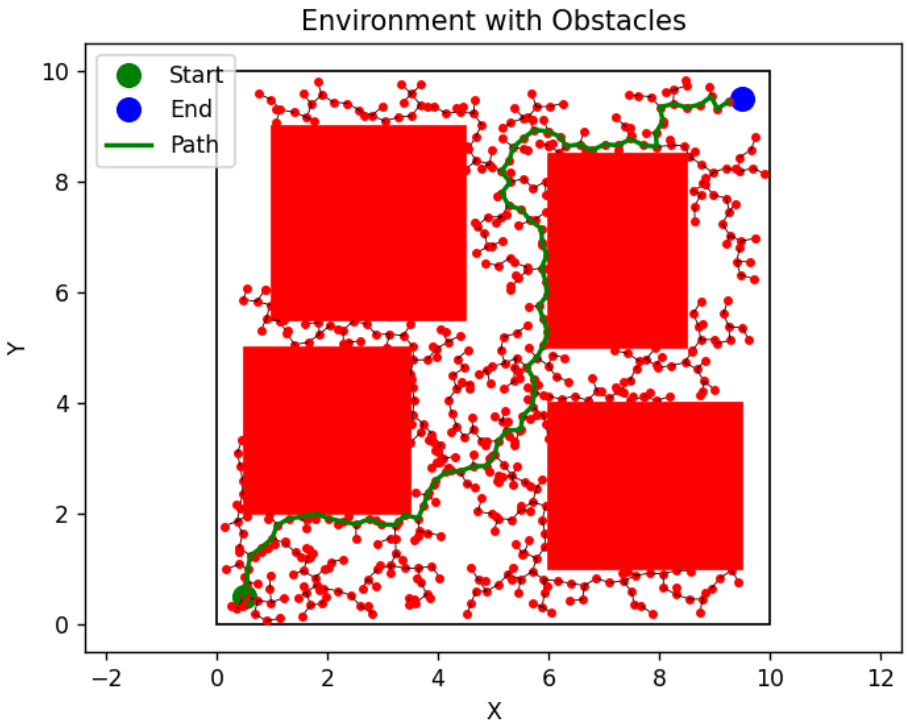


Figure 1: basic

0.2 Trajectory Smoothing

To address the non-smoothness of the RRT path, we employ the Ramer-Douglas-Peucker algorithm. This algorithm recursively simplifies the path by removing intermediate points while preserving its shape.

```
def ramer_douglas_peucker(points, epsilon, environment):
    ...
```

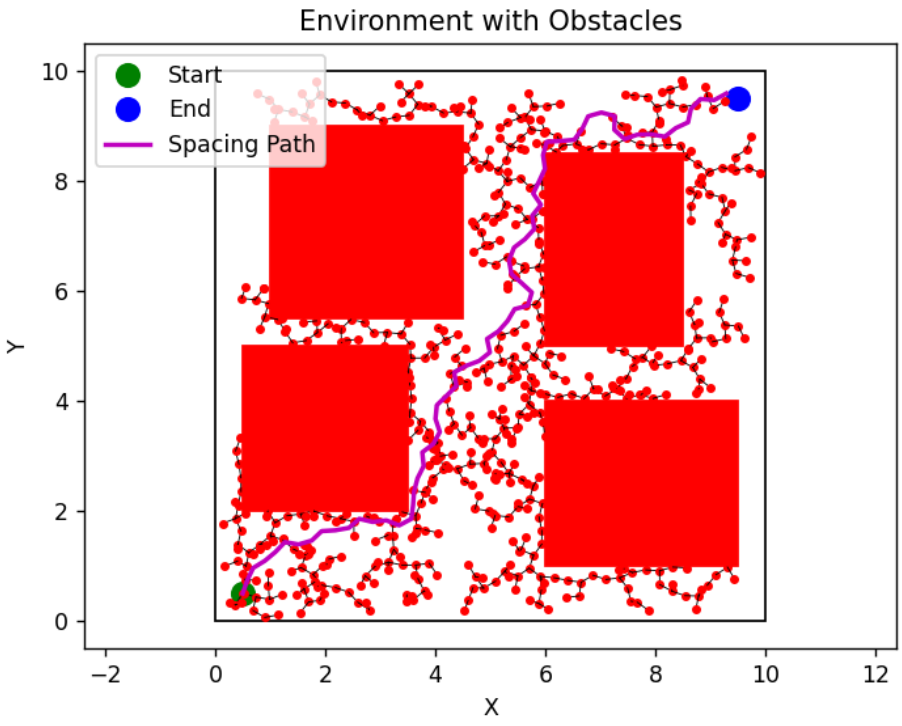
- The ramer_douglas_peucker function implements the Ramer-Douglas-Peucker algorithm for trajectory smoothing.
- It recursively simplifies a path by removing intermediate points while preserving its shape.
- Collision checking ensures that the simplified path avoids obstacles.

0.3 Spacing Consideration

We enhance the RRT algorithm to consider spacing away from obstacles. Random points are more likely to be sampled at a distance from the edges of obstacles to ensure safe navigation.

```
def rrt_with_spacing(start_point, end_point, env, step_size=0.25,
max_iter=2000, spacing=1):
    ...
```

- The rrt_with_spacing function enhances the RRT algorithm to consider spacing away from obstacles.
- It generates a path with points spaced away from obstacle edges to ensure safe navigation.



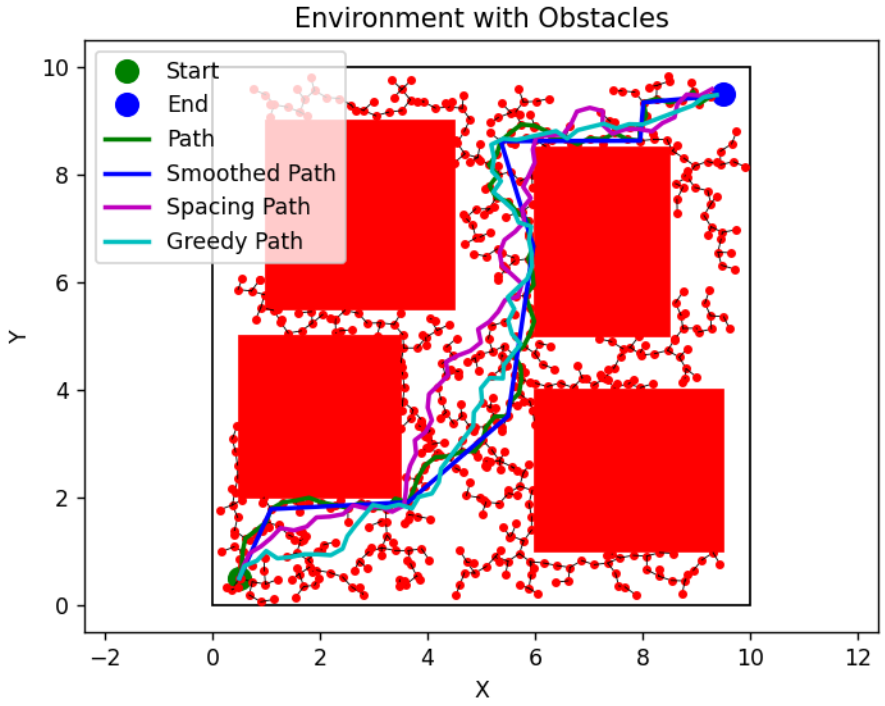
0.4 Greedy Approach

We incorporate a greedy approach where points closer to the goal are more likely to be chosen during sampling. This enhances the convergence speed towards the goal.

```
def rrt_with_greedy(start_point, end_point, env, step_size=0.25,
max_iter=2000):
    ...
```

– The rrt_with_greedy function incorporates a greedy approach in the RRT algorithm.

– It biases point selection towards the goal, improving convergence speed.

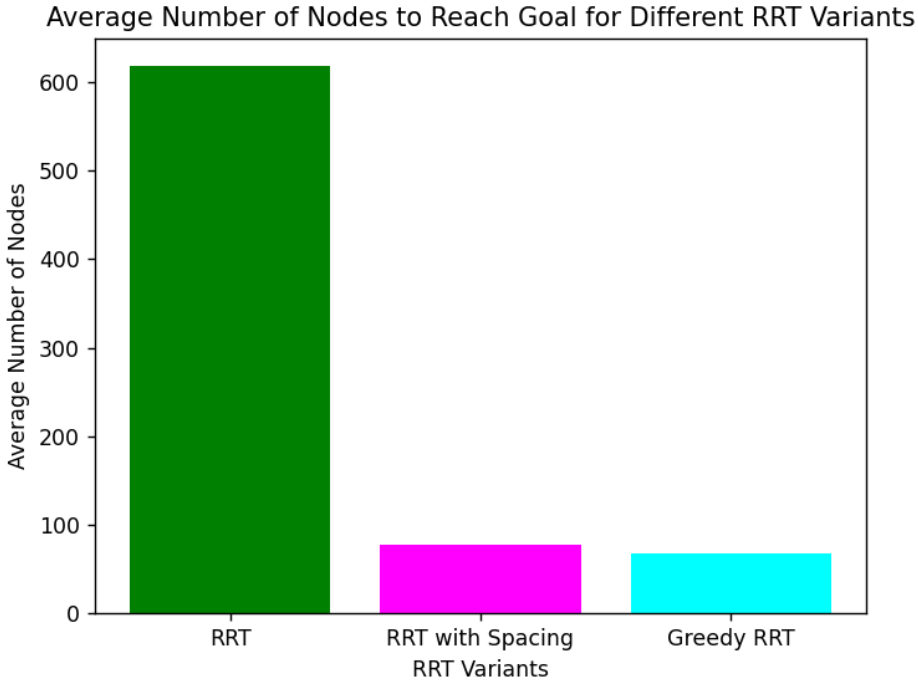


0.5 Visualization and Analysis

The implementation includes visualization functions to plot the environment, obstacles, RRT tree, and final paths. Additionally, we analyze the average number of nodes required to reach the goal for different variants of the RRT algorithm.

Average node counts:

We conducted simulations with 1000 iterations for each variant and calculated the average number of nodes needed to reach the goal. The results are summarized in the bar chart below.



Conclusion

In conclusion, we successfully implemented the RRT algorithm for path planning in autonomous robotics. By incorporating trajectory smoothing, spacing considerations, and a greedy approach, we improved the efficiency and effectiveness of the algorithm. Our analysis demonstrates the impact of these enhancements on the convergence towards the goal and the average number of nodes required.

Building a regex engine:

FSM are like roadmaps that help computers understand and follow a series of steps to do something. Think of them as guides for computer programs, showing them what to do next based on what's happening now. These maps have specific places and routes (transitions) between them. FSMs are handy for lots of things, like making traffic lights work, guiding vending machines, or even controlling video game characters.

In this assignment, we made a simple tool that uses FSMs to match patterns in text. You give it a pattern (like a special sequence of characters), and it checks if that pattern appears in a piece of text you provide. If it finds a match, it highlights that part of the text.

Transitions library:

- The transitions library in Python is a finite-state machine implementation for Python. It provides a convenient way to create and manage finite-state machines (FSMs) in Python code.

Converting Regular expression to FSM :

Converting a regular expression (regex) into a Finite State Machine (FSM) involves translating the abstract pattern defined by the regex into a concrete, state-based model capable of efficiently recognizing matching strings.

The process commences with parsing the regex to identify its components, including literals, metacharacters, and quantifiers. Subsequently, an FSM is constructed wherein each state represents a potential match state, and transitions between states are determined by the input characters.

Object-Oriented Programming (OOP) is a computer programming paradigm that revolves around organizing software design around data, or objects, rather than functions and logic. We are going to utilize the RRT algorithm within the OOP paradigm.

Special characters in the regex, such as '.', '*', '+', necessitate special handling in FSM construction:

- '.' represents any single character. In the FSM, this is implemented as transitions from the current state to multiple possible next states, each representing a specific character.
- '*' indicates zero or more occurrences of the preceding character or group. This is realized as looping transitions in the FSM, enabling the machine to iterate over the matching process.
- '+' is similar to '*', but mandates at least one occurrence of the preceding character or group. This is enacted through a combination of the corresponding '*' transition followed by a transition for the required single occurrence.

The resultant FSM furnishes a structured framework for proficiently matching input strings based on the regex pattern's specifications, facilitating robust pattern recognition across diverse applications.

1 State diagrams:

1.0.1 Direct matches:

The FSM diagram consists of two states: start and end.The initial state is start, and it transitions to the end state upon encountering the character 'a'.This pattern represents a direct match, meaning the input string is accepted only if it contains the specified character exactly as given in the pattern.

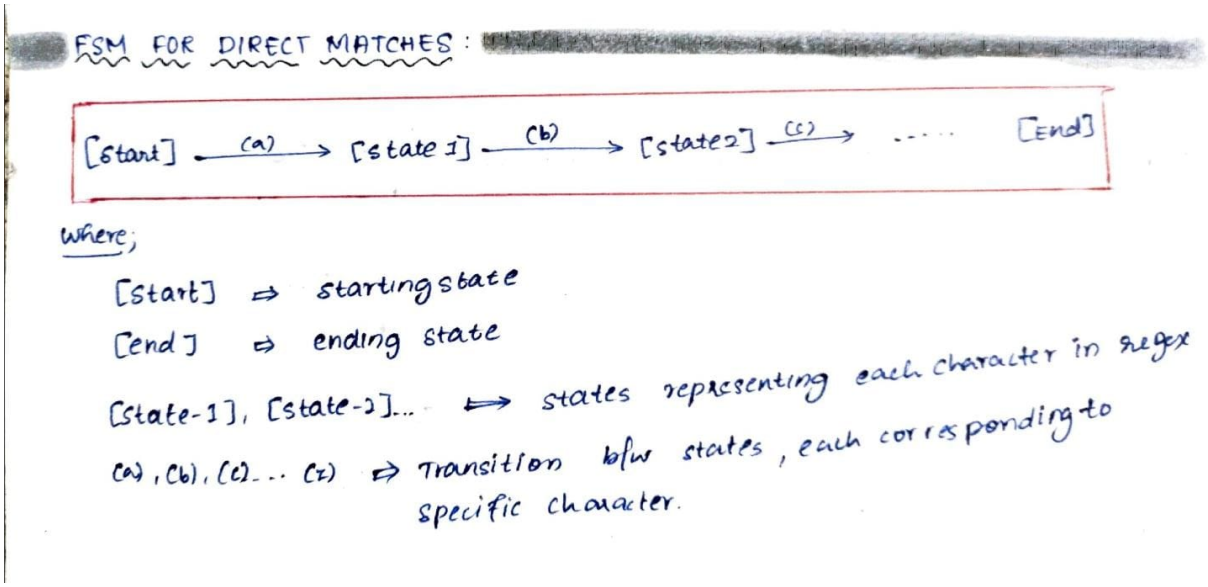


Figure 2: direct match

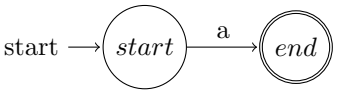


Figure 3: State Diagram for Direct Matches

1.0.2 Any number of characters:

The FSM diagram includes three states: start, middle, and end.From the start state, the FSM transitions to the middle state upon encountering the character 'b'. It stays in the start state if it encounters 'a' repeatedly.From the middle state, the FSM transitions to the end state upon encountering 'b'. It stays in the middle state if it encounters 'a' repeatedly.This pattern matches strings where 'b' is preceded by one or more 'a's, possibly separated by other characters.

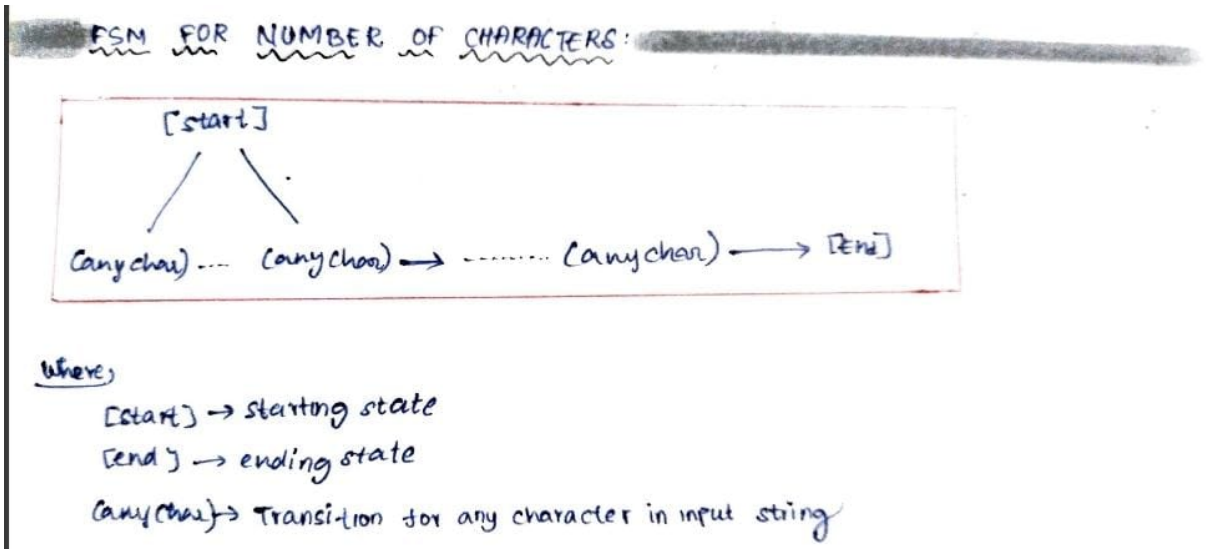
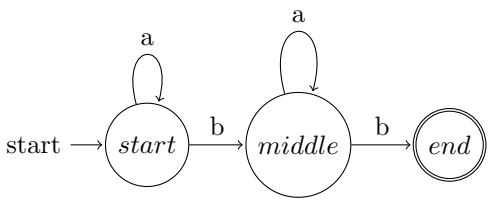


Figure 4: any number of characters



1.0.3 Wildcard Character:

The FSM diagram contains two states: start and end.From the start state, the FSM transitions to the end state upon encountering any character, indicated by the '.' symbol. This pattern matches any single character in the input string.

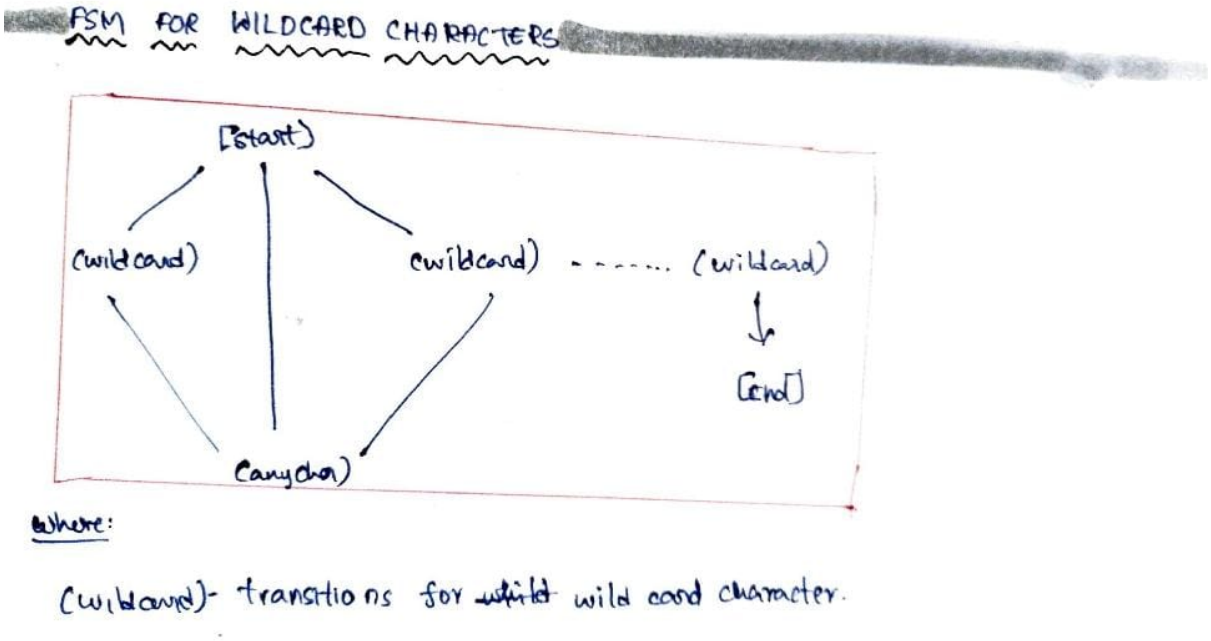


Figure 5: wild card characters

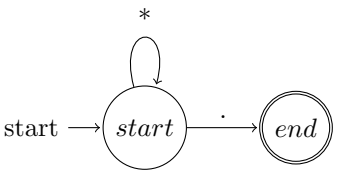


Figure 6: State Diagram for Wildcard Character

1.0.4 Multiple matches:

The FSM diagram consists of three states: start, middle, and end. From the start state, the FSM transitions to the middle state upon encountering the character 'b'. It stays in the start state if it encounters 'a' repeatedly. From the middle state, the FSM transitions to the end state upon encountering 'b'. It stays in the middle state if it encounters 'a' repeatedly.This pattern matches strings where 'b' is preceded by one or more 'a's, possibly separated by other characters.

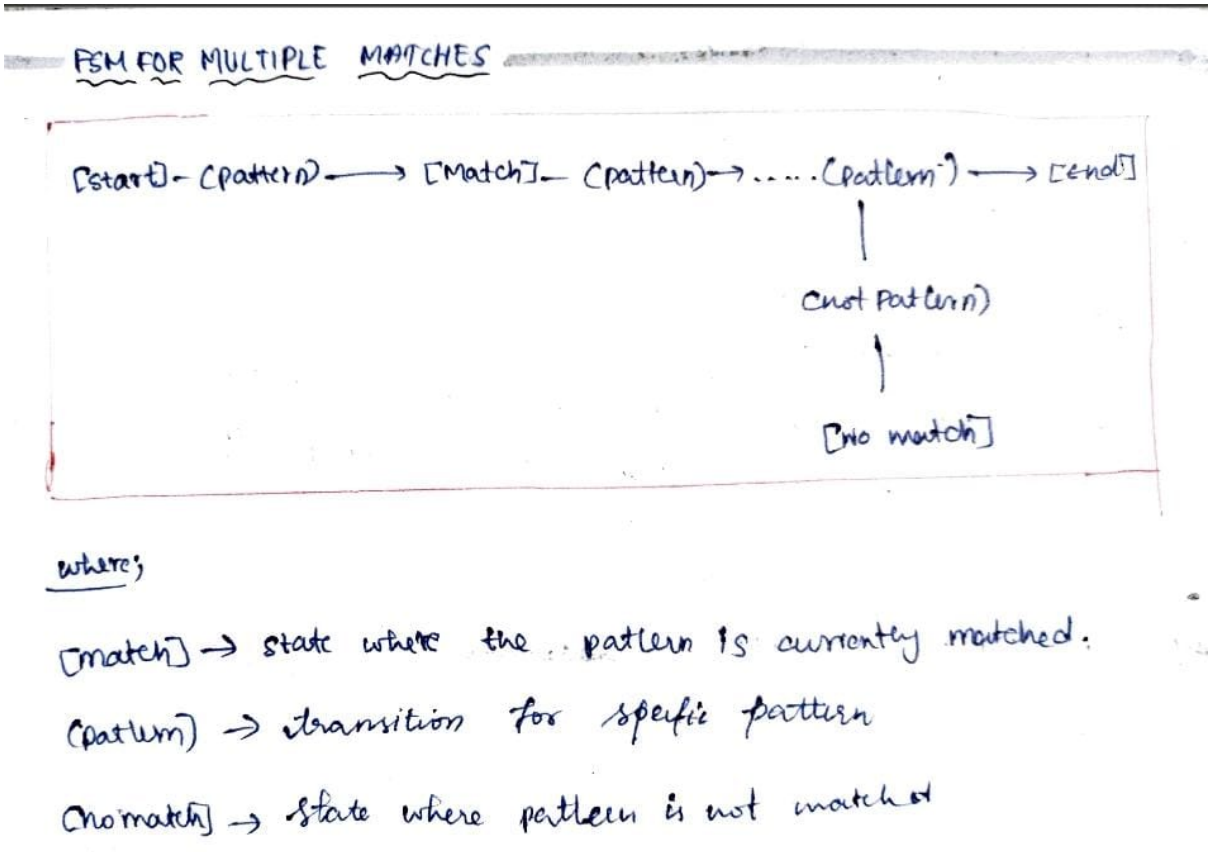


Figure 7: multiple matches

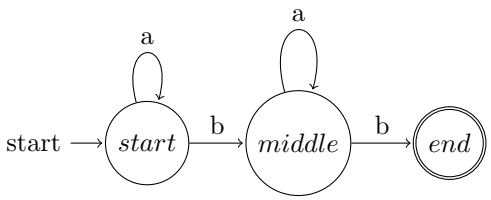


Figure 8: State Diagram for Multiple Matches

Conclusion:

The regex engine efficiently transforms regular expressions into FSMs using OOPS, facilitating accurate pattern matching in input strings. Its systematic design ensures precise handling of direct matches, variable character lengths, wildcard characters, and multiple matches. Integration of logging functionality enhances error detection and debugging, ensuring reliable execution.

Visual state diagrams provide clarity on the FSM's behavior, aiding comprehension. In summary, the regex engine presents a flexible solution for pattern matching tasks, demonstrating adeptness in applying theoretical concepts to practical applications and serving as a valuable asset for text processing needs.

Insights:

I learned about pathfinding algorithms and their purpose. Then, I delved into implementing the RRT algorithm, exploring its effectiveness and making modifications. Finally, I compared the outcomes of each variation. From the second question, I grasped the concept of FSM and its practical application. Regularly using regex commands, I gained insight into how they are coded, enhancing my understanding of FSMs. Overall this assignment enhanced my skillset.