LaTeX REPORT

TAKE HOME END SEMESTER .

Course Number: [”ID2090”]

Submitted by:

Name: SASTIKAA S
Roll Number: MM23B062

Date: May 25, 2024

0.1 Introduction

This is a latex report contains analysis of Question 1,2 of Take home end semester examination.

0.2 Path Planning with RRT Algorithm:

Path planning is a crucial aspect of autonomous robotics systems, enabling them to navigate complex environments efficiently. RRT algorithm is a sampling-based approach widely used for path planning when the environment map is available. This report explores enhancements to the RRT algorithm and their impact on path planning efficiency.

Objective of the question:

- 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.

Setting up the environment:

We import matplotlib module (for graph and plotting purpose), numpy module (for numerical computations) and random module. After this we create a rectangle which acts as the walls of the environment.with the help of random.randint function we generate obstacles at random points(assuring that the obstacles don't overlap each other).

0.3 Implementing RRT algorithm:

The RRT algorithm builds a tree structure by iteratively sampling random points in the environment and connecting them to the nearest existing node. The algorithm continues until a path to the goal is found.

code explanation

- `class Node`: Represents a point in the RRT tree.
- `def distance(p1, p2)`: This function calculates the distance between two points using numpy's `hypot` function.
- `def nearest_node()`: This function finds the node in a list of nodes that is closest to a given random point.
- `def steer()`: This function extends the tree from the nearest node.
- `def line_collision_check()`: This function verifies path feasibility.
- `def extract_path()`: This function retrieves the path to the goal.

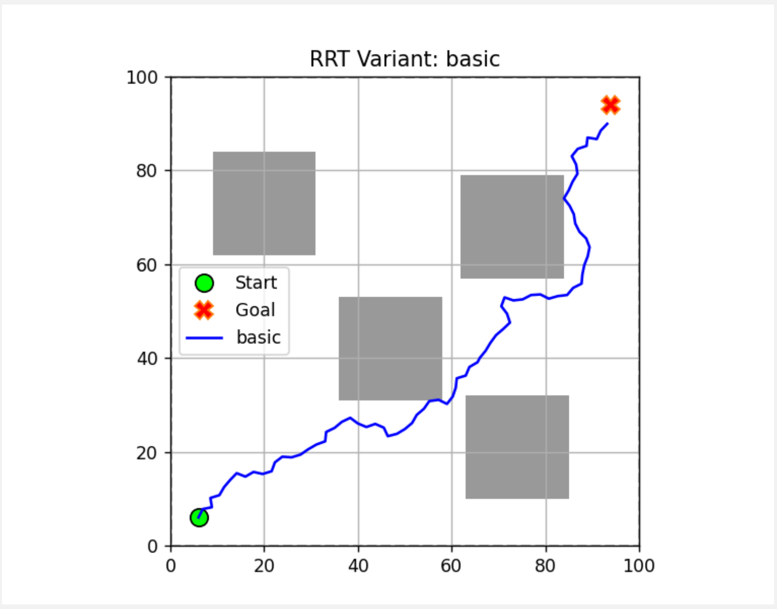


Figure 1: basic

0.4 Trajectory smoothing:

Trajectory smoothing is essential to ensure that the generated path is feasible for robotic motion, reducing unnecessary sharp turns and improving navigation efficiency.

code explanation

- def smooth_path(): this function attempts to smooth a given path by iteratively removing intermediate points if a direct path between randomly selected pairs of points is possible and does not collide with any obstacles. It repeats this process until either the path cannot be further optimized or a maximum number of iterations is reached.

0.5 Spacing between obstacles :

We have to take it to account about spacing while path planning is done to prevent collisions and a safe navigation for the robot. This is achieved by Implementing a buffer around obstacles due to which the random point sampling is done away from obstacle edges.

code explanation

- def find_free_point():This function generates random points within a defined space, ensuring they are not within any obstacles with a specified buffer zone.

0.6 Greedy approach :

Greedy algorithm prioritizes points closest to the goal, favoring paths that show promising progress towards achieving the objective. By favoring such paths,this algorithm can potentially reduce the overall search effort and converge faster towards the goal compared to random selection strategies.

code explanation

- The ' rrt ' function is modified to bias random point sampling towards the goal with a certain probability .This encourages the algorithm to explore areas closer to the goal more frequently .

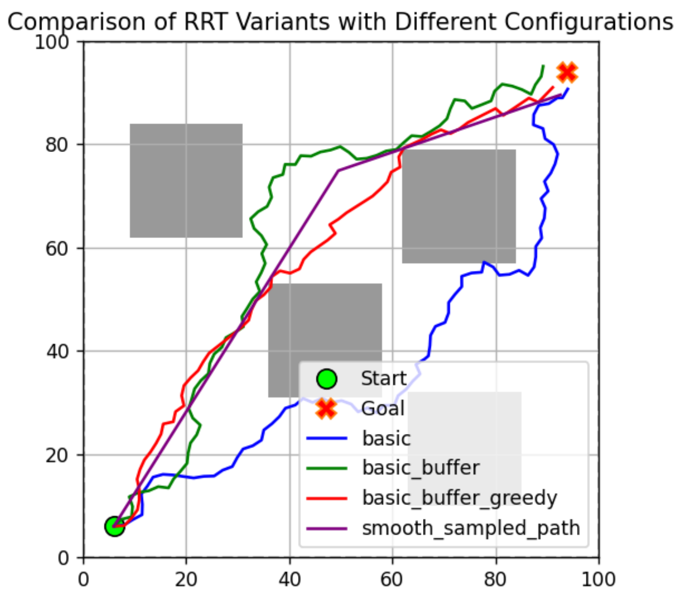


Figure 2: All algorithms

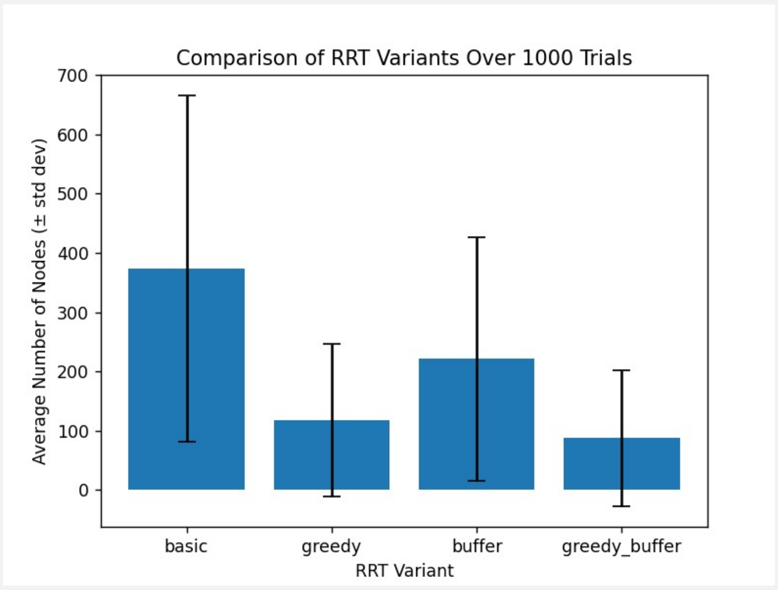
0.7 Simulation and Results:

We conducted simulations with different variants of the RRT algorithm, varying parameters such as environment size, obstacle density, and sampling strategies. The simulation results show the average number of nodes required to reach the goal for each RRT variant, indicating the performance and efficiency of each approach. We discuss the effectiveness of each RRT variant in terms of path planning performance and computational complexity. Insights are provided on potential improvements and future research directions.

- **Basic RRT:** Serving as the reference point, this variant relies on purely random sampling, typically resulting in a higher node count compared to other variants.
- **RRT with Buffer:** Adding buffer zones increases safety but it can also increase the path length due to reduced space for sampling.
- **Greedy RRT:** Employing a greedy algorithm, this variant notably reduces the average node count by prioritizing sampling towards the goal configuration.

- **Greedy RRT with Buffer:** Striking a balance between efficiency and safety, this variant offers a compromise by combining the advantages of the greedy algorithm with the safety enhancement provided by buffer zones.

RESULT



It takes about 5-10 min to run 1000 simulations and get the out put of the plot

0.8 Closure:

This assignment demonstrated the versatility and adaptability of the RRT algorithm in addressing various challenges in robotic path planning, from generating feasible paths to optimizing for efficiency and safety. By combining theoretical concepts with practical implementations and analysis, valuable insights were gained into the complexities of real-world navigation scenarios and the importance of algorithmic optimizations for autonomous systems.

0.9 Automata (Building a regex engine) :

FSM stands for Finite State Machine. It's a mathematical model used in computer science to represent systems or processes with a finite number of states, transitions between those states, and actions associated with those transitions. It's commonly used in designing software for things like traffic lights, vending machines, and digital circuits.

In this assignment, we implemented a basic regular expression engine using finite state machines (FSM). The program accepts a regular expression pattern and a test string, and it highlights the matched portions of the test string based on the pattern.

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. we used this library in our code.

Logging module:

This module will allow us to log important events, errors, or debug information during the execution of our regular expression engine.I have used this in this assignment for logging errors.

0.10 Regular expression to FSM conversion :

Converting regex into FSM involves transforming the abstract pattern defined by the regex into a concrete, state-based model capable of efficiently recognizing matching strings.

- The process begins with parsing the regex to identify its components, including literals, metacharacters, and quantifiers. Then, an FSM is constructed where each state represents a possible match state, and transitions between states are determined by the input characters.

OOPS is a computer programming model that organizes software design around data, or objects, rather than functions and logic. We are going to use rrt algorithm in oops.

Special characters in the regex, such as '.', '*', '+', require special handling in the 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 implemented as looping transitions in the FSM, allowing the machine to repeat the matching process.
- + : Similar to ", but requires at least one occurrence of the preceding character or group. This is implemented as a combination of the corresponding " transition followed by a transition for the required single occurrence.

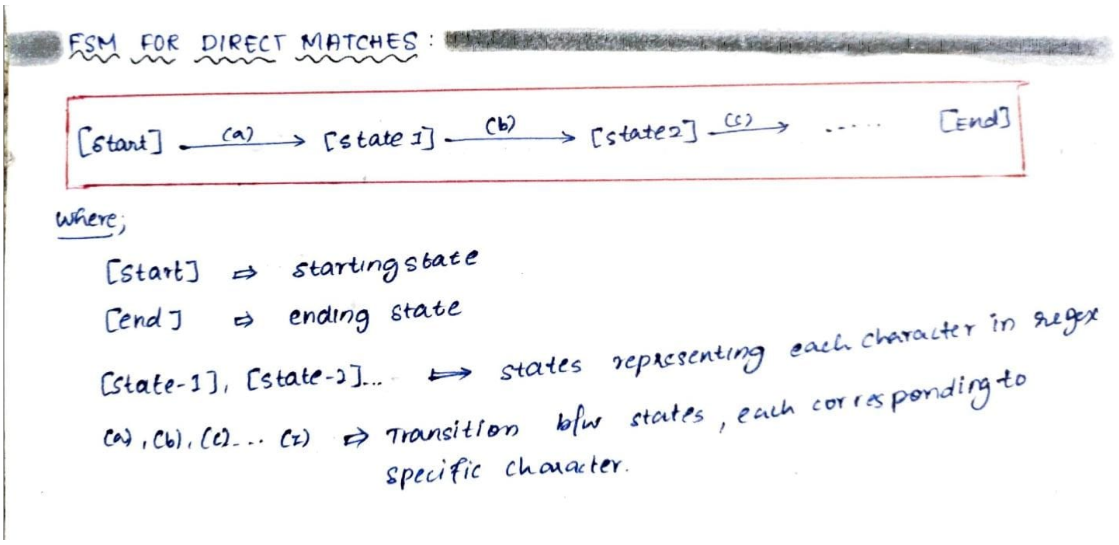
The constructed FSM provides a structured framework for efficiently matching input strings based on the regex pattern's specifications, facilitating robust pattern recognition in various applications.

```
class RegexFSM(object):
    def __init__(self, pattern):

    (#CONSTRUCTOR CODE)

    # Method to build FSM from the pattern
    def _build_fsm(self):

    (# FSM building code)

    # Callback for character match
    def on_match(self, event):

    # Callback to save the match
    def save_match(self, event):

    (# Saving matched portions)

    # Method to process input string and highlightmatches
    def process_string(self, input_string):

    (# Processing input string)

    # Helper method to get highlighted text
    def get_highlighted_text(self, input_string):

    (# Highlighting matched portions)
```

0.11 State diagrams:

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.



FSM FOR DIRECT MATCHES :

[Start] —(a)→ [state 1] —(b)→ [state2] —(c)→ ..... [End]

Where;

[start] ⇒ starting state

[end] ⇒ ending state

[state-1], [state-2]... ⟺ states representing each character in regex

(a), (b), (c)... (z) ⇒ Transition b/w states, each corresponding to specific character.

Figure 3: direct match

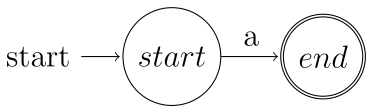


Figure 4: State Diagram for Direct Matches

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.

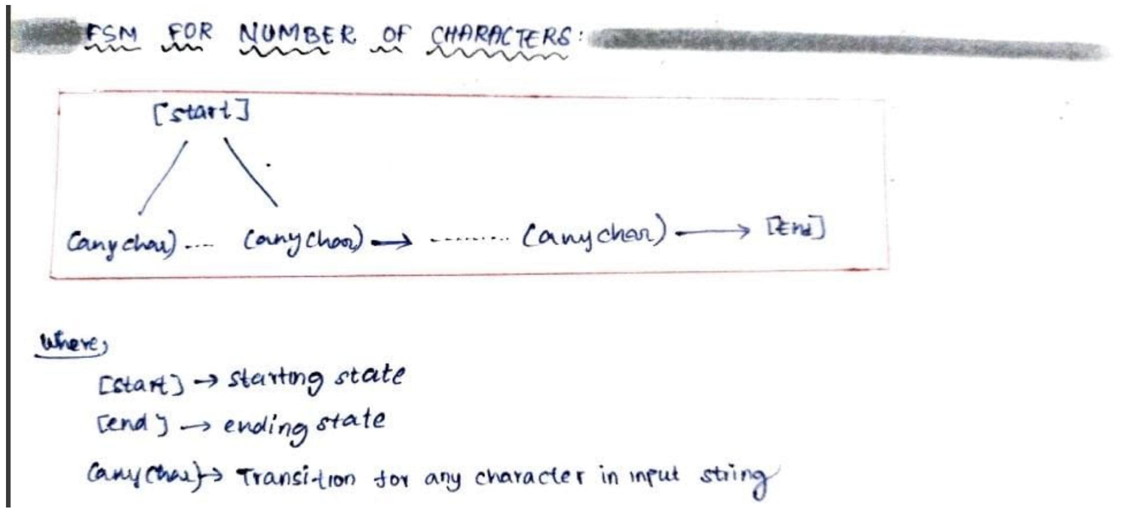
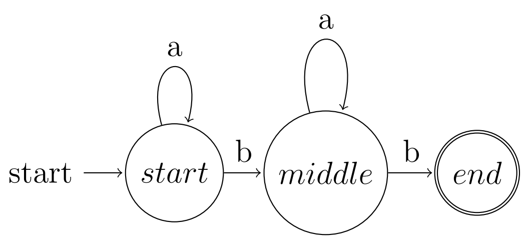

FSM FOR NUMBER OF CHARACTERS:
[start]
(any char) ---- (any char) → .... (any char) → [End]
where,
[start] → starting state
[end] → ending state
(any char) → Transition for any character in input string

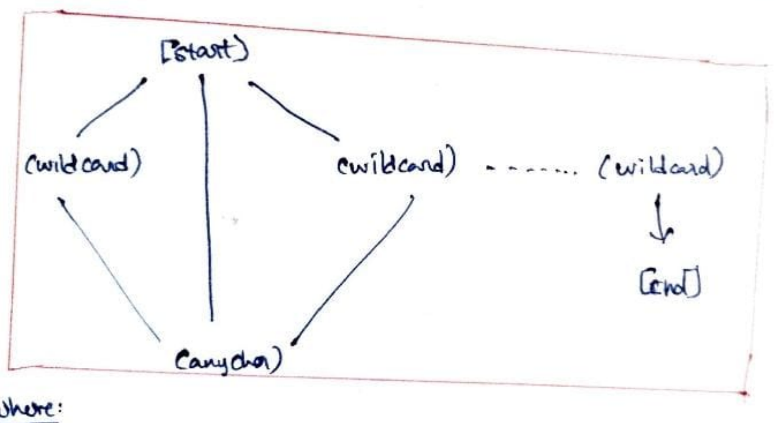
Figure 5: any number of characters



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.

FSM FOR WILDCARD CHARACTERS



Where:

(wildcard)- transitions for ~~whild~~ wild card character.

Figure 6: wild card characters

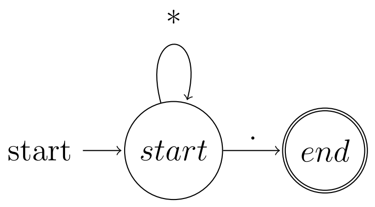
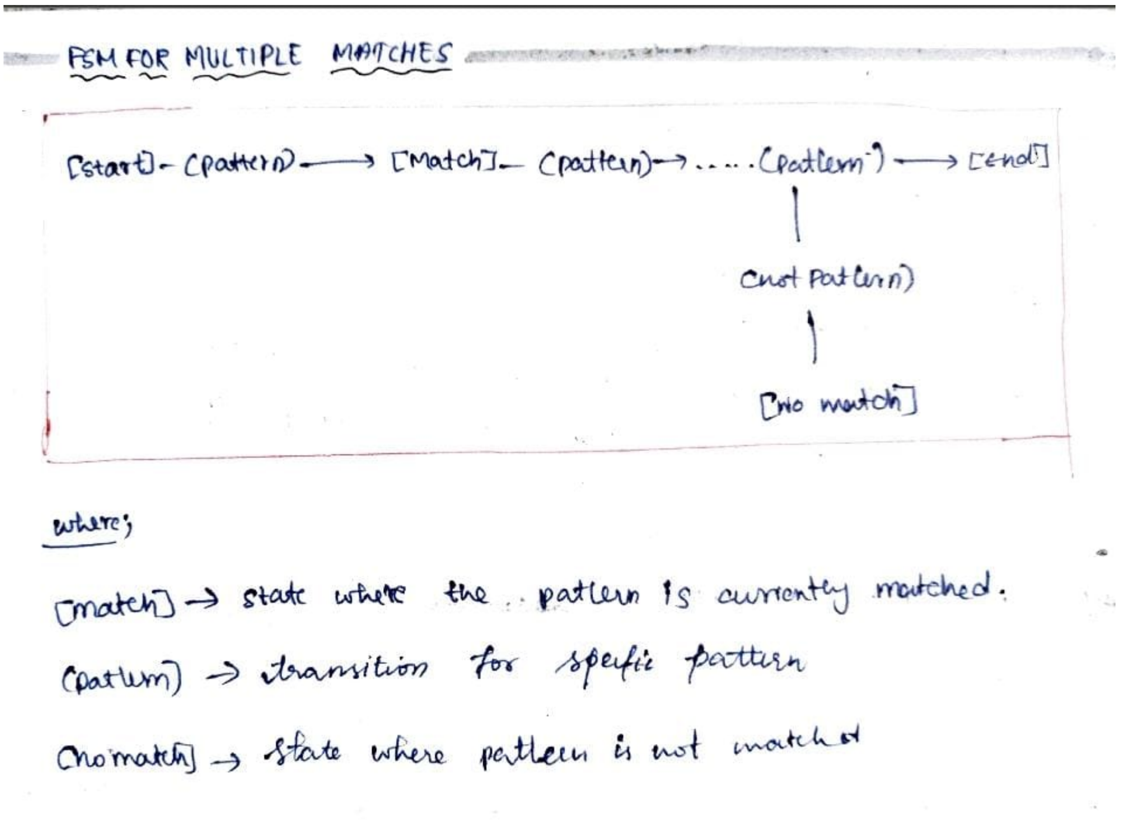


Figure 7: State Diagram for Wildcard Character

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.



FSM FOR MULTIPLE MATCHES

$$\begin{tikzcd}
\text{[start]} \arrow[r, dash] & \text{(pattern)} \arrow[r] & \text{[Match]} \arrow[r, dash] & \text{(pattern)} \arrow[r] & \text{.....} & \text{(pattern)} \arrow[r] \arrow[d, dash] & \text{[end]} \\
 & & & & & \text{(not pattern)} \arrow[d, dash] & \\
 & & & & & \text{[no match]} &
\end{tikzcd}$$

where;

[match] → state where the pattern is currently matched.

(pattern) → transition for specific pattern

[no match] → state where pattern is not matched

Figure 8: multiple matches

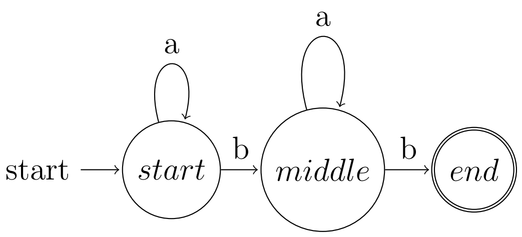
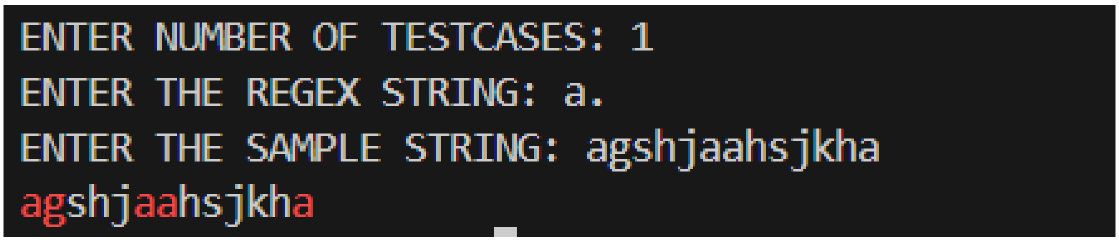


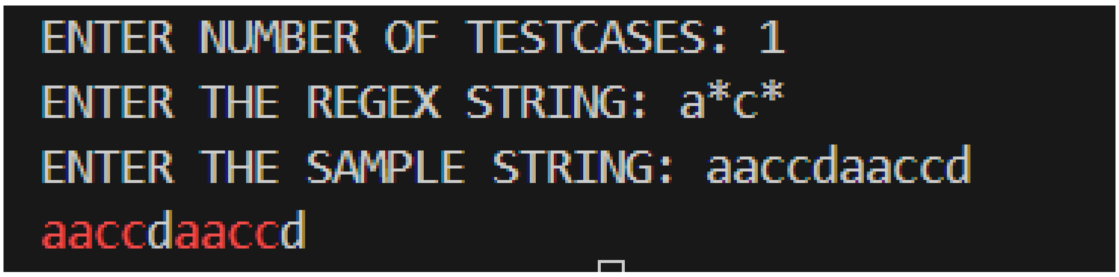
Figure 9: State Diagram for Multiple Matches

0.12 Execution:



```
ENTER NUMBER OF TESTCASES: 1
ENTER THE REGEX STRING: a.
ENTER THE SAMPLE STRING: agshjaahsjkha
agshjaahsjkha
```

Figure 10: output



```
ENTER NUMBER OF TESTCASES: 1
ENTER THE REGEX STRING: a*c*
ENTER THE SAMPLE STRING: aaccdaaccd
aaccdaaccd
```

Figure 11: output

0.13 Closure:

The implemented regex engine adeptly converts regular expressions into FSM using OOPS. This FSM effectively matches patterns against input strings, fulfilling the task's requirements

Through systematic design, the engine handles direct matches, any number of characters, wildcard characters, and multiple matches with precision. Logging functionality enhances debugging, ensuring robustness in execution.

State diagrams offer visual insight into the FSM's behavior, aiding understanding. In conclusion, the regex engine offers a versatile solution for pattern matching tasks, showcasing proficiency in bridging theory and practice, and stands as a valuable tool for text processing applications.

0.14 Takeaways:

- From the 1st question I learnt about path finding algorithms and why we use them. Then I learnt how to implement the rrt algorithm which is effective and interesting. Various modifications were done and finally i compared the results from each case.

- From the second question I learnt what is a fsm and how to use it. Using regex command more frequently i never wondered how was it coded. This question gave me an idea on how it was made. I got a grip on fsm from this question.