# ID2090 End Semester Report

Arivoli Ramamoorthy ee23b008

May 2024

## 1 Path planning

### 1.1 Task Description

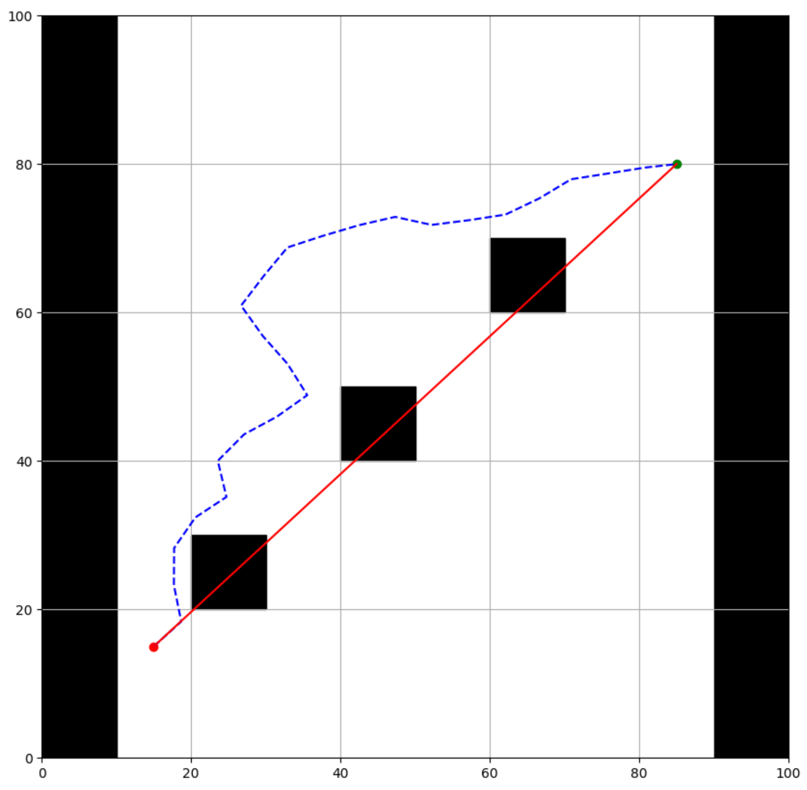
Your task is to write a program that plots a rectangle of some arbitrary width and height, which acts as the walls of the environment, and place 3-4 obstacles, which are rectangles of around one-tenth the size of the bounding wall. The placement of the obstacles can be random or fixed. Choose two points -start and goal. With this, you are expected to do the following:

1. Implement the RRT algorithm in this environment.
2. Since RRT is a random sampling algorithm, the path produced by it is not smooth, which in real life is not a feasible trajectory. So, use the path from the previous part and do trajectory smoothing on it.
3. RRT also doesn't consider the spacing necessary between obstacles and the path that a bot might follow. So, implement RRT such that the random points are more likely at a spacing away from the edges of the obstacle. This spacing is a parameter you can set as you see fit.
4. A greedy approach would be choosing points nearer to the goal more likely. Implement this in RRT algorithm. The nearer here can be any logical distance function such as L1 or L2 norm.
5. Write a report on your implementation of the previous parts and give plots on the average number of nodes needed to reach the goal - this is done by running the simulation a large number of times and taking the average; a good number would be 1000 for each variant. Setup an environment as shown Figure 1 to check your implementation of RRT from part 3 and write about it in the the report.

This report describes the implementation and results of two path planning algorithms: the standard RRT (rapidly exploring random tree) and an enhanced version with a greedy sampling strategy (Greedy RRT). These algorithms are designed to find a path from a start point node to a goal point in a 2D environment with obstacles.

## 1.2 Environment image

Like given in the assignment Figure1, and mentioned in the task description, 2 sides are placed with rectangles which act as the walls of the environment, and 3 obstacles are placed of around size one tenth of the bounding wall (I've fixed the placements, but it can be randomized by using random library and a few variables instead of hardcoding positions).



## 1.3 Initalizing Nodes

```python
class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.parent = None

# Finding distance between nodes
def distance(node1, node2):
    dist =  np.sqrt((node1.x - node2.x)**2 + (node1.y - node2.y)
    **2)
    return dist

# Randomly select nodes from current
```

```python
def sample_random_node(width, height):
        # Random uniform chooses numbers between specified width/height
    return Node(random.uniform(0, width), random.uniform(0, height))

# Finds the nearest node
def nearest_node(nodes, random_node):
    def distance_key(node):
        return distance(node, random_node)
    return min(nodes, key=distance_key)

def movement(from_node, to_node, max_dist):
    if distance(from_node, to_node) <= max_dist:
        return to_node
    theta = np.arctan2(to_node.y - from_node.y, to_node.x - from_node.x)
    return Node(from_node.x + max_dist * np.cos(theta), from_node.y + max_dist * np.sin(theta))

def possible_path(node1, node2, obstacles, safe=2):
    for x, y, w, h in obstacles:
        if x - safe <= min(node1.x, node2.x) <= x + w + safe and y - safe <= min(node1.y, node2.y) <= y + h + safe:
            return False
    return True
```

The functions sample_random_node, nearest_node, movement, possible_path are used in the RRT function and are the basic functions required for the code.

## 1.4 Implementing RRT

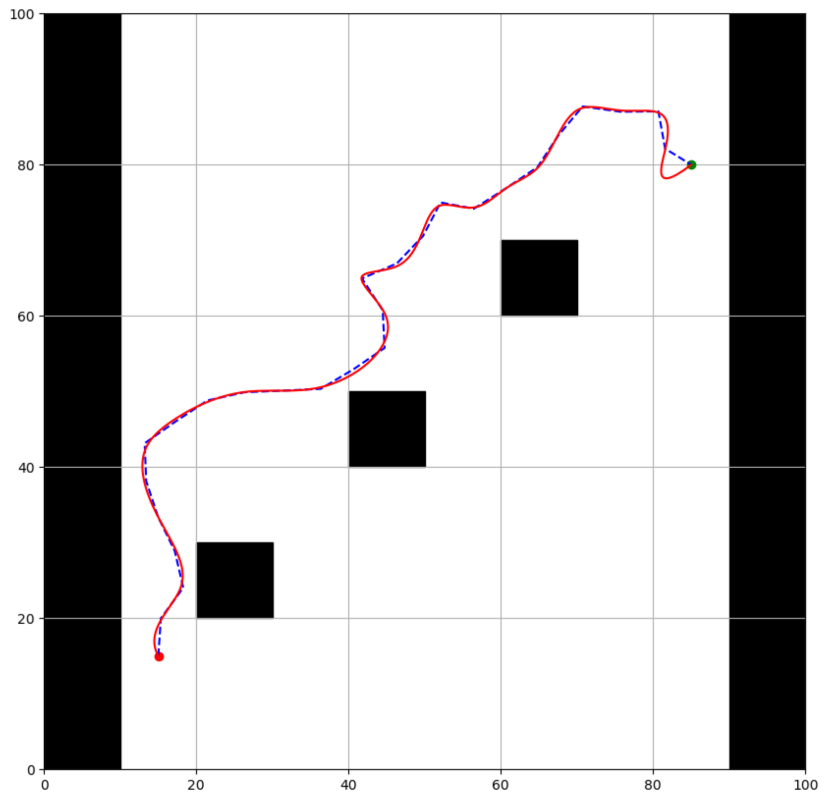
Took help from chatgpt for this, initially I tried implementing the pseudocode from wikipedia, but it didn't quite work out.

```python
def rrt(start, goal, width, height, obstacles, max_dist, tolerance, max_iter=1000):
    nodes = [start]
    for i in range(max_iter):
        rand_node = sample_random_node(width, height)
        nearest = nearest_node(nodes, rand_node)
        new_node = movement(nearest, rand_node, max_dist)
        if possible_path(nearest, new_node, obstacles):
            new_node.parent = nearest
            nodes.append(new_node)
            if distance(new_node, goal) <= tolerance:
                goal.parent = new_node
                nodes.append(goal)
                return nodes
    return None
```

## 1.5 Smoothening Trajectory

Smoothening the trajectory using B-spline representation to get a smooth curve.

```python
def smooth_path(path, obstacles, s=2):
    if not path:
        return path
    x = [node.x for node in path]
    y = [node.y for node in path]
    tck, u = splprep([x, y], s=s)
    unew = np.linspace(0, 1.0, num=500)
    out = splev(unew, tck)
    smooth_nodes = [Node(x, y) for x, y in zip(out[0], out[1])]
    return smooth_nodes
```



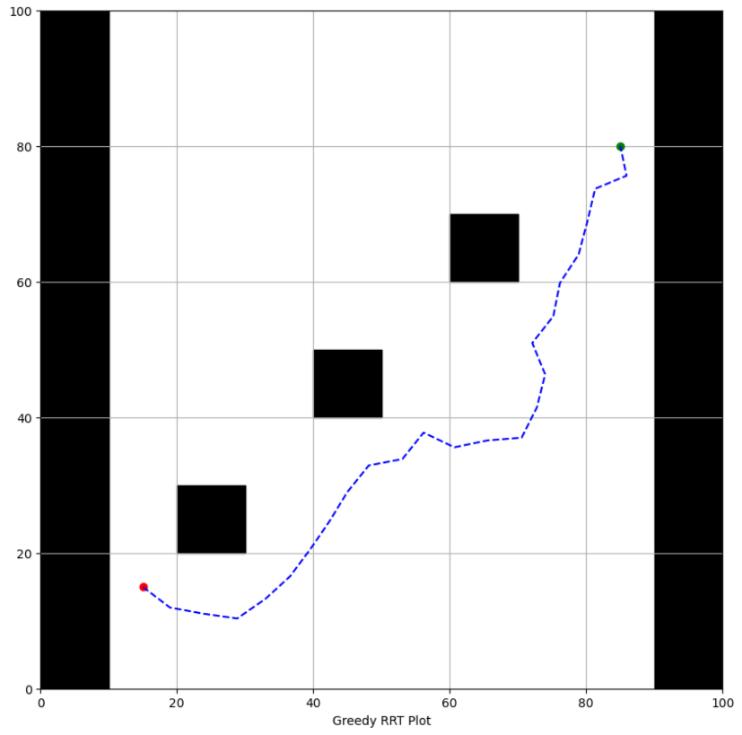
## 1.6 Edges of an obstacle

So far in the report, subpart (a) and subpart (b) have been covered (implementing RRT and smoothening). In the task description, subpart(c) wanted us to maintain a certain distance away from the edges of the obstacles. And for this, I've implemented the 'safe' variable which makes the centre of the moving body stay away from the edge of the obstacle by atleast 'safe' distance.

```python
def possible_path(node1, node2, obstacles, safe=2):
    for x, y, w, h in obstacles:
        if x - safe <= min(node1.x, node2.x) <= x + w + safe and y
    - safe <= min(node1.y, node2.y) <= y + h + safe:
            return False
    return True
```

## 1.7 Greedy approach

Greedy approach works by occasionally biasing the random sampling towards the goal, increasing the likelihood of finding a path more quickly. (RRT is completely random hence taking more time). the 'occasionaly' is decided by the greedy factor. Whenever random.random is less than greedy factor (initially set at a 20% chance as greedy factor is hardcoded as 0.2), the bias towards the goal is applied.



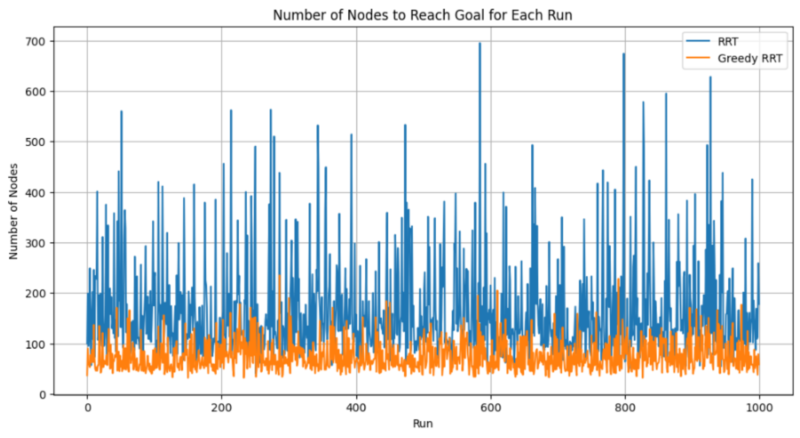
```
def greedy_sample_random_node(goal, width, height, greedy_factor
    =0.2):
    if random.random() < greedy_factor:
        return Node(goal.x + random.uniform(-5, 5), goal.y + random
    .uniform(-5, 5))
    else:
        return sample_random_node(width, height)

```

```
def rrt_with_greedy(start, goal, width, height, obstacles, max_dist
, tolerance, max_iter=1000, greedy_factor=0.2):
    nodes = [start]
    for i in range(max_iter):
        rand_node = greedy_sample_random_node(goal, width, height,
    greedy_factor)
        nearest = nearest_node(nodes, rand_node)
        new_node = movement(nearest, rand_node, max_dist)
        if possible_path(nearest, new_node, obstacles):
            new_node.parent = nearest
            nodes.append(new_node)
            if distance(new_node, goal) <= tolerance:
                goal.parent = new_node
                nodes.append(goal)
                return nodes
    return None
```

The rest of the process remains similar to the standard RRT algorithm, ensuring that the path is valid and avoids obstacles.

## 1.8 RRT vs Greedy RRT

```
def average_nodes_rrt(runs, start, goal, width, height, obstacles,
    max_dist, tolerance, max_iter=1000):
    node_counts = []
    for _ in range(runs):
        nodes = rrt(start, goal, width, height, obstacles, max_dist
, tolerance, max_iter)
        if nodes:
            node_counts.append(len(nodes))
    return np.mean(node_counts)

def average_nodes_rrt_greedy(runs, start, goal, width, height,
    obstacles, max_dist, tolerance, max_iter=1000, greedy_factor
    =0.2):
    node_counts = []
    for _ in range(runs):
        nodes = rrt_with_greedy(start, goal, width, height,
    obstacles, max_dist, tolerance, max_iter, greedy_factor)
        if nodes:
            node_counts.append(len(nodes))
    return np.mean(node_counts)

# Parameters for simulation
runs = 1000

# Calculate average number of nodes
avg_nodes_rrt = average_nodes_rrt(runs, start, goal, width, height,
     obstacles, max_dist, tolerance)
avg_nodes_rrt_greedy = average_nodes_rrt_greedy(runs, start, goal,
    width, height, obstacles, max_dist, tolerance)

print(f"Average number of nodes (RRT): {avg_nodes_rrt}")
print(f"Average number of nodes (Greedy RRT): {avg_nodes_rrt_greedy
    }")
```

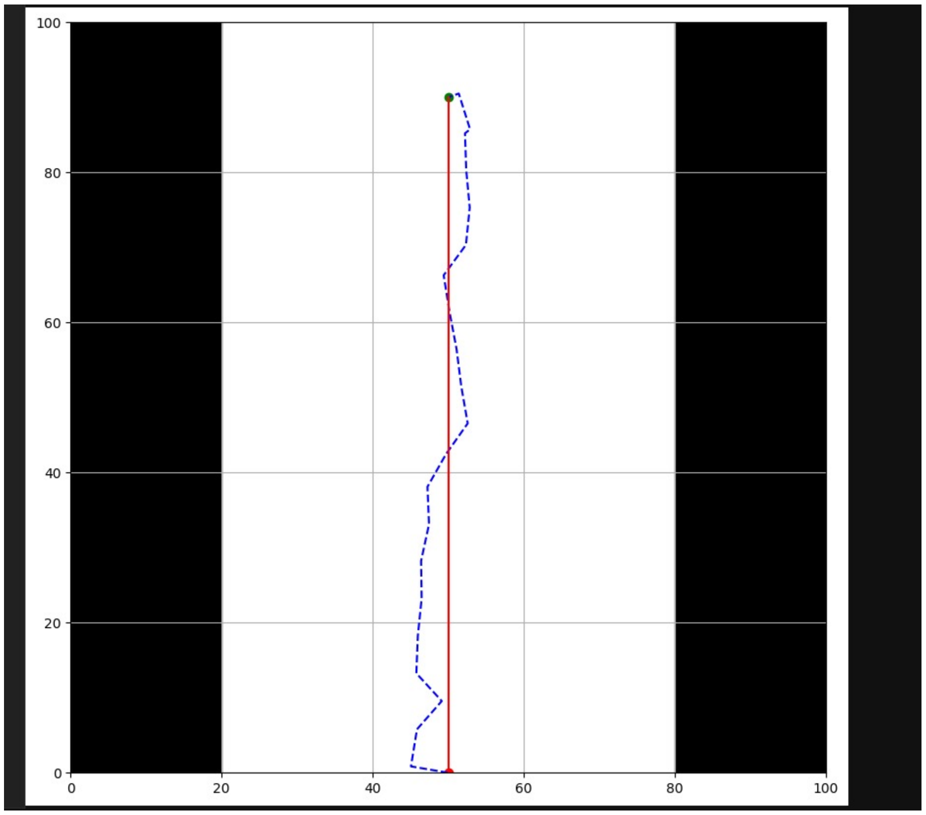


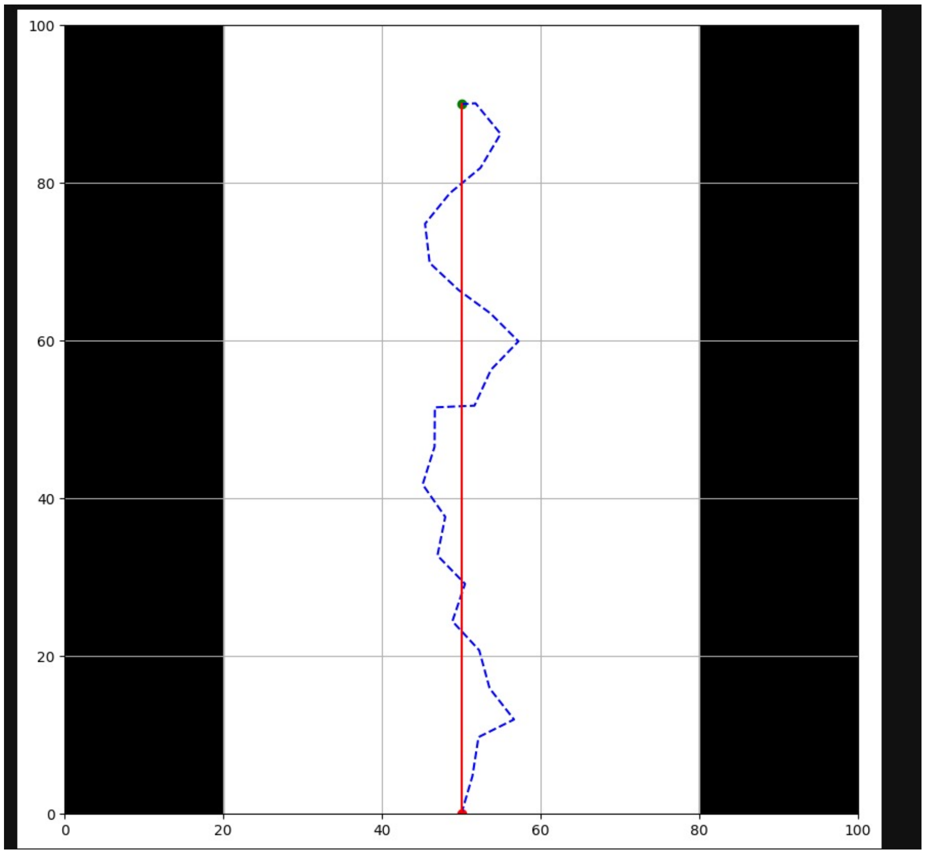
```
Average number of nodes (RRT): 163.086
Average number of nodes (Greedy RRT): 74.813
```

Clearly, the Greedy RRT is more efficient as expected.
Code for the plotting:

```python
# Parameters for simulation
runs = 1000
width, height = 100, 100
obstacles = [(20, 20, 10, 10), (40, 40, 10, 10), (60, 60, 10, 10)
    ,(0, 0, 10, 100),(90, 0, 10, 100)]
start = Node(15, 15)
goal = Node(85, 80)
max_dist = 5
tolerance = 5

# Collect node counts for each run
nodes_count_rrt = average_nodes_rrt(runs, start, goal, width,
    height, obstacles, max_dist, tolerance)
nodes_count_rrt_greedy = average_nodes_rrt_greedy(runs, start, goal
    , width, height, obstacles, max_dist, tolerance)

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(nodes_count_rrt, label='RRT')
plt.plot(nodes_count_rrt_greedy, label='Greedy RRT')
plt.xlabel('Run')
plt.ylabel('Number of Nodes')
plt.title('Number of Nodes to Reach Goal for Each Run')
plt.legend()
plt.grid(True)
plt.show()

# Calculate and print average number of nodes
avg_nodes_rrt = np.mean(nodes_count_rrt)
avg_nodes_rrt_greedy = np.mean(nodes_count_rrt_greedy)
print(f"Average number of nodes (RRT): {avg_nodes_rrt}")
print(f"Average number of nodes (Greedy RRT): {avg_nodes_rrt_greedy
    }")
```

## 1.9 Test environment for Part 3, section c





With a 'safe' (spacing parameter) distance of 25(very high safe distance to show that my code works), we can see that, RRT tends to stay away from the edges by a lot. Proves correct implementation of part 3.

# 2 Automata

## 2.1 Task Description

In this task, you will implement a basic regex engine that converts a regular expression into a basic finite state machine, in which we pass a string and match the expression. You need to do the following:

1. Implement a program that converts a regular expression into an FSM. A suggestion would be to use OOPS to build the state machine.
2. Write a report on your implementation of a regular expression engine and give the finite state machine's state diagram for the below cases.

The FSM should be able to match:

1. Direct Matches
2. Any number of Characters
3. Wildcard Character
4. Multiple Matches

## 2.2 Introduction

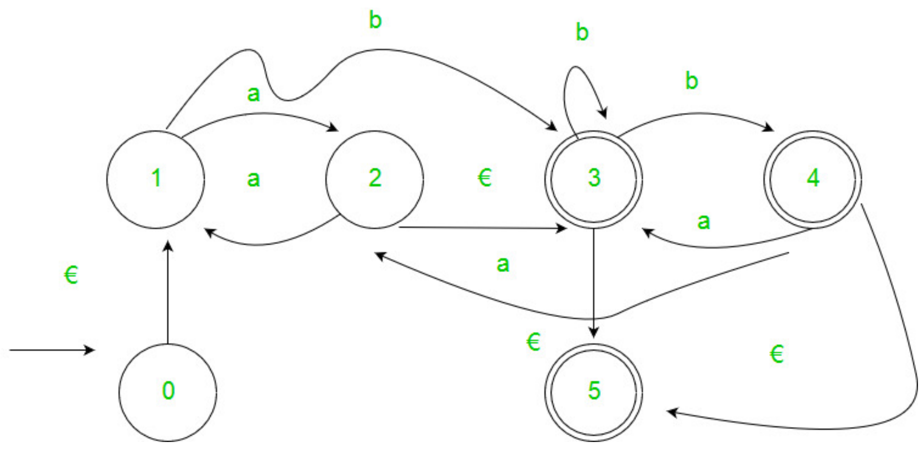
This report outlines the design and implementation of a simple regex parser using a Finite State Machine (FSM). The parser is capable of handling basic regex patterns, including the use of wildcards * (zero or more of the preceding character), . (any single character) and + (one or more of the preceding character).

## 2.3 FSM design

The FSM is designed to process an input string against a specified regular expression. The key components of the FSM include:

- **States:** Represented by characters in the regex.
- **Transitions:** Movement between states based on input characters.
- **Current State:** The state the FSM is currently in.
- **Previous State:** The state the FSM was in before the current state.
- **State Index:** The position of the current state in the regex.

The FSM starts at the first character of the regex and attempts to transition through each character of the input string. This process of it going letter by letter (state by state) makes it a finite state machine



## 2.4 Implementation

### 2.4.1 FSM Initialization

Class definition for FSM

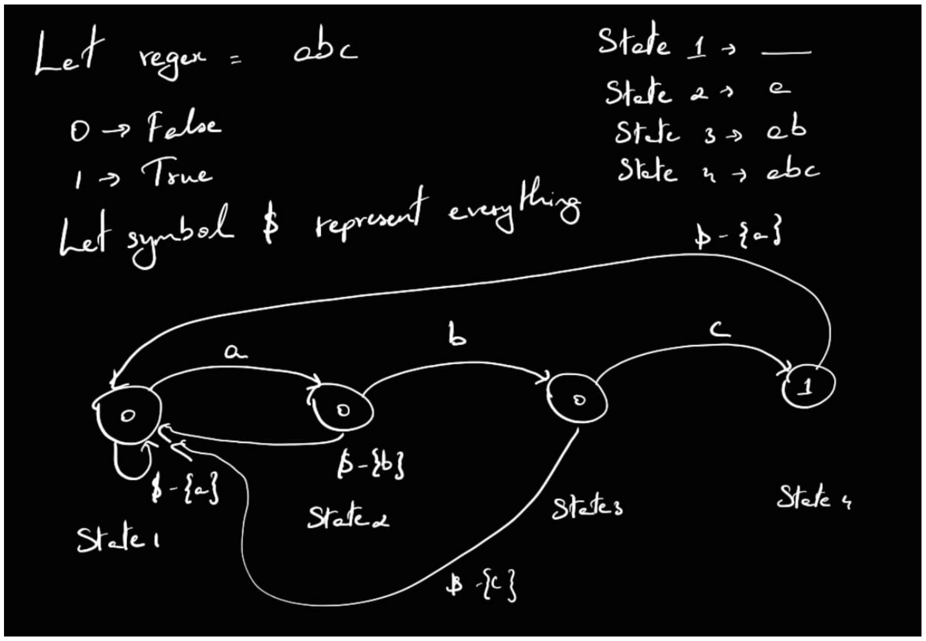
```
class FSM:
    #constructor
    def __init__(self,regexp):
        #contains the number of states(each character is
        # a state)

        ##regular expression
        self.regexp=regexp
        #the current state (string)
        self.current_state = regexp[0]
        self.prev_state = self.current_state
        #the total number of states
        self.states = len(regexp)
        #idx of the state
        self.state_number=0
```

### 2.4.2 State Transitions

Code for state transitions, including wildcards The comments of the code explain what each set of code is doing.

```
    def state_transition(self,input):
        takePrev=False ##to indicate match is tiill previous
    element
        Matched = False
        Reset = False

        # print(self.current_state)

        if (self.current_state not in self.wildcards):
            if (input == self.regexp[self.state_number]):
                #checking if it is eligible for a state transition
```

```python
            self.state_number+=1
            if (self.state_number==self.states):
                #if succesfully matched
                Matched = True
                #resetting the index of the state, and current state back to null
                self.state_number=0
                self.current_state=self.regexp[0]
                return Matched, Reset, takePrev
            #updating the current state
            self.prev_state=self.current_state
            self.current_state=self.regexp[self.state_number]
        else:

            if (self.regexp[self.state_number+1]=='*'):
                Matched=True
                takePrev=True
                self.current_state=self.regexp[0]
                self.prev_state=self.current_state
                self.state_number=0
                return Matched, Reset, takePrev

            Reset = True
            self.current_state=self.regexp[0]
            self.state_number=0
            self.prev_state=self.current_state
            return Matched, Reset, takePrev

    elif (self.current_state=="+" or self.current_state=="*"):
        ##what to do if currently we have hit a *, previous state becomes important
        #if the input is same as the previous state, we need not do anything,
        #if the input is different, we need to check the next state, (in the regex)
        # if the next state is same, we can move on
        if (self.prev_state!=input):
            self.state_number+=1

            ##are we done?(with the regex)
            if (self.state_number==self.states):
                self.current_state=self.regexp[0]
                self.prev_state=self.current_state
                self.state_number=0
                if (self.regexp[0]==input):
                    self.state_number=1
                    self.current_state=self.regexp[1]
                    self.prev_state=self.regexp[0]
                Matched=True
                takePrev = True
                return Matched, Reset, takePrev
            if (input == self.regexp[self.state_number]):

                self.state_number+=1

                if (self.state_number==self.states):
                    Matched=True
                    self.state_number=0
                    self.current_state=self.regexp[0]
                    self.prev_state=self.current_state
                    return Matched, Reset, takePrev

                self.current_state=self.regexp[self.state_number]
                self.prev_state=self.current_state
            else:
                self.prev_state=self.current_state
                self.current_state=self.regexp[self.state_number]
                Matched=True
                takePrev=True
                self.state_number=0
                self.current_state=self.regexp[0]
                self.prev_state=self.current_state
                return Matched,Reset,takePrev

    elif(self.current_state=="."):
        self.state_number+=1
        if (self.state_number==self.states):
            #if succesfully matched
            Matched = True
            #resetting the index of the state, and current state back to null
            self.state_number=0
            self.current_state=self.regexp[0]
            return Matched, Reset, takePrev
        #updating the current state
        self.prev_state=self.current_state
        self.current_state=self.regexp[self.state_number]
    return Matched, Reset, takePrev
```



A drawn representation of how matching would work using FSM.

- When the state 4 is reached, output is 1, which indicates that a match is found.
- When + wildcard is used, a small difference would be that, whenever the preceding character is, it stays in the present state, and when input is different, it goes back to initial state, or next state (if input matches)
- When . wildcard character is used, it will always go to next state no matter what input.
- when * wildcard is used, preceding character need not be there, so, the state before that also has output 1 and is considered a match.

### 2.4.3 Execution

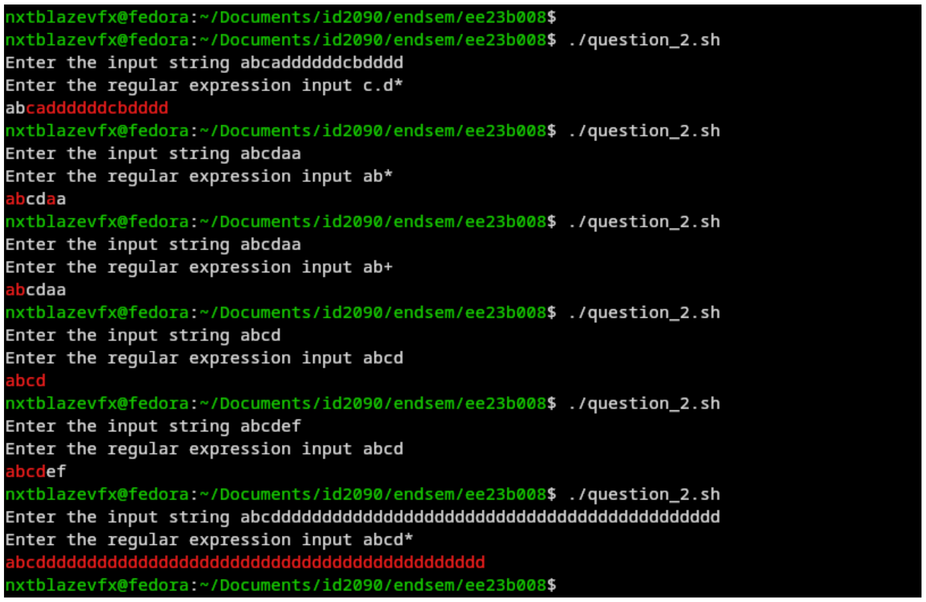
```python
def color_string(input_string):
    colored_string = ''
    colored_string += '\033[91m'  # ANSI escape code for red color
    (you can adjust color as needed)
    colored_string += input_string  # Add the colored part
    colored_string += '\033[0m'  # Reset color to default
    return colored_string

#taking my inputs
input_string = input("Enter the input string ")
```

```
regex_string = input("Enter the regular expression input ")

#calling my class
fsm = FSM(regex_string)
# print(fsm.states)
initial_match_idx=0


matches = []


for i in range(len(input_string)):
    # print(input_string[i],fsm.current_state, fsm.prev_state, fsm.
    state_number)
    Matched, Reset, takePrev = fsm.state_transition(input_string[i
    ])
    if Reset == True:
        initial_match_idx=i+1
    if Matched == True:
        if takePrev==False:
            matches.append((initial_match_idx, i))
            # print("Matcheda", initial_match_idx, i)
            # print(color_string(input_string,matches))
            initial_match_idx=i+1
        else:
            matches.append((initial_match_idx, i-1))
            # print("Matchedb", initial_match_idx, i-1)
            # print(color_string(input_string,matches))
            initial_match_idx=i

    if (i==len(input_string)-1):
        if fsm.current_state=="*" or fsm.current_state=='+':
            matches.append((initial_match_idx, i))
        if  fsm.state_number+1<fsm.states:
             if  fsm.regexp[fsm.state_number+1]=="*":
                matches.append((initial_match_idx, i))

            # print("Matchedc", initial_match_idx, i)
            # print(color_string(input_string,matches))

color_indices=[]
for i in matches:
    for j in range(i[0],i[1]+1):
        color_indices.append(j)


for i in range(len(input_string)):
    if i in color_indices:
        print(color_string(input_string[i]), end='')
    else:
        print(input_string[i], end='')
```

### 2.4.4 Supported Wildcards

'*': Matches zero or more occurrences of the previous character. '.': Matches any single character. '+': Matches one or more occurrences of the previous character.

### 2.5 Outputs



```
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$ ./question_2.sh
Enter the input string abcaddddddcbdddd
Enter the regular expression input c.d*
abcaddddddcbdddd
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$ ./question_2.sh
Enter the input string abcdaa
Enter the regular expression input ab*
abcdaa
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$ ./question_2.sh
Enter the input string abcdaa
Enter the regular expression input ab+
abcdaa
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$ ./question_2.sh
Enter the input string abcd
Enter the regular expression input abcd
abcd
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$ ./question_2.sh
Enter the input string abcdef
Enter the regular expression input abcd
abcdef
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$ ./question_2.sh
Enter the input string abcddddddddddddddddddddddddddddddddddddddddddddd
Enter the regular expression input abcd*
abcddddddddddddddddddddddddddddddddddddddddddddd
nxtblazevfx@fedora:~/Documents/id2090/endsem/ee23b008$
```

These testcases show that all the required criterias are met. Direct matches, Wildcard Characters (+,*,.), Any number of input length, multiple wildcard characters in one regex, and multiple matches in same testcase. Multiple corner cases were checked and rectified.

# 3 References:

- ChatGPT for few parts of first code.
- Wikipedia
- Geeks for Geeks for algorithms
- Github repos for regex (none of them are working, so the whole regex parser was coded on my own)