Final Report - End Semester Examination - ID2090

Krishna Murari Chivukula

May 22, 2024

Abstract

This is a report on the take home end semester examination for the course ID2090.

1 Path Planning

1.1 Problem Abstract

This problem dealt with path planning using an algorithm known as **Rapidly Exploring Random Trees**, or RRT. Essentially, RRT is a sampling-based path planning algorithm, often used in autonomous robotics systems when the map of the environment is available. I used **Python** for the implementation and **Matplotlib** for visualisation.

1.2 An Insight Into the Working of RRT:

RRT is a specific variation of robotics algorithms , essential for path planning. The fundamental algorithm of **RRT** is summarized as follows:

- Initialize an empty tree with the start node as the root and repeat until the goal node is reached or a maximum number of iterations is reached.
- This involves sampling a random point in the environment space under consideration and finding the nearest node in the tree to this random point.
- It is followed by extending the tree from the nearest node towards the random point by a small step size, creating a new node, and checking if it is valid.
- A valid path is when the connect between the randomly sampled point and the node in the tree dosen't pass through an obstacle in the environment.
- If it is valid, it should be added to the tree and connected to the nearest node. If it is close enough to the goal node, connect them and terminate the algorithm. Finally, trace back the path from the goal node to the start node in the tree.

1.3 Why RRT?

In this section we'll delve into a comparative measure between **RRT** and other path planning algorithms , namely **Dijkstra's** and **A***.

- **Dijkstra's Algorithm**:
  1. Dijkstra's algorithm is a classic pathfinding algorithm used to find the shortest path between nodes in a graph.
  2. It guarantees finding the shortest path from a start node to all other nodes in the graph.
  3. However, it explores nodes uniformly without considering any heuristic, which can lead to inefficient exploration in scenarios where the goal is known or where certain paths are more promising than others.
- **A* (A-star) Algorithm**:

1. A* builds upon Dijkstra's algorithm by incorporating a heuristic function that guides the search towards the goal node.
2. It combines the cost of reaching a node from the start with an estimate of the cost to reach the goal from that node, usually denoted by the heuristic function.
3. A* is more efficient than Dijkstra's algorithm in terms of exploration because it prioritizes paths that seem to be closer to the goal, thus reducing the search space. However, it's important to choose an admissible heuristic to ensure the algorithm finds the optimal solution.

- **RRT (Rapidly-exploring Random Tree)**:

1. RRT is a probabilistically complete algorithm commonly used in motion planning for robotics.
2. It randomly samples the search space and incrementally grows a tree structure towards the goal.
3. RRT is particularly effective in high-dimensional and complex spaces because it focuses exploration on areas that have not been adequately explored, thereby efficiently navigating complex environments.
4. Unlike Dijkstra's and A*, RRT does not guarantee optimality in finding the shortest path, but it often provides good solutions quickly, especially in dynamic or uncertain environments.

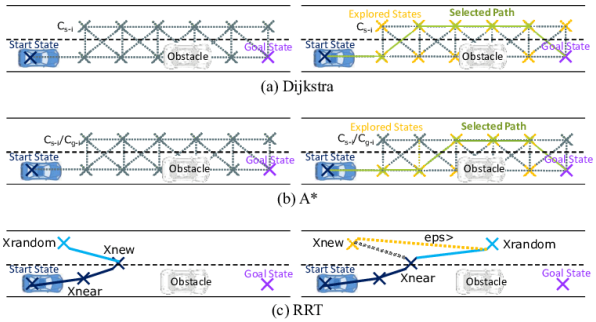


Figure 1: A Comparision Between The Algorithms

1.4 Pseudo Code for RRT Implementation:

```
Env //region that identifies success
Counter = 0 //keeps track of iterations
lim = n //number of iterations algorithm should run for
G(V,E) //Graph containing edges and vertices, initialized as empty
While counter < lim:
    Xnew  = RandomPosition()
    if IsInObstacle(Xnew) == True:
        continue
    Xnearest = Nearest(G(V,E),Xnew) //find nearest vertex
    Link = Chain(Xnew,Xnearest)
    G.append(Link)
    if Xnew in Env:
        Return G
Return G
```

1.5 Normal RRT Implementation

I simulated a rectangular environment , and placed 3-4 rectangular obstacles of dimensions around $\frac{1}{10}$th the environment dimensions at random locations. **RRT** was then implemented in this environment.

```python
#Importing essential libraries
import numpy as np
import matplotlib.pyplot as plt
#Defining a class to initialize the environment as a rectangle , with a certain width and height
class Environment:
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height

    def contains(self, point):
        return (self.x <= point[0] <= self.x + self.width and
                self.y <= point[1] <= self.y + self.height)
#Defining a class to generate nodes
class Node:
    def __init__(self, point, parent=None):
        self.point = np.array(point)
        self.parent = parent
#Defining class for RRT parameters
class RRT:
    def __init__(self, start, goal, bounds, obstacles, max_iters=1000, step_size=10):
        self.start = Node(start)
        self.goal = Node(goal)
        self.bounds = bounds
        self.obstacles = obstacles
        self.max_iters = max_iters
        self.step_size = step_size
        self.nodes = [self.start]
#Generating Random points in vicinity of current node of tree
    def rand_point(self):
        return [np.random.uniform(self.bounds[0], self.bounds[0] + self.bounds[2]),
                np.random.uniform(self.bounds[1], self.bounds[1] + self.bounds[3])]
#Returning nearest distance node
    def nearest_node(self, point):
        distances = [np.linalg.norm(np.array(node.point) - np.array(point)) for node in self.nodes]
        return self.nodes[np.argmin(distances)]
#Creating new point at this nearest distance
    def new_point(self, nearest, target):
        direction = np.array(target) - np.array(nearest.point)
        magnitude = np.linalg.norm(direction)
        if magnitude <= self.step_size:
            return target
        else:
            return nearest.point + (direction / magnitude) * self.step_size
#Check if path lies outside obstacle
    def check_collision(self, point):
        for obstacle in self.obstacles:
            if obstacle.contains(point):
                return True
```

```
        return False
#Create a path connecting this node and point
    def create_path(self):
        for _ in range(self.max_iters):
            random_point = self.rand_point()
            nearest_node = self.nearest_node(random_point)
            new_point = self.new_point(nearest_node, random_point)
            if not self.check_collision(new_point):
                new_node = Node(new_point, nearest_node)
                self.nodes.append(new_node)
                if np.linalg.norm(np.array(new_point) - np.array(self.
    goal.point)) < self.step_size:
                    self.goal.parent = new_node
                    return True
        return False
#Backtrack the path from the goal node to the start node in the tree
    def backtrack_path(self):
        path = []
        current = self.goal
        while current is not None:
            path.append(current.point)
            current = current.parent
        return path[::-1]
#Plotting the entire layout
    def plot(self):
        plt.figure(figsize=(8, 8))
        plt.axis('equal')
        plt.plot(self.start.point[0], self.start.point[1], 'go',
    markersize=10)
        plt.plot(self.goal.point[0], self.goal.point[1], 'ro',
    markersize=10)
        for obstacle in self.obstacles:
            plt.gca().add_patch(plt.Rectangle((obstacle.x, obstacle.y)
    , obstacle.width, obstacle.height, color='gray'))
        for node in self.nodes:
            if node.parent:
                plt.plot([node.point[0], node.parent.point[0]], [node.
    point[1], node.parent.point[1]], 'k-')
        path = self.backtrack_path()
        if path:
            plt.plot([point[0] for point in path], [point[1] for point
     in path], 'b-')
        plt.xlim(self.bounds[0], self.bounds[0] + self.bounds[2])
        plt.ylim(self.bounds[1], self.bounds[1] + self.bounds[3])
        plt.show()

# We Define start and goal points, which can be modified appropriately
start = (50, 50)
goal = (450, 450)
# x, y, width, height respectively , for all below parameters
bounds = (0, 0, 500, 500)
#We define width and height of parameters
obstacles = [Environment(110, 150, 50, 50), Environment(200, 250, 70,
    30), Environment(400, 400, 80, 40)] #

rrt = RRT(start, goal, bounds, obstacles)
```

```
#This caters to whether the algorithm was able to obtain a path
    between the necessary positions
if rrt.create_path():
    print("Successful Path Obtained")
    rrt.plot()
else:
    print("No Path Detected")
```

Listing 1: Normal RRT Implementation

The above implementation gave me an **RRT** as depicted below. A point to be noted is that the path was obtained only after multiple tries for the same start and end goals and obstacle positions. For the first few tries, it resulted in no path detected. This result shows us that **RRT** generates different paths every time we run it. This fact is obvious since we randomly sample points everytime we simulate a run.

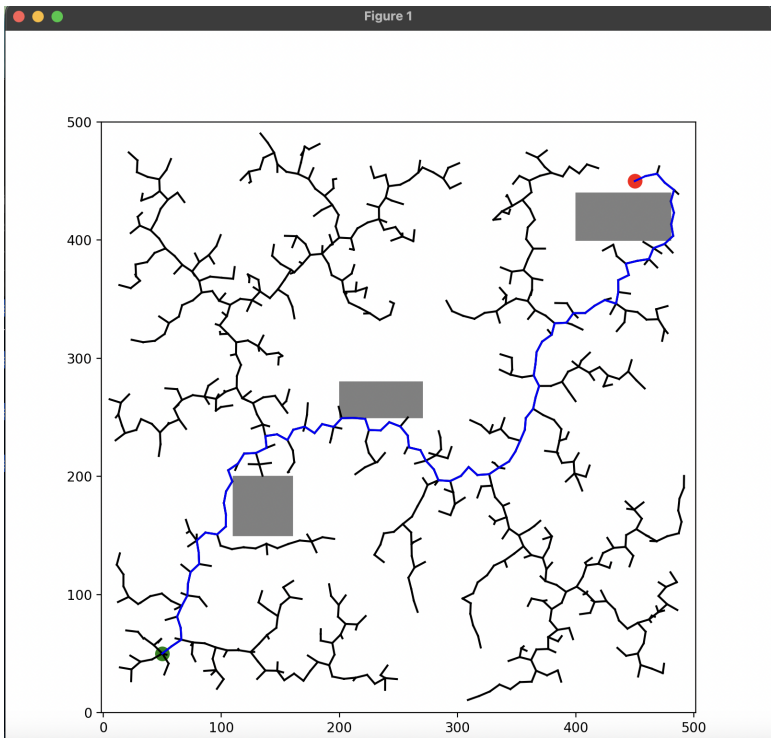


Figure 2: Normal **RRT** Implementation

1.6 Trajectory Smoothing:

Since **RRT** is a random sampling algorithm, the path produced by it is not smooth, which in real life is not a feasible trajectory. In order to perform trajectory smoothing , I implemented a function in the previous code titled `smooth_path`. This uses **Cubic Spline Interpolation** to smoothen the path made by the normal implementation. To use the `CubicSpline` function , I had to import it from `scipy.interpolate`. The remaining implementation remains the same.

1.6.1 A Small Note On Cubic Spline Interpolation:

I learnt about this technique from my friends who had taken the course EE1103-Numerical Methods in their 1st semester.
Cubic spline interpolation is a method used to interpolate(essentially connect) points within a given set of data points. It involves constructing a piecewise cubic polynomial that passes through each data point and has continuous first and second derivatives at each point. This results in a smooth curve

that approximates the data. Thus , we smoothen the otherwise somewhat "discrete" trajectory that isn't smooth.

```python
#The class and function declaration remains the same until this point , after which we declare a function to smoothen the path given
# by RRT. For this specific case I used Cubic Spline Interpolation.
    def smooth_path(self, path):
        if len(path) < 4:
            return path

        x = [point[0] for point in path]
        y = [point[1] for point in path]

        t = np.linspace(0, 1, len(path))
        cs_x = CubicSpline(t, x)
        cs_y = CubicSpline(t, y)

        new_t = np.linspace(0, 1, len(path) * 10)
        smooth_path = np.column_stack((cs_x(new_t), cs_y(new_t)))

        return smooth_path
```

Listing 2: Smoothened RRT Implementation

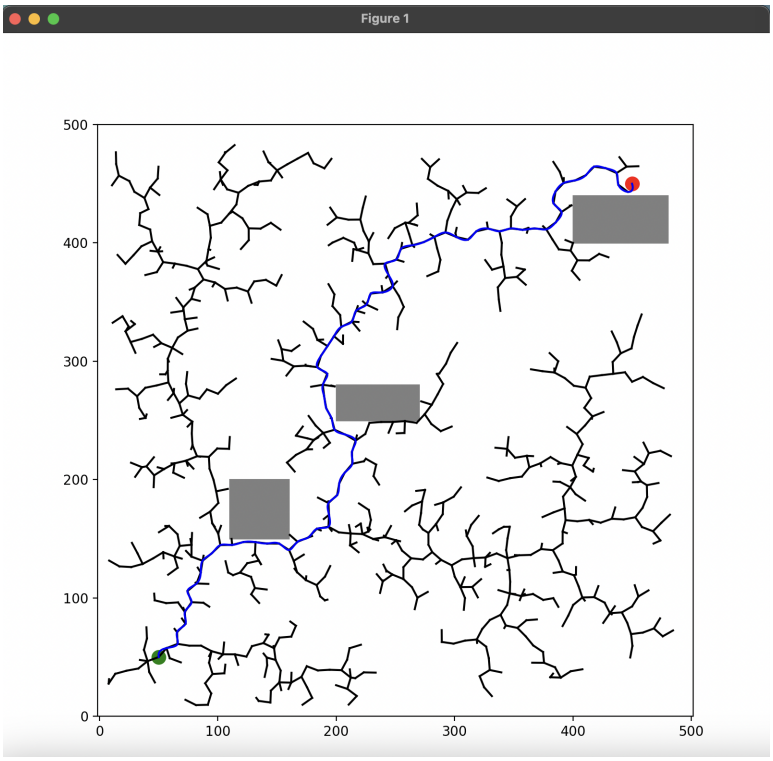


Figure 3: Smoothened **RRT** Implementation

1.7 RRT Implemented With Spacing

For this again , I just modified the `random_point` function to sample the random points slightly away from the obstacles under consideration. This was done by introducing a parameter `self.spacing`. Overall , the path seemed to shift away from the obstacles. I changed the location of obstacles for this trial.

```python
# Modified RRT implementation with obstacle spacing consideration
class RRTWithSpacing(RRT):
    def __init__(self, start, goal, bounds, obstacles, spacing, max_iters=1000, step_size=10):
        super().__init__(start, goal, bounds, obstacles, max_iters, step_size)
        self.spacing = spacing

    def rand_point(self):
        while True:
            x = np.random.uniform(self.bounds[0] + self.spacing, self.bounds[0] + self.bounds[2] - self.spacing)
            y = np.random.uniform(self.bounds[1] + self.spacing, self.bounds[1] + self.bounds[3] - self.spacing)
            if all(not obstacle.contains([x, y]) for obstacle in self.obstacles):
                return [x, y]
```

Listing 3: Spaced RRT Implementation

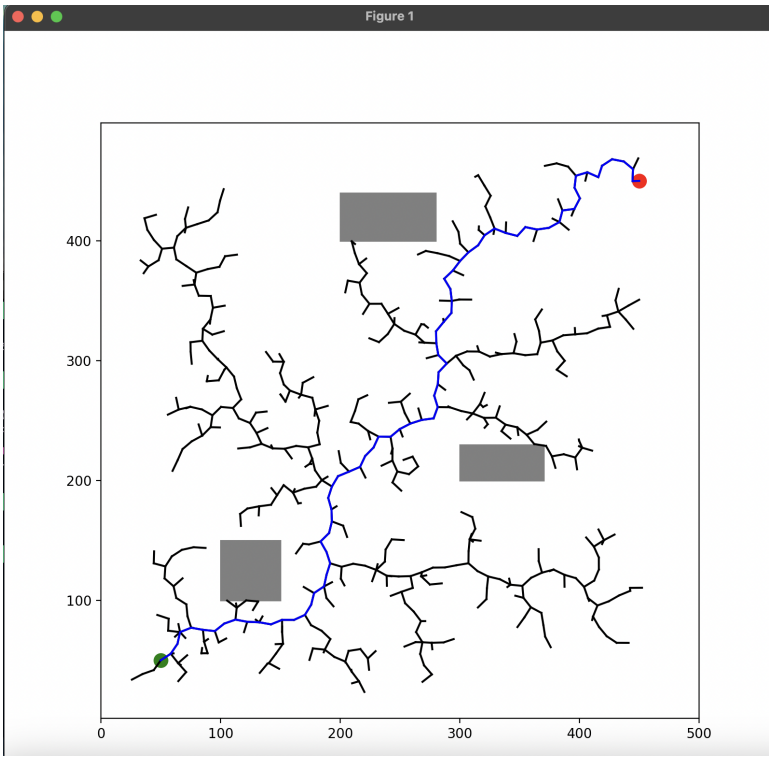


Figure 4: Spaced **RRT** Implementation

1.8 Greedy RRT:

We define a "greedy" approach as one that would involve choosing points nearer to the goal more likely. For more likely, I modified the `rand_point` method to bias the sampling towards the goal. This was done by adjusting the probability distribution from which random points are sampled. Essentialy, wherever I defined the function to generate random points near the node of a tree there I biased the point generation closer to the end point .I defined this bias standard by taking probability of exactly choosing the goal point itself , and tweaking that value. I used the Euclidean , L2 norm for distance calculation.

- I introduced a probability `probab_goal` which determines the chance of directly sampling the goal.
- If a random value is less than `probab_goal`, the function returns the goal point.
- Otherwise, it samples a random point but biases it towards the goal. This biasing is done by calculating a connect from the randomly sampled point to the goal and then scaling this connect by a factor (0.1 in this case - `probab_goal`) before adding it to the sampled point.
- Below is the modified `rand_point` method. The remaining implementation remains same.

```
    #Generating Random points in vicinity of current node of tree
    def rand_point(self):
        # Probability of choosing goal point
        probab_goal = 0.1  # This value we can adjust , based of how greedy we can be
        if np.random.uniform() < probab_goal:
            return self.goal.point
        else:
            # Sample random points biased towards the goal
            # Weighted average of current point and goal point (L2 NORM)
            rand_x = np.random.uniform(self.bounds[0], self.bounds[0] + self.bounds[2])
            rand_y = np.random.uniform(self.bounds[1], self.bounds[1] + self.bounds[3])
            greedy_point = np.array(self.goal.point) - np.array([rand_x, rand_y])
            # Scaling the distance to control the bias parameter
            greedy_pt = np.array([rand_x, rand_y]) + 0.1 * greedy_pt
            return greedy_pt.tolist()
```

Listing 4: Greedy RRT Implementation

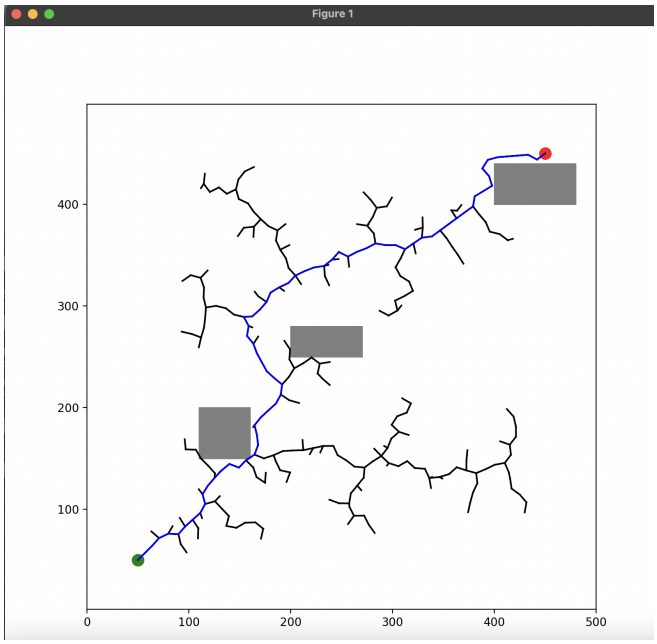


Figure 5: Greedy **RRT** Implementation

- Notice how the path seems much less branched out than normal **RRT**, since now we have defined an additional constraint on the way the random points are sampled. Hence , the tree has some form of initial knowledge on the sampling process and does not need to branch more.

1.9 Comparison Plot:

- This section deals with a comparison between the different variants of **RRT**, namely , the normal implementation , implementation with spacing and the greedy implementation of **RRT**.
- Essentially , I included the classes for all methodologies in a single code and simulated multiple runs. The function `simulate_rrt` was called to check the total number of nodes involved in each implementation.
- This was then plotted in a bar-chart as depicted below.

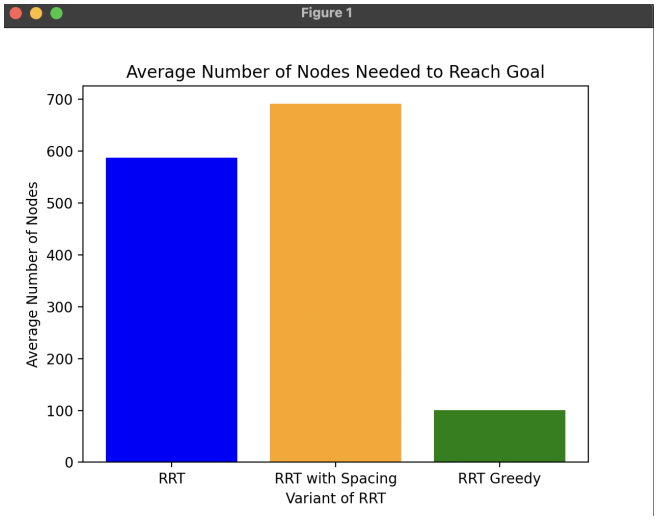


Figure 6: Comparision of **RRT** Implementation Types

- The greedy implementation took the least number of nodes as expected and seen from the above branching in **Subsection 1.8**. This was followed by normal **RRT** implementation and **RRT** with spacing took the most number of nodes required. This is because the branching goes wider around the obstacles involved which necessitates a greater number of nodes.
- The function `simulate_rrt` is depicted below.

```
def simulate_rrt(rrt_class, start, goal, bounds, obstacles,
 spacing=None, max_iters=1000, step_size=10, num_simulations
 =1000):
    num_nodes_list = []
    for _ in range(num_simulations):
        rrt = rrt_class(start, goal, bounds, obstacles, max_iters,
    step_size) if spacing is None else rrt_class(start, goal,
 bounds, obstacles, spacing, max_iters, step_size)
        if rrt.generate_path():
            num_nodes_list.append(len(rrt.nodes))
    return np.mean(num_nodes_list)


```

Listing 5: Greedy RRT Implementation

2 Automata

2.1 Problem Abstract

This problem dealt with the implementation of a basic regex engine that converts a regular expression into a basic finite state machine(FSM), in which we pass an input string and match the expression to specific portions of the string. The regex engine supports direct matches, any number of characters (*), wildcard characters (.), and combinations of these.

2.2 Implementation

2.2.1 State Class

The State class represents a state in the FSM. Each state has a set of transitions to other states based on input characters and epsilon (ϵ) transitions which do not consume any input. An epsilon transition allows an automation to change its state spontaneously, i.e. without consuming any input symbol.

```
class State:
    def __init__(self):
        self.transitions = {}
        self.epsilon_transitions = []

    def add_transition(self, char, state):
        if char not in self.transitions:
            self.transitions[char] = []
        self.transitions[char].append(state)

    def add_epsilon_transition(self, state):
        self.epsilon_transitions.append(state)

    def get_transitions(self, char):
        return self.transitions.get(char, []) + self.transitions.get(
   None, [])

    def get_epsilon_transitions(self):
        return self.epsilon_transitions
```

Listing 6: State Class

2.2.2 FSM Class

The FSM class represents the overall nature of the finite state machine. It has several methods to add transitions, build the FSM from a regex, and match strings against the FSM.

- Initialization (`__init__`): The FSM class initializes with a `start_state` and an `accept_state`, both instances of the State class.
- Add Transitions (`add_transition` and `add_epsilon_transition`):
  1. `add_transition(from_state, to_state, char)`: Adds a transition from `from_state` to `to_state` on reading char.
  2. `add_epsilon_transition(from_state, to_state)`: Adds an epsilon transition (a transition without consuming any input) from `from_state` to `to_state`.
- Build from Regex (`build_from_regex`): Constructs the FSM based on a given regex pattern.
- Match (`match`): Determines whether a given string matches the pattern described by the FSM.

Building FSM from Regex (`build_from_regex`):
This method constructs the FSM to recognize a given regex pattern.
Overall analysis:

- Initialization: Starts with the `start_state` and iterates over each character in the regex.
- Case Handling:
  1. If the character is *, it represents zero or more occurrences of the previous character/state. It creates epsilon transitions that loop back to the current state.
  2. For each regular character or wildcard (.), it creates new states and transitions accordingly.
  3. If a character is followed by *, it creates epsilon transitions between the current and next state, allowing zero or more repetitions.
- Finalization: Ensures that the `accept_state` can be reached from the current states via epsilon transitions.
- Matching Strings (`match`): This method checks if any substring of the given string matches the pattern defined by the FSM.
- Epsilon Closure (`epsilon_closure`): Computes the epsilon closure of a set of states, which includes all states reachable via epsilon transitions.
- Matching Process:
  1. Iterates over each possible starting position in the string.
  2. For each starting position, computes the set of current states including epsilon transitions. For each character in the string from the current starting position, updates the current states based on the transitions for the character and includes epsilon transitions.
  3. If the `accept_state` is reached, records the span (start and end positions) where the match is found.

```
class FSM:
    def __init__(self):
        self.start_state = State()
        self.accept_state = State()

    def add_transition(self, from_state, to_state, char):
        from_state.add_transition(char, to_state)

    def add_epsilon_transition(self, from_state, to_state):
        from_state.add_epsilon_transition(to_state)

    def build_from_regex(self, regex):
        current_states = [self.start_state]
        i = 0
        while i < len(regex):
            char = regex[i]
            if char == '*':
                # * means zero or more of the previous state
                for state in current_states:
                    self.add_epsilon_transition(state, state)
                i += 1
                continue

            next_states = [State() for _ in current_states]
            for from_state, to_state in zip(current_states,
    next_states):
                if char == '.':
                    self.add_transition(from_state, to_state, None)  #
     Wildcard transition
                else:
```

```
                    self.add_transition(from_state, to_state, char)
                if i + 1 < len(regex) and regex[i + 1] == '*':
                    self.add_epsilon_transition(to_state, from_state)
                    self.add_epsilon_transition(from_state, to_state)
            current_states = next_states
            i += 1

        for state in current_states:
            self.add_epsilon_transition(state, self.accept_state)

    def match(self, string):
        def epsilon_closure(states):
            stack = list(states)
            closure = set(states)
            while stack:
                state = stack.pop()
                for next_state in state.get_epsilon_transitions():
                    if next_state not in closure:
                        closure.add(next_state)
                        stack.append(next_state)
            return closure

        match_spans = []
        n = len(string)

        for i in range(n):
            current_states = epsilon_closure({self.start_state})
            j = i
            while j < n:
                new_states = set()
                for state in current_states:
                    new_states.update(state.get_transitions(string[j])
)
                    new_states.update(state.get_transitions(None))
                current_states = epsilon_closure(new_states)
                if self.accept_state in current_states:
                    match_spans.append((i, j + 1))
                    break
                j += 1

        return match_spans
```

Listing 7: FSM Class

2.2.3 Highlighting Matches

The `highlight_match` function takes the string and a list of match spans (tuples of start and end indices) and constructs the highlighted string using ANSI escape codes. I have specifically used to ANSI escape code for highlighting the matched part in red.

ANSI escape codes are sequences of characters that alter the appearance and behavior of text in terminal output. These sequences begin with the escape character \033 followed by various codes that specify the desired formatting.

```python
def highlight_match(string, match_spans):
    highlighted_string = ""
    start_highlight = "\033[31m"
    end_highlight = "\033[0m"

    last_end = 0
    for start, end in match_spans:
        highlighted_string += string[last_end:start]
        highlighted_string += start_highlight + string[start:end] +
    end_highlight
        last_end = end

    highlighted_string += string[last_end:]
    return highlighted_string
```

Listing 8: Highlighting Matched Portion

2.2.4 Testing the Engine:

The `test_regex_engine` function demonstrates the regex matching and highlights the matched portions in the test strings.

```python
def test_regex_engine():
    regex = "a*c*" #Can be modified appropriately
    fsm = FSM()
    fsm.build_from_regex(regex)

    test_strings = ["baacc"] #Can be modified , giving multiple
    entries in the list
    for s in test_strings:
        match_spans = fsm.match(s)
        highlighted = highlight_match(s, match_spans)
        print(f"{highlighted}") #Prints highlighted portion

if __name__ == "__main__":
    test_regex_engine()
```

Listing 9: Final Output Portion

2.2.5 Sample Output:

For the regex 'a*c*' and the input string 'baacc', the following output was obtained. The machine was tested for various different matches and worked appropriately for all of them , including '*' and '.' operations.

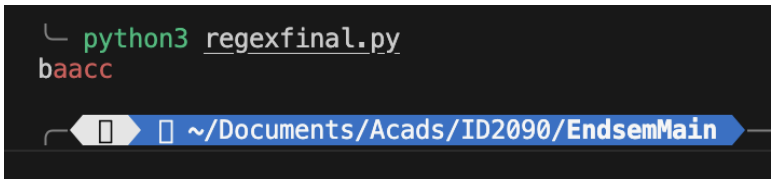


Figure 7: Sample Output

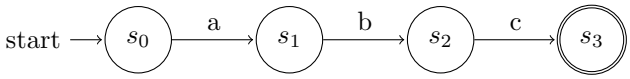
2.3 Finite State Machine(FSM) Diagrams

This section shows the Finite State Machine Diagrams for each of the cases covered by the regex engine. Direct matches, multiple characters and matches as well as wildcard characters were dealt with. I used `tikzpicture` for displaying the FSM in a graph-like fashion with nodes and branches. Finite State Machines (FSMs) are models used to design and analyze the behavior of particular systems. They consist of a finite number of states, transitions between these states, and actions. FSM diagrams are graphical representations of these machines, offering a visual way to understand and design systems with distinct states and transitions involved.

- The states are represented by circles and signify a particular condition in the system.
- Transitions are represented by arrows connecting states. They indicate the movement from one state to another based on certain conditions or inputs.
- The inital state is typically marked with an arrow pointing to it from nowhere or with a distinct marker, usually titled "start". It denotes the starting point of the FSM. Final states are represented by double circles.
- Input Alphabet is a set of symbols that trigger transitions. Every transition is labeled with an input symbol from this alphabet.

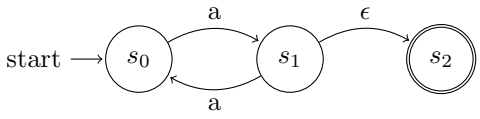
2.3.1 Direct Matches

This type handles exact, direct matches to character.For example , using the regex **abc**.



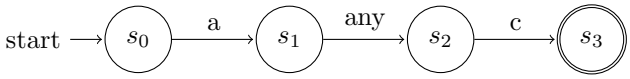
2.3.2 Any Number of Characters (*)

This case handles the * operator, which matches zero or more occurrences of the preceding element, e.g., the regex **a***.



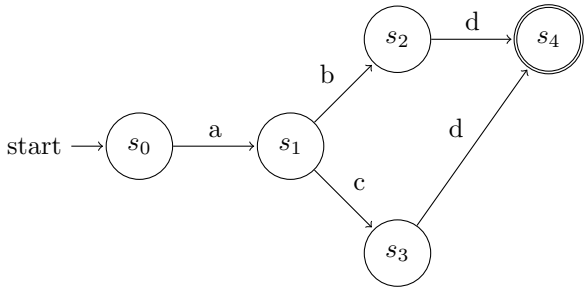
2.3.3 Wildcard Character (.)

This case handles the wildcard character '.' , which matches any single character, e.g., the regex **a.c**.



2.3.4 Multiple Matches

This case handles multiple matches or a combination of patterns, e.g., the regex **a(b—c)d**.



3 References:

3.0.1 Path Planning Using RRT:

- A comparision between RRT, Dijkstra's and A* was derived from here - https://www.researchgate.net/figure/Illustrations-of-the-processes-of-a-Dijkstra-b-A-and-c-RRT_fig4_333124691.
- For understanding RRT and the general idea behind it's working as well as the pseudo-code involved in the implementation, I referred to the Medium blog - https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378
- Wikipedia article on RRT - https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree#:~:text=A%20rapidly%20exploring%20random%20tree,building%20a%20space%2Dfilling%20tree.
- Youtube Video on RRT - https://www.youtube.com/watch?v=0b3BIJkQJEw

3.0.2 Automata Using Regex:

- Documentation of the FSM package- https://pyfsm.readthedocs.io/en/latest/reference/fsm.html
- Regex Documentation - https://docs.python.org/3/library/re.html.
- Finite State Machines - https://brilliant.org/wiki/finite-state-machines/#:~:text=A%20finite%20state%20machine%20(sometimes,state%20automata%20generate%20regular%20languages.
- Reference For Coding A Regex Engine from scratch - https://www.youtube.com/watch?v=fgp0tKWYQWY.

Apart from this , I used StackOverFlow, ChatGPT and a bunch of other websites for references, fixing bugs and tackling errors in the logic pertaining to code. Furthermore , I used these platforms to learn more on implementation of RRT, FSMs as well as Regex and how each of these actually work.