Final Reflection

D V Anantha Padmanabh
ME23B012

May 2024

Task 1

RRT

1.1 Introduction

1.1.1 RRT

Rapidly-exploring Random Tree (RRT) is a path planning algorithm used in robotics and other fields. It works by randomly sampling points (nodes) and building a tree like path. This algorithm is good enough and fast and is used when the most efficient path is not required.

1.2 Implementation

1.2.1 Overview

The RRT algorithm here is implemented using python. It was chosen not to use any of Python OOPs other than the node class. It creates a predefined environment in matplotlib, containing start, end and obstacles as specified in the problem statement.

The implementation has the given features which can be toggled:

- **Greediness**: The RRT algorithm can be executed in a biased way so that nodes towards the goal are sampled more.
- **Clearance**: The RRT can find nodes such that are a fixed distance away from the obstacles such that there is a clearance distance.
- **Smoothing**: The path traced by the RRT with line segments can be approximated to a high degree polynomial for smooth fit.

1.2.2 Code Reflection

The functions used in the script are explained briefly below:

- **Node class**: The node class defines the node object(the points) having the attributes *position* and *parent* which stores the coordinates and parent node which the node was generated from respectively.
- ***is_collision_free* function**: this function checks if the nodes in the path are not on the obstacles and also implements the ***min_distance_to_obstacle*** function which will be explained separately.
- ***reconstruct_path* function**: this function constructs the path as a list of numpy arrays having the coordinates by backtracking from the goal to the start.
- ***visualize_rrt* function**: this function, using matplotlib creates the environment with obstacles and traces the path and all the nodes.

***rrt* Function**

The `rrt` function is the core of the program, implementing the RRT algorithm. It starts by initializing a tree with the start node. Then, it enters a loop where it samples a random point in the environment. The function then identifies the nearest node in the tree to this random point. It moves from this nearest node towards the random point, taking a step size of 1 while ensuring the path is collision-free. This process continues until the goal is reached or the maximum number of iterations is hit.

***smooth_path* Function**

This function take the nodes in the path and fits a high degree polynomial(15 here to be precise) along it to get a smooth trajectory. Numpy functions were used for doing it as seen in the script. The smoothing is optional and can be set to `True/False`

***min_distance_to_obstacle* Function**

This function when given a value('None' if not being used) ensures that the nodes are a fixed distance away from the obstacles. However the current implementation doesn't ensure that the line segments between nodes and the fit curve is also farther than `min_distance_to_obstacle`.

***biased_sampling* Function**

This function implements the greedy RRT algorithm. It does it by choosing the random point to near the goal by a bias factor so that the direction of travel is closer to the goal.

1.2.3 The Complete Script

```
import numpy as np
import matplotlib.pyplot as plt


# Node Class
class Node:
    def __init__(self, position):
        self.position = position
        self.parent = None


# Greedy Sampling
def biased_sampling(goal, bias):
    if np.random.random() < bias:  # Bias towards the goal
        return goal
    else:
        return np.random.uniform([0, 0], [width, height])


# Define the minimum distance to be maintained from the obstacles
def min_distance_to_obstacle(point, obstacle_positions, obstacle_sizes):
    min_distance = float("inf")
    for obs_pos, obs_size in zip(obstacle_positions, obstacle_sizes):
        dx = max(obs_pos[0] - point[0], 0, point[0] - obs_pos[0] - obs_size[0])
        dy = max(obs_pos[1] - point[1], 0, point[1] - obs_pos[1] - obs_size[1])
        min_distance = min(min_distance, np.sqrt(dx * dx + dy * dy))
    return min_distance
```

```
# Check if the path between nodes do not intersect with the obstacles
def is_collision_free(
    start, end, obstacle_positions, obstacle_sizes, min_obstacle_dist
):
    for obstacle_position, obstacle_size in zip(obstacle_positions, obstacle_sizes):
        if not (
            end[0] < obstacle_position[0]
            or start[0] > obstacle_position[0] + obstacle_size[0]
            or end[1] < obstacle_position[1]
            or start[1] > obstacle_position[1] + obstacle_size[1]
        ):
            return False

        # Check if the new node is at least min_obstacle_dist away from all obstacles
        if min_obstacle_dist is not None:
            dx = max(
                obstacle_position[0] - end[0],
                0,
                end[0] - obstacle_position[0] - obstacle_size[0],
            )
            dy = max(
                obstacle_position[1] - end[1],
                0,
                end[1] - obstacle_position[1] - obstacle_size[1],
            )
            distance = np.sqrt(dx * dx + dy * dy)
            if distance < min_obstacle_dist:
                return False

    return True


# Reconstruct the path from the start node to the goal node
def reconstruct_path(node):
    path = []
    while node.parent is not None:
        path.append(node.position)
        node = node.parent
    path.append(start)
    return path[::-1]


# Smoothen the path using a polynomial
def smooth_path(path):
    path = np.array(path)
    t = np.arange(path.shape[0])

    # Fit a polynomial to the x and y coordinates separately
    poly_x = np.polyfit(t, path[:, 0], deg=15)  # Degree of the polynomial is 15
    poly_y = np.polyfit(t, path[:, 1], deg=15)

    # Find the polynomial functions
    poly_x_fn = np.poly1d(poly_x)
    poly_y_fn = np.poly1d(poly_y)
    # Generate smooth t values
    smooth_t = np.linspace(t.min(), t.max(), 500)

    # Find the smooth path
    smooth_path = np.vstack([poly_x_fn(smooth_t), poly_y_fn(smooth_t)]).T

    return smooth_path


# RRT Algorithm
def rrt(
    start,
    goal,
    obstacle_positions,
    obstacle_sizes,
    smoothen=False,
    max_iterations=10000,
):
    global required_iterations
    tree = [Node(start)]
    all_paths = []  # Store all attempted paths

    for i in range(max_iterations):
        random_point = biased_sampling(goal, bias)
        nearest_node = min(
            tree, key=lambda node: np.linalg.norm(node.position - random_point)
        )
        direction = random_point - nearest_node.position
        direction = direction.astype(float)
        direction /= np.linalg.norm(direction)

        new_node_position = nearest_node.position + direction * 1
        # Step size is 1 because the direction is already normalized

        if is_collision_free(
            nearest_node.position,
            new_node_position,
            obstacle_positions,
            obstacle_sizes,
            min_obstacle_dist,
        ):
            new_node = Node(new_node_position)
            new_node.parent = nearest_node
            tree.append(new_node)
            all_paths.append(
                (nearest_node.position, new_node_position)
            )  # Store the attempted path

            if np.linalg.norm(new_node.position - goal) < 1:
                required_iterations = i
                path = reconstruct_path(new_node)
                return path, all_paths
    return None, all_paths


# Visualize the RRT
def visualize_rrt(
    start,
    goal,
    obstacle_positions,
    obstacle_sizes,
    path=None,
    all_paths=None,
    display_tree=False,
):
    plt.figure(figsize=(8, 8))
    plt.xlim(0, width)
    plt.ylim(0, height)
    plt.grid(False)
    plt.title("Rapidly-exploring Random Tree")

    for obstacle_position, obstacle_size in zip(obstacle_positions, obstacle_sizes):
        obstacle_rect = plt.Rectangle(
            obstacle_position, obstacle_size[0], obstacle_size[1], fc="gray"
        )
        plt.gca().add_patch(obstacle_rect)

    plt.plot(start[0], start[1], "go", markersize=10)
    plt.plot(goal[0], goal[1], "ro", markersize=10)

    if display_tree and all_paths is not None:
        for start, end in all_paths:
            plt.plot(
                [start[0], end[0]], [start[1], end[1]], "y.-"
            )  # Plot all attempted paths

    if path is not None:
        if smoothen:
            path = smooth_path(path)
        plt.plot([p[0] for p in path], [p[1] for p in path], "b-")

    plt.show()


# Define the intialising parameters
width = 40  # Dimensions of the environment
height = 30
start = np.array([1, 1])  # Start position
goal = np.array([35, 28])  # Goal position
# start, goal = goal, start

# The data was made on the following environment
# obstacle_positions = [(3, 3), (10, 10), (15, 15), (25, 20)]  # Obstacle positions
# obstacle_sizes = [(5, 5), (4, 4), (3, 7), (6, 2)]

# New environment
```

```
obstacle_positions = [(3, 3), (10, 10), (20, 3), (25, 20)]  # Obstacle positions
obstacle_sizes = [(5, 5), (4, 4), (3, 7), (6, 2)]

# #Custom Environment
# start = np.array([20, 1])
# goal = np.array([20, 29])
# obstacle_positions = [(1,1),(24,1)]
# obstacle_sizes = [(15,28),(15,28)]

min_obstacle_dist = None  # Minimum distance to be maintained from the obstacles
bias = 0  # Bias towards the goal
smoothen = False  # Smoothen the path

# Run the RRT algorithm
path, all_paths = rrt(start, goal, obstacle_positions, obstacle_sizes, smoothen)
# print(required_iterations)
visualize_rrt(
    start,
    goal,
    obstacle_positions,
    obstacle_sizes,
    path,
    all_paths,
    display_tree=True,
)
```

1.3 Observations and Conclusion

1.3.1 RRT in Action

Normal RRT

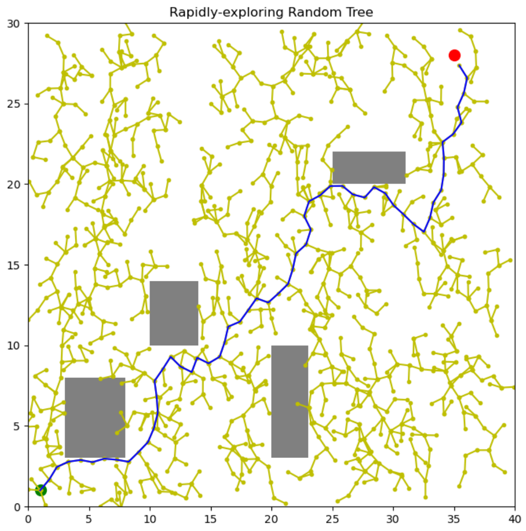


Figure 1.1: Normal RRT

RRT with clearance

`min_distance_to_obstacle = 1`

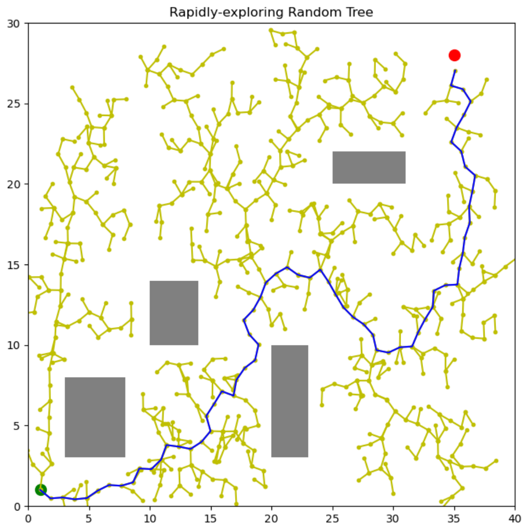


Figure 1.2: RRT with clearance

Greedy/Biased RRT

```
bias = 1
```

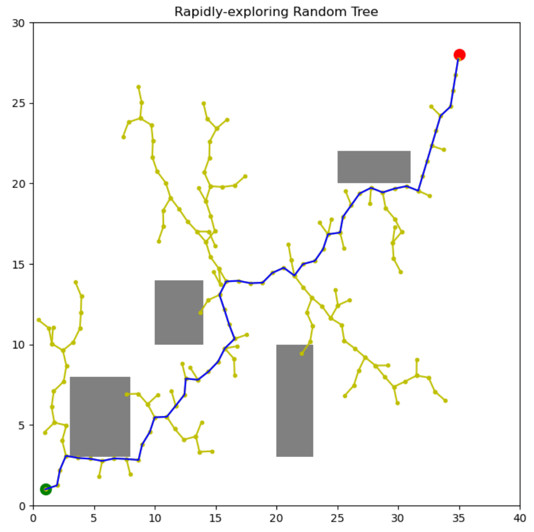


Figure 1.3: Biased RRT

Biased RRT with Clearance

```
bias = 1
min_distance_to_obstacle = 1
```

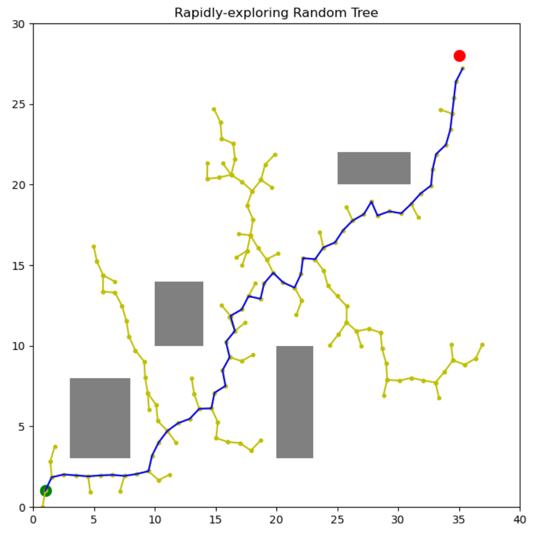


Figure 1.4: Biased RRT with Clearance

Smooth RRT

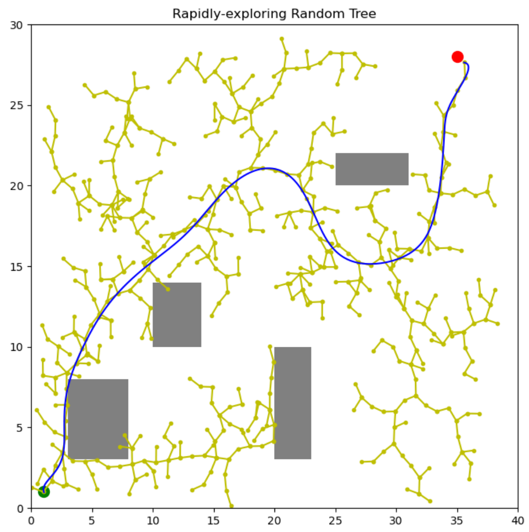


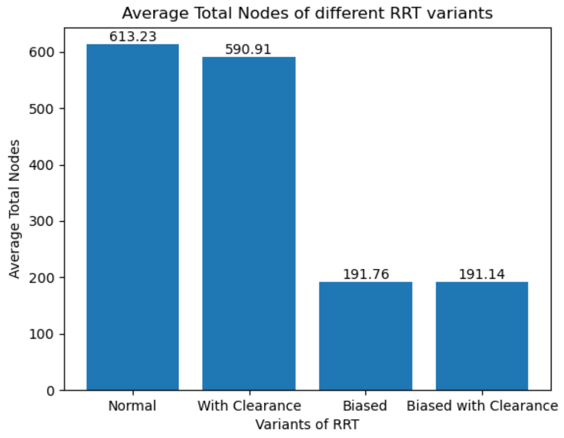
Figure 1.5: Smooth RRT

Note: The normal RRT explores a lot of the environment as it is not restricted to anything whereas the other variants restrict the exploration of the environment.

1.3.2 Data Plots

Average Total Nodes

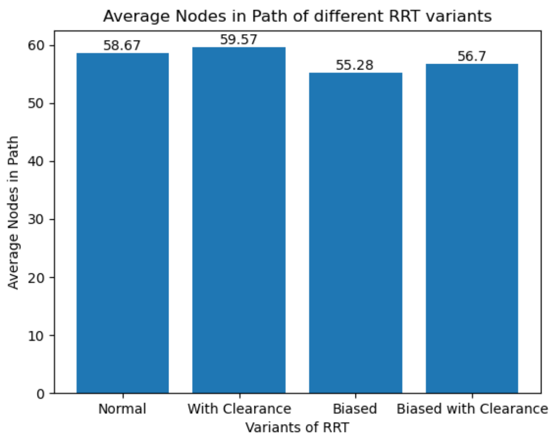
This bar graph shows the average total nodes i.e every node that has been checked of each variant.



The figure implies that the nodes explored are far less than normal RRT which is also evident from the previous figures.

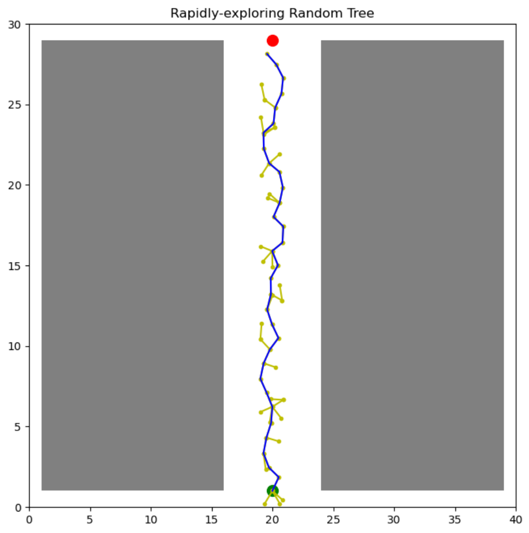
Average Nodes in Path

This bar graph shows the average nodes in the final path of each variant.



1.3.3 Custom Environment

The following shows the **RRT with clearance** implemented in the custom environment given in the problem statement.



Task 2

Automata

2.1 Introduction

2.1.1 Regex

Regular expressions(Regex) are a sequence of characters that define a search pattern for text data. Regex is powerful for tasks like web scraping to extract specific data like emails which have a unique pattern in them.

2.1.2 FSM

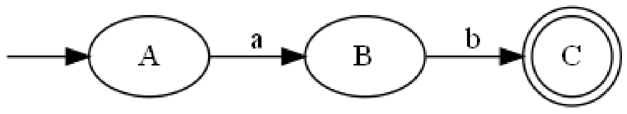
Finite State Machine(FSM) is a mathematical model for computations. It has a finite number of states which and will be in one of them at a given time. The state can 'transition' to another state for a specific input.

2.1.3 Regex as FSM

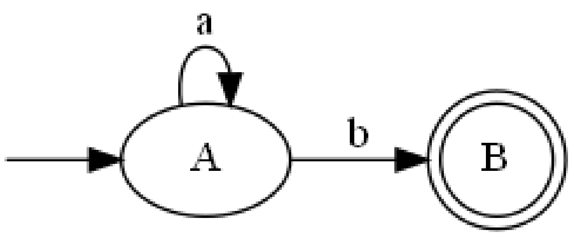
A regular expression can be converted into a FSM by specifying the states and transitions. This then can be applied on a string and the pattern can be matched by sending the characters into the fsm.

Graph Representation of FSM for different Regex

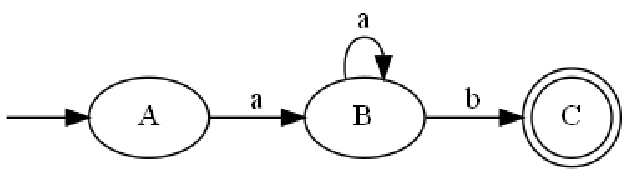
- /ab/



- /a*b/



- /a+b/



Images taken from link

2.2 Implementation

2.2.1 Overview

The implementation is done in python without using OOPs and any external libraries. The current implementation is a very basic one which includes only '*','+' and '.' characters and multiple matching.

- $*$ character when preceded by a main character checks for zero or more occurrences of the main character.
- $+$ character when preceded by a main character checks for one or more occurrences of the main character.
- . is the wildcard character and can represent any ASCII character.

2.2.2 Code Reflection

The functions in the script are briefly explained here.

- ***set_transition* function**: it is used to set the transitions for the FSM. It takes in the transitions dictionary, a character, and a state, and sets the transition for the given character to the given state.
- ***get_transition* function**: it is used to get the transition for a given character from the transitions dictionary. If the exact transition is not found, it tries to get the transition for the '.' wildcard character.
- ***colorize_matches* function**: it is used to colorize the matched parts in the input string. It takes in the input string and a list of matches, and returns the string with the matched parts colorized in red.

build_fsm Function

This function is used to build the FSM. It takes the regex pattern as input and builds an FSM based on it. The FSM is stored as a dictionary where each key is a state and the value is another dictionary representing the transitions from that state.

match Function

This function is used to check if a given string matches the regex pattern represented by the FSM. It takes in the FSM and the string to be checked. It starts from the initial state of the FSM and follows the transitions based on the characters in the string.

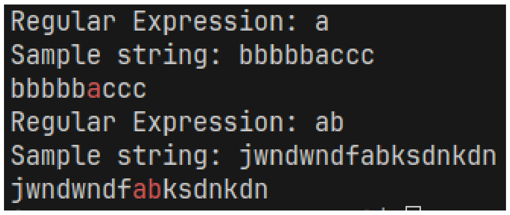
2.3 The Complete Script

```
# Set the transitions for the FSM
def set_transition(transitions, char, state):
    transitions[char] = state


# Get the transition for the FSM
def get_transition(transitions, char):
    # If the exact transition is not found, try with the '.' wildcard
    if char not in transitions:
        return transitions.get((char[0], "."))
    return transitions.get(char)


# Build the FSM from the pattern
def build_fsm(pattern):
    transitions = {}  # Transition table
    end_states = set()  # Set of end states
    current_state = 0  # Start state is 0
    next_char = None
    for i in range(len(pattern)):
        char = pattern[i]
        if char == "*" or char == "+":
            continue
        next_char = pattern[i + 1] if i + 1 < len(pattern) else None
        if next_char == "*":
            set_transition(transitions, (current_state, char), current_state)
            # Set the transition back to the current state
        elif next_char == "+":
            next_state = current_state + 1
            set_transition(transitions, (current_state, char), next_state)
            current_state = next_state
            set_transition(transitions, (current_state, char), current_state)
            """ Set transition to the next state and back to the current state
              as '+' means one or more"""
        elif char == ".":
            next_state = current_state + 1
            for c in range(256):
                set_transition(transitions, (current_state, chr(c)), next_state)
                """ Set the transition to any ASCII character
                    as '.' means any character"""
            current_state = next_state
        else:
            next_state = current_state + 1
            set_transition(transitions, (current_state, char), next_state)
            # Set the transition to the next state
            current_state = next_state
    end_states.add(current_state)
    return transitions, end_states


# Match the string with the FSM
def match(transitions, end_states, string):
    current_state = 0
    match_start = 0
    matches = []
    for i, char in enumerate(string):
        next_state = get_transition(transitions, (current_state, char))
        if next_state is None:
            next_state = get_transition(transitions, (current_state, None))
        if next_state is None:
            if current_state in end_states:
                matches.append((match_start, i))
            match_start = i + 1
            current_state = 0
        else:
            current_state = next_state
    if current_state in end_states:
        matches.append((match_start, len(string)))
    return matches


# Colorize the matched parts in the string
def colorize_matches(string, matches):
    colorized_string = ""
    last_end = 0
    for start, end in matches:
        colorized_string += (
            string[last_end:start] + "\033[31m" + string[start:end] + "\033[0m"
        )
        # Colorize the matched part in red
        last_end = end
    colorized_string += string[last_end:]
    return colorized_string


# Get user input
pattern = input("Regular Expression: ")
sample_string = input("Sample string: ")


# Build the FSM and match the sample string
transitions, end_states = build_fsm(pattern)

match_results = match(transitions, end_states, sample_string)

# Print the sample string with the matched parts colorized in red
print(colorize_matches(sample_string, match_results))
```

2.4 Observations and Conclusion

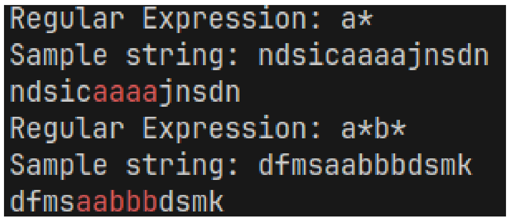
2.4.1 The cases

(a) Direct Matches



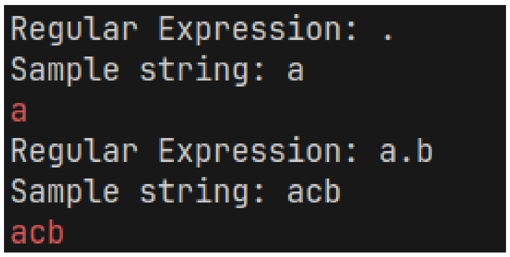
```
Regular Expression: a
Sample string: bbbbbaccc
bbbbbaccc
Regular Expression: ab
Sample string: jwndwndfabksdnkdn
jwndwndfabksdnkdn
```

(b) Any number of characters



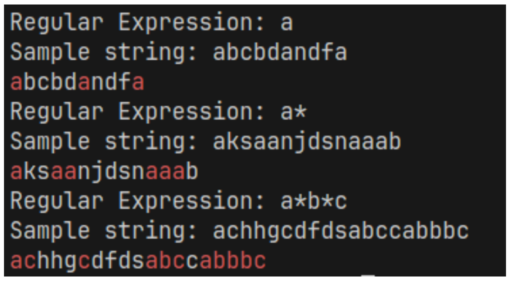
```
Regular Expression: a*
Sample string: ndsicaaaajnsdn
ndsicaaaajnsdn
Regular Expression: a*b*
Sample string: dfmsaabbbdsmk
dfmsaabbbdsmk
```

(c) Wildcard Chracter



```
Regular Expression: .
Sample string: a
a
Regular Expression: a.b
Sample string: acb
acb
```

(d) Multiple Matches



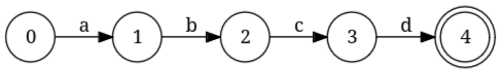
```
Regular Expression: a
Sample string: abcbdandfa
abcbdandfa
Regular Expression: a*
Sample string: aksaanjdsnaaab
aksaanjdsnaaab
Regular Expression: a*b*c
Sample string: achhgcdfdsabccabbbc
achhgcdfdsabccabbbc
```

2.4.2 Sample Test Case from Problem Statement

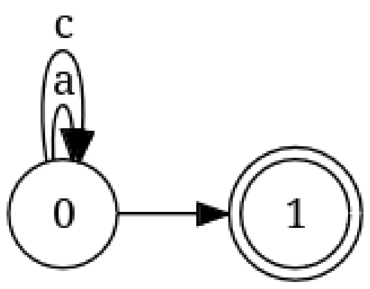
The graphs for the given testcases are shown below:

- **Regular Expression**: abcd
  Sample string: abcdef

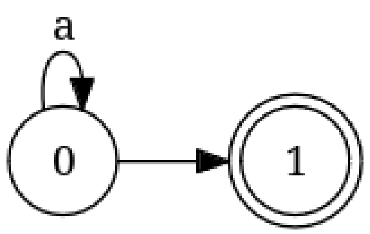
$$\begin{tikzcd}
0 \arrow[r, "a"] & 1 \arrow[r, "b"] & 2 \arrow[r, "c"] & 3 \arrow[r, "d"] & \boxed{4}
\end{tikzcd}$$



- **Regular Expression**: a*c*
  Sample string: baacc

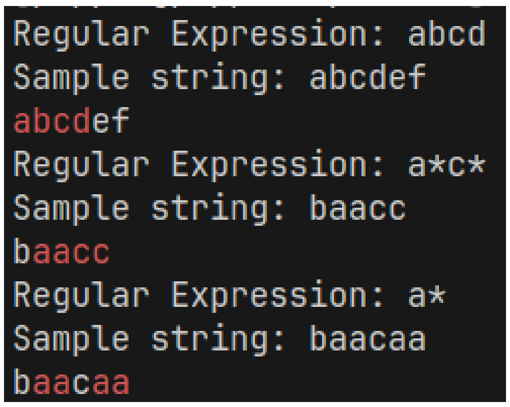


- **Regular Expression**: a*
  Sample string: baaccaa



Graphs made using graphviz

The terminal output for these are



```
Regular Expression: abcd
Sample string: abcdef
abcdef
Regular Expression: a*c*
Sample string: baacc
baacc
Regular Expression: a*
Sample string: baacaa
baacaa
```