ID2090 Take Home Assignment

Anton Beny M S, ME23B015

May 2024

Contents

1 Path Planning **3**
- 1.1 Introduction . . . 3
- 1.2 Implementation . . . 3
  - 1.2.1 Node . . . 3
  - 1.2.2 Obstacle . . . 3
  - 1.2.3 Environment . . . 3
  - 1.2.4 RRT . . . 5
  - 1.2.5 Smoothing . . . 6
  - 1.2.6 Spacing . . . 7
  - 1.2.7 Greed . . . 8
- 1.3 Code . . . 10
  - 1.3.1 Imports . . . 10
  - 1.3.2 Node . . . 10
  - 1.3.3 Obstacle . . . 11
  - 1.3.4 Environment . . . 12
  - 1.3.5 RRT . . . 15
  - 1.3.6 Usage . . . 21
- 1.4 Results . . . 24
  - 1.4.1 Environments . . . 25
  - 1.4.2 Path Length . . . 25
  - 1.4.3 The Effect of Spacing . . . 27
  - 1.4.4 Time Complexity . . . 28

2 Automata **30**
- 2.1 Introduction . . . 30
- 2.2 Implementation . . . 30
  - 2.2.1 FSM . . . 30
  - 2.2.2 Direct Match . . . 30
  - 2.2.3 The Asterisk . . . 31
  - 2.2.4 Wildcards and other Special Characters . . . 31
  - 2.2.5 Multiple Matches . . . 33
  - 2.2.6 To Infinity and Beyond . . . 33
  - 2.2.7 Character Sets . . . 34
  - 2.2.8 Escape from Reality . . . 35
  - 2.2.9 Alternation . . . 36
- 2.3 Code . . . 36

2.3.1 Imports 36
2.3.2 FSM 36
2.3.3 Usage 45
2.4 Results 46
2.4.1 Some FSMs 46
2.4.2 Greedy Matches 48

Task 1

Path Planning

1.1 Introduction

In this task, we are asked to simulate an environment with a starting position, a goal and some obstacles. We have to implement an RRT (Rapidly-exploring Random Tree) algorithm to find a path from the start to the goal, while avoiding the obstacles.

1.2 Implementation

1.2.1 Node

The tree is made up of nodes. Each node has a position, and a parent.

1.2.2 Obstacle

The obstacles are modelled as rectangles. Each obstacle is defined by the bottom-left and top-right corners. The bounding walls are also considered as obstacles.

1.2.3 Environment

The Environment is modelled as a 2D grid, with the bottom-left corner as $(0, 0)$ and the top-right corner as $(50, 50)$. I used three different environments for testing the algorithm.

1. **Environment 1**: This is a simple environment with only walls and **no obstacles** between the start and the goal. The start position is $(12, 12)$ and the goal position is $(38, 38)$.

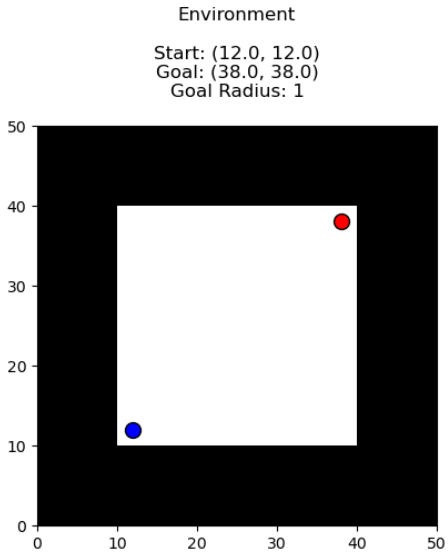


Figure 1.1: Environment 1

2. **Environment 2**: This is a slightly more complex environment with walls and **a few obstacles** between the start and the goal. The start position is $(12, 12)$ and the goal position is $(38, 38)$.

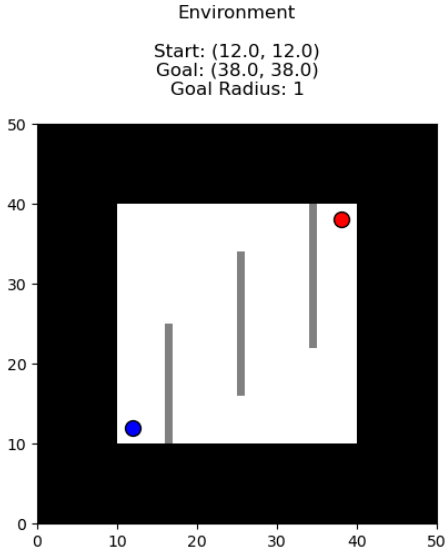


Figure 1.2: Environment 2

3. **Environment 3**: This is a complex environment with walls and is desigened to be a **maze** with only one path from the start to the goal. The start position is $(12, 12)$ and the goal position is $(36, 38)$.

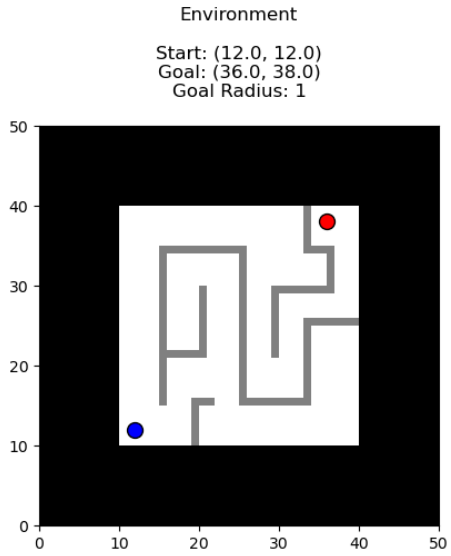


Figure 1.3: Environment 3

1.2.4 RRT

RRT is a simple path planning algorithm that works by building a tree of nodes. It starts with the start node and keeps adding nodes to the tree, until it reaches the goal. The algorithm works as follows:

1. START with an environment which has a start position, a goal position and some obstacles. Initially, the tree has only the start node.
2. Generate a random node within the environment.
3. Find the nearest node in the tree to the random node.
4. Move from the nearest node towards the random node, but only by a certain distance (step_size).
5. Add the new node to the tree.
6. If the new node is close enough to the goal, then END, else goto Step 2.

It uses the term **Rapidly-exploring** because, initally, when the tree is small and there is a higher chance of the random node being in an empty space. Therefore, it is more likely to explore the empty spaces first. It is also **Random**, because, well... the nodes are generated randomly. And it is a **Tree**, because the nodes are connected in a tree structure, that is, each node has only one parent, and there are no loops.

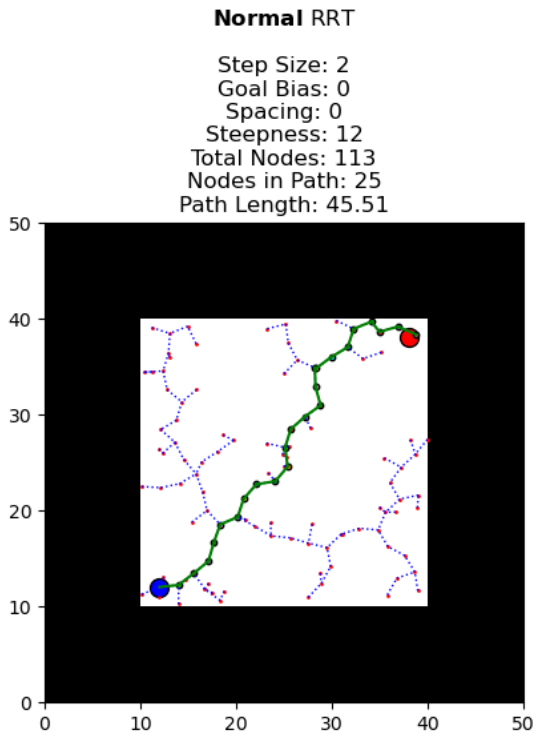
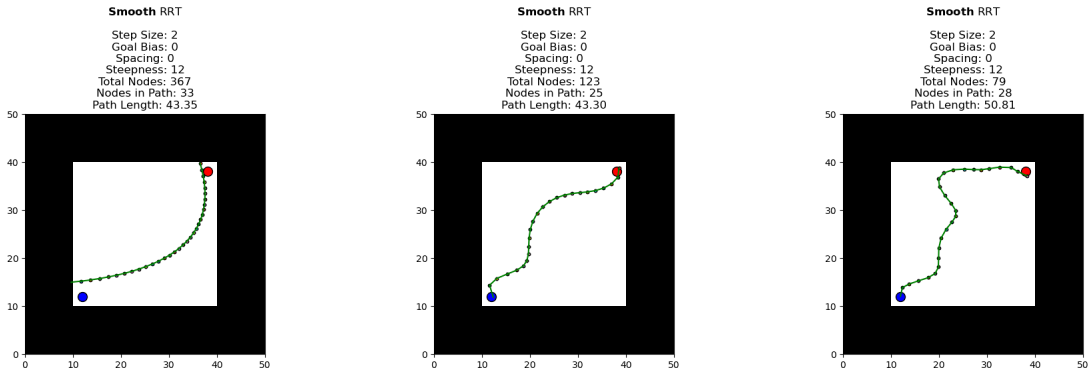


Figure 1.4: RRT Algorithm for Environment 1

1.2.5 Smoothing

The path generated by the RRT algorithm almost always has a lot of sharp turns. This is because, the nodes are generated randomly, and the path is not very smooth. To make the path smoother, a simple option is to do **Polynomial Regression** in order to fit a smooth curve to the path.



(a) Degree 2 Polynomial (b) Degree 7 Polynomial (c) Degree 12 Polynomial

However, as we can see, Polynomial Regression is not very good at smoothing the path. Therefore, I am going to use **Moving Polynomial Regression**. In this, I will take a window of points around a point, and fit a polynomial to these points. This way, the path will be much smoother. This is commonly referred to as **Savitzky-Golay Filtering**.

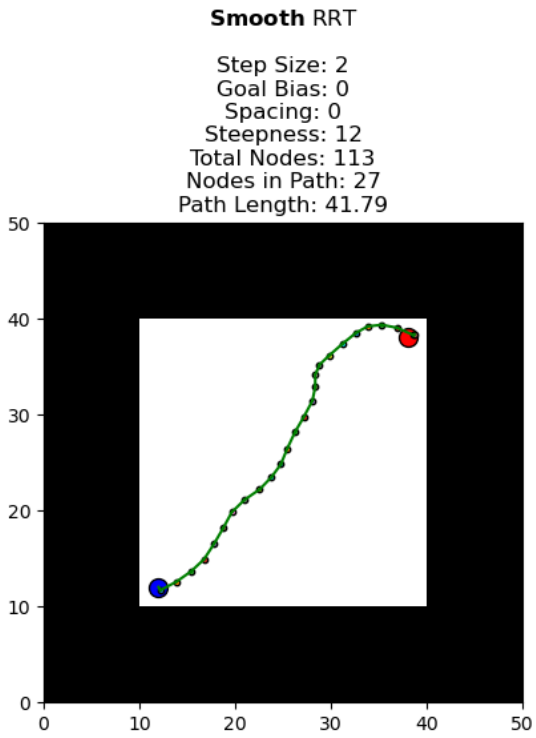


Figure 1.6: Smooth Path for Environment 1 using Savitzky-Golay Filtering

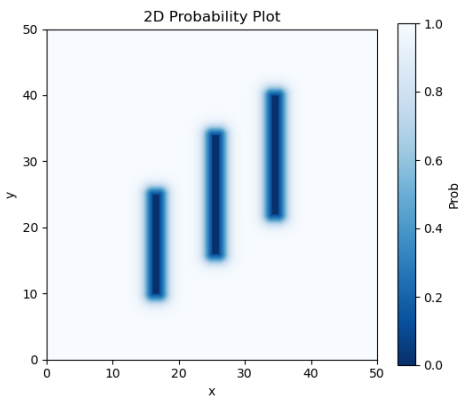
There are other methods of smoothing, such as **Bezier Curves**, **B-Splines**, etc. However, I am going to use Savitky-Golay Filtering, as it is simple and effective.

1.2.6 Spacing

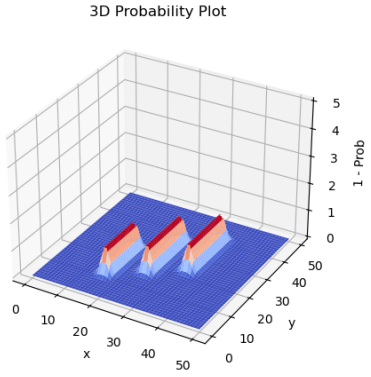
In a normal RRT, the nodes are generated randomly. In order to account for obstacles, we simply implement in such a way that, if there is a collision with an obstacle, then we can discard the node, and try with a new random node.

However, it doesn't take into account the spacing between the obstacle and the bot. Without spacing, RRT can generate a path that could be very close to the obstacle or could pass very close at the corners.

In order to prevent this, we just adding padding around each obstacle. The amount of padding is the spacing parameter. I implemented this in a probabilistic way. The probability of placing a node in a particular position is given by a sigmoid function of the distance of the node from the closest obstacle such that, the probability of placing a node at a distance = spacing from the obstacle is 0.5, and the probability decreases as we move closer to the obstacle, and increases as we move away from the obstacle. The plots of the probability for Environment 2 are shown below.

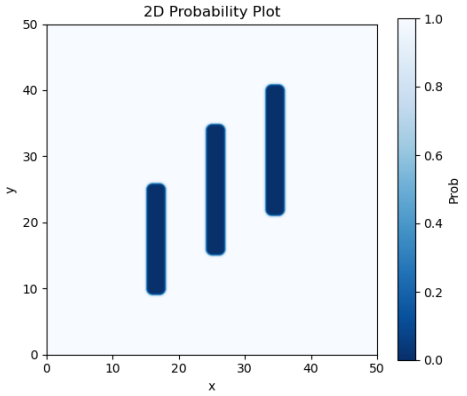


(a) 2D Plot of the Spacing Function

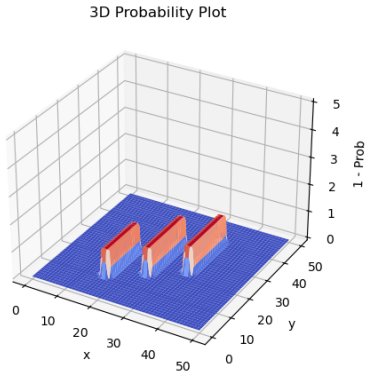


(b) 3D Plot of the Spacing Function

Figure 1.7: Spacing Function for Environment 2 with **Low Steepness**



(a) 2D Plot of the Spacing Function

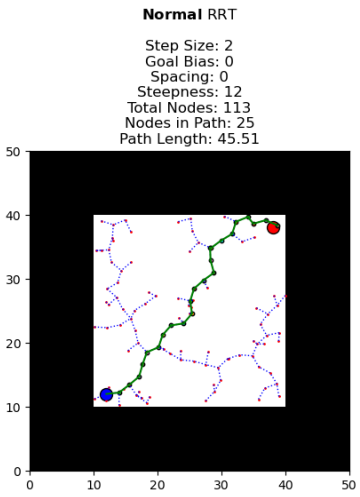


(b) 3D Plot of the Spacing Function

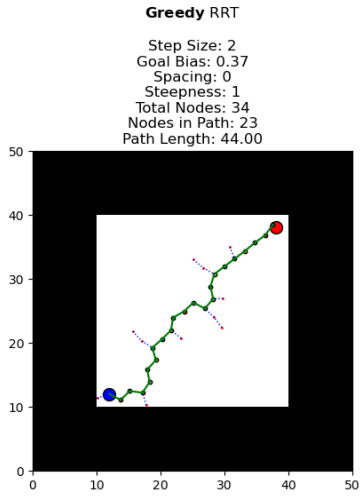
Figure 1.8: Spacing Function for Environment 2 with **High Steepness**

1.2.7 Greed

In a greedy approach to the RRT algorithm, we can add a parameter called `goal_bias`. If `goal_bias` = 0, then the nodes are purely random. If not, there is a chance that the random node is the goal node itself, or a node close to the goal node. This way, the algorithm finds the goal much faster.



(a) Environment 1: `goal_bias` = 0



(b) Environment 1: `goal_bias` = 0.37

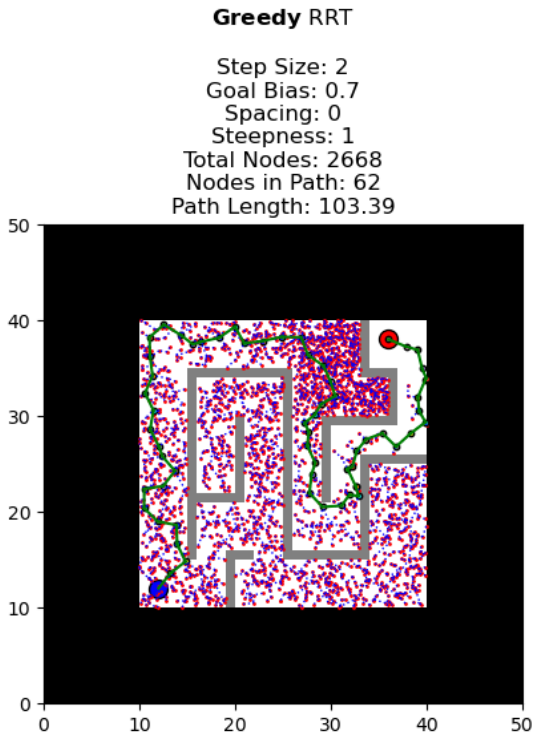


Figure 1.10: Environment 3 with **High** `goal_bias` = 0.7.

If the `goal_bias` is high, then the algorithm will find the goal much faster in environments where the goal is easy to reach such as Environment 1 as shown in Figure 1.9b. However in complex environments such as Environment 3, the algorithm might not find the goal at all. Notice the high concentration of nodes near the goal in Figure 1.10.

1.3 Code

I implemented the RRT algorithm in Python. The whole code and its explanation is given here:

1.3.1 Imports

I used `matplotlib` to plot the environment and the path. I also used `numpy` for some mathematical operations. The `random` library is used to generate the random numbers. I also used `copy` to make deep copies of the environment. The `math` library is used for mathematical operations, such as `sqrt` and `exp`

Listing 1.1: Imports

```
#!/usr/bin/python3

import copy
import math
import os
import random
import time
from typing import Union

import matplotlib.patches as patches
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

1.3.2 Node

I defined a class `Node`, which stores the position of the node and the parent of the node, which is later used to construct the path. It has a `__repr__` method, that returns a string representation of the node in the form `(x, y)`.

Listing 1.2: Node Class

```
class Node:
    def __init__(self, pos: tuple[float, float], parent: Union["Node"
        , None] = None):
        self.pos = pos
        self.x, self.y = pos
        self.parent = parent

    def __repr__(self) -> str:
        return f"({self.x:.1f}, {self.y:.1f})"
```

It also has a `distance` method, that calculates the Euclidean distance between two nodes.

Listing 1.3: `Node.distance`

```
    def distance(self, other: Union["Node", None]) -> float:
```

```
        if other is None:
            return 0
        return math.sqrt((self.x - other.x) ** 2 + (self.y - other.y)
            ** 2)
```

1.3.3 Obstacle

The next class is `Obstacle`. It stores the bottom-left and top-right corners, width, height of the obstacle. It also has an attribute `is_wall` in order to denote if the obstacle is a wall or not.

Listing 1.4: Obstacle Class

```
class Obstacle:
    def __init__(self, corner1: Node, corner2: Node, is_wall: bool =
        False):
        self.corner1 = Node((min(corner1.x, corner2.x), min(corner1.y
            , corner2.y)))
        self.corner2 = Node((max(corner1.x, corner2.x), max(corner1.y
            , corner2.y)))
        self.width = self.corner2.x - self.corner1.x
        self.height = self.corner2.y - self.corner1.y
        self.is_wall = is_wall
```

The functions of the `Obstacle` class are:

- `__contains__`: checks if a Node is within the obstacle. I used `__contains__` method, so that I can use the `in` operator to check if a node is within the obstacle.

  Listing 1.5: `Obstacle.__contains__`

  ```
  def __contains__(self, node: Node) -> bool:
      x1, y1 = self.corner1.x, self.corner1.y
      x2, y2 = self.corner2.x, self.corner2.y

      return x1 <= node.x <= x2 and y1 <= node.y <= y2
  ```

- `distance`: calculates the distance of a node from the obstacle. The distance is the distance from the node to the closest point on the obstacle. It does this by finding which side of the obstacle the node is on, and then calculating the distance accordingly. If the node is to the right, left, above or below the obstacle, then the distance is the perpendicular distance from the node to the closest side of the obstacle. If the node is to the top-right, top-left, bottom-right or bottom-left of the obstacle, then the distance is the distance from the node to the closest corner of the obstacle.

  Listing 1.6: Obstacle.distance

  ```
  def distance(self, node: Node) -> float:
      xc, yc = (self.corner1.x + self.corner2.x) / 2, (
          self.corner1.y + self.corner2.y
      ) / 2
  ```

```
        r = 0
        x1, y1 = node.x, node.y
        x1, y1 = x1 - xc, y1 - yc
        xl, xr = self.corner1.x, self.corner2.x
        yb, yt = self.corner1.y, self.corner2.y
        xl, xr = xl - xc, xr - xc
        yb, yt = yb - yc, yt - yc

        if y1 > yt:
            if x1 > xr:
                r = math.sqrt((x1 - xr) ** 2 + (y1 - yt) ** 2)
            elif x1 < xl:
                r = math.sqrt((x1 - xl) ** 2 + (y1 - yt) ** 2)
            else:
                r = yt - y1
        elif y1 < yb:
            if x1 > xr:
                r = math.sqrt((x1 - xr) ** 2 + (y1 - yb) ** 2)
            elif x1 < xl:
                r = math.sqrt((x1 - xl) ** 2 + (y1 - yb) ** 2)
            else:
                r = y1 - yb
        else:
            if x1 > xr:
                r = xr - x1
            elif x1 < xl:
                r = x1 - xl

        return abs(r)
```

1.3.4 Environment

Then I defined a class `Environment`, that stores environment-specific information such as obstacles, start and goal positions. It also stores the goal-radius, which is the radius around the goal node, within which the goal is considered to be reached.

Listing 1.7: Environment Class

```
class Environment:
    def __init__(
        self,
        dimensions: tuple[float, float],
        start: Node,
        goal: Node,
        obstacles: Union[list[Obstacle], None] = None,
        goal_radius: float = 1.0,
```

```
):
    if obstacles is None:
        obstacles = []
    self.width, self.height = dimensions
    self.start = start
    self.goal = goal
    self.goal_radius = goal_radius
    self.obstacles = obstacles
```

The functions of the `Environment` class are:

- `add_obstacle`: adds an obstacle to the environment.

Listing 1.8: `Environment.add_obstacle`

```
def add_obstacle(self, obstacle: Obstacle):
    self.obstacles.append(obstacle)
```

- `random_node`: generates a random node in the environment. This random could be anywhere in the environment, even within the obstacles.

Listing 1.9: `Environment.random_node`

```
def random_node(self, goal_bias: float = 0.2) -> Node:
    if random.random() <= goal_bias:
        if random.random() <= goal_bias:
            return self.goal
        else:
            rand_nodes = [
                Node(
                    (
                        random.uniform(0, self.width),
                        random.uniform(0, self.height),
                    )
                )
                for _ in range(10)
            ]
            return min(rand_nodes, key=lambda node: node.distance
               (self.goal))
    else:
        return Node(
            (
                random.uniform(0, self.width),
                random.uniform(0, self.height),
            )
        )
```

- `is_valid_node`: returns a boolean value, indicating if the node is within the environment and not within any obstacle.

Listing 1.10: `Environment.is_valid_node`

```
def is_valid_node(self, node: Node) -> bool:
    return (
        0 <= node.x <= self.width
        and 0 <= node.y <= self.height
        and all(node not in obstacle for obstacle in self.
           obstacles)
    )
```

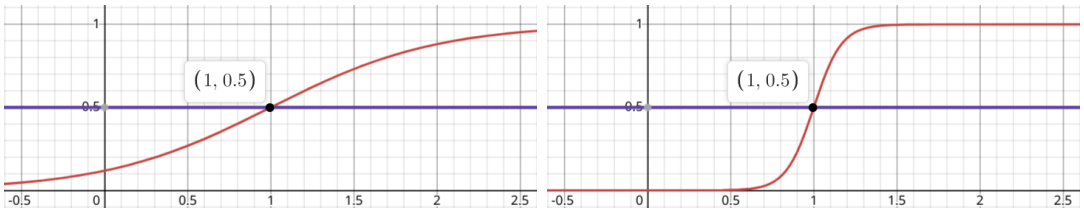
- `prob`: returns the probability of placing a node in that particular position. The probability is 0 inside obstacles, and there is a gradient from 0 to 1 as we move away from the obstacles. The gradient is a `sigmoid` function of the distance of the node from the closest obstacle given by `Obstacle.distance`. If r is this distance, then the probability is given by

$$P(r) = \frac{1}{1 + e^{-k(r-s)}}$$

. Here k is a constant that determines the **steepness** of the gradient. For large k, the gradient is much more sharp. s is the spacing parameter. It is such that, the probability of finding a node at a distance $= s$ from the obstacle is 0.5. This is how I implemented the spacing parameter.

Listing 1.11: `Environment.prob`

```
def prob(self, node: Node, spacing: float, steepness: float = 12)
    -> float:
    if not self.obstacles:
        return 1
    if not self.is_valid_node(node):
        return 0
    min_r = min(obstacle.distance(node) for obstacle in self.
       obstacles)
    probability = 1 / (1 + math.exp(-steepness * (min_r - spacing
       )))
    return probability
```



(a) Sigmoid Function with **Low Steepness**

(b) Sigmoid Function with **High Steepness**

- `is_suitable_node`: returns the probability of placing a node in that particular position using the `prob` method. It also checks if the line between the nearest node and the new node is valid using the `is_valid_line` method.

Listing 1.12: Environment.is_suitable_node

```
def is_suitable_node(
    self, nearest: Node, new_node: Node, spacing: float,
       steepness: float = 12
) -> float:
    if spacing < 0:
        return 1
    if self.is_valid_line(nearest, new_node):
        return self.prob(new_node, spacing, steepness)
    return 0
```

- is_valid_line: returns a boolean value, indicating if the line between two nodes is valid. This is done by dividing the line into small segments and checking if each point is valid.

Listing 1.13: Environment.is_valid_line

```
def is_valid_line(self, node1: Node, node2: Node) -> bool:
    dx = node2.x - node1.x
    dy = node2.y - node1.y
    for t in np.linspace(0, 1, 10):
        x = node1.x + t * dx
        y = node1.y + t * dy
        if any(Node((x, y)) in obstacle for obstacle in self.
           obstacles):
            return False

    return True
```

1.3.5 RRT

Finally, we have the RRT class, which uses the functions of all the previously mentioned classes to implement the RRT algorithm. I store a deepcopy of the environment, so that I can modify the environment in the RRT class, without affecting the original environment. I also store a list of nodes, and other parameters such as goal_bias, step_size, steepness. Finally, I also store the target node (the node that is close to the goal) and the path.

Listing 1.14: RRT Class

```
class RRT:
    def __init__(
        self,
        environment: Environment,
        step_size: float = 2,
        max_iter: int = 10000,
        goal_bias: float = 0.2,
        spacing: float = 1.0,
        steepness: float = 12,
    ):
```

```
        self.environment = copy.deepcopy(environment)
        self.step_size = step_size
        self.max_iter = max_iter
        self.nodes = [environment.start]
        self.target = None
        self.goal_bias = goal_bias
        self.spacing = spacing
        self.steepness = steepness

        self.length = 0
        self.path = []
```

The functions of the RRT class are:

- reset: It is a simple function that resets the nodes, the target node and the path and the length of the path.

Listing 1.15: RRT.reset

```
def reset(self):
    self.nodes = [self.environment.start]
    self.target = None
    self.length = 0
    self.path = []
```

- plan: It runs a loop for max_iter number of times. In each iteration, it calls the new function which returns a new node to be placed. This node is appended to nodes. If this node is within the goal-radius, then the target node is set to this node, and the loop is broken. After the loop, the path is generated by calling the construct function.

Listing 1.16: RRT.plan

```
def plan(self):
    for _ in range(self.max_iter):
        new_node = self.new()
        self.nodes.append(new_node)

        if self.goal_reached(new_node):
            self.target = new_node
            break
```

- new: This function generates a random node using Environment.random_node. It then finds the nearest node to this random node by calling the nearest function. It generates the new node by moving from the nearest node towards the random node by a distance step_size. The probability of placing this node is calculated using the Environment.is_suitable_node function. Depending on this probability, the node is either placed or discarded. If the node is discarded, a new node is generated, and the process is repeated. If the node is placed, then it is given to the plan function.

Listing 1.17: RRT.new

```
def new(self) -> Node:
    target = self.environment.random_node(self.goal_bias)
    nearest = self.nearest(target)
    rel_pos = Node((target.x - nearest.x, target.y - nearest.y))
    rel_pos_len = rel_pos.distance(Node((0, 0)))

    step_size = min(self.step_size, rel_pos_len)

    new_pos = Node(
        (
            nearest.x + step_size * rel_pos.x / rel_pos_len,
            nearest.y + step_size * rel_pos.y / rel_pos_len,
        )
    )

    new_node = Node((new_pos.x, new_pos.y), nearest)

    probability = self.environment.is_suitable_node(
        nearest, new_node, self.spacing, self.steepness
    )

    if random.random() <= probability:
        return new_node
    else:
        return self.new()
```

- nearest: This function finds the nearest node in the tree to the given node. It does this by iterating through all the nodes and finding the node with the minimum distance, and returns this node.

Listing 1.18: RRT.nearest

```
def nearest(self, target: Node) -> Node:
    return min(self.nodes, key=lambda node: node.distance(target)
       )
```

- goal_reached: This function checks if the target node is within the goal-radius, and returns the boolean value.

Listing 1.19: RRT.goal_reached

```
def goal_reached(self, node: Node) -> bool:
    return node.distance(self.environment.goal) <= self.
       environment.goal_radius
```

- construct: This function constructs the path from the start node to the target node. It does this by starting from the target node, and moving to its parent, and then moving to

the parent of the parent, and so on, until the start node is reached. Finally, it reverses the path, so that the start node is the first element of the path, and returns it.

Listing 1.20: `RRT.construct`

```
def construct(self):
    ptr = self.target
    while ptr is not None:
        self.path.append(ptr)
        self.length += ptr.distance(ptr.parent)
        ptr = ptr.parent
    self.path.reverse()
```

- `plot`: This function is used to plot the environment, the obstacles, the path and the nodes, using `matplotlib`.

Listing 1.21: `RRT.plot`

```
def plot(
    self,
    path: Union[list[Node], None] = None,
    smooth: bool = False,
    length: float = 0,
):
    if path is None:
        if smooth:
            path, length = self.smoothen_path()
        else:
            path = self.path

    if self.spacing >= 0:
        fig, ax = self.environment.plot(view=False)
    else:
        fig, ax = plt.subplots(figsize=(6, 6))
        ax.set_xlim(0, self.environment.width)
        ax.set_ylim(0, self.environment.height)

    if not smooth:
        for node in self.nodes:
            if node.parent is not None:
                plt.plot(
                    [node.x, node.parent.x],
                    [node.y, node.parent.y],
                    color="blue",
                    linestyle="dotted",
                    linewidth=1,
                )
                plt.scatter(node.x, node.y, s=1, edgecolors="red"
                   )
```

```
if self.target is not None:
    for i in range(1, len(path)):
        plt.plot(
            [path[i].x, path[i - 1].x],
            [path[i].y, path[i - 1].y],
            "g-",
            linewidth=1.5,
        )
        plt.scatter(path[i].x, path[i].y, s=10, edgecolors="
           black")

no_of_nodes = len(self.nodes)
no_of_nodes_in_path = len(path)
total_length = length if smooth else self.length

titles = []
if smooth:
    titles.append("Smooth")
if self.spacing > 0:
    titles.append("Spaced")
if self.goal_bias > 0:
    if self.spacing > 0:
        titles.remove("Spaced")
        titles.append("Speedy")
    else:
        titles.append("Greedy")
if not titles:
    titles = ["Normal"]

title = "␣".join(titles)

details = (
    rf"$\bf{{{title}}}$"
    f"␣RRT\n\n"
    f"Step␣Size:␣{self.step_size}\n"
    f"Goal␣Bias:␣{self.goal_bias}\n"
    f"Spacing:␣{self.spacing}\n"
    f"Steepness:␣{self.steepness}\n"
    f"Total␣Nodes:␣{no_of_nodes}\n"
    f"Nodes␣in␣Path:␣{no_of_nodes_in_path}\n"
    f"Path␣Length:␣{total_length:.2f}"
)
plt.title(details)

fig.subplots_adjust(left=0.2, right=0.8, top=0.7, bottom=0.1)
```

```
    plt.show()
    plt.close()
```

- smoothen_path: This function uses scipy.signal._savitzky_golay.savgol_filter to smoothen the path. It takes a window size of 7 and a polynomial degree of 2. It returns the smoothened path and the length of the path.

Listing 1.22: RRT.smoothen_path

```
def smoothen_path(
    self, window_length: int = 7, polyorder: int = 2
) -> tuple[list[Node], float]:
    from scipy.signal import savgol_filter

    x = [node.x for node in self.path]
    y = [node.y for node in self.path]

    x_smooth = savgol_filter(x, window_length=window_length,
       polyorder=polyorder)
    y_smooth = savgol_filter(y, window_length=window_length,
       polyorder=polyorder)

    x_smooth = np.concatenate(([x[0]], x_smooth, [x[-1]]))
    y_smooth = np.concatenate(([y[0]], y_smooth, [y[-1]]))

    smooth_path = [Node((x, y)) for x, y in zip(x_smooth,
       y_smooth)]
    length = 0
    for i in range(1, len(smooth_path)):
        length += smooth_path[i].distance(smooth_path[i - 1])

    return smooth_path, length
```

- plot_prob: This function is used to plot the probability of placing a node at each point in the environment. It uses the Environment.is_suitable_node function to calculate the probability.

Listing 1.23: RRT.plot_prob

```
def plot_prob(self):
    x = np.linspace(0, self.environment.width, 500)
    y = np.linspace(0, self.environment.height, 500)
    x, y = np.meshgrid(x, y)

    z = np.zeros_like(x)
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            node = Node((x[i, j], y[i, j]))
```

```
                z[i, j] = self.environment.prob(node, self.spacing,
                    self.steepness)

        plt.figure(figsize=(6, 5))
        plt.imshow(
            z,
            origin="lower",
            extent=(0, self.environment.width, 0, self.environment.
                height),
            cmap="Blues_r",
        )
        plt.colorbar(label="Prob")
        plt.title("2D Probability Plot")
        plt.xlabel("x")
        plt.ylabel("y")

        z = 1 - z
        fig = plt.figure(figsize=(6, 5))
        ax = fig.add_subplot(111, projection="3d")
        ax.plot_surface(x, y, z, cmap="coolwarm")
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_zlabel("1 - Prob")
        ax.set_zlim(0, 5)
        plt.title("3D Probability Plot")

        plt.show()
```

1.3.6 Usage

Now that we have defined all the classes and functions, we can use them to run the RRT algorithm. Here, I have given an example of how to use it to implement the RRT algorithm to find a path in Environment 2.

Firstly, I defined a function that returns an environment from Environment 1, Environment 2 or Environment 3 based on the parameter.

Listing 1.24: create_env

```
def create_env(env_type: int = 2) -> Environment:
    env = Environment(
        (50, 50),
        start=Node((12, 12)),
        goal=Node((36 if env_type == 3 else 38, 38)),
        goal_radius=1,
    )
    env.add_obstacle(Obstacle(Node((0, 0)), Node((10, 50)), is_wall=
        True))
```

```
    env.add_obstacle(Obstacle(Node((40, 0)), Node((50, 50)), is_wall=
       True))
    env.add_obstacle(Obstacle(Node((10, 0)), Node((40, 10)), is_wall=
       True))
    env.add_obstacle(Obstacle(Node((10, 40)), Node((40, 50)), is_wall
       =True))

    if env_type == 2:
        env.add_obstacle(Obstacle(Node((16, 10)), Node((17, 25))))
        env.add_obstacle(Obstacle(Node((25, 16)), Node((26, 34))))
        env.add_obstacle(Obstacle(Node((34, 22)), Node((35, 40))))
    elif env_type == 3:
        env.add_obstacle(Obstacle(Node((15, 15)), Node((16, 35))))
        env.add_obstacle(Obstacle(Node((19, 10)), Node((20, 16))))
        env.add_obstacle(Obstacle(Node((20, 15)), Node((22, 16))))
        env.add_obstacle(Obstacle(Node((16, 21)), Node((20, 22))))
        env.add_obstacle(Obstacle(Node((20, 21)), Node((21, 30))))
        env.add_obstacle(Obstacle(Node((16, 34)), Node((26, 35))))
        env.add_obstacle(Obstacle(Node((25, 15)), Node((26, 35))))
        env.add_obstacle(Obstacle(Node((26, 15)), Node((34, 16))))
        env.add_obstacle(Obstacle(Node((33, 34)), Node((34, 40))))
        env.add_obstacle(Obstacle(Node((34, 34)), Node((37, 35))))
        env.add_obstacle(Obstacle(Node((36, 30)), Node((37, 35))))
        env.add_obstacle(Obstacle(Node((29, 29)), Node((37, 30))))
        env.add_obstacle(Obstacle(Node((29, 21)), Node((30, 30))))
        env.add_obstacle(Obstacle(Node((33, 16)), Node((34, 26))))
        env.add_obstacle(Obstacle(Node((34, 25)), Node((40, 26))))

    return env
```

Now, I can use this function to create an environment, and then use the `RRT` class to find a path in this environment.

Listing 1.25: Usage

```
env = create_env(2)

rrt = RRT(env, step_size=2, max_iter=10000, goal_bias=0.37, spacing
   =1, steepness=12)
rrt.plan()
rrt.construct()

rrt.plot()
rrt.plot(smooth=True)
```

In order to get some insights into the performance of each variant, I defined a function `test`, which runs the RRT algorithm for each variant, and stores the data as a `csv` file.

Listing 1.26: Testing

```
def test(rrt_list: list[RRT], env: Environment, no_of_tests: int = 1)
    :
    rrts = []
    if "Normal" in rrt_list:
        normal_rrt = RRT(env, step_size=2, max_iter=10000, goal_bias
            =0, spacing=0)
        rrts.append(("Normal", normal_rrt))
    if "Spaced" in rrt_list:
        spaced_rrt = RRT(
            env, step_size=2, max_iter=10000, goal_bias=0, spacing=1,
                steepness=12
        )
        rrts.append(("Spaced", spaced_rrt))
    if "Greedy" in rrt_list:
        greedy_rrt = RRT(
            env, step_size=2, max_iter=10000, goal_bias=0.37, spacing
               =0, steepness=12
        )
        rrts.append(("Greedy", greedy_rrt))
    if "Speedy" in rrt_list:
        speedy_rrt = RRT(
            env, step_size=2, max_iter=10000, goal_bias=0.37, spacing
               =1, steepness=12
        )
        rrts.append(("Speedy", speedy_rrt))

    filepath = f"data/{len(env.obstacles)}_obstacles"
    if not os.path.exists(filepath):
        os.makedirs(filepath)
    file = open(f"{filepath}/rrt_data.csv", "w")
    file.write(
        f"RRT,Step␣Size,Goal␣Bias,Spacing,Steepness,Path␣Length,Path␣
            Nodes"
        f",Smooth␣Length,Smooth␣Nodes,Total␣Nodes,Time␣Taken,
            Obstacles\n"
    )

    no_of_obstacles = len(env.obstacles)

    for _ in range(no_of_tests):
        print(f"\nTest␣{_␣+␣1}␣of␣{no_of_tests}:")
        for name, rrt in rrts:
            try:
                start = time.time()
                rrt.plan()
                rrt.construct()
```

```
                smooth_path, smooth_length = rrt.smoothen_path(
                    window_length=5, polyorder=2
                )
                smooth_nodes = len(smooth_path)
                time_taken = time.time() - start
                print(f"{name} RRT in {time_taken:.2f} seconds.")

                path_length = rrt.length
                path_nodes = len(rrt.path)
                total_nodes = len(rrt.nodes)

                file.write(
                    f"{name},"
                    f"{rrt.step_size},"
                    f"{rrt.goal_bias},"
                    f"{rrt.spacing},"
                    f"{rrt.steepness},"
                    f"{path_length},"
                    f"{path_nodes},"
                    f"{smooth_length},"
                    f"{smooth_nodes},"
                    f"{total_nodes},"
                    f"{time_taken},"
                    f"{no_of_obstacles}\n"
                )
                file.flush()

                rrt.reset()
            except Exception:
                pass

    file.close()

rrt_list = ("Normal", "Spaced", "Greedy", "Speedy")
test(rrt_list, env, no_of_tests=1000)
```

1.4 Results

By using the `test` function, I ran the RRT algorithm for each variant 1000 times, and stored the data in a `csv` file. I then used this data to plot the performance of each variant. In order to compare all of them, I used the same parameters for each variant, except for the goal bias and the spacing. The parameters used were:

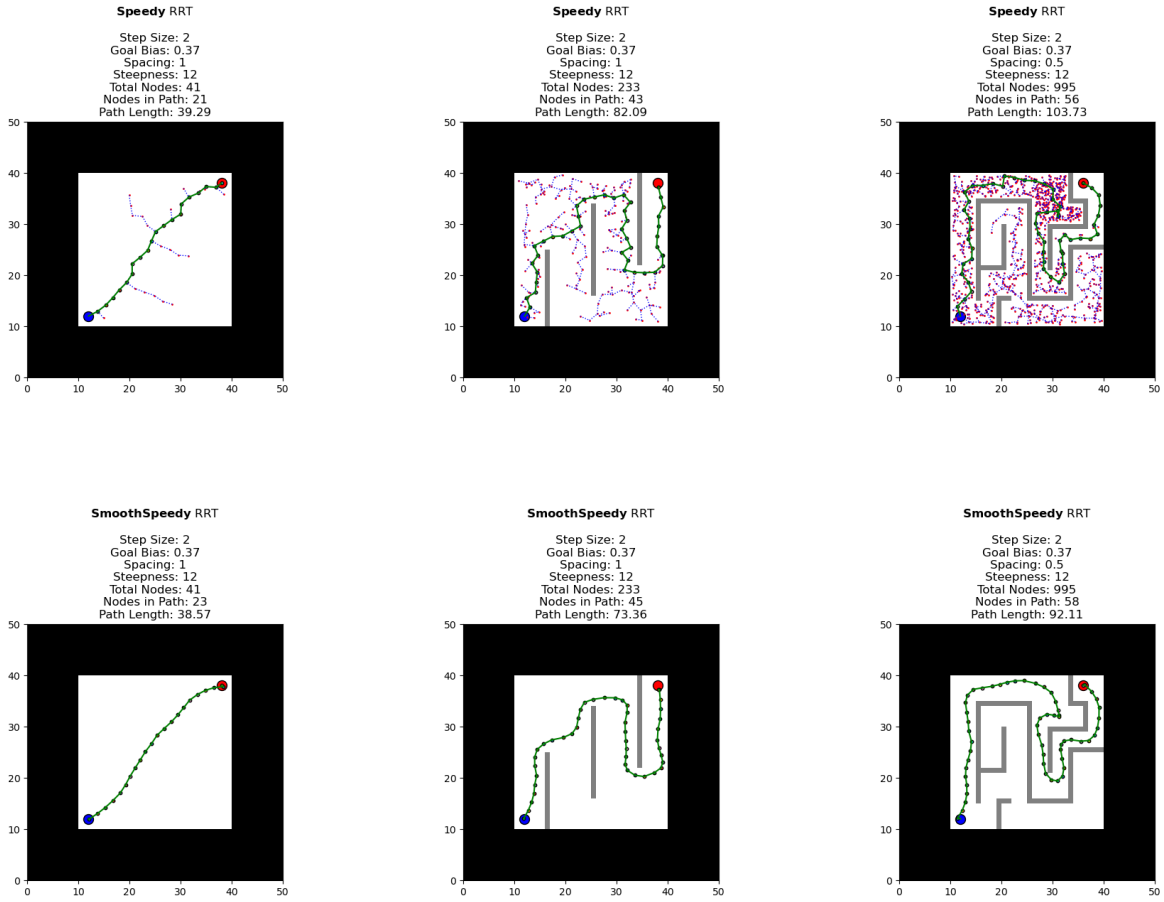
- Normal: `goal_bias` $= 0$, `spacing` $= 0$
- Spaced: `goal_bias` $= 0$, `spacing` $= 1$

- Greedy: `goal_bias` $= 0.37$, `spacing` $= 0$
- Speedy: `goal_bias` $= 0.37$, `spacing` $= 1$ [1]

[2]

1.4.1 Environments

Before we look at the results, let's look at the environments and the smooth paths for each of them.



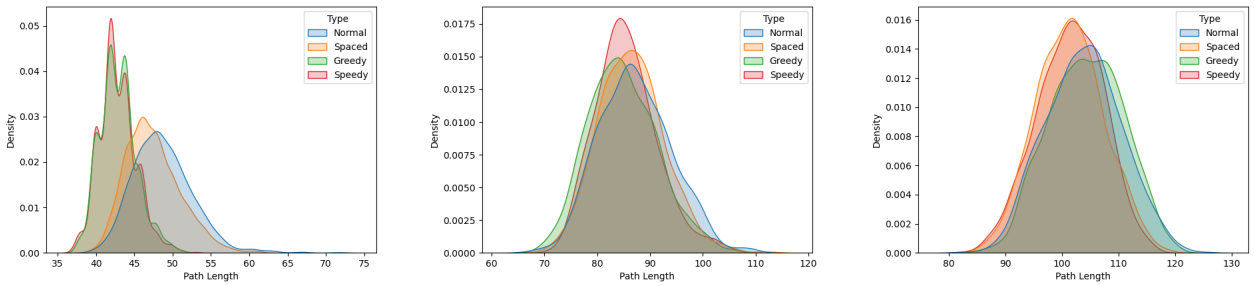
1.4.2 Path Length

I am going to be using Path Length in order to compare each variant, as it is a better indicator than the number of nodes in the path.

First, we have the frequency of the path length for each variant in each environment.

[1] Speedy is the combination of Greedy and Spaced. Literally, it is **Sp**aced + Gr**eedy.**

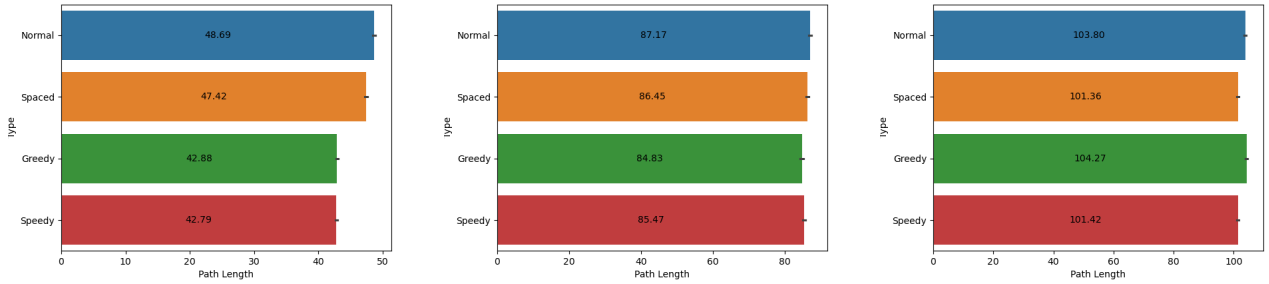
[2] The values of `spacing` was chosen as 0.5 for Environment 3, as the obstacles were much closer to each other.



(a) Environment 1 (b) Environment 2 (c) Environment 3

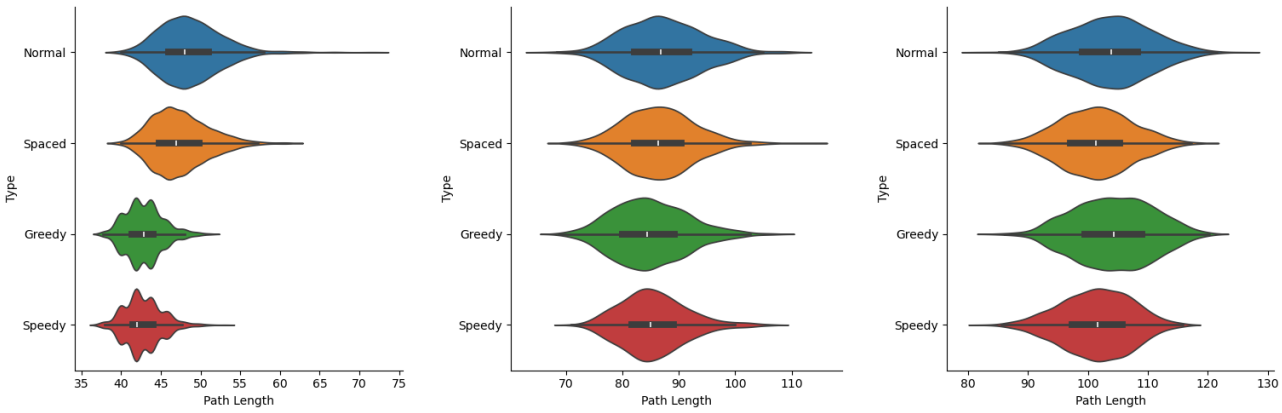
Figure 1.14: Path Length Frequency

We can also plot the average path length for each variant in each environment.



(a) Environment 1 (b) Environment 2 (c) Environment 3

This can be better visualized using a violin plot as shown below:



(a) Environment 1 (b) Environment 2 (c) Environment 3

Figure 1.16: The white line in the middle of the violin plot represents the median. The thick black bar represents the interquartile range.

Conclusions

- We can observe that the `Greedy` variant has the best performance in environments with fewer obstacles such as Environment 1 and Environment 2.

- In environments with more obstacles such as Environment 3, the `Spaced` variant performs the best. This is because the obstacles are much closer to each other, and the `Spacing` helps in avoiding these obstacles.
- Overall the `Speedy` variant performs the best, as it has both `Goal Bias` and `Spacing`.
- As expected, the `Normal` variant performs the worst in all environments.

1.4.3 The Effect of Spacing

For part (c) of the Task, We are asked to check our implementation of `Spacing` for an environment as shown here:

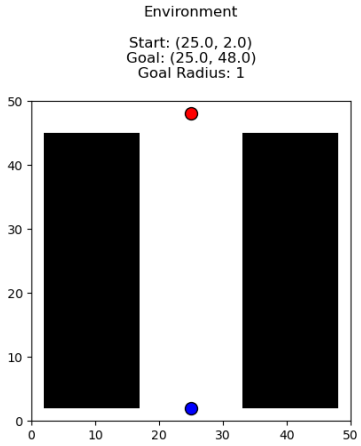


Figure 1.17: Environment

Without any spacing and without any goal bias, the path found by the RRT algorithm is as shown below:

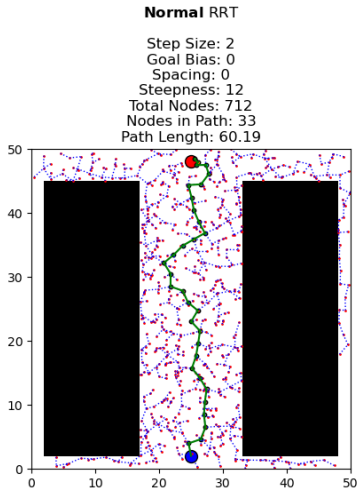
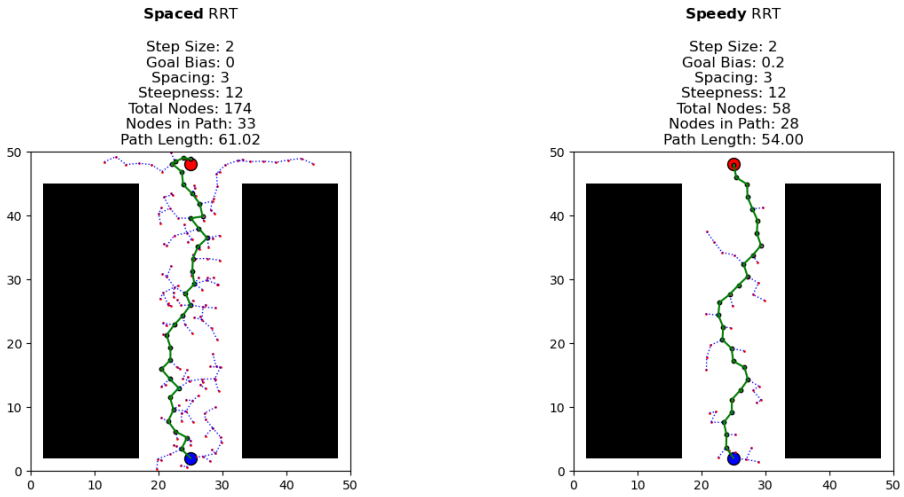


Figure 1.18: Environment

Observe the path, it goes very close to the obstacles. This is not ideal and it can be fixed by using the `Spacing` parameter. I used a `Spacing` of 3 for this environment, and the path found by the RRT algorithm is as shown below:

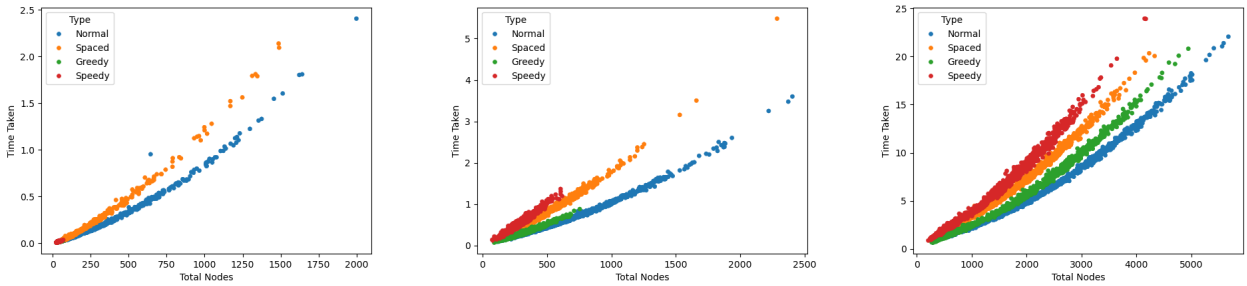


(a) Spaced (b) Speedy

We can clearly notice, that the Spaced variants have much better paths. It doesn't try to go through the small gaps between the obstacles and the wall.

1.4.4 Time Complexity

One interesting thing to note is the time taken by each variant. I am going to plot the Time Taken for the RRT algorithm for each variant vs The total number of obstacles in the environment.



(a) Environment 1 (b) Environment 2 (c) Environment 3

Figure 1.20: Total Nodes vs Time Taken

Conclusions

- There is a direct correlation between the Total Number of Obstacles and the Time Taken by the RRT algorithm. The relationship is exponential, and the shape of the curve remains the same for all variants for each environment.

- For environments with fewer obstacles, the `Greedy` variants take significantly less time than the `non-Greedy` variants.
- However, in Environment 3, all the variants take a similar amount of time, as the obstacles are much closer to each other.
- We can see, for a given Number of Nodes, the `Speedy` variant takes the most time, as it has to check for both `Spacing` and `Goal Bias`. The `Normal` variant takes the least time, as it doesn't have to check for either.
- One interesting thing to note is that the `Spaced` variant takes more time than the `Greedy` variant in Environment 2. This makes sense considering that, the program generates a lot more random nodes, as a lot of them are discarded due to the `Spacing`. Meanwhie, for `Greedy`, the program has a chance to choose a closer node, but it doesn't have to generate as many random nodes.

Task 2

Automata

2.1 Introduction

In this task, we have to implement a simple regex engine using FSM (Finite State Machine). We are given certain simplifications and we have to implement an engine that can match the following:

1. Direct match
2. Any number of characters
3. Wildcard Characters
4. Multiple matches

2.2 Implementation

2.2.1 FSM

To quote from the Wikipedia article on FSM:

> A Finite State Machine (FSM) is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

This doesn't help much, so let's look at a simple example, that is relevant to our task of regular expression matching.

2.2.2 Direct Match

Consider the regular expression `abcd`. This is a direct match, i.e., it matches exactly for the string `abcd`. We can represent this as a FSM, as shown below:

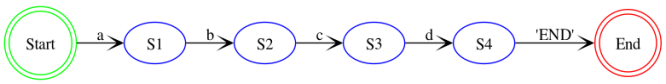


Figure 2.1: FSM for the regular expression `abcd`

In this, we have 6 states. The first one is the 'Start' and the final one is the 'End'. We will have one pointer that will move from one state to another, based on the input string. If at any point, the pointer reaches the 'END' state, then it means that we have a match.

This works in the following manner:

1. The pointer starts at the 'START' state.
2. If the FSM encounters the character 'a', then it moves to the next state (S1).
3. In this new state, the FSM checks for the character 'b'. If it is 'b', then it moves to the next state (S2). If it is not 'b', then the FSM resets to the 'START' state.
4. This process continues until the pointer reaches the 'END' state. If the pointer reaches the 'END' state, then it means that we have a match.

This is the simplest of the FSMs, and we can build upon this to match more complex regular expressions.

2.2.3 The Asterisk

In regex, the asterisk (*) is used to match zero or more occurrences of the preceding character. For example, the regex `ab*c` will match `ac`, `abc`, `abbc`, etc.

One of the simplest ways to implement this is to use a self-loop in the FSM. This means that the current state can loop back to itself, if the character matches. This is shown in the figure below:

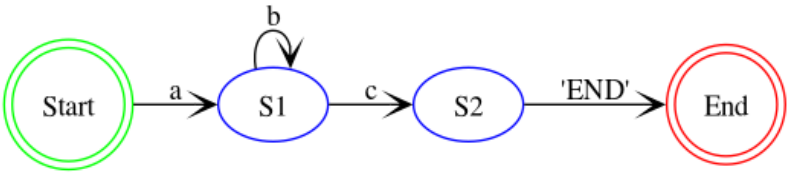


Figure 2.2: FSM for the regular expression `ab*c`

If the FSM gets the character `a`, it transitions to S1. At this state, the FSM can either move to the next state (S2) for the character 'c', or it can loop back to the same state for the character 'b'. This way, the FSM can match any number of 'b's, and then finally match 'c'. If the FSM encounters any other character, then it resets to the 'START' state.

2.2.4 Wildcards and other Special Characters

Regex has some special characters. For example:

1. `.` - The **dot** or **Wildcard** can match any character. For example, the regex `ab.d` will match `abcd`, `abbd`, `abzd`, etc.

   The implementation of the dot, is pretty simple. We can just add an edge corresponding to the dot, from the current state to the next state. We can check for the dot using a simple if condition within `FSM.match` function.

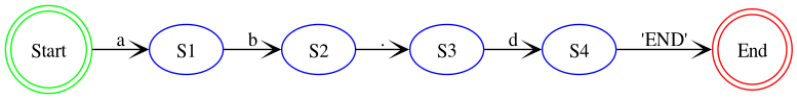


Figure 2.3: FSM for the regular expression `ab.d`

2. `+` - The **plus** matches one or more occurrences of the preceding character. For the `+`, observe that it is somewhat similar to the implementation of `*`, that we did earlier. Instead of a self-loop to the current state like in the case of `*`, we can add the self-loop to a new state. This will make it equivalent to using with the `*`. For example, `ab+c` will be equivalent to `abb*c`.

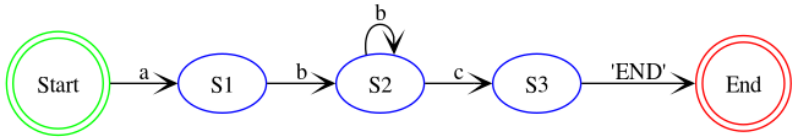


Figure 2.4: FSM for the regular expression `ab+c`

3. `?` - The **question mark** matches zero or one occurrence of the preceding character. Implementation of the `?` is also simple. We can have two transitions from the current state, one for the one occurrence and one for zero occurrences.

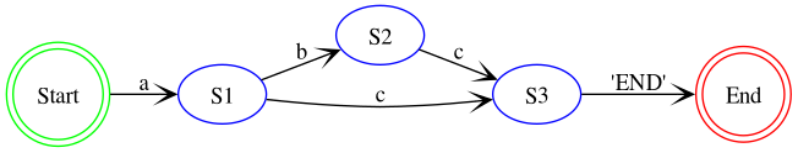


Figure 2.5: FSM for the regular expression `ab?c`. Notice the **two** transitions from State 1 to State 2 and State 3.

4. `^` and `$` - The **caret** matches the start of the string, and the **dollar** matches the end of the string. For `^` and `$`, it is simpler to check the start and end of the string during matching (like in the case of the Wildcard operator: `.`), rather than building a FSM for it.

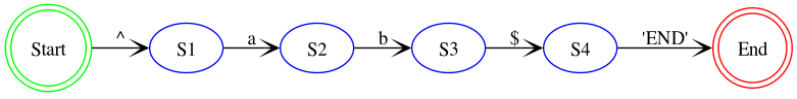


Figure 2.6: FSM for the regular expression `^ab$`

2.2.5 Multiple Matches

In order to match multiple occurrences, I implemented the `match` function in such a way that it has one or more pointers.

- I first go through the string and check if a character is directly connected to the START state. This means that, the character is valid start for a pattern.
- With this list of pointers, I select one pointer at a time, and move it through the FSM. If at any point, there are more than one valid transitions, then I create a new pointer for each transition. This way, I can match multiple occurrences.
- In a way, I am basically traversing the FSM using **Depth First Search (DFS)**. I keep track of the pointers that are valid at each step, and move them accordingly.
- Note that, I have only one moving pointer at any given time, although I have a list of valid pointers. This particular pointer either reaches the END state if it is a valid match, else it is discarded, and we start using the next valid pointer. This satisfies the condition that the FSM can be in exactly one state at any given time.
- I maintain a list of valid matches. If a pointer reaches the END state, then I add the start and end index of the match to the list. After the list of all pointers is exhausted, I return this list of valid matches.
- The indices within this list, are part of the match. I used the python library `termcolor` to highlight the matches in the string.

2.2.6 To Infinity and Beyond

Until this point, we have implemented the basic regex engine. We can use `*, +, ., ?, ^, $` and multiple matches. However, this is a very very small subset of what regex can do. Here are some of the things that we can do to improvise the engine:

- Until now, I have not talked about any usage of backtracking. Most regex engines use backtracking to match the regex. Consider the regular expression `ab*bc`. Ideally, the engine should match `abbbc`, but it is not able to at the current stage. To understand why this happens: We know that there is a self loop going from State 2 to State 2. This works for cases like `ab*` or `ab*c`. However, in the case of `ab*b`, the engine **overwrites** the self loop transition, and creates a new transition from State 2 to State 3 for the character 'b'. This is because, currently the FSM is designed to be deterministic, and it cannot have multiple transitions for the same character from the same state. This is the problem that backtracking can solve. However, there is another way to fix this. We can use a **Non-Deterministic Finite Automata (NFA)**. In an NFA, we can have multiple transitions for the same character from the same state. This is exactly what we need for the regex engine. In our example, we can have a transition from State 2 to State 2 for the character 'b' and another transition from State 2 to State 3 for the character 'b'. This way, we can match `abbbc`.

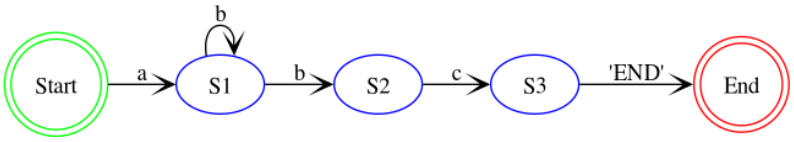


Figure 2.7: FSM for the regular expression `ab*bc`

- We can also implement **Character Sets** (`[ ]`), **Negated Character Sets** (`[^]`) and **Alternation** (`|`). This also means we need our engine to be able to handle **Ranges** like `[a-z]` and `[^0-9]`.
- However, I am **not** going to implement **Quantifiers**, **Grouping**, **Capturing**, **Lookahead** and **Lookbehind**, as they are more complex and require backtracking.

2.2.7 Character Sets

Character sets are used to match any one of the characters within the square brackets. For example, the regex `[abc]` will match any one of the characters `a`, `b` or `c`. We can implement this by adding multiple transitions from the current state to the next state, for each character in the set.

But before that, we need to keep in mind that there are also ranges in character sets. For example, `[a-z]` will match any character from `a` to `z`. One way to implement this is by having 26 transitions from the current state to the next state, for each character in the range. However, this is not very memory efficient. Moreover, we also need to handle negated character sets like `[^a-z]`.

In order to implement this:

- I created another class `Token`. A `Token`, is the smallest unit in the regex engine. It can be a character, a wildcard, a range, etc.
- It has an attribute `negated`, which is `True` if the token is negated.
- Until now, we have been using characters to build our transitions. Now, we will use `Tokens`, which gives us a lot more flexibility, and makes the implementation easier in the cases of Negated Character Sets and Ranges.
- Tokens also solve the problem where a new transition with the same character overwrites the previous transition. This doesn't happen with Tokens, as they are unique objects. This allows us to have multiple transitions for the same character from the same state.

Coming back to character sets, it was done in a simple way, as there can be no nesting or special characters within the character set. I just added each element of the set as a `Token` and added transitions for each of them, where all of them lead to the same state.

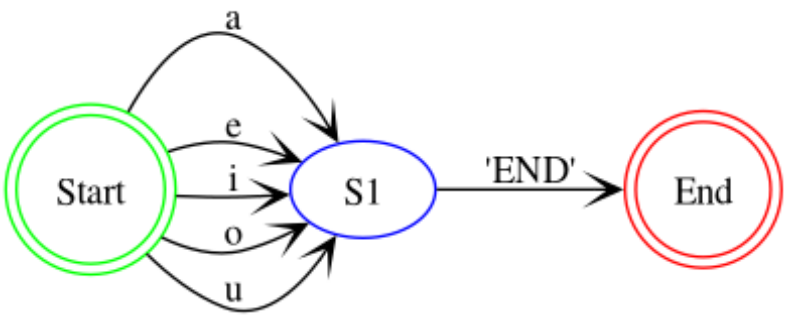


Figure 2.8: FSM for the regular expression `[aeiou]`

The best part about this is that, becuase ranges are also `Tokens`, they will now work with special characters like `.`, `*`, etc. For example, the regex `[a-z]*` will match any string that has only lowercase alphabets.

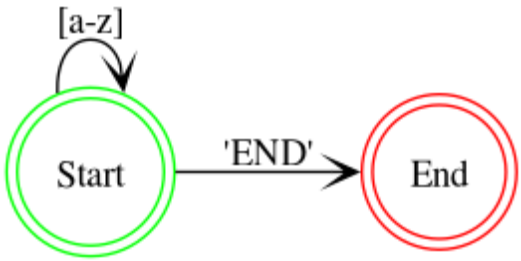


Figure 2.9: FSM for the regular expression `[a-z]*`

2.2.8 Escape from Reality

In regex, escaped characters are used to match special characters. For example, will match the character `*`, instead of matching zero or more occurrences of the preceding character. This is easy to implement, as we can just add an attribute `escaped` to the `Token` class, and check for it in the `match` function. The Escape Characters that I implemented are:

- `\w` - Matches any word character or underscore. It is equivalent to `[a-zA-Z0-9_]`.
- `\W` - Matches any non-word character and non-underscores. It is equivalent to `[^a-zA-Z0-9_]`.
- `\d` - Matches any digit. It is equivalent to `[0-9]`.
- `\D` - Matches any non-digit. It is equivalent to `[^0-9]`.
- `\s` - Matches any whitespace character such as spaces, tabs and line breaks.
- `\S` - Matches any non-whitespace character.
- `\b` - Matches a word boundary.
- `\B` - Matches a non word-boundaries.

I also added a few more Escape characters (that are not very common):

- `\t` - Matches a tab character.
- `\n` - Matches a newline character.
- `\v` - Matches a vertical tab character.
- `\f` - Matches a form feed character.
- `\r` - Matches a carriage return character.

With this, we can finally do something useful, such as `[\w]+@[\w]+\.[\w]+`. This regular expression can match basic email addresses.

2.2.9 Alternation

The `|` character acts like a boolean OR. For example, the regex `good|bad` will match either `good` or `bad`. This is simple to implement. Assuming that the regular expression doesn't have any parantheses, we can just split the regex at the `|` character, and build the FSM for each part. We can then merge the FSMs by adding transitions from the START state to the FSMs of each part. Of course, we have to consider the case where the `|` character is escaped, in which case it should be treated as a normal character.

2.3 Code

Throughout the implementation, I am working with the FSM by considering it as a directed graph. Each node in the graph is a `State` and each edge corresponds to a transition. The transitions are stored in a dictionary, where the key is a `Token`, and the value is the next state.

2.3.1 Imports

I used `Graphviz` to plot the FSMs, and `termcolor` to highlight the matches in the string.

Listing 2.1: Imports

```
#!/usr/bin/python3

from typing import Union

from graphviz import Digraph
from termcolor import colored
```

2.3.2 FSM

The whole implementation is done within the `FSM` class. The class has some subclasses like `Token` and `State`.

Listing 2.2: FSM

```
class FSM:
    def __init__(self, regexp: str):
        self.regexp = regexp
        self.start = FSM.State(is_start=True)
        self.end = FSM.State(is_end=True)
        self.split_and_compile()
```

Token

The `Token`, as mentioned earlier, is the smallest unit in the regex engine. It can be a character, a wildcard, a range, etc. It has attributes like `negated` and `escaped`. The `__repr__` function is used to represent the token in a human-readable format.

Listing 2.3: `FSM.Token`

```
    class Token:
        def __init__(
            self,
            range: Union[str, tuple[str, str]],
            negated: bool = False,
            escaped: bool = False,
        ):
            if isinstance(range, tuple):
                start, end = range
            else:
                start, end = range, range
            self.start = start
            self.end = end
            self.length = ord(end) - ord(start) + 1
            self.negated = negated
            self.escaped = escaped

        def __repr__(self) -> str:
            if "END" == self.start:
                return "END"
            if "\\" == self.start:
                return r"\\"
            pre = "^" if self.negated else "" + r"\\" if self.escaped
                else ""
            if self.length == 1:
                return f"{pre}{self.start}"
            return f"{pre}[{self.start}-{self.end}]"
```

The `FSM.Token` class has one function `__contains__`, which is used to check if a character is in the range of the token. It returns a boolean value which depends on the `negated` attribute.

Listing 2.4: `FSM.Token.__contains__`

```
        def __contains__(self, char: str) -> bool:
            return (self.start <= char <= self.end) ^ self.negated
```

State

The `State` class is used to represent a node in the FSM. It has attributes like `is_start` and `is_end`. Most Importantly, It has the dictionary `transitions`, which stores the transitions from the current state to the next state. The `__repr__` function is used to represent the state in a human-readable format.

Listing 2.5: `FSM.State`

```
state_counter = 1

class State:
    def __init__(self, is_start: bool = False, is_end: bool =
       False):
        self.id = FSM.state_counter
        FSM.state_counter += 0 if is_start or is_end else 1
        self.transitions = {}
        self.is_start = is_start
        self.is_end = is_end

    def __repr__(self) -> str:
        return (
            f"Start" if self.is_start else f"End" if self.is_end
               else f"S{self.id}"
        )
```

The `FSM.State` class has a function `link`, which is used to add a transition from the current state to the target state.

Listing 2.6: `FSM.State.link`

```
        def link(self, token: Union["FSM.Token", str], target: "FSM.
           State"):
            self.transitions[token] = target
```

FSM Functions

Now, the functions of the `FSM` class are explained here:

- `split_and_compile`: This function splits the regular expression at the `|` character, and compiles the FSM for each part. It then merges the FSMs by linking these FSMs with the START and END state by calling the `FSM.compile` function. It also takes care of cases where the `|` character is escaped.

  Listing 2.7: `FSM.split_and_compile`

  ```
  def split_and_compile(self):
  ```

```
    split_idxs = []
    for i, char in enumerate(self.regexp):
        if char == "|":
            if i > 0 and self.regexp[i - 1] == "\\":
                continue
            split_idxs.append(i)

    prev = 0
    for split_idx in split_idxs:
        subexpr = self.regexp[prev:split_idx]
        prev = split_idx + 1
        self.compile(subexpr, self.start, self.end)

    subexpr = self.regexp[prev:]
    self.compile(subexpr, self.start, self.end)
```

- `compile`: This function is used to compile the FSM for a given regular expression. It does this by going through the regular expression character by character, and building the FSM accordingly. This is the heart of the regex engine. It uses subfunctions like `build` and `link` to build the FSM. Its functioning is explained here:
  - `FSM.compile.build`: This function looks at the next character in the regular expression, and decides how to link the current state to the next state. It uses the `link` function to add the transitions.
  - `FSM.compile.link`: This function is used to add transitions from the current state to the target state. It also takes care of adding transitions for multiple parents. The cases of multiple parents arise when we have operators such as `*`, and `?` which can have zero occurrences. This means that the state before these can also lead to the target state. Therefore, the states before the `current` state are maintained in a list called `parents`.
  - With the help of these functions, a forward pass is made through the regular expression, and the FSM is built accordingly, considering features like **Character Sets**, **Negated Character Sets**, **Ranges**, **Escaped Characters**, etc.
  - Finally, the function adds transitions from the `parents` and `current` state to the `END` state.

Listing 2.8: `FSM.compile`

```
def compile(self, regexp: str, start: "FSM.State", end: "FSM.
   State"):
    current = start
    parents = []
    i = 0

    def build(
        tokens: Union["FSM.Token", set["FSM.Token"]],
        next_char: str,
        target: Union["FSM.State", None],
```

```
):
    nonlocal current, parents
    if next_char == "*":
        current, parent = link(tokens, current, current,
           parents)
        parents.append(parent)
    elif next_char == "+":
        current, _ = link(tokens, current, target, parents)
        link(tokens, target, target, parents)
        parents = []
    elif next_char == "?":
        current, parent = link(tokens, current, target,
           parents)
        parents.append(parent)
    else:
        current, _ = link(tokens, current, target, parents)
        parents = []

def link(
    tokens: Union["FSM.Token", set["FSM.Token"], str],
    current: "FSM.State",
    target: "FSM.State",
    parents: list["FSM.State"],
):
    if isinstance(tokens, FSM.Token) or isinstance(tokens,
       str):
        tokens = {tokens}
    for token in tokens:
        current.link(token, target)
        for parent in parents:
            parent.link(token, target)

    return target, current

while i < len(regexp):
    char = regexp[i]
    if char in ("*", "+", "?"):
        i += 1
        continue

    if char == "[":
        i += 1
        negated = regexp[i] == "^"
        i += 1 if negated else 0
        tokens = set()
        while regexp[i] != "]":
            escaped = False
```

```
                if regexp[i] == "\\":
                    escaped = True
                    i += 1
                if (
                    regexp[i + 1] == "-"
                    and regexp[i + 2] != "]"
                    and ord(regexp[i]) < ord(regexp[i + 2])
                ):
                    tokens.add(
                        FSM.Token((regexp[i], regexp[i + 2]),
                           negated, False)
                    )
                    i += 2
                else:
                    tokens.add(FSM.Token(regexp[i], negated,
                       escaped))
                i += 1

            next_char = regexp[i + 1] if i < len(regexp) - 1 else
                None
            target = FSM.State() if next_char != "*" else None
            build(tokens, next_char, target)
        else:
            escaped = False
            if char == "\\":
                escaped = True
                i += 1
                char = regexp[i]
            next_char = regexp[i + 1] if i < len(regexp) - 1 else
                None
            target = FSM.State() if next_char != "*" else None
            build(FSM.Token(char, escaped=escaped), next_char,
               target)
        i += 1

    for parent in parents:
        parent.link("END", end)
    parents = []
    link("END", current, end, parents)
```

- `match`: This function is used to match the regular expression with a given string. Its functioning is explained here:
  - It maintains a list of valid pointers, and moves them through the FSM. If a pointer reaches the END state, then it is added to the list of valid matches. The indices of these matches are then returned.

– It also implements the ^ and $ characters, by checking for these transitions at the start and end of the string.

– The function has a subfunction `FSM.match.transition` which looks at the current character in the string and all the transitions from the current state, and moves the pointer accordingly. This is the function that implements the **Depth First Search (DFS)**. Some examples are:

  * If the transition is \d and the current character is a digit, then the pointer moves to the corresponding state.
  * If there is a transition for the character ., then the pointer moves to the corresponding state, irrespective of the current character.

Listing 2.9: `FSM.match`

```
def match(self, string: str) -> set[int]:
    matched_chars = set()

    def transition(string, ptr, idx, start):
        nonlocal matched_chars
        char = string[idx] if idx < len(string) else None
        if idx in matched_chars:
            return
        for token in ptr.transitions:
            if "END" == token:
                matched_chars.update(list(range(start, idx)))
                continue

            if idx == len(string):
                if "$" == token.start:
                    transition(string, ptr.transitions[token],
                       idx, start)
            else:
                if token.escaped:
                    if token.start in (
                        *("w", "W", "d", "D", "s", "S"),
                        *("t", "n", "v", "f", "r"),
                    ):
                        if (
                            ("w" == token.start and char.isalnum
                               () or char == "_")
                            or (
                                "W" == token.start
                                and not char.isalnum()
                                and char != "_"
                            )
                            or ("d" == token.start and char.
                               isdigit())
```

```
                or ("D" == token.start and not char.
                   isdigit())
                or ("s" == token.start and char.
                   isspace())
                or ("S" == token.start and not char.
                   isspace())
                or ("t" == token.start and ord(char)
                   == 9)
                or ("n" == token.start and ord(char)
                   == 10)
                or ("v" == token.start and ord(char)
                   == 11)
                or ("f" == token.start and ord(char)
                   == 12)
                or ("r" == token.start and ord(char)
                   == 13)
            ):
                transition(
                    string, ptr.transitions[token],
                       idx + 1, start
                )
        elif token.start in ("b", "B"):
            if (
                (
                    "b" == token.start
                    and (idx == 0 or string[idx - 1].
                       isspace())
                    and string[idx].isalnum()
                )
                or (
                    "b" == token.start
                    and (
                        idx == len(string) - 1
                        or string[idx + 1].isspace()
                    )
                    and string[idx].isalnum()
                )
                or (
                    "B" == token.start
                    and not (idx == 0 or string[idx -
                        1].isspace())
                    and string[idx].isalnum()
                )
                or (
                    "B" == token.start
                    and not (
                        idx == len(string) - 1
```

```
                                    or string[idx + 1].isspace()
                                )
                                and string[idx].isalnum()
                            )
                        ):
                            transition(string, ptr.transitions[
                               token], idx, start)
                    elif char in token:
                        transition(string, ptr.transitions[token
                           ], idx + 1, start)
                else:
                    if idx == 0 and "^" == token.start:
                        transition(string, ptr.transitions[token
                           ], idx, start)

                    if char in token or "." == token.start:
                        transition(string, ptr.transitions[token
                           ], idx + 1, start)

    for idx, char in enumerate(string):
        transition(string, self.start, idx, idx)

    return matched_chars
```

- `plot`: Finally, I have a function to plot the FSM. This function uses the `Graphviz` library to plot the FSM. It uses the `Digraph` class to create the directed graph, and adds the nodes and edges accordingly. The function then saves the graph as a `.png` file.

Listing 2.10: `FSM.plot`

```
def plot(self, filename=None, view: bool = False):
    g = Digraph(format="png")
    g.attr(rankdir="LR", compound="true", concentrate="true")

    state_counter = 1
    states = [(self.start, "Start")]
    names = {id(self.start): "Start"}

    while states:
        ptr, ptr_name = states.pop()
        for token, target in ptr.transitions.items():
            if id(target) not in names:
                target_name = target.__repr__()
                names[id(target)] = target_name
                state_counter += 1
                states.append((target, target_name))
            else:
                target_name = names[id(target)]
```

```
                g.edge(ptr_name, target_name, label=token.__repr__(),
                    arrowhead="vee")

            if ptr.is_end:
                g.node(ptr_name, shape="doublecircle", color="red")
            elif ptr.is_start:
                g.node(ptr_name, shape="doublecircle", color="green")
            else:
                g.node(ptr_name, color="blue")

        if filename is None:
            filename = self.regexp.replace("*", "8")

        if view:
            g.view(cleanup=True)
        else:
            g.render(
                filename=rf"data/images/{filename}",
                format="png",
                cleanup=True,
            )
```

2.3.3 Usage

I have a simple function `highlight` to highlight the matches in the string. It uses the `termcolor` library to color the matches in red.

Listing 2.11: `highlight`

```
def highlight(string: str, matched_chars: set[int]) -> str:
    highlighted = list(string)

    highlighted = [
        colored(char, "red") if idx in matched_chars else char
        for idx, char in enumerate(highlighted)
    ]

    return "".join(highlighted)
```

Finally, the usage of the code is shown here:

Listing 2.12: Usage

```
regexp = input("Regular␣Expression:␣")
fsm = FSM(regexp)

string = input("Sample␣string:␣")
matches = fsm.match(string)
```

```
highlighted = highlight(string, matches)

print(highlighted)
fsm.plot(view=True)
```

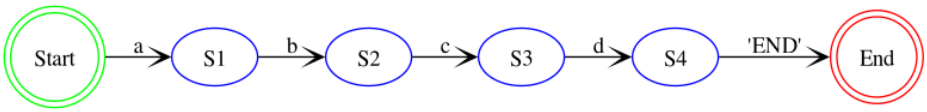
2.4 Results

Here is a list of all the features that I implemented in the regex engine:

- Direct Matches
- Wildcard Character
- Special Characters: `*`, `+`, `?`, `^`, `$`
- Multiple Matches
- Character Sets, Negated Character Sets and Ranges
- All Escape Characters
- Alternation `|`

2.4.1 Some FSMs

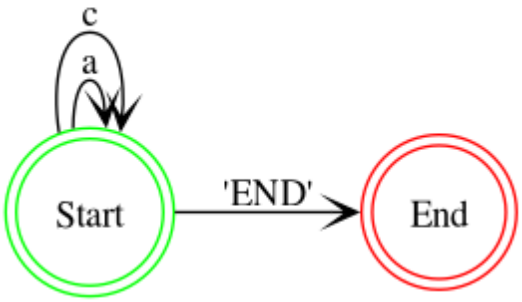
Here are the FSMs for some of the regular expressions that I tested:



(a) FSM for the regular expression `abcd`

```
Regular Expression: abcd
Sample string: abcdef
abcdef
```

(b) Direct Match

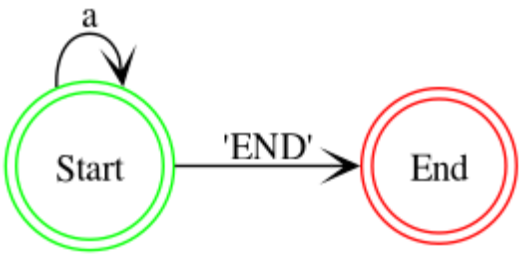


(a) FSM for the regular expression `a*c*`

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

(b) Any Number of 'a's and 'c's

Figure 2.11: Special Characters

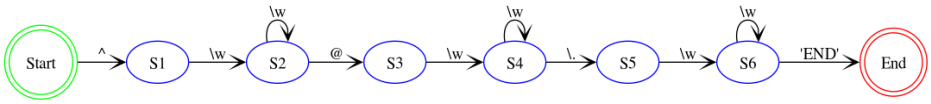


(a) FSM for the regular expression `a*`

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

(b) Multiple Matches

Figure 2.12: Special Characters

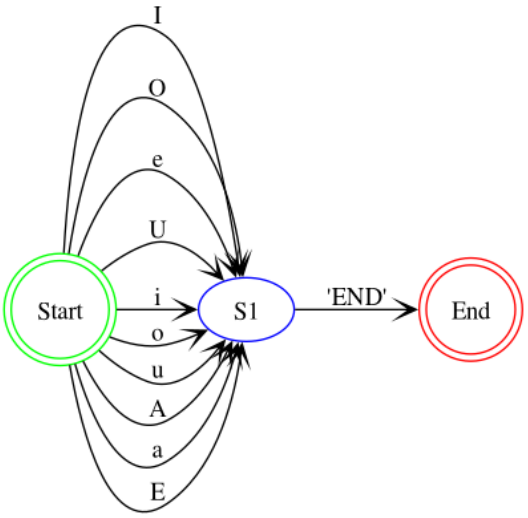


(a) FSM for matching simple email addresses

```
Regular Expression: ^[\w]+@[\w]+\.[\w]+
Sample string: j0hn_d0e@example.com
j0hn_d0e@example.com
```

(b) Matched Email Address

Figure 2.13: Escape Characters

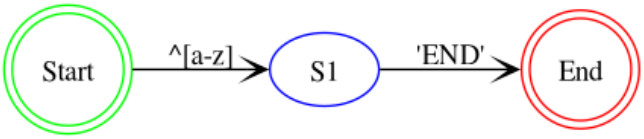


(a) FSM for matching vowels

```
Regular Expression: [aeiouAEIOU]
Sample string: Anton Beny M S
Anton Beny M S
```

(b) Matched Vowels

Figure 2.14: Character Sets

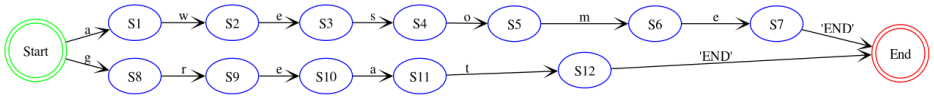


(a) FSM for [ˆa-z]

```
Regular Expression: [^a-z]
Sample string: abcABCdef123
abcABCdef123
```

(b) Matches for non-lowercase alphabets

Figure 2.15: Negated Character Sets



(a) FSM for the regular expression `awesome|great`

```
Regular Expression: awesome|great
Sample string: regex is awesome and great!
regex is awesome and great!
```

(b) Alternation

2.4.2 Greedy Matches

One interesting thing that I learnt while implementing the regex engine is that, the engine is **greedy**.

It is shown in the examples below:

```
Regular Expression: ab*
Sample string: aaabbbbbaaaabbbb
aaabbbbbaaaabbbb
```

```
Regular Expression: aab*
Sample string: aaabbbbbaaaabbbb
aaabbbbbaaaabbbb
```

```
Regular Expression: aaab*
Sample string: aaabbbbbaaaabbbb
aaabbbbbaaaabbbb
```

```
Regular Expression: aaaab*
Sample string: aaabbbbbaaaabbbb
aaabbbbbaaaabbbb
```

We can notice in the second example, that the engine **does not** match the string. This is because the first two `a`'s are **used** in matching the pattern. As the engine is `greedy`, it now will not consider the second `a` as a valid start for the pattern. Therefore, it **does not** match the middle part of the string.