TAKE HOME END SEMESTER

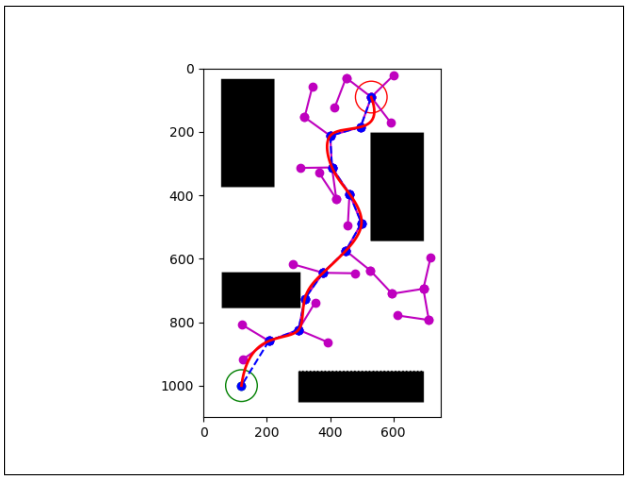
Vakati Venkata Akshit MM23B009

21 May 2024

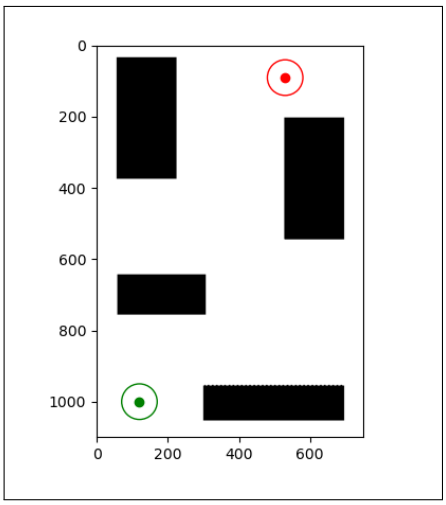
1 Path planning:

In this report, we will discuss the implementation of the Rapidly-exploring Random Tree (RRT) algorithm using Python. The algorithm is used to find a path from a starting point to a goal point in a given grid-based environment.
The RRT algorithm constructs a tree rooted at the starting point. It iteratively samples random points in the grid, extends the tree toward these points, and connects them to the nearest point in the tree, if feasible.

1.1 Setup



(a) RRT path along with smooth path from start to goal.



(b) Setup of grid

- **start** = [530.0, 90.0]
- **goal** = [120.0, 1000.0]
- The image is loaded and resized to dimensions 750×1100.
- The image is converted to grayscale and inverted.
- The inverted image is binarized, resulting in a binary grid where 1 represents foreground (white) and 0 represents background (black).
- The resulting binary grid is saved as a numpy array (`.npy`).

1.2 Usage of the files:

- When you untar the ***mm23b009.tar*** file a directory named *mm23b009* would be created
- Going to ***question_1*** run *question_1a.sh* and *question_1b.sh* . run these files so that **RRT plots** would be created. One for the Given Environment and other for my custom grid.

./question_1a.sh

./question_1b.sh

1.3 Code Analysis for normal RRT

The `TreeNode` class represents a node in the RRT tree structure with attributes for X and Y coordinates (`locationX`, `locationY`), child nodes (`children`), and a parent node (`parent`).

The `RRTAlgorithm` class implements the Rapidly-exploring Random Tree (RRT) algorithm for path planning. It is initialized with parameters including the starting point (`start`), the goal point (`goal`), the maximum number of iterations (`numIterations`), the grid environment representing obstacles (`grid`), and the step size for extending the tree (`stepSize`).

- `start` (starting point)
- `goal` (goal point)
- `numIterations` (maximum iterations)
- `grid` (grid environment with obstacles)
- `stepSize` (step size for extending the tree)

The algorithm:

- Starts with `start` as the root node and `goal` as the target.
- Iteratively extends the tree towards random points (`sampleAPoint`) in the grid.
- Finds the nearest node (`findNearest`) and steers towards it (`steerToPoint`).
- Checks obstacle collision (`isInObstacle`) before adding a new node (`addChild`).
- Terminates when the goal is reached (`goalFound`) and traces back the path (`retraceRRTPath`).

The algorithm computes Euclidean distances (`distance`) and uses cubic splines for path smoothing.

1.4 Greedy Nature of the Goal RRT Algorithm

The **"greediness"** in the **Greedy RRT** algorithm primarily refers to the **goal-biased sampling** strategy employed during the path planning process. This strategy enhances the algorithm's efficiency, convergence, and adaptability:

- **Goal-Biased Sampling:** Points closer to the goal are prioritized during sampling, accelerating the search towards the goal and increasing the likelihood of finding a path sooner.
- **Enhanced Convergence:** By favoring points near the goal, the algorithm accelerates the exploration towards the desired destination, reducing the number of iterations needed to connect the start and goal points.

- **Improved Efficiency:** In complex environments with obstacles, the Greedy RRT navigates more effectively by focusing exploration efforts on promising regions of the grid.
- **Reduced Computational Cost:** The goal bias decreases the number of iterations required to find a valid path, making the algorithm suitable for real-time applications.
- **Path Quality:** Despite the bias, the algorithm explores the entire grid, ensuring high-quality paths that are further refined and smoothed using cubic splines.
- **Adaptability:** The algorithm adjusts the bias based on obstacles and goal location, making it adaptable to different environments and grid configurations.

In summary, the Greedy RRT algorithm's **greediness** is evident in its **goal-biased sampling**, which accelerates the search towards the goal while improving efficiency, reducing computational cost, and maintaining high path quality in complex environments.

1.5 Comparison of RRT and Greedy RRT Over 1000 Runs

The histograms depict the performance differences between the standard RRT and the Greedy RRT algorithms over 1000 runs. Below are key points for each plot:

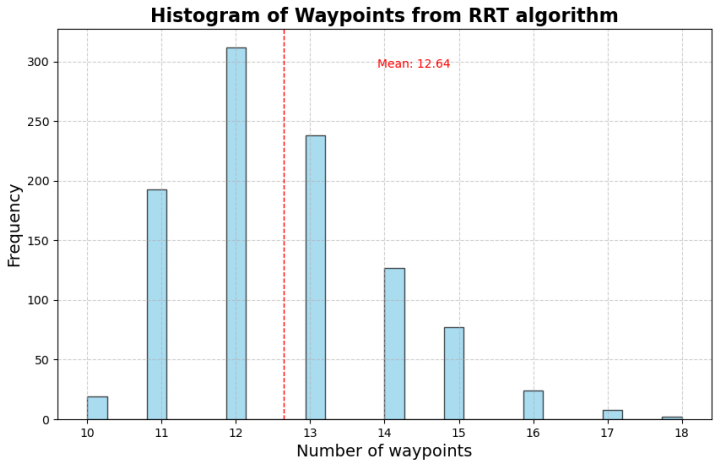


Figure 2: Histogram for RRT algorithm run over 1000 times

Standard RRT Histogram:

- Shows the distribution of path lengths or iterations needed for standard RRT to find a valid path.
- Typically displays a wider spread, indicating higher variability in performance.
- Longer tails suggest occasional significantly longer paths or more iterations.
- The mean and median values are higher compared to Greedy RRT, reflecting less efficiency.
- The mean path length or iteration count is 12.64, reflecting less efficiency compared to Greedy RRT.
- all the output paths of 1000 runs are attached under **/question_1/plots/RRT/***

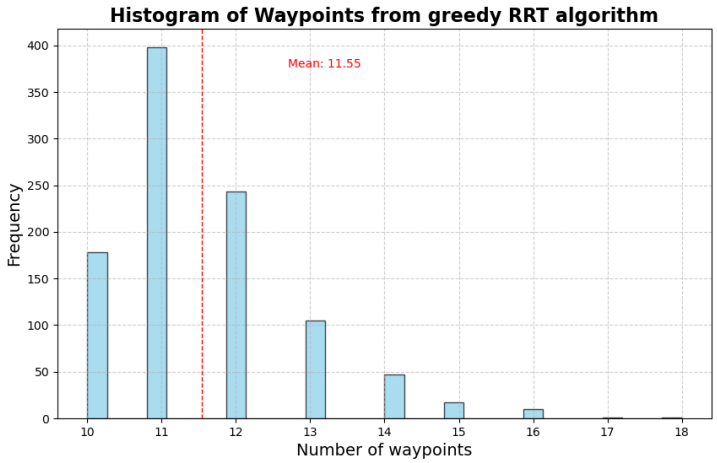


Figure 3: Histogram for goal RRT algorithm run over 1000 times

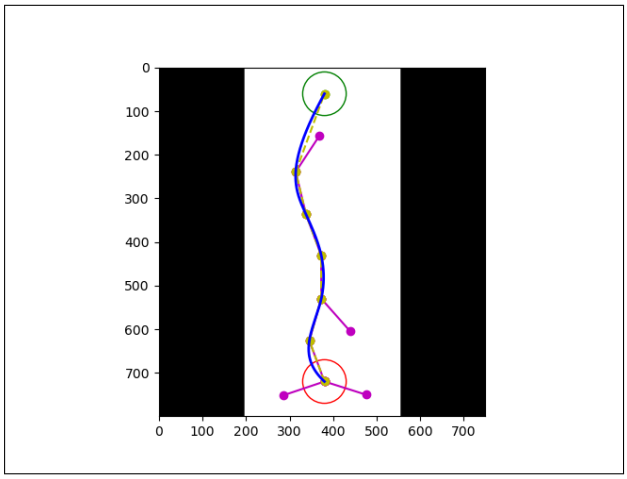
Greedy RRT Histogram:

- Illustrates the distribution of path lengths or iterations for Greedy RRT.
- Generally exhibits a narrower spread, indicating more consistent performance.
- Shorter tails suggest fewer instances of significantly long paths or iterations.
- The mean and median values are lower compared to standard RRT, highlighting improved efficiency.
- The mean path length or iteration count is 11.55, highlighting improved efficiency compared to standard RRT
- all the output paths of 1000 runs are attached under **/question_1/plots/gRRT/***

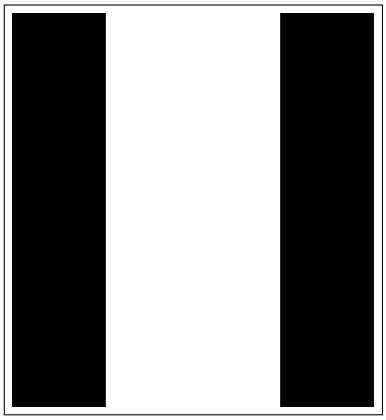
Performance Comparison:

- Greedy RRT consistently outperforms standard RRT in terms of path length and iteration count.
- The goal-biased sampling in Greedy RRT leads to faster convergence and shorter paths.
- Standard RRT shows greater variability, with some runs requiring significantly more iterations.
- The histograms visually demonstrate the effectiveness of the goal-biased strategy in reducing computational cost and improving reliability. .

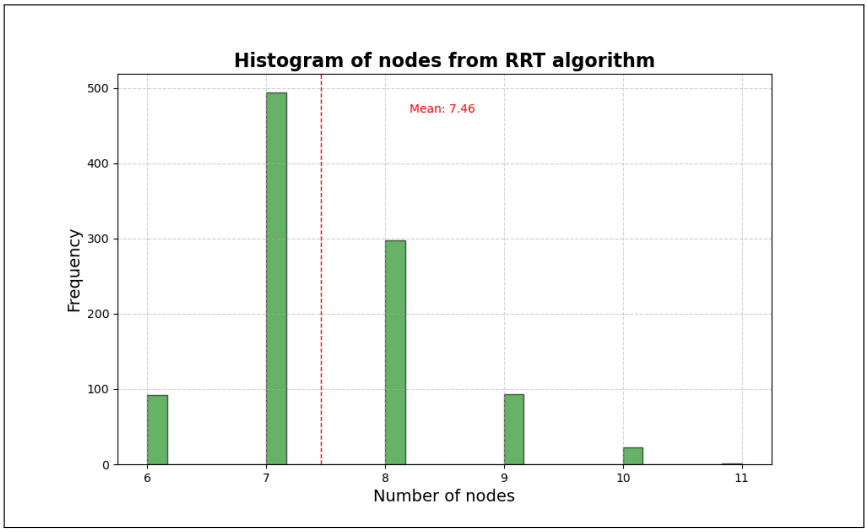
1.6 Analysis of the given environment :



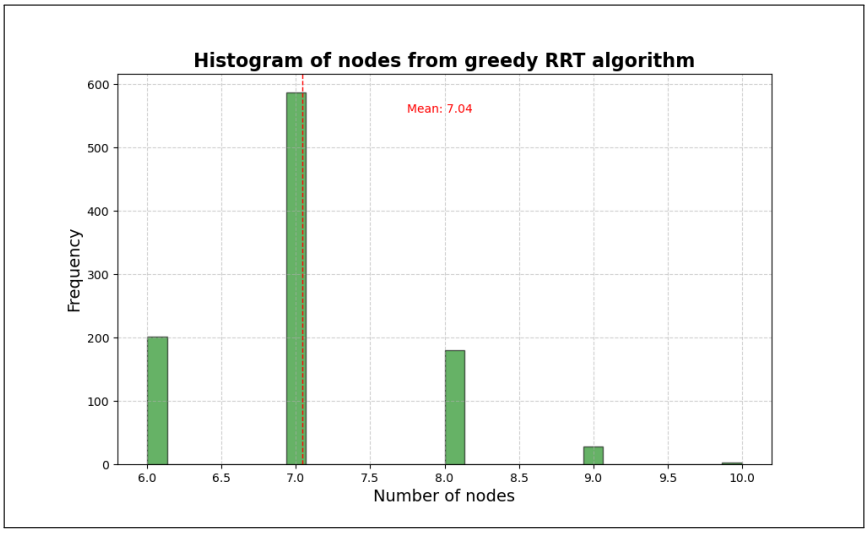
(a) RRT path along with smooth path from start to goal.



(b) Setup of grid



(a) RRT path along with smooth path from start to goal.



(b) Setup of grid

2 Automata

Finite state machines (FSMs) are abstract models of computation used to design both computer programs and sequential logic circuits. An FSM consists of a finite number of states, transitions between those states, and actions. In the context of regex, the states represent different parts of the pattern, and transitions occur based on the characters of the input string.

2.1 Regex Engine Implementation

The code demonstrates the implementation of a regex engine that supports direct matches, any number of characters, wildcard characters, and multiple matches. The implementation includes the following steps:

1. **Regex to NFA Conversion**: The input regex is converted into a non-deterministic finite automaton (NFA) using the 'automata' library.
2. **NFA to DFA Conversion**: The NFA is then converted into a deterministic finite automaton (DFA).
3. **FSM Class**: A class 'FSM' is defined to handle DFA operations such as matching strings, finding matches, and highlighting matched parts of the string.
4. **Highlighting Matches**: The code highlights matches in the input string by surrounding them with ANSI color codes.

2.2 Code Explanation

The following sections explain the implementation of the `FSM` class and related functions.

2.2.1 Imports

The necessary modules for DFAs and NFAs are imported:

- `from automata.fa.dfa import DFA`
- `from automata.fa.nfa import NFA`

2.2.2 FSM Class

- The `FSM` class is initialized with a DFA and defines colors for highlighting:
  - `__init__` method sets up the DFA and color codes.
- The `match` method checks if a string is accepted by the DFA:
  - It returns `True` if the DFA accepts the input string, otherwise `False`.
- The `find_matches` method identifies all substrings in the input string that match the DFA:
  - It iterates over all possible substrings and uses the `match` method to check for acceptance.
  - It returns a list of tuples indicating the start and end indices of matched substrings.
- The `highlight_match` method uses `find_matches` to highlight matching substrings:
  - If matches are found, it calls the `color_text` method to apply the highlight.

  - It returns the original string with highlighted matches.

- The `color_text` method applies specified colors to matched substrings:
  - It merges overlapping and contiguous matched indices.
  - It constructs a new string with color codes applied to matched substrings.

2.2.3 Regex to FSM Conversion

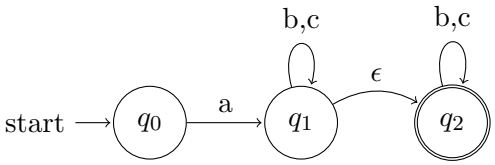
The `regex_to_fsm` function converts a regex pattern to an FSM:

- It first converts the regex pattern to an NFA.
- Then, it converts the NFA to a DFA.
- Finally, it returns an instance of the `FSM` class initialized with the DFA.

Example:
Consider the regular expression: `a(b|c)*`

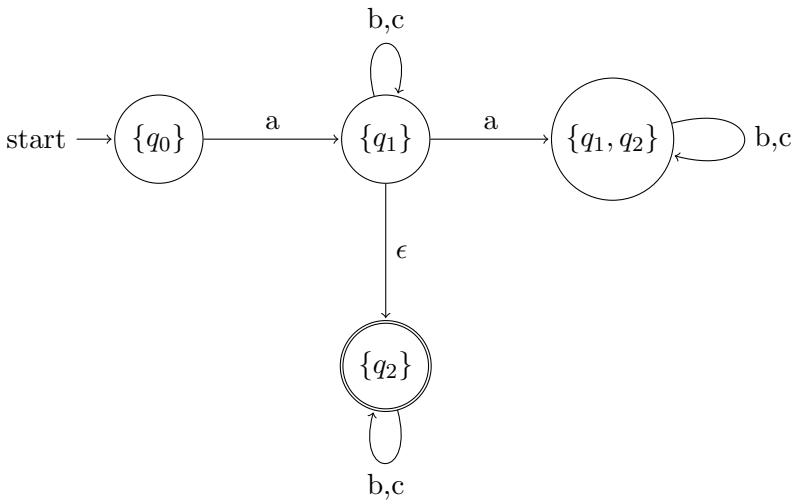
1. **NFA Construction:**
   - Start with the initial state.
   - Transition on `a` to a new state.
   - From this new state, transition on `b` or `c` to a looped state (representing the Kleene star operation).

NFA State Diagram:



2. **DFA Construction:**
   - Convert the NFA to an equivalent DFA by creating states that represent combinations of NFA states.

DFA State Diagram:



In these diagrams:

- The NFA includes epsilon transitions (ϵ) and multiple states to represent non-determinism.
- The DFA converts this into a deterministic model where each state represents a combination of NFA states, and there are no epsilon transitions.

Rules for FSM:

- A Finite State Machine (FSM) is defined by a finite set of states, including an initial state and one or more accepting states.
- Transitions between these states are triggered by input symbols.
- An FSM can be either a Non-deterministic Finite Automaton (NFA) or a Deterministic Finite Automaton (DFA).
- An NFA allows multiple transitions for the same input from a given state and can include epsilon transitions (transitions without input).
- A DFA has exactly one transition per input symbol from each state and does not include epsilon transitions.
- The conversion from NFA to DFA involves creating a state in the DFA for each possible combination of NFA states.
- This conversion ensures that the DFA is deterministic and can be implemented efficiently.

2.2.4 Main Function

The `main` function performs the following tasks:

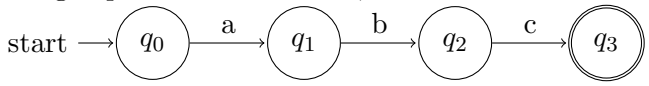
- It reads a regex pattern and a sample string from the user.
- It creates an FSM using the `regex_to_fsm` function.
- It highlights matches in the sample string using the FSM and prints the highlighted string.

2.3 State Diagram Examples

Here, we provide state diagrams for the specified cases:

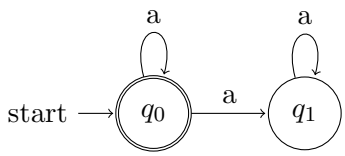
2.3.1 Direct Matches

For a regex pattern such as `abc`, the FSM would have the following states:



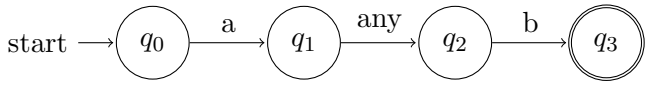
2.3.2 Any Number of Characters

For a regex pattern such as `a*`, the FSM would be:



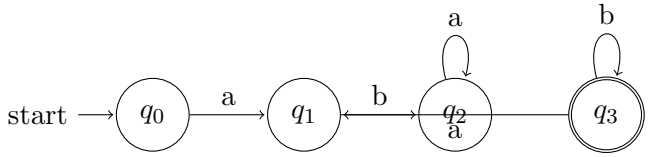
2.3.3 Wildcard Character

For a regex pattern such as `a.b`, the FSM would look like:



2.3.4 Multiple Matches

For a regex pattern such as `(ab)+`, the FSM would be:



2.4 USAGE:

- In *question_2* directory there is a file named **regex_fsm.py**
- run the file as:

  python3 regex_fsm.py

- Sample usage of the **regex_fsm.py** for `a*b(d|c)`:

  Regular Expression: `a*b(d|c)`

  Sample string: `ssaaaaaabdrrrraaaaabceeee`

  Output: `ssaaaaaabdrrrraaaaabceeee`

NOTE

This regex parser will only work for below combinations:

- `*`: Kleene star operation, language repeated zero or more times. Ex: `a*`, `(ab)*`
- `+`: Kleene plus operation, language repeated one or more times. Ex: `a+`, `(ab)+`
- `?`: Language repeated zero or one time. Ex: `a?`
- Concatenation. Ex: `abcd`
- `|`: Union. Ex: `a|b`
- `&`: Intersection. Ex: `a&b`
- `.`: Wildcard. Ex: `a.b`
- `^`: Shuffle. Ex: `a^b`
- {}: Quantifiers expressing finite repetitions. Ex: `a{1,2}`, `a{3,}`
- `()`: The empty string.
- `(...)`: Grouping.

References

[1] S. M. LaValle, *Rapidly Exploring Random Trees: A New Tool for Path Planning*, Technical Report TR 98-11, Computer Science Department, Iowa State University, 1998. `https://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf`

[2] Steven M. Lavalle, James J. Kuffner, *Progress and Prospects: Rapidly Exploring Random Trees (RRTs)*, Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA), 2000. DOI: 10.1109/ROBOT.2000.844730.

[3] Sertac Karaman, Emilio Frazzoli, *Sampling-based Algorithms for Optimal Motion Planning*, International Journal of Robotics Research, vol. 30, no. 7, pp. 846-894, June 2011. DOI: 10.1177/0278364911406761.

[4] *RRT Implementation by nimRobotics*, Available at: `https://github.com/nimRobotics/RRT`.

[5] Wikipedia contributors, *Rapidly exploring random tree — Wikipedia, The Free Encyclopedia*, 2024, `https://en.wikipedia.org/w/index.php?title=Rapidly_exploring_random_tree&oldid=1211274166`, [Online; accessed 24-May-2024].

[6] Michael Sipser, *Introduction to the Theory of Computation*, 3rd Edition, Cengage Learning, 2012.

[7] Peter Linz, *An Introduction to Formal Languages and Automata*, 6th Edition, Jones & Bartlett Learning, 2016.

[8] Jiacun Wang, *Formal Methods in Computer Science*, CRC Press, 2019, p. 34. ISBN 978-1-4987-7532-8.

[9] John E. Hopcroft, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 1st Edition, Addison-Wesley, 1979. ISBN 978-0-201-02988-8.

[10] Jeffrey E.F. Friedl, *Mastering Regular Expressions*, 3rd Edition, O'Reilly Media, 2006.

[11] Jan Goyvaerts, Steven Levithan, *Regular Expressions Cookbook*, 2nd Edition, O'Reilly Media, 2012.

[12] Wikipedia contributors, *Finite-state machine — Wikipedia,*, 2023, `https://en.wikipedia.org/w/index.php?title=Finite-state_machine&oldid=1187480774`.

[13] Wikipedia contributors, *Regular expression — Wikipedia,*, 2024, `https://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=1225278254`.

[14] *A Python library for simulating finite automata, pushdown automata, and Turing machines* | ***automata-lib***, Available at: `https://github.com/caleb531/automata/tree/main/automata/regex` | `https://pypi.org/project/automata-lib/`.