AI and Robotics

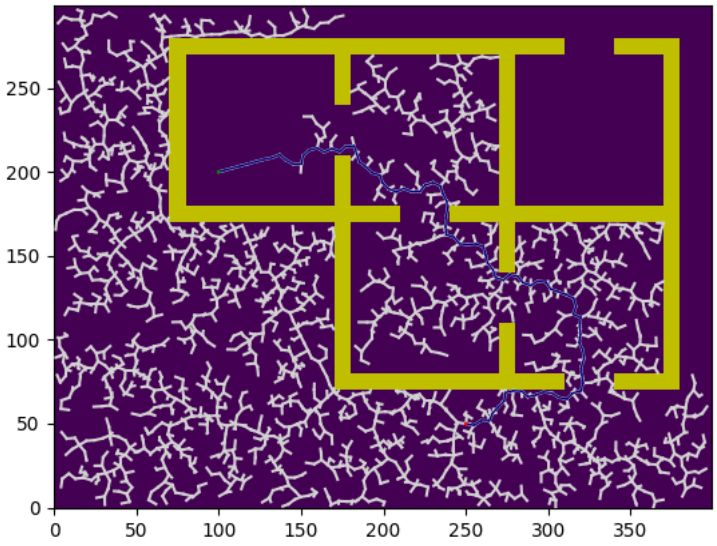
HW2

02360006

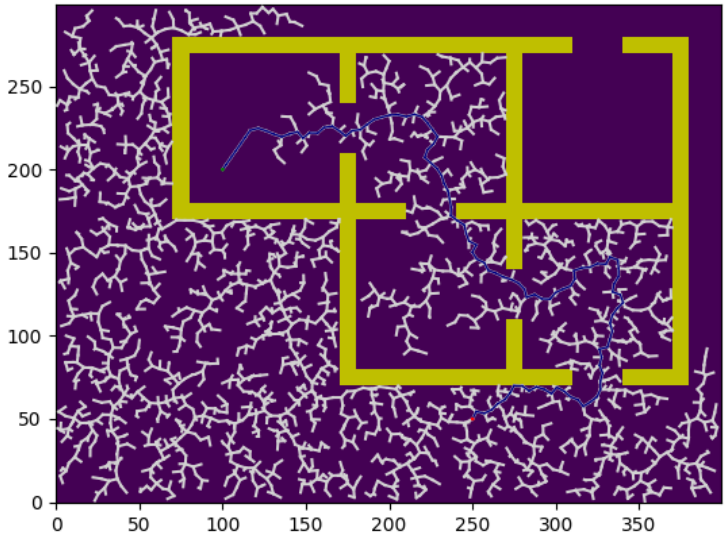
	סטודנט א'	סטודנט ב'
שם	אדם עאסי	הגר בר
מספר סטודנט	211696216	205564784

Part 3: RRT Algorithm

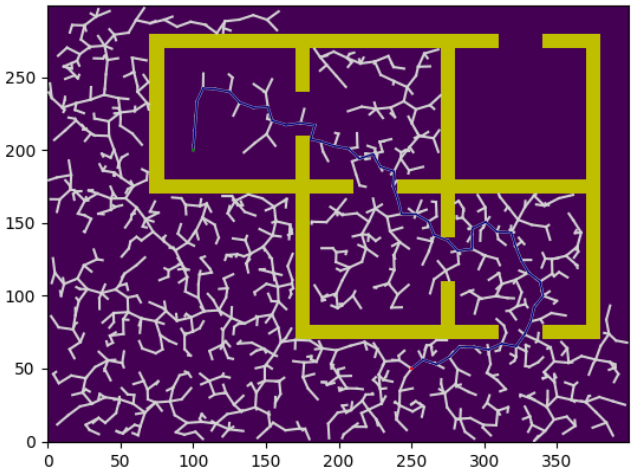
1. We will implement RRT algorithm, bias the sampling to pick the goal of 5% and 40%.The images can be seen in Section 2, where we implemented each Extend mode according to the requirements.
2. We completed the RRT algorithm, and in the algorithm, it can be seen that we run the code 10 times to obtain the average results, as the model is stochastic. According to the instructions, we will present the results by running the algorithm on Map 2.We will present the results of the RRT algorithm for the 3 different η values (where the runtime and path length data are the averages of 10 runs, and we perform Extend according to E_2).



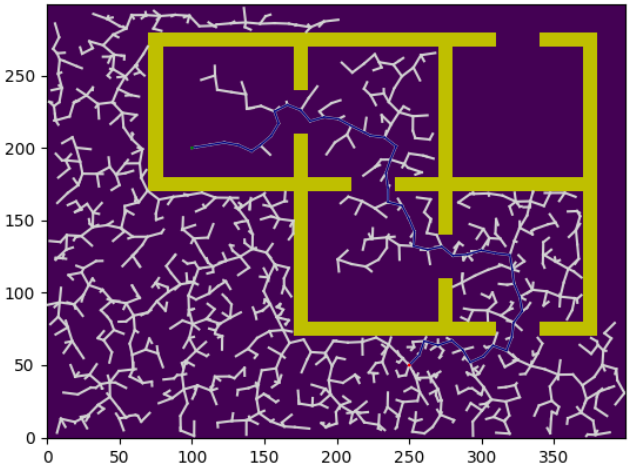
Picture number (2) : Running RRT with Extend by E_2 on map number 2, with $\eta = 5$ and probability 5% .
***Average cost of path*: 527.4607, *Average time*: 37.6288**



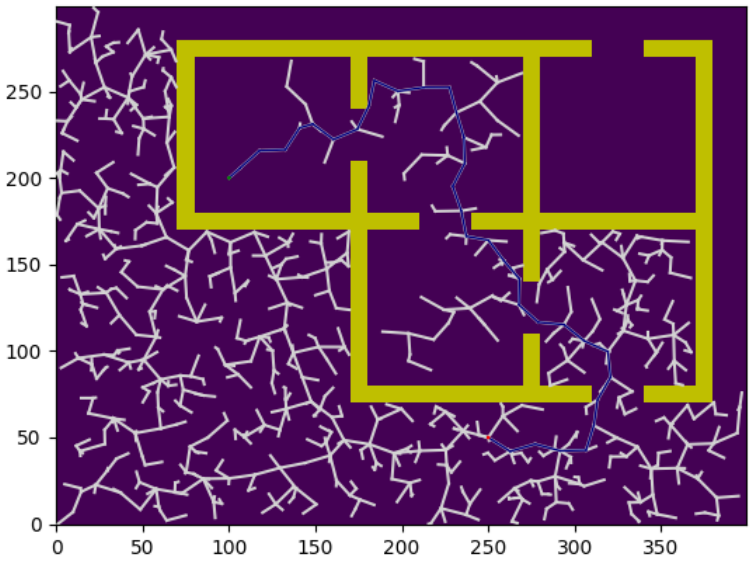
Picture number (1): Running RRT with Extend by E_2 on map number 2, with $\eta = 5$ and probability 5%.
***Average cost of path*: 508.6879 , *Average time*: 21.3719**



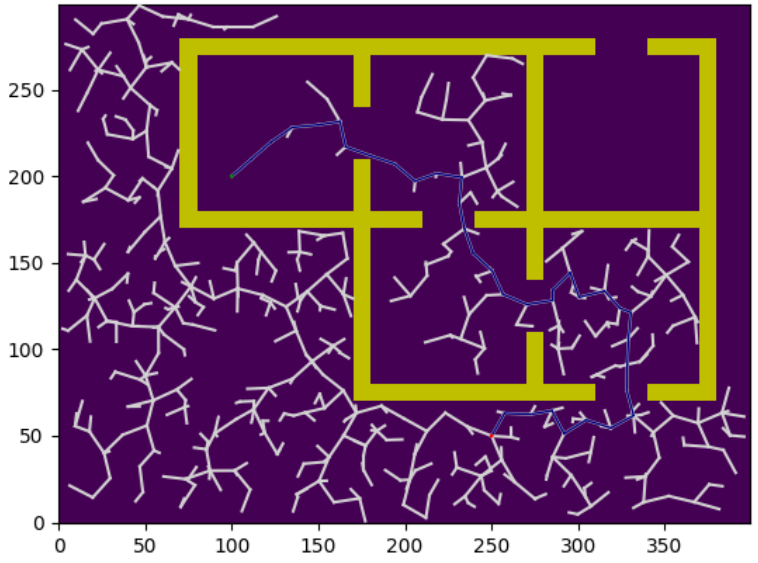
Picture number (3): Running RRT with Extend by E_2 on map number 2, with $\eta = 10$ and probability 5%
***Average cost of path*: 512.49209, *Average time*: 8.1595**



Picture number (4): Running RRT with Extend by E_2 on map number 2, with $\eta = 10$ and probability 40%
***Average cost of path*: 505.1577, *Average time*: 9.6544**



Picture number (5): Running RRT with Extend by E_2 on map number 2, with $\eta = 15$ and probability 40%
***Average cost of path*: 511.8251, *Average time*: 4.5443**



Picture number (6): Running RRT with Extend by E_2 on map number 2, with $\eta = 15$ and probability 5%
***Average cost of path*: 522.6156, *Average time*: 2.3280**

We will summarize the results in a table for E_2:

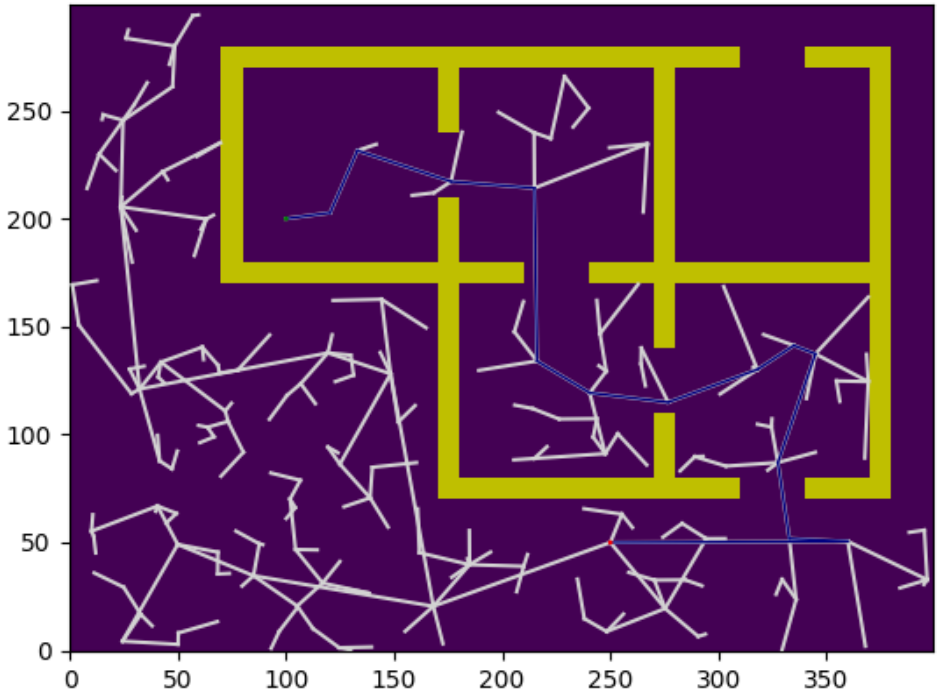
η\Probability	Pr = 5%		Pr = 40%	
	Average Cost	Average Time	Average Cost	Average Time
$\eta = 5$	508.6879	21.3719	527.4607	**37.6288**
$\eta = 10$	512.49209	8.1595	505.1577	**9.6544**
$\eta = 15$	522.6156	2.3280	511.8251	**4.5443**

From the table above, it can be observed that as we increase η, the runtime decreases because the step size and convergence rate increase. However, this also results in missing more optimal paths (with lower cost).
For η=5, we obtained a significantly higher average runtime compared to the other η values. Therefore, we will not choose this η, as it leads to slower solutions.
Since the model is stochastic, 10 runs are not truly sufficient to determine the best η in terms of cost, even though we achieved the lowest average cost with η=10 .
What was consistent, however, was the convergence time, where $t_{\eta=5} > t_{\eta=10} > t_{\eta=15}$ across all runs.

We aim to balance the tradeoff between reasonable runtime and low cost.
Therefore, we will choose $\eta = 10$.

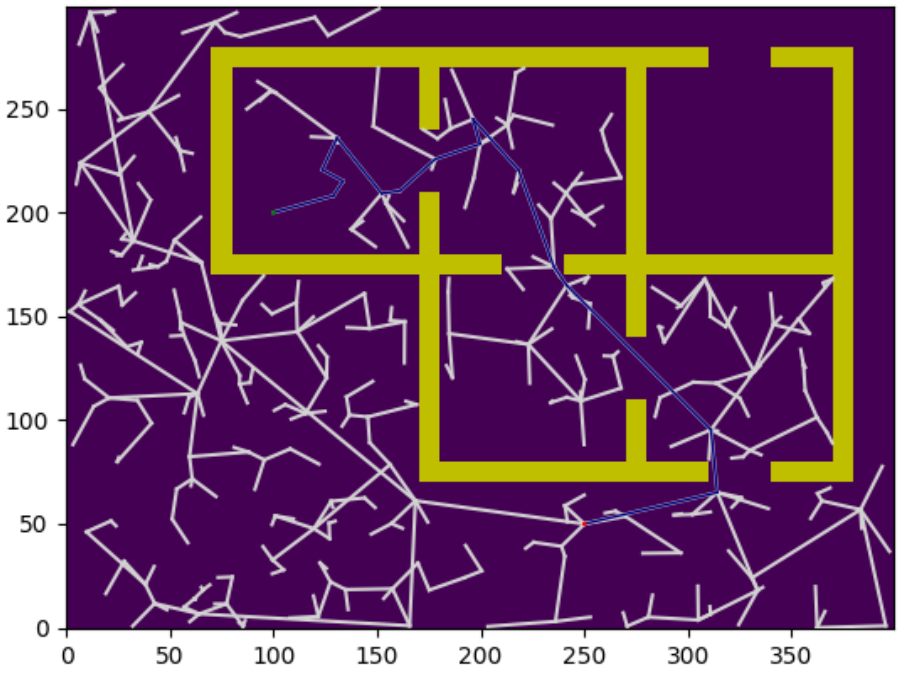
Now, we will run RRT algorithm for E_1:

The results obtained are $\text{Cost} = 582.13, \text{time} = 0.30$ when the probability is 5%.



Picture number (7): Running RRT with Extend by E_1 on map number2 , with probability 5%
***Average cost of path*: 598.3084, *Average time* 1.0411**

Another run was obtained with $Cost = 456.02$, $time = 1.16$ when the probability is 40%.



Picture number (8): Running RRT with Extend by E_1 on map number 2, with probability 40%.
***Average cost of path*: 604.3049, *Average time* 2.1332**

We will summarize the results in a table for E_1:

$\mathbf{Pr = 5\%}$		$\mathbf{Pr = 40\%}$	
Average Cost	Average Time	Average Cost	Average Time
598.3084	1.0411	604.3049	$\mathbf{2.1332}$

3. In the previous sections, we performed 10 iterations for each probability we were asked to prioritize for **extend** using E_1, and 10 iterations for each probability and η value provided.

For E_1, where the nearest neighbor tries to extend all the way to the sampled point, the lowest average runtime was received.

For E_2, where the nearest neighbor tries to extend to the sampled point only by a step size η, the paths obtained were cheaper on average compared to E_1, but at the cost of increased runtime. As explained in the previous section, we chose $\eta = 10$ for this case.

When selecting an **extend strategy** or step size, the choice depends on what is more important: path computation time or the lowest path cost. We selected **extend** according to E_2 because there was a significant difference in the average path cost. We chose $\eta = 10$ because it offers a balance between a relatively low-cost path and reasonable runtime (a tolerable trade-off).

Regarding Probabilities:

1. For E_1, a significantly shorter runtime was achieved with a 40% probability, but the difference in path cost was not substantial. It's possible that with a much larger number of runs, the results might vary slightly.
2. For E_2, the runtime increased as the probability rose from 5% to 40%. However, the paths were shorter, except for η=5. Again, due to the stochastic nature of the model and the averaging over only 10 runs, it is possible that a larger number of runs could yield different results.

Conclusion:

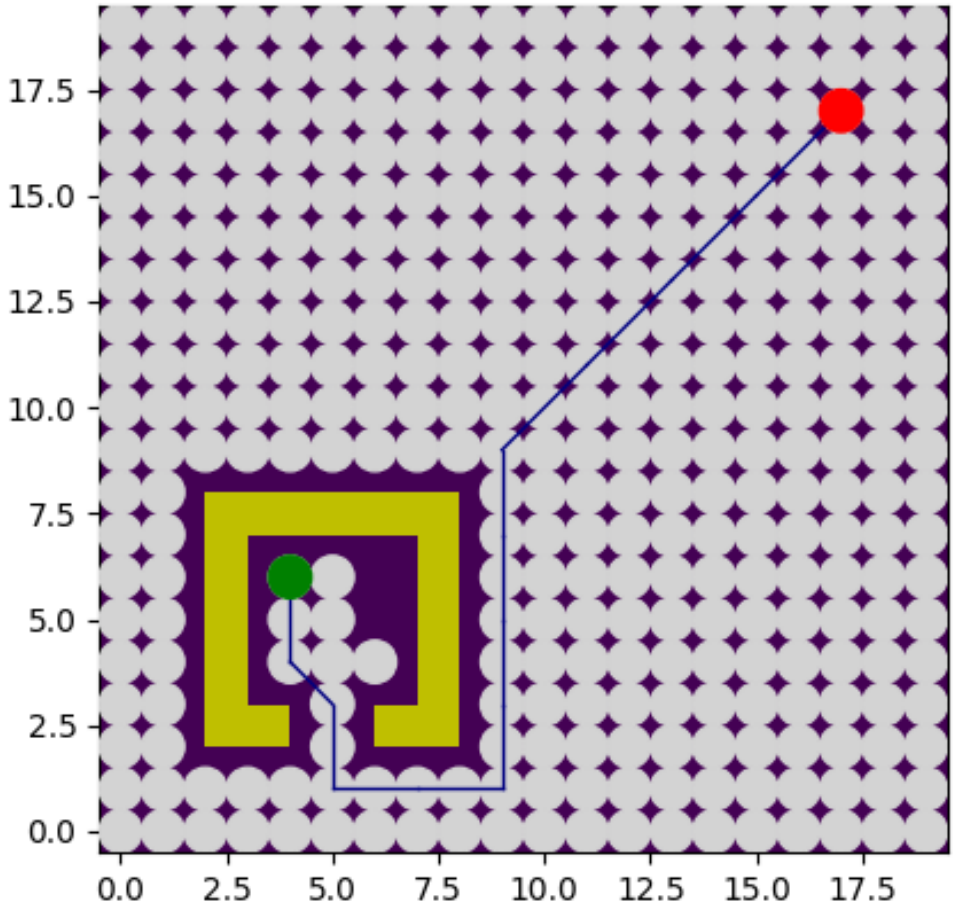
The trade-off between runtime and path cost heavily influences the choice of η and the **extend strategy**. For this task, we opted for E_2 with $\eta = 10$, as it provided a reasonable compromise between runtime and path cost.

Part 4: RCS Algorithm

1. We implemented the RCS algorithm using a min-heap. The primary sorting criterion was the rank, and the secondary criterion used in cases of identical ranks was the state. This was done to select the step that would lead to a solution with minimal cost. The rank of the root was initialized to 0, and the rank of each node increased by 1 compared to its parent node.
   In addition to movements up, down, left, and right, we also allowed diagonal movements.

2. We obtained a path with a length of 28.7279 consisting of 14 steps, counting the initial state as a step with COARSE resolution. Out of all the steps, only one step was not COARSE.
   When including the initial state as COARSE, we get:

$$\%fine_{steps} = \frac{1}{14} \cdot 100 \approx 7.1428\%, \qquad \%coarse_{steps} = \frac{13}{14} \cdot 100 \approx 92.85$$



Picture number (9): Running RCS on map number 1.

Part 5: Bonus

1. The input of the robot according to the paper is: how much of the needle to insert into the body – l, and at what angle – θ.

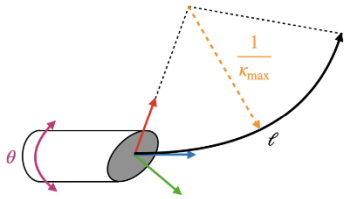


Fig. 2. The kinematics of a bevel-tip steerable needle. The needle can be inserted (characterized by ℓ) and axially rotated at its base (characterized by θ).

2. Information Stored in Each Node:

1. **Rank**:
    - The **sorting criterion** used to prioritize nodes. Nodes with a **lower rank** have higher priority and are added to the OPEN list earlier.
    - The rank of a node is calculated as:

$$Rank(v.parent) + a(l_l(M_v) + l_\theta(M_v) + b$$

        - a,b: Constants for weighting the contributions of the resolution levels.
        - $l_l(M_v)$: Insertion resolution level of the motion primitive.
        - $l_\theta(M_v)$: Rotation resolution level of the motion primitive.
    - Alternatively, recursively:

$$Rank(v) = Rank(v.parent) + \left(l_l(M_v) + l_\theta(M_v)\right) + 1$$

    - The **root node's rank** is initialized to 0.
2. **State**:
    - Represents the **configuration of the needle**:

$$Configuration = (P, \theta)$$

        - $P \in R^3$: The position of the needle in Cartesian coordinates (x, y, z).

        - $\theta \epsilon SO(3)$: The orientation of the needle in 3D space, which can be represented using:
            - Rotation matrices.
            - Quaternions.
            - Euler angles.
    - Together, P and θ provide the complete configuration of the needle, enabling an accurate model in the robot's configuration space.
3. **Resolution**:
    - Indicates the **resolution level** used to reach the current node.
    - Tracks the refinement applied in terms of insertion length (l_l) and rotation angle (l_θ).
4. **Cost**:
    - Represents the **cost** from the initial node to the current node.
    - Cost may consider:
        - The number of actions performed by the robot.
        - The distance traveled.
        - The energy expended.
    - In the RCS implementation, the **rank** of the node was used as its cost.
5. **Parent**:
    - Points to the **parent node** that performed the **Expand** action to generate the current node.
    - Represents the segment of the path connecting the parent node to the current node.
    - Critical for reconstructing the path after reaching the goal state.

6. **Motion Primitive**:
    - Describes the **action** or **motion** applied to the parent node to create the current node.
    - Examples include:
        - Small rotational changes.
        - Translations.
        - Complex motion patterns designed for needle guidance.

This structure ensures that each node contains all the necessary information for ranking, path reconstruction, cost computation, and incremental refinement during the RCS algorithm.

3. If I were to implement the node structure for the steerable needle motion planning, it could look like:

```
class Node:
    def __init__(self, state, resolution, rank, parent=None, motion_primitive=None, cost=0.0):
        """
        Represents a node in the search space.
        Args:
            state (dict): The configuration of the needle as a dictionary:
                          {'p': np.array([x, y, z]),                      # Position
                           'theta': np.array([thetax, thetay, thetaz, thetaw])}  # Orientation
(thetauaternion)
            resolution (str): The resolution type ('coarse' or 'fine').
            rank (float): Priority for expansion (based on cost, resolution, or heuristic).
            parent (Node, optional): Reference to the parent node. Defaults to None.
            motion_primitive (tuple, optional): Action/move applied to generate this node. Defaults to None.
            cost (float, optional): Cumulative cost from the start node. Defaults to 0.0.
        """
        # State of the node: position (p) and orientation (theta)
        self.state = state

        # Resolution type ('coarse' or 'fine')
        self.resolution = resolution

        # Priority for expansion (cost, heuristic, etc.)
        self.rank = rank

        # Reference to the parent node for path reconstruction
        self.parent = parent

        # Motion primitive that generated this node
        self.motion_primitive = motion_primitive

        # Cumulative cost from the start node to this node
        self.cost = cost

    def __lt__(self, other):
        """
        Less-than operator for comparing nodes in priority thetaueues.
        Nodes with lower rank are given higher priority.
        """
        return self.rank =< other.rank
```

4. The **OPEN list** in the steerable needle motion planning algorithm is ordered by the following criteria:
   1) Nodes in the OPEN list are primarily ordered by their rank, which is calculated as:

$$Rank(v) = Rank(v.parent) + l_\ell(M_v) + l_\theta(M_v) + 1$$

   2) Secondary Metric - Cost-Plus-Heuristic ($f(v)$):

   For resolution-optimal planning (RCS*), nodes with the same rank are further ordered by the f-value:

$$f(v) = C(v) + h(v)$$

- $C(v)$: The cumulative cost from the root to the node v, computed using the cost function.
- $h(v)$: A heuristic estimate of the cost from the node v to the goal.

5. **extracting from the OPEN list** involves more than simply popping the lowest-ranked node (as would be done in a typical heap). The process integrates the **"n" look-ahead parameter** to balance exploration and optimization.

   Extraction Method:

   1. Prioritization by Rank and Look-Ahead:
      - The nodes in the OPEN list are primarily ordered by their **rank**
      - A look-ahead parameter n_{la} allows the algorithm to consider nodes with slightly higher rank for extraction. Specifically:

        Nodes with ranks up to $Rank_{mln} + n_{la}$ where $Rank_{mln}$ is the minimum rank in the OPEN list, are eligible for extraction.
   2. Secondary Sorting Within the Look-Ahead Range:
      - Among the nodes with rank $Rank(v) \leq Rank_{mln} + n_{la}$,the algorithm prioritizes nodes based on a secondary cost metric:

        $$f(v) = C(v) + h(v)$$
      - This sorting ensures that within the look-ahead range, nodes closer to the goal and with lower cost are expanded earlier.

6. The second metric used during the extraction process in the algorithm is the **cost-plus-heuristic metric**:

   $$f(v) = C(v) + h(v)$$

   Cost-Like Metric $C(v)$:

- This represents the **cumulative cost** of the path from the root (start configuration) to the current node v.
- It is computed based on a given cost function, such as:
  - **Trajectory length**: The sum of distances along the path.
  - **Obstacle clearance**: A penalty for proximity to obstacles.
  - **Cost map values**: Derived from medical images or other domain-specific data.
- $C(v)$ quantifies the cost already incurred in reaching node v.

Heuristic-Like Metric $h(v)$:

- This is an **estimate of the cost** required to move from the current node v to the goal.
- Commonly used heuristics include:
    - The **Euclidean distance** from v to the goal.
    - **Dubins curve length** (for curvature-constrained motion planning).
- $h(v)$ predicts future cost based on domain knowledge or geometric properties, guiding the search toward the goal.

7. The heuristic h(v) used in the paper is **admissible** because it is based on geometric metrics (Euclidean distance or Dubins curves), which are guaranteed not to overestimate the actual cost to the goal. This property allows the algorithm to explore efficiently while maintaining the guarantees of optimality provided by the RCS* planner.
8. **1. Data Structure Choice for the CLOSED List**

- Use a **hash table (dictionary)** or a similar structure for the CLOSED list:
    - **Key**: A unique identifier for the node's state.
    - **Value**: The associated node data.
- Hash tables provide **O(1)** average-time complexity for insertions and lookups, making them highly efficient for detecting duplicates.

2. Node Values to Compare
Instead of comparing all node data, only a subset of values that uniquely identify the state should be used. These include:
a. **Configuration (Position and Orientation)**:

- Use the position (p) and orientation (q) of the node as a unique identifier.
- For example, in 2D or 3D motion planning:

$$key = (round(p_x, k), round(p_y, k), round(p_z, k)q)$$

b. **Resolution Levels**:

- Include insertion and rotation resolution levels $l_l, l_\theta)$ in the key. These are critical for distinguishing between coarse and refined states.

3. Avoid Comparing All Node Data

- Comparing full node data (e.g., costs, parent references, motion primitives) is unnecessary and computationally expensive.
- Instead:
  - Use only the **state variables** (position, orientation, resolution levels).
  - Store additional node data (e.g., cost) as part of the hash table value for efficient updates when needed.

9. In the paper, the **refined expansion approach** is applied incrementally instead of simultaneously applying all "fine set" actions. This means that when a node v is expanded, its parent v.parent is revisited and refined with additional motion primitives of higher resolution. This strategy balances efficiency and precision by progressively increasing the resolution.

   Initial Coarse Expansion:

- Suppose the parent node v.parent was expanded with a **coarse motion primitive**:

  $$Mcoarse = (\kappa, \delta\ell coarse, \delta\theta coarse)$$

  - κ: Curvature.
  - $\delta\ell coarse$: Large insertion length (e.g., $\ell coarse = 10$).
  - $\delta\theta coarse$: Coarse rotation angle (e.g., $\delta\theta coarse = \frac{\pi}{2}$).

Refinement: Higher Resolution:

- During refinement, **finer motion primitives** are generated by halving the resolution levels for both insertion and rotation.

The algorithm selectively applies these finer primitives to v.paren:

- Adds nodes for motions with $\delta l_{refined}$ and $\delta\theta_{refined}$.
- Prioritizes these nodes based on the updated rank:

$$Rank(v) = rank(v.parent) + l_l(M_v) + l_\theta(M_v) + 1$$

For example:

- A new child node v′ is created using:

  $M' = (\kappa, 5, \frac{\pi}{4})$

- Another child v′′ might be created using:

  $M'' = (\kappa, 5, 0)$

10. In the refined primitive motion context described in the paper, the number of controls applied to a node grows dynamically as the insertion level l_l and angle level l_θ are refined step-by-step. For instance, starting with a coarse motion primitive where $\delta l = \delta l_{mAx}$and angle $\delta\theta = \frac{\pi}{2}$, the refinement process generates finer primitives such as $\delta l = \frac{\delta l_{mAx}}{2}$and angle$\delta\theta = \frac{\pi}{4}$. This process continues until the minimum resolution Rmin is reached. In contrast, in vanilla RCS, the number of controls remains constant as only the coarsest motion primitives are applied, such as $\delta\theta\epsilon\left\{\frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi\right\}$ and $\delta l = \delta l_{mAx}$ with no further refinement. Thus, the refined approach involves a growing number of controls, while the vanilla RCS applies a fixed set of controls throughout.