Using Machine Learning to Improve Sampling in SBMPs

236901 - Lab Final Project

Tamir Offen
Student ID: 211621479
tamiroffen@campus.technion.ac.il

Hazem Irshed
Student ID: 211928569
ir@campus.technion.ac.il

Spring 2024

1 Introduction

In class we learned about exact methods to solve motion planning problems, that relied on an explicit representation of the free space. Unfortunately, explicitly representing the free space is not feasible in many real world problems. Instead, we learned that we can approximate the free space using a roadmap graph that is constructed by sampling configurations in the configuration space. An important part of this new algorithm is how to sample the configuration space. The easiest and most common sampler is uniform random rejection sampling, which samples a random point in the configuration space and checks if it is collision-free. While this sampler is fast and simple to implement, we thought it is a waste that it is not learning from the dataset of samples that it is generating.

This project idea is based off of "Machine Learning Guided Exploration for Sampling-based Motion Planning Algorithms". Code was written by Tamir

and Hazem, and environment was adapted from homework 3's 4-link robot arm environment. Code can be found here.

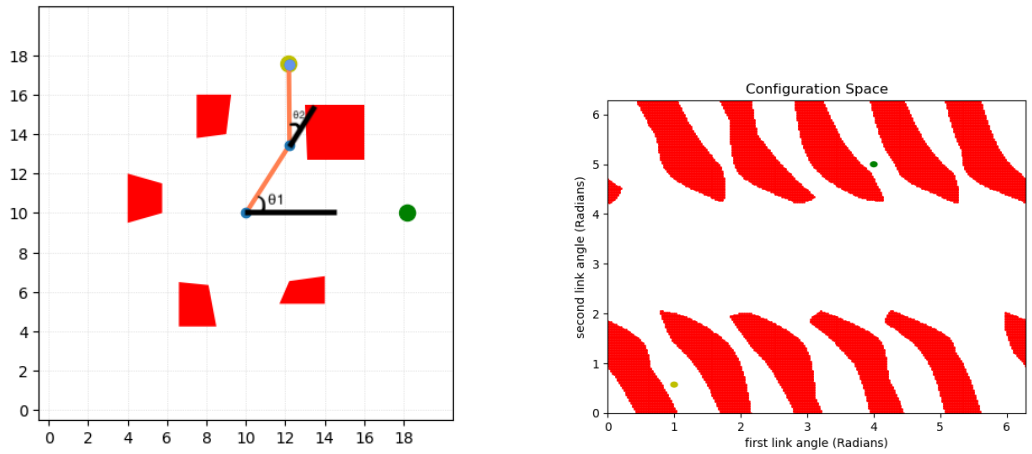
2 Problem and Environment Formulation

In this lab, we decided to work with a two-link robot. We represent a configuration of the robot as two values $(\theta_1, \theta_2 : \theta_1, \theta_2 \in [0, 2\pi))$, such that each value corresponds to the angle of a link. This makes our configuration space 2D. A 2 dimensional configuration space makes visualization easier by graphing it along with the sampled points onto a 2D plane.

The specific environment that we chose to use is shown in figure 1 along with its config space. It is similar to the environment that was used here. The algorithms in this report are not dependent on a specific setup, and could work with any setup as long as a `random sampler` and a `legal configuration checker` functions are provided. We chose to use this workspace because it has tight corridors, making it interesting to see if the online adaptive sampler can reach them, while avoiding sampling the obstacles.

Let $\mathcal{X} \subset \mathbf{SO(2)}$, where $\mathcal{X}$ is our configuration space and $\mathbf{SO(2)}$ is the 2D rotation group. Let $\mathcal{X}_{obs}$ be our obstacle space and $\mathcal{X}_{free} = \mathcal{X} \setminus \mathcal{X}_{obs}$ be our free space. Let $x_{init}, x_{goal} \in \mathcal{X}_{free}$ be our initial and goal configurations respectively. Let $\mathcal{G}(V, E)$ be our graph, where V represents sampled configurations and E represents edges between them (using the local planner).
Let $g : V \to \mathbb{R}_{>=0}$ be the cost to come function. $g(v) =$ path cost from x_{init} to v. $g^*(v)$ is the optimal cost to come function.
Let $h : V \to \mathbb{R}_{>=0}$ be a heuristic function. $h(v) =$ estimate of optimal cost between v and x_{goal}. We assume that h is admissible. It is well known that inadmissible heuristics can be used to speed-up the search, but they may lead to sub-optimal paths.
Our goal is to find a path in $\mathcal{G}$ that starts at x_{init} and finishes at x_{goal} with the lowest path cost. Note that this is a path in the configuration space.

Like what was mentioned in the introduction, the performance of SBMP relies heavily on the sampler. In order to better sample points from the configuration space, we want to bias the sampler to choose points that are likely to be a part of an optimal path. The *relevant* region of $\mathcal{X}_{free}$ is the set of points x for which the optimal cost-to-come value of x plus the optimal



(a) Workspace $\mathcal{W}$ (b) Configuration Space $\mathcal{X}$

Figure 1: Robot Environment . (b) shows $\mathcal{X}_{obs}$ in red.

cost estimate from x to x_{goal} is less than the optimal cost-to-come value of x_{goal}. From claim 1, we can see that sampling points from $\mathcal{X} \setminus \mathcal{X}_{rel}$ will lead to solutions that are not optimal in their length.

$$\mathcal{X}_{rel} = \{\underbrace{x \in \mathcal{X}_{\text{free}}}_{(1)} : \underbrace{g^*(x) + h(x) \leq g^*(x_{\text{goal}})}_{(2)}\}$$

Claim 1: Let $x \in \mathcal{X}_{free}$. If $g^*(x) + h(x) > g^*(x_{goal})$, then x cannot be on an optimal path from x_{init} to x_{goal} in $\mathcal{X}_{free}$.

> Proof: Suppose by contradiction that there exists $x_0 \in \mathcal{X}_{free}$ that is on an optimal path from x_{init} to x_{goal} such that $g^*(x_0)+h(x_0) > g^*(x_{goal})$. Because h is admissible, $h^*(x_0) \geq h(x_0)$. Because x_0 is on an optimal path, $g^*(x_{goal}) = g^*(x_0) + h^*(x_0) \geq g^*(x_0) + h(x_0)$. Therefore, we find that x_0 satisfies $g^*(x_0) + h(x_0) \leq g^*(x_{goal})$, which is a contradiction to our assumption.

We can learn to sample points from $\mathcal{X}_{rel}$ by breaking up sampling into two parts:

1. Classification: $x_{sample} \overset{?}{\in} \mathcal{X}_{free}$, which is condition (1).

2. Regression: predict $g^*(x_{sample})$, which is condition (2).

We can check if x_{sample} is in $\mathcal{X}_{rel}$ by checking both (1) and (2).

3 Learning $\mathcal{X}_{free}$ and $\mathcal{X}_{obs}$

Problem statement: given our training set at iteration m, we want to find a function $\hat{y}_{\text{CS}} : \mathcal{X} \to \{\underbrace{-1}_{\text{col.}}, \underbrace{1}_{\text{col. free}}\}$ where, with high probability, it is correct.

The training set is $\mathcal{D} = \{\underbrace{(x^{(i)}, y^{(i)})}_{\text{config. and label at iter. i}} : i \in [m]\}$. Notice that our dataset increases with every iteration, which will make our model improve with time. Real collision checking is performed only for points that are classified as collision-free. We do this to not have to call the collision checker every time we sample a point, but we also do not want to have a situation where we include a point that is illegal (in collision) in our tree (i.e fale negatives).

There are two approaches to learning $\hat{y}_{\text{CS}}$. One approach is to use a model with a fixed number of parameters. Some examples include linear regression and neural networks. The advantage of these kind of models is that the model complexity is known and is constant throughout training. A disadvantage of using a parametric model for SBMP is that parametric models require assumptions about the form of the underlying data distribution. There is no reason for us to assume that our environment will have some underlying pattern, and for it to not be completely random and irregular. The other approach is a non-parametric model. These models do not have a fixed complexity, but instead grow with the size of the dataset. This is their main downside when it comes to SBMP. Our dataset of sampled points will most likely be very large, making each learning step more and more expensive. Because they do not require an assumption about the underlying distribution of the data, they can make a good fit for $\hat{y}_{\text{CS}}$.

To model $\mathcal{X}_{free}$ and $\mathcal{X}_{obs}$, we used kernel density estimation (KDE) to get class conditional distributions and calculate $\hat{y}_{\text{CS}}$ using a Bayesian decision rule. KDE is a non-parametric method that is used to estimate the probability density function of a random variable. The main idea behind KDE is that

each data point lends some local density to the final density function. For example, if (θ_1, θ_2) is in collision, then probably $(\theta_1 \pm \epsilon, \theta_2 \pm \epsilon)$ is also in collision. The kernel density estimator $\hat{f}_X(x)$ is given by:

$$\hat{f}_X(x) = \frac{1}{m} \sum_{i=1}^{m} K_H \left(x - x^{(i)}\right) \tag{1}$$

where K_H is the kernel function and $H = diag(h, ..., h)$ is the bandwidth matrix. We chose to use the Epanechnikov kernel function, but we didn't find a big difference in performance if we used other kernel functions. On the other hand, we found that the bandwidth matrix is very important when it comes to performance. We set $h = k \cdot \sqrt{\frac{log(|\mathcal{D}|)}{|\mathcal{D}|}}$ such that $|\mathcal{D}|$ is the size of the dataset at the certain iteration. Intuitively, we can think of it as controlling the size of ϵ in the above example. Therefore, k needs to be tuned according to the environment. From figure 2 below, we chose $k = 0.7$ because for $k \geq 0.75$ we get over smoothed decision regions and for $k \leq 0.65$ we get under smoothed decision regions.

The following is the adaptive sampler procedure, which can be found in `AdaptiveSampler2D.py`:

1. Initialize X_{obs} and X_{free} to be empty sets. These two sets will be used to keep track of points that we have sampled, separated into their appropriate category. $X = X_{obs} \cup X_{free}$. Initialize a pdf $\hat{f}_X$ to be uniform over the configuration space $\mathcal{X}$. This is our sampling function, providing us with probabilities over $\mathcal{X}$ for where the model thinks $\mathcal{X}_{free}$ is. In code we made $\mathcal{X}$ discrete using 2D NumPy array, see figure 3 below.

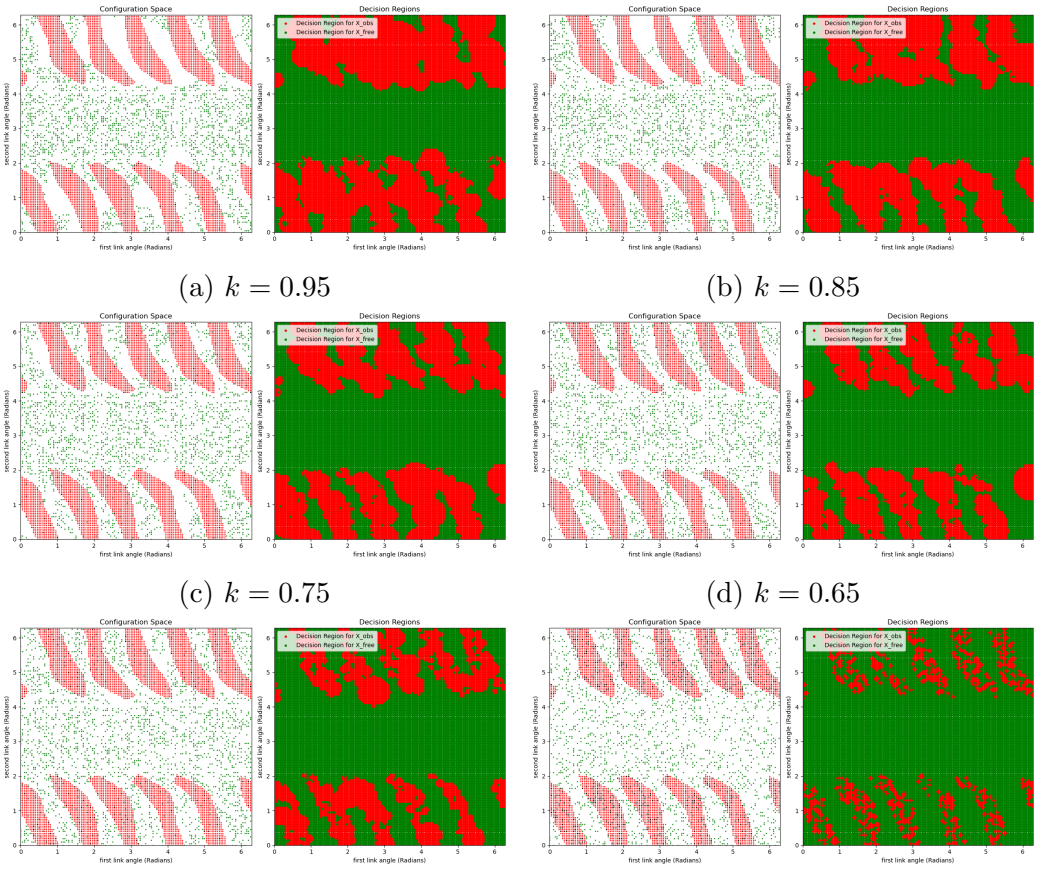
2. For number of samples:

   (a) Sample point x_{sample} using $\hat{f}_X$.

   (b) If x not `inCollision`, add x_{sample} to X_{free} and continue to next sample.

   (c) Else, x is `inCollision`, add x_{sample} to X_{obs} and update $\hat{f}_X$ with the following steps.

   (d) Calculate $\tilde{X}_{free} = \{x \in \mathcal{X} : \hat{y}_{\text{CS}}(x) = 1\}$ and $\tilde{X}_{obs} = \mathcal{X} \setminus \tilde{X}_{free}$



(a) $k = 0.95$ (b) $k = 0.85$

(c) $k = 0.75$ (d) $k = 0.65$

(e) $k = 0.55$ (f) Uniform sampling (constant h)

Figure 2: The effect of k on sampling (2,500 points)

i. Calculate $q_{free}(x) = \frac{1}{\hat{f}_X(x)} \cdot p(x|y=1) \cdot p(y=1)$. $p(x|y=1)$ is calculated using equation 1 with X_{free}. $p(y=1)$ is $\frac{|X_{free}|}{|X|}$.

ii. Calculate $q_{obs}(x)$ similarly to $q_{free}(x)$.

iii. $\hat{y}_{\text{CS}}(x) = 1$ if $q_{free}(x) \geq q_{obs}(x)$, else -1.

(e) Update the pdf:

$$\hat{f}_X(x) = \begin{cases} \frac{1}{|\tilde{X}_{free}|} & \text{if } x \in \tilde{X}_{free} \\ 0 & \text{if } x \in \tilde{X}_{obs} \end{cases}$$

From the sampling procedure we can notice a couple things:

- We are learning the configuration space online, with no preprocessing.
- We do not change the sampling pdf as long as we sample from the free space. This might lead to areas of the configuration space that never get sampled. Figure 2a shows this, where the bandwidth radius is very high, and in some tight passages little to no points get sampled. The bandwidth radius effectively controls the exploration vs. exploitation bias.
- Sampling a point that is in collision and updating the pdf is where the bulk of the runtime takes place. For every point in the configuration space, we need to predict if it is in collision or not using steps d.i to d.iii. The configuration space is continuous, but in code we made it discrete with a given resolution. We update our sampling pdf by making sure we don't sample in the region where we got the in collision sample.

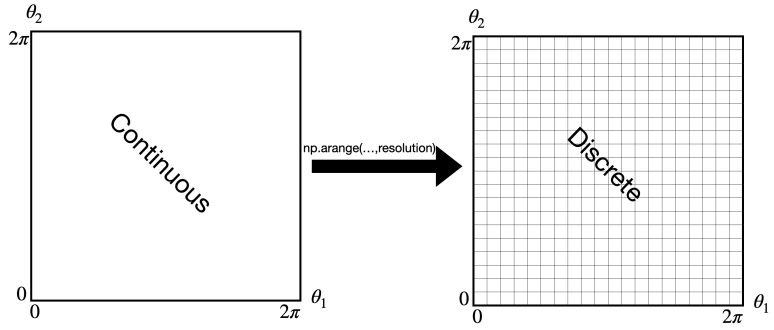


Figure 3: Continuous 2D space to a discrete 2D space.

As we can see from figure 4, there exists areas in the configuration space that the adaptive sampler might never sample from. One reason for this is because of the size of h in the bandwidth matrix. Starting form $|\mathcal{D}| > 2$, h is monotonically decreasing, so the radius of the data point decreases with every iteration. In figure 4, at point $(2.2, 0.3)$ we can see that it was one of the first points to be sampled and was in-collision, and because the h value was high, the adaptive sampler gave this point a large circle. Simply decreasing or increasing the scaling factor k in h will not fix this problem, as can be seen in figure 2. Instead, there needs to be a part in the adaptive sampler

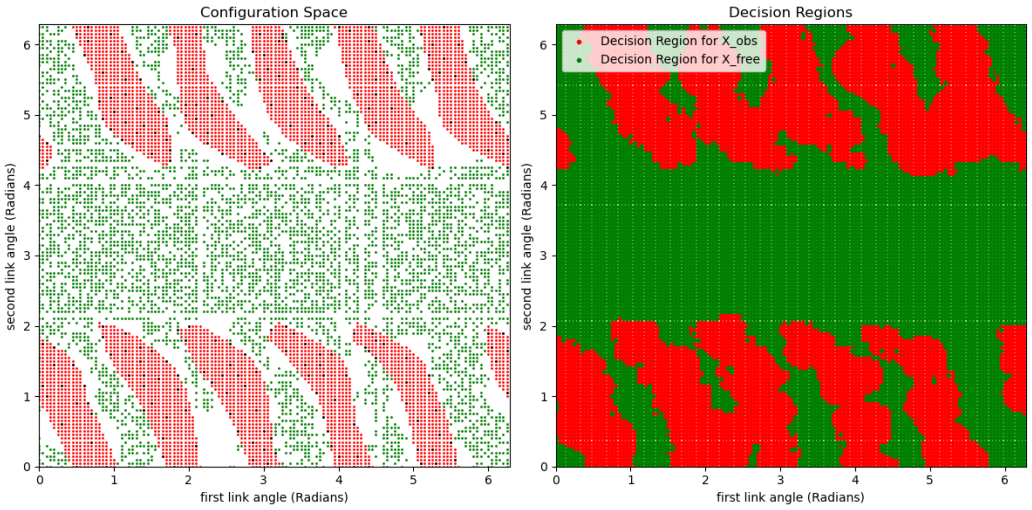


Figure 4: 7500 samples

algorithm that will allow for exploration in in-collision decision regions, and updating the decision region accordingly.

4 Learning the Optimal Cost-to-come Function g^*

In this section, we explain a method for learning the cost-to-come (or cost-to-go) value in the context of sampling-based motion planning algorithms using machine learning algorithm (regression). Given the set of training data $\mathcal{D} = \{(x^{(i)}, y^{(i)}) : i = 1, \ldots, m\}$ where the pair $(x^{(i)}, y^{(i)})$ denotes a randomly drawn point along with its LMC value computed by the replanning procedure at the ith iteration, we aim to find a function $\hat{y}_{cv} : X \to \mathbb{R}$ that estimates the cost-to-come value of a given point x.

Locally Minimal Cost (LMC) Definition

The Locally Minimal Cost (LMC) value is a crucial concept in incremental search algorithms, representing a one-step lookahead value based on g-values, but potentially more informed than the g-values themselves (it estimates the g-value of vertices not included in the tree). The LMC value of a given point

is calculated as the minimum g-value among all its neighboring points plus the cost of moving from the neighbor to the point in question. This means that the LMC value captures the best possible path cost from the start point to the given point through any of its neighbors.
Formally, the LMC value of a point x can be the solution of the following Bellman-type equation:

$$\text{LMC}(x) = \min_{x' \in \text{neighbors}(x)} (g(x') + c(x', x))$$

where $g(x')$ is the g-value of the neighboring point x', and $c(x', x)$ is the cost of moving from x' to x.
The LMC value of the starting point is typically zero, reflecting no movement cost from the start. For any other point, the LMC value provides a heuristic estimate of the minimal cost to reach that point, incorporating the cost information from its immediate surroundings. By using LMC values, the algorithm can make more informed decisions about the optimal path, leading to more efficient pathfinding and replanning.

Locally Weighted Learning-Based Regression

Due to the incremental nature of sampling-based algorithms, we consider locally weighted learning-based methods [**?**], a form of lazy learning, to address the regression problem because of their ease of training. In this method, the training data is stored in memory, and only a small subset is retrieved to answer a specific query. The relevance of the data is measured using a distance function (e.g., nearby points have similar features). For regression problems, the method fits a surface to nearby points using distance-weighted regression as follows:

$$\hat{y}_{cv}(x) = \frac{\sum_i y^{(i)} w(x, x^{(i)})}{\sum_i w(x, x^{(i)})}$$

The weighting function $w(x, x')$ measures the relevance of two points and can be defined using a kernel function, e.g., $w(x, x') = K_{H_v}(x - x')$ where $H_v = \text{diag}(h_v, \ldots, h_v)$ is computed from equation $h = k \cdot \sqrt{\frac{log(|\mathcal{V}|)}{|\mathcal{V}|}}$ according to the size of vertex set V. In the proposed approach, whenever a new sample is examined for inclusion in the graph, its cost-to-come value is estimated using the LMC values of the neighbor vertices according to the above equation.

Then, the new sample is included in the graph if its approximate cost-to-come value satisfies the following inequality:

$$X_{\text{rel}} = \{x \in X_{\text{free}} : \hat{g}(x) + h(x) < \text{lmc}(x^*_{\text{goal}})\}$$

In our approach to estimating the Locally Minimal Cost (LMC) values using regression, obtaining sufficient samples and their corresponding LMC values from previous iterations is essential. Initially, we computed the g-value for the first 10% of sampled points and considered them relevant to accurately estimate subsequent configurations. This subset provided a foundational understanding of traversal costs in the environment.

Algorithm 1 Relevancy Check Procedure

1: **procedure** IsRelevant(G, X, x_{new})
2: $(V, E) \leftarrow G$; $(X_{\text{obs}}, X_{\text{free}}) \leftarrow X$
3: $P_{p,\text{obs}} = |X_{\text{obs}}|/|X|$, $P_{p,\text{free}} = |X_{\text{free}}|/|X|$
4: $S_{\text{obs}} \leftarrow \text{Near}(X_{\text{obs}}, x_{\text{new}}, |X_{\text{obs}}|)$; $P_{c,\text{obs}} = 0$
5: **for** $x' \in S_{\text{obs}}$ **do**
6: $P_{c,\text{obs}} = P_{c,\text{obs}} + K_{H_o}(x_{\text{new}} - x')$
7: **end for**
8: $S_{\text{free}} \leftarrow \text{Near}(X_{\text{free}}, x_{\text{new}}, |X_{\text{free}}|)$; $P_{c,\text{free}} = 0$
9: **for** $x' \in S_{\text{free}}$ **do**
10: $P_{c,\text{free}} = P_{c,\text{free}} + K_{H_f}(x_{\text{new}} - x')$
11: **end for**
12: $P_{c,\text{obs}} = P_{c,\text{obs}}/|S_{\text{obs}}|$, $P_{c,\text{free}} = P_{c,\text{free}}/|S_{\text{free}}|$
13: $q_{\text{obs}}(x_{\text{new}}) = P_{c,\text{obs}} \cdot P_{p,\text{obs}}$
14: $q_{\text{free}}(x_{\text{new}}) = P_{c,\text{free}} \cdot P_{p,\text{free}}$
15: **if** $q_{\text{free}}(x_{\text{new}}) \geq q_{\text{obs}}(x_{\text{new}})$ **then**
16: $\hat{g}(x_{\text{new}}) = 0$; $w_{\text{total}} = 0$
17: $X_{\text{near}} \leftarrow \text{Near}(G, x_{\text{new}}, |V|)$
18: **for** $x' \in X_{\text{near}}$ **do**
19: $w(x_{\text{new}}, x') = K_{H_v}(x_{\text{new}} - x')$
20: $\hat{g}(x_{\text{new}}) = \hat{g}(x_{\text{new}}) + \text{lmc}(x')w(x_{\text{new}}, x')$
21: $w_{\text{total}} = w_{\text{total}} + w(x_{\text{new}}, x')$
22: **end for**
23: $\hat{g}(x_{\text{new}}) = \hat{g}(x_{\text{new}})/w_{\text{total}}$
24: $\text{Key}(x_{\text{new}}) = (\hat{g}(x_{\text{new}}) + h(x), \hat{g}(x_{\text{new}}))$
25: **return** $\text{Key}(x_{\text{new}}) \prec \text{Key}(v^*_{\text{goal}})$
26: **end if**
27: **return** False
28: **end procedure**

Algorithm Explanation: Relevancy Check Procedure

- **Lines 1-2: Initialization**
  - G represents the graph structure containing vertices V and edges E.
  - X is partitioned into X_{obs} (obstacles) and X_{free} (free space).

- **Lines 3-4: Probability Calculation**
  - $P_{p,\text{obs}}$ and $P_{p,\text{free}}$ calculate the probability of sampling from X_{obs} and X_{free} respectively.
- **Lines 5-7: Near Set Calculation (Obstacles)**
  - S_{obs} identifies nearby points in X_{obs} relative to x_{new}.
  - Pc, obs aggregates the kernel function values $K_{H_o}(x_{\text{new}} - x')$ for all $x' \in S_{\text{obs}}$.
- **Lines 8-11: Near Set Calculation (Free Space)**
  - S_{free} identifies nearby points in X_{free} relative to x_{new}.
  - Pc, free aggregates the kernel function values $K_{H_f}(x_{\text{new}} - x')$ for all $x' \in S_{\text{free}}$.
- **Lines 12-14: Probability Calculation (Continued)**
  - Pc, obs and Pc, free are normalized by the size of their respective near sets.
  - $q_{\text{obs}}(x_{\text{new}})$ and $q_{\text{free}}(x_{\text{new}})$ compute the probabilities of x_{new} being from X_{obs} and X_{free} respectively.
- **Lines 15-26: Decision Making and Weighted Cost Calculation**
  - Compares $q_{\text{free}}(x_{\text{new}})$ and $q_{\text{obs}}(x_{\text{new}})$ to determine if x_{new} is more likely to be relevant in X_{free}.
  - If $q_{\text{free}}(x_{\text{new}}) \geq q_{\text{obs}}(x_{\text{new}})$, sets $\hat{g}(x_{\text{new}})$ to zero and initializes w_{total}.
  - Calculates weighted sums $w(x_{\text{new}}, x')$ for all x' in X_{near}.
  - Computes $\hat{g}(x_{\text{new}})$ using the locally minimal cost $\text{lmc}(x')$ and kernel weights $w(x_{\text{new}}, x')$.
  - Normalizes $\hat{g}(x_{\text{new}})$ by w_{total} to obtain $g^*(x_{\text{new}})$.
  - Computes $\text{Key}(x_{\text{new}})$ based on $g^*(x_{\text{new}})$ and a heuristic $h(x)$, and compares it to $\text{Key}(v^*_{\text{goal}})$.
- **Lines 27-28: Final Decision**
  - Returns **True** if x_{new} is relevant (i.e., passes the condition), otherwise returns **False**.

Exploration Enhancement

In addition acknowledging the complexities of exploration in challenging environments, we also included 10% of initially deemed irrelevant configurations in our relevant set. This strategy enhances exploration capabilities, particularly in environments where obstacles create intricate pathfinding challenges. In such scenarios, relying solely on the relevant region may limit our ability to find feasible paths promptly. By considering vertices from the initially deemed irrelevant region, we increase the chances of discovering viable paths, even if they are suboptimal. This adaptive approach balances efficiency with robust exploration, ensuring more reliable pathfinding outcomes.

Algorithm 2 Wrapper for IsRelevant

```
procedure WrapperIsRelevant(G, X, x_new, iter_num)
    result ← IsRelevant(G, X, x_new, iter_num)
    if result = False and random(0, 1) ≤ 0.1 then
        return True
    else
        return result
    end if
end procedure
```

The following figure illustrates the configuration space after 10,000 random iterations. The starting point is at (2, 2) and the goal point is at (5, 5). Red configurations represent regions that are not relevant, while green configurations represent regions that are relevant. Additionally, due to considering 10% of the non-relevant configurations as relevant, we observe green spots in different areas that do not apply the relevant condition strictly.

5 Integration into an RRT-like Algorithm

The following algorithm outlines the integration of adaptive sampling and checks for configurations within the relevant region using the Tree (RRT) algorithm, ensuring efficient motion planning in the configuration space X. Here, the algorithm initializes a tree T rooted at x_{start} and enters a `while` loop until a maximum number of iterations n is reached. Inside the loop, it samples a state x_{rand} using the adaptive sampler we implemented in the first

Algorithm 3 Modified RRT Algorithm with Adaptive Sampling

Require: Configuration space X, start configuration x_{start}, goal region X_{goal}, number of iterations n, steering parameter η

T.init(x_{start})
$i \leftarrow 0$
while $i < n$ **do**
    $x_{\text{rand}} \leftarrow$ adaptive_sampler(X)
    $x_{\text{near}} \leftarrow$ nearest_neighbor(T, x_{rand})
    $x_{\text{new}} \leftarrow$ extend($x_{\text{rand}}, x_{\text{near}}, \eta$)
    if WrapperIsRelevant(x_{new}) **then**
        $i \leftarrow i + 1$
        if collision_free($x_{\text{near}}, x_{\text{new}}$) **then**
            T.add_vertex(x_{new})
            T.add_edge($x_{\text{near}}, x_{\text{new}}$)
            if $x_{\text{new}} == X_{\text{goal}}$ **then**
                return path from x_{start} to x_{new}
            end if
        end if
    end if
end while
return []

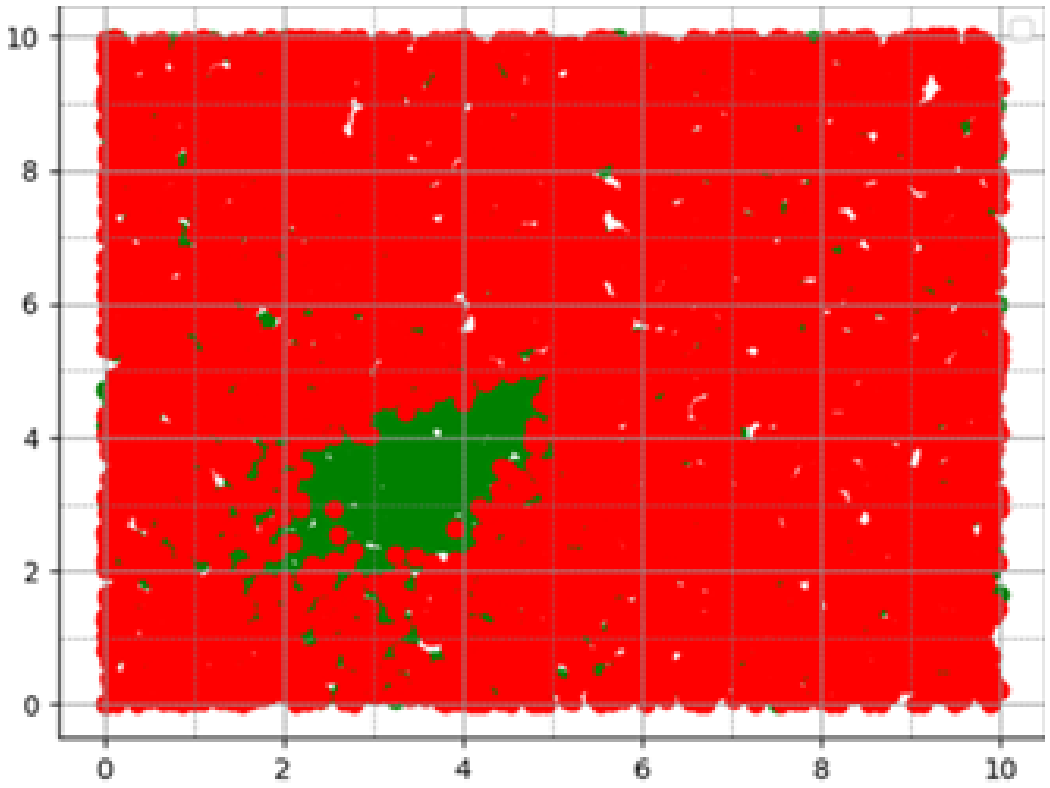


Figure 5: Configuration space after 10,000 random iterations

part of the project, finds the nearest neighbor x_{near}, and extends towards x_{rand} with a steering parameter η.
The `WrapperIsRelevant` function checks the relevance of x_{new}. If it returns true, indicating x_{new} is relevant based on the second part of the project, the algorithm proceeds to check if the path between x_{near} and x_{new} is collision-free using the `collision_free` function. If the path is clear, x_{new} is added as a vertex to the tree T and an edge is added between x_{near} and x_{new}.
If x_{new} equals X_{goal}, the algorithm returns the path from x_{start} to x_{new}. Otherwise, it returns an empty path if the goal is not reached within n iterations.

6 Results

We are sharing some simulation results that demonstrate the performance of our modified algorithm integrated with the RRT algorithm. The results can

be accessed using the following link: here.
In our simulations, our sampler failed to reach the goal point in this case. After exploring the region, it discovered that the best region to sample in was the middle of the space, as it was furthest from obstacles. This behavior indicates that the sampler performed its required job of identifying safer sampling regions, but it led to not finding a path. Figure 6 shows the sampling after 1000 iterations, illustrating this behavior.
The simulations linked above show trajectories from the relevant region, confirming our observations. We ran additional simulations and obtained similar results. However, it is important to note that this approach sometimes fails to find a path because the relevant region algorithm may determine that there are no further enhancing samples that would improve the current sampling.

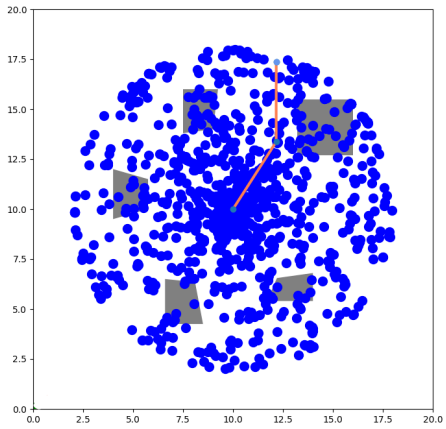


Figure 6: Sampling after 1000 iterations showing the concentration of samples in the middle region.