

# Identifying Limitations in RRT Star

Prithviraj Khelkar  
pkhelkar@bu.edu

Jonas Raedlar  
jraedler@bu.edu

Shiven Sharma  
shivenrk@bu.edu

## Abstract

*Motion planning algorithms play a pivotal role in robotics, with many state-of-the-art algorithms and planners available across multiple platforms and libraries. One of the algorithms is the Rapidly exploring random tree (RRT) and RRT\* algorithms, known for its probabilistic completeness and asymptotic optimality. However, RRT\* itself introduces computational complexities and limitations. This report presents a comprehensive investigation into the constraints of RRT\*. We conduct various experiments on a variety of scenarios in which the algorithm might be used, including 2 and 3-dimensional spaces, as well as stationary and dynamic scenes, using Intel's VTune profiler. Additionally, a state-of-the-art implementation of the Open Motion Planning Library (OMPL) is detailed. The report concludes with insights into the algorithm's performance, laying the foundation for future enhancements and applications.*

## 1. Introduction

The robotics computational pipeline comprises three crucial components: perception, mapping and localization, and motion planning and control. After gaining insight into its current environment (perception) and determining its position relative to this environment and its obstacles (mapping and localization), the robot needs to calculate a collision-free path to reach its goal state (motion planning and control). To efficiently determine such a safe path, numerous algorithms have been proposed, with a significant subset relying on sampling for increased efficiency (Lamiraud and Laumond [6]).

Among the various algorithms designed to address the challenges of motion planning, Rapidly Exploring Random Trees (RRT) have emerged as a powerful and widely used tool in the field of robotics. Developed to efficiently explore complex, high-dimensional configuration spaces, RRT algorithms aim to find feasible paths for robots or autonomous systems. Despite their effectiveness, the original RRT algorithm often provided suboptimal and inefficient paths, particularly in scenarios requiring optimization.

In response to this limitation, the RRT\* (RRT Star) al-

gorithm was introduced by Steven M. LaValle and James J. Kuffner Jr [7]. This algorithm represents a significant advancement, building upon the foundation of RRT by incorporating incremental optimization techniques. Unlike its predecessor, RRT\*, in its various implementations, aims not only to discover feasible paths but also to identify paths that approach optimality based on a specified cost function. This enhancement is particularly valuable in applications where path optimization is crucial, such as in robotics, where minimizing energy consumption, time, or other performance metrics is essential.

Despite the remarkable advancements and successes achieved by the RRT\* algorithm, it is imperative to recognize and understand its limitations. No algorithm is immune to challenges and constraints, and RRT\* is no exception. Understanding these limitations becomes crucial for researchers, engineers, and practitioners seeking to deploy RRT\* in real-world scenarios. Only through a thorough investigation and comprehension of these constraints can informed decisions be made regarding the suitability of RRT\* for specific applications. Additionally, such insights serve as a foundation for researchers committed to refining and enhancing the algorithm or exploring alternative strategies to mitigate its limitations.

In this report, we explore the limitations inherent in the RRT\* algorithm. By closely examining these shortcomings, our objective is to provide a comprehensive understanding of the scenarios and conditions under which RRT\* may encounter challenges or exhibit suboptimal performance.

## 2. Background

Mobile robots have become integral components of various applications, ranging from life services and industrial transportation to agriculture. Their versatility in navigating diverse environments has led to significant research focus on enhancing their capabilities, with path planning standing out as a critical element. Path planning involves determining a collision-free path from an initial state to a target state in the configuration space, often optimizing criteria such as the length of the path.

Numerous path planning algorithms have been developed to address the complexity of this task. Among them,

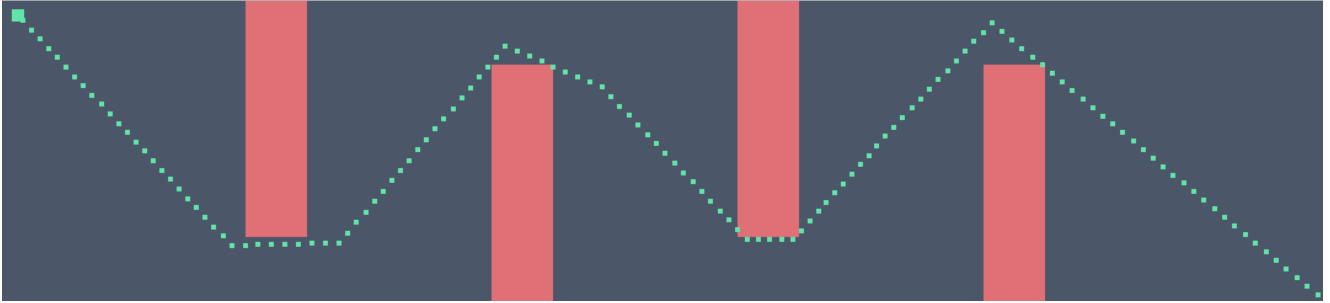


Figure 1. Path with four stationary obstacles

artificial potential field (APF) algorithms use virtual potential fields to guide robot motion, resulting in smooth and reliable paths. However, they are susceptible to local optima. Intelligent bionic algorithms, inspired by natural processes, such as genetic algorithms and ant colony algorithms, offer solutions to complex problems but face limitations in processing speed and local optima. Grid-based searches like A\* meet real-time and optimization requirements but are confined to discrete low-dimensional spaces, facing exponential computational challenges with increasing dimensionality.

Sampling-based algorithms, particularly the Rapidly-exploring Random Tree (RRT) algorithm, have gained prominence for their applicability in high-dimensional configuration spaces. The RRT algorithm ensures probabilistic completeness but suffers from limitations like low convergence rates and fluctuating path quality. To address these issues, researchers have proposed enhancements, leading to the development of the RRT\* algorithm which provides asymptotic optimality.

The key differences that provide asymptotic optimality, while still maintaining the probabilistic completeness property, are 1) the consideration of path costs when a new sample is added to the existing tree and 2) the rewiring step that calculates whether this new sample, as a potential parent node for neighboring nodes, reduces the path cost for these nodes. The original RRT\* paper provides a pseudocode implementation of the algorithm (Ding et al. [4], Qureshi and Ayaz [8], Karaman and Frazzoli [5]).

RRT\* is usually run on a fairly typical system. In our search, we have found that the algorithm is often run on an Intel Core i5 processor with 2 to 3 GHz and 4+ GB of RAM (Karaman and Frazzoli [5]).

Despite its advantages, RRT\* exhibits limitations in terms of searching efficiency. Subsequent efforts, such as the RRT\*-smart algorithm, introduce intelligent sampling

and path optimization techniques for faster convergence. However, challenges persist, notably in the escalating number of nodes, posing a significant memory challenge.

Researchers have further refined the RRT\* algorithm to address specific drawbacks. The RRT\*-FN algorithm limits the maximum number of nodes, optimizing memory usage. The Informed RRT\* algorithm restricts node sampling through a hyper-ellipsoid sampling subset, improving convergence rates but encountering challenges in certain environments. The Quick-RRT\* algorithm modifies the growth strategy of the tree based on the triangular inequality, enhancing convergence rates, though it may face inefficiencies in narrow environments.

### 3. Motivation

The RRT\* algorithm, with its invaluable properties of probabilistic completeness and asymptotic optimality, has become a pivotal tool in the domain of motion planning for robotics. However, the integration of these advanced features comes at a discernible cost, sparking the need for a closer examination of its limitations and the motivation to address them.

Foremost among these limitations is the significantly lower convergence rate of RRT\* compared to its predecessor, the original RRT algorithm. The introduction of incremental optimization techniques, while enhancing optimality, impose substantial computational complexity. Specifically, RRT\* necessitates a nearest neighbor search for each newly sampled point to determine the lowest path cost among existing paths. This process contributes to a 49% increase in latency, rendering RRT\* up to 8 times slower than the original RRT algorithm in experiments (Bakhshalipour et al. [2]).

The slow convergence rate is exacerbated in high-dimensional spaces, where RRT\* demands an extensive number of samples for thorough exploration. This require-

ment not only elongates computational time but also places significant strain on memory resources. High memory demands are further compounded by the low node utilization rate, as a considerable portion of sampled nodes minimally contributes to finding an optimal path (Ding et al. [4], Qureshi and Ayaz [8]).

In addition to these challenges, RRT\* encounters significant difficulties in dynamic scenes with moving objects. Its response time can often be not fast enough to determine a collision-free path before an object moves into the said path, presenting a critical limitation for real-world applications where dynamic environments are common.

Furthermore, RRT\* faces challenges when applied to robots with higher degrees of freedom, such as quadrupeds and humanoids. While its performance is satisfactory for robots with fewer degrees of freedom, like drones or cars, the algorithm becomes considerably more time-consuming and less effective when dealing with the intricacies of more complex robotic systems.

Recognizing these limitations prompts exploration into potential benefits that could arise from addressing the challenges faced by the RRT\* algorithm. Tackling the issue of slow convergence holds the promise of expediting robotic operations, minimizing the time spent on calculating collision-free paths. The resultant reduction in latency within the robotic computational pipeline, a critical component of real-time decision-making, brings us closer to the seamless integration of robots into everyday life.

Moreover, an improvement in motion planning efficiency directly impacts power consumption. By reducing computational demands and achieving a higher convergence rate, robots employing RRT\* would consume less energy during path planning. For battery-powered robots, this translates into an extended operational time and reduced recharging frequency, enhancing the feasibility of deploying robots in real-world applications.

In summary, the motivation to address the limitations of the RRT\* algorithm extends beyond academic considerations. It holds the potential to significantly impact the landscape of robotics by greatly enhancing the efficiency and viability of robots for real-world applications. This pursuit not only fuels innovation in the realm of robotic research but also brings us closer to a future where robots can seamlessly and reliably assist in diverse human activities.

## 4. Methodology

### 4.1. System

To conduct our experiments, the RRT\* algorithm will be executed on a computing system with an Intel CPU, as seen in Table 1.

Component	HP ENVY Laptop 13-ba1xxx
Processor	11th Gen Intel(R) Core(TM) i7-1165G7 Base Speed: 2.80 GHz Cores: 4 Logical Processors: 8
Caches	L1 Cache: 320 KB L2 Cache: 5.0 MB L3 Cache: 12.0 MB
Memory	Total Physical Memory: 15.8 GB Memory Speed: 3200 MHz
GPU	Intel(R) Iris(R) Xe Graphics Shared GPU Memory: 7.9 GB

Table 1. Specifications of the Machine Used

### 4.2. Profiler

To gain deeper insights into the performance of the RRT\* algorithm and identify potential bottlenecks, we employed Intel’s VTune Profiler on the Intel-based system. VTune is a powerful performance analysis tool provided by Intel, designed to identify hotspots in code execution, assess memory usage, and pinpoint areas where optimizations can be applied.

### 4.3. Implementation

The Optimal Rapidly-exploring Random Trees (RRT\*) algorithm is an integral part of the Motion class within the Open Motion Planning Library (OMPL). As a geometric planner in OMPL, RRT\* is implemented in C++ and inherits from `ompl::base::Planner`.

The RRT\* algorithm requires a `base::SpaceInformationPtr` for proper initialization. This pointer encapsulates information about the configuration space, allowing RRT\* to adapt to the specific constraints of the robotic system.

The provided code implements the RRT\* algorithm [1] in C++ using the Open Motion Planning Library (OMPL) for a robotic system. Below are the key aspects of the implementation:

#### 1. State Space

- Representation of the space where planning is performed.
- Contains information of the dimensionality of the space. Using `addDimension(double, double)`.
- For example, `RealVectorStateSpace`

#### 2. Validity Checker:

- This function is called on every instance of the State, to validate if it’s satisfactory or not.

- For example, `StateValidityChecker` checks the validity of states with respect to obstacles in the environment.
- It can be used for collision detections.

### 3. Setup

- Create the set of classes typically needed to solve a geometric problem.
- Used to set start and goal states (using `ScopedState`), validity checkers, optimization objectives.
- It's also used to instantiate the planner (RRT\*).
- For example, `SimpleSetup`.
- A setup instance, after the planner has been assigned, is used to solve for the path.

### 4.4. Measurements

In our study of the RRT\* algorithm, we crafted a diverse set of inputs to gauge its performance across a spectrum of scenarios. Initially, we generated 10 distinct inputs featuring varying numbers of stationary obstacles, seeking to encapsulate a broad range of environmental complexities. To delve deeper into the algorithm's behavior, we singled out the four most significant inputs from this set for more detailed analysis.

Beyond static environments, our exploration extended to inputs featuring dynamic elements—moving obstacles and robots. This aims to uncover the algorithm's efficacy when confronted with moving objects, specifically with a focus on the obstacle size and their velocities. By systematically varying these parameters, we aspire to delineate the algorithm's success thresholds under dynamic conditions.

Expanding the scope, we introduced an input with slightly elevated dimensions, introducing an additional layer of complexity to the algorithm's exploration and path-planning capabilities. This augmentation enables us to assess the algorithm's adaptability and efficiency when faced with scenarios of increased spatial intricacy.

Analyzing the state-of-the-art implementation's impact on the computing system involves a meticulous examination of profiler-derived data. We inspected the efficiency of RRT\*'s CPU usage through pipeline slot percentages in different categories through VTune's Microarchitecture Exploration, conducted a profound investigation into specific algorithmic parts consuming significant time through the profiler's Hotspot Analysis, and pinpointed issues like cache misses and latency specifics through its Memory Access functionality.

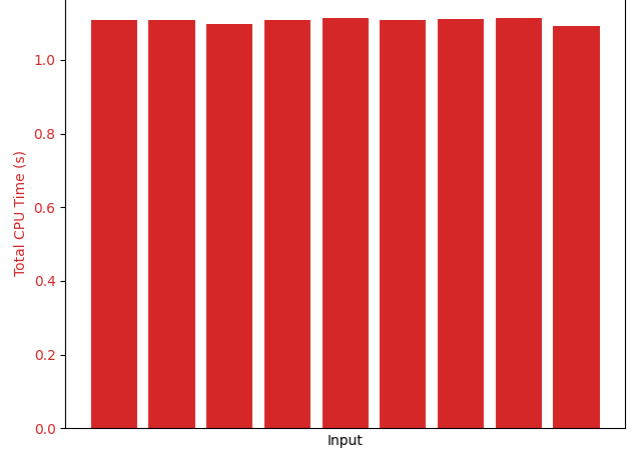


Figure 2. Comparison of CPU time

## 5. Data and Analysis

### 5.1. Stationary scenes

For our first analysis, we created 10 distinct inputs featuring varying numbers of stationary obstacles. From these we selected the four most significant inputs for more detailed analysis, as seen in table [Table 2](#).

We observe that most of these data points are very close to each other across all four inputs, specifically in terms of total time. Since the algorithm runs the `ompl::base::StateValidityChecker.isValid` function in every state, testing if the current state is valid (meaning that the state is within bounds and not intersecting with an obstacle) or not, we can observe a very slight increase in the total time as the number of states increases. As we see that the total time barely changes as the number of obstacles increases from 1 to 10+ (as seen in [Figure 2](#)), we decided to further analyze the performance of the subfunctions, as we believed that one or more functions might be imposing a bottleneck on the algorithm. We'll proceed with this analysis in a later section of this report.

Another noticeable trend is the relationship between the number of iterations and the number of vertices. For input *a*, the vertices are the least, which makes sense, as we have only one obstacle so the tree would be short. Hence it takes the least number of iterations. Input *d*, however, does not have the highest number of iterations and vertices, meaning that the algorithm most likely inversely relates to the size of the obstacles, or simply put, directly relates to the space available. This also explains input *c*'s higher cost, iteration number, and vertex number in contrast to input *d*'s: the space available (total area - area of the obstacles) is significantly less in *c* than in *d*.

Moreover, while RRT\* significantly improves the quality of the paths compared to its predecessor, it still does not




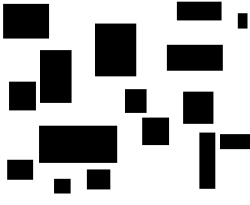
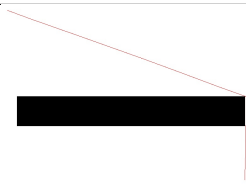
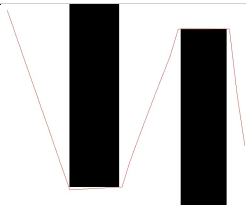
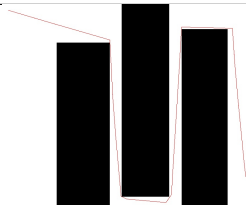

Input a	Input b	Input c	Input d
			
			
Obstacles: 1 Initial Solution cost: 1159.89 Final Solution cost: 1078.36 Iterations: 82 Vertices: 50 New States created: 2001 Rewire options: 464191 Total time: 1.144s	Obstacles: 2 Initial Solution cost: 2013.33 Final Solution cost: 1844.72 Iterations: 249 Vertices: 122 New States created: 1963 Rewire options: 453817 Total time: 1.156s	Obstacles: 3 Initial Solution cost: 2261.12 Final Solution cost: 2187.61 Iterations: 918 Vertices: 303 New States created: 1928 Rewire options: 333292 Total time: 1.204s	Obstacles: 10+ Initial Solution cost: 1138.77 Final Solution cost: 1027.56 Iterations: 336 Vertices: 157 New States created: 1918 Rewire options: 441572 Total time: 1.208s

Table 2. Description of the images.

return the most optimal path available. Table 3 shows the error in each of the four inputs as shown in Table 2.

Input	Actual Path (m)	Ideal Path (m)	Error (percent)
a	31.7	31	2.25
b	54	52.2	3.45
c	65	61.4	5.87
d	31.1	28.7	8.36

Table 3. Cost of Paths per Input

## 5.2. Dynamic Scenes (Obstacles in motion)

Secondly, we note that the algorithm isn't quite able to compute a feasible solution when one or more obstacle starts to move. For the purpose of our simulation, we stated an obstacle would begin with `DELTA_TIME` being equal to 16 (an obstacle moves by one unit every `DELTA_TIME` ms). We can proportionately translate that to the real world relative to the robot's dimensions, and we make the following observations:

- The robot's computation time is about a thousand times bigger than `DELTA_TIME`, where we adjust `DELTA_TIME` to mimic the real world as closely as possible.

- Hence by the time the robot has calculated an efficient path, obstacles have already moved into the said path, often rendering the path invalid.

After further analysis of the algorithm, specifically of the `StateValidityChecker` and the `RealVectorStateSpace`, we noted that the library does not provide an efficient way to predict where an obstacle is going to be depending on its past location. This limits the algorithm's real-world applications for dynamic environments, as a robot using RRT\* would only work if the maximum velocity of all the objects in the scene is carefully clamped.

From Table 2, we see that the solver, on average, takes 1.2 seconds to compute a path (2 DOF). To accommodate this, an obstacle has to move at most 1 meter in 1.2 seconds, which is 0.83 m/s or 83.34 cm/s, which is clearly too slow.

One possible way to tackle this would be to keep track of every obstacle's previous location and to estimate its velocity. Then we can predict where the object is going to be during the next iteration of the solver and use that to calculate our path more efficiently. This solution, although very imperfect, works in certain real-world scenarios. We test this in the following scenario: a robot is mapping its way across a room when a sliding door starts to close. We run both the original RRT\* and a modified version that considers the abovementioned datapoints and obtain the following



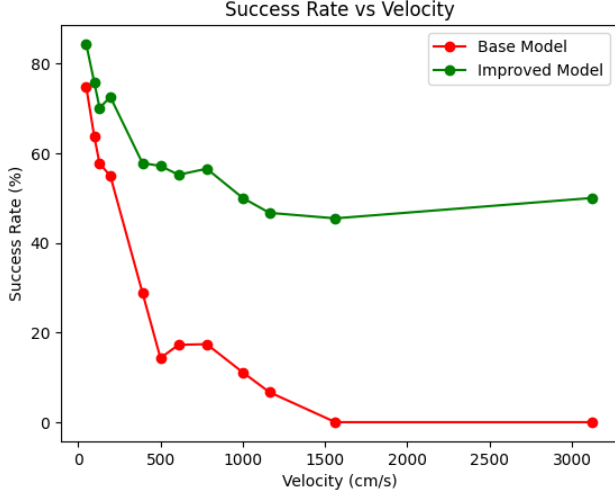


Figure 3. Obstacle velocity vs success rate

data: Figure 3. Here, we calculate the number of times the solver runs and the number of times it fails, enabling us to calculate the success rate.

We can see that our improved algorithm does considerably better at computing the path successfully. For slower velocities, it’s almost at a 90 percent success rate, while the original version has less than 80%. But also for faster velocities, the improved algorithm has a 50 percent success rate while the base algorithm flattens down to zero.

Note that this chart is valid for this particular instance of the scene, with one obstacle moving - the door. It may not be accurate for other scenarios.

### 5.3. Measurement Data for Subfunctions

In order to tackle this limitation at its foundation, however, we need to determine the exact weaknesses of RRT\*. After all, resolving this issue at the base level will lead to more significant improvements across all its use cases. Therefore, we took the four inputs from Table 2 and used VTune to get information about the algorithm’s execution, such as the time during which the CPU is actively executing RRT\*, the number of actually retired instructions, the distribution of pipeline slots (how many are retired, front-end bound, back-end bound, and bad speculation), etc. We obtained this data using VTune’s Performance Snapshot, Hotspot Analysis, and Microarchitecture Exploration. Aside from gathering this data for the entire RRT\* algorithm, we also gathered it for six subfunctions: the three most expensive subfunctions (in terms of CPU usage time) coming from the OMPL library (referenced here as *NearestNeighbor*, *isCostBetterThan*, and *Distance*), one helper-function that does the collision check (referenced here as *isStateValid*), and two subfunctions from the std library: *Pop\_heap\_hole\_by\_index* and the *less* operation. We took

an interest in the *Pop\_heap\_hole\_by\_index* function, as it is part of RRT\*’s rewiring process, an essential element of the algorithm. Although we don’t expect any serious possible improvements for the *less* operation, we decided to look at it anyway as it takes up a significant amount of CPU usage time.

After testing with the aforementioned four different inputs, each characterized by varying degrees of complexity, it became evident that the execution time, as depicted in Figure 4 by the Total CPU-Self time metric, varies exclusively for the *isStateValid* function. Intriguingly, the remaining five subfunctions have almost a consistent execution time across the four inputs.

An important observation is the correlation between the execution time of *isStateValid* and the complexity of the input. Specifically, as the input progresses from the relatively straightforward implementation in input *a* to the more intricate input *d*, the execution time increases from 57.4ms to 102.0ms. This is most likely due to the fact that the subfunction *isStateValid* samples random points and tests if they collide with an obstacle. Thus, it makes sense that more obstacles in the input suggest a longer convergence time, as more samples will be required.

### 5.4. Comparison of Inputs

Upon a detailed comparison of the four inputs, some noteworthy trends and variations emerge in key metrics as seen in Table 4: in terms of the total CPU time, input *d* has the highest duration at 1.208 seconds, suggesting a potential correlation between the complexity of the input and execution time. While the number of instructions retired paints a different picture, with input *b* having the lowest count instead of input *a*, input *d* still requires the highest amount of retired instructions.

Input	Total CPU Time (s)	Instructions Retired	IPC	Stores
<i>a</i>	1.144	7,116,200,000	1.276	339,800,000
<i>b</i>	1.156	3,685,080,000	1.199	97,300,000
<i>c</i>	1.204	6,670,160,000	1.244	290,800,000
<i>d</i>	1.208	8,595,720,000	1.277	367,700,000

Table 4. CPU Time data for different inputs

Further exploration into the cache hierarchy, displayed in Table 5, reveals more intriguing insights: the L1 Bound sees substantial variation, with input *d* reaching the highest percentage at 11.5%, indicating a potentially higher demand for L1 cache. Meanwhile, the L2 Bound experiences the lowest values in Input *d* at 1.6%, suggesting efficient utilization of the L2 cache. But the total memory (combining L1, L2, L3, and DRAM) suggests a familiar pattern with input *d* highest and input *a* lowest.

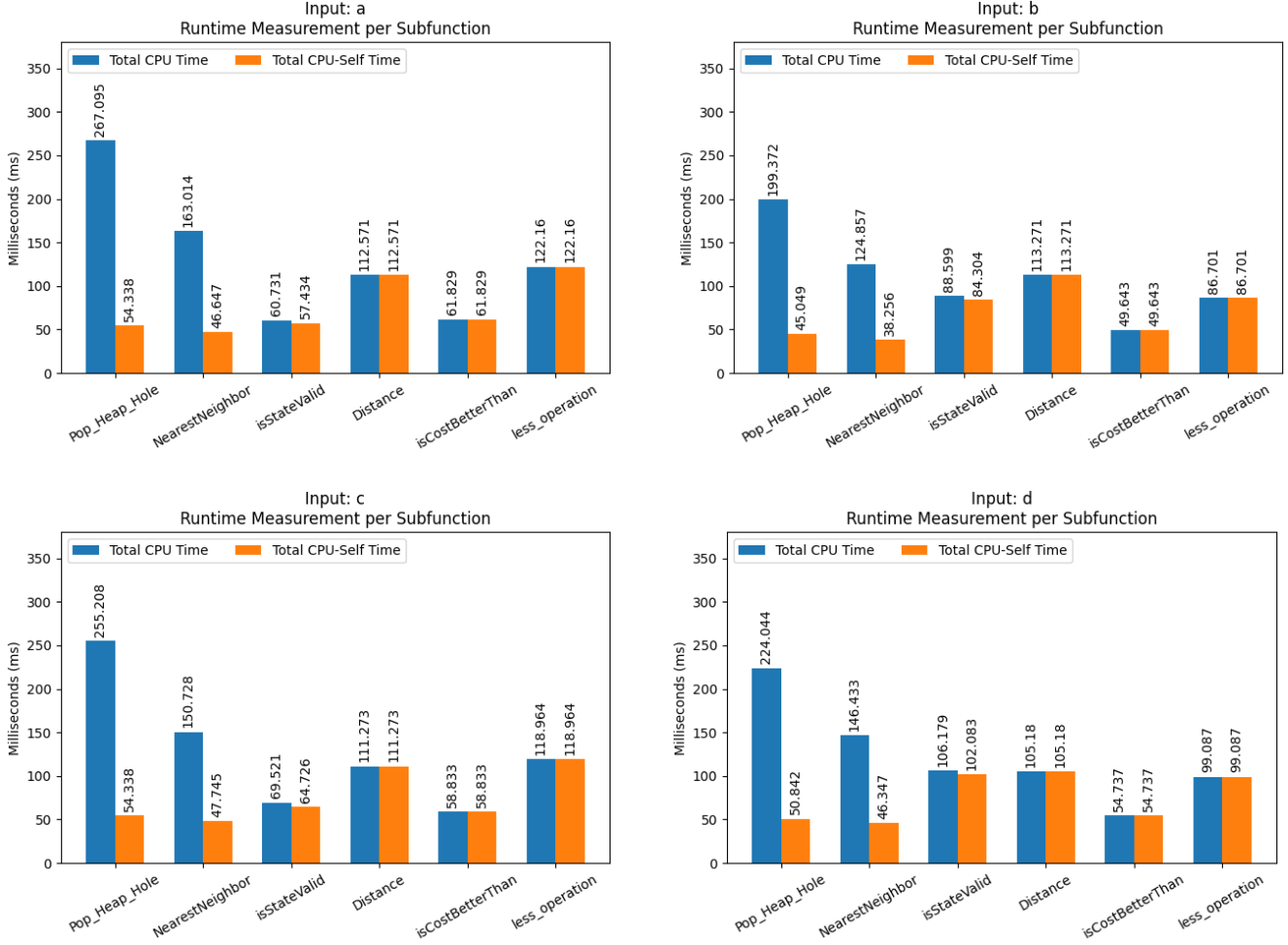


Figure 4. Each plot contains the measurements for Total CPU Time (blue) and Total CPU-Self Time (orange) in ms for the 6 different subfunctions for a different input. Total CPU Time refers to the total amount of time the subfunction is executing on the CPU (including all the subfunctions those subfunctions call), whereas the Total CPU-Self Time only refers to the time without any subfunctions that are called within the subfunction in question.

Input	LLC Miss Count	L1 Bound	L2 Bound	L3 Bound	DRAM Bound
a	490,196	6.5%	1.7%	1.8%	4.3%
b	460,184	5.8%	2.1%	3.5%	4.7%
c	605,242	5.2%	2.1%	3.1%	4.7%
d	515,206	11.5%	1.6%	1.5%	4.2%

Table 5. Memory analytics for different inputs

Based on these findings, and given the fact that input *d* is also the input that represents a real-life environment the closest, we have decided to only focus on *d* for the rest of our analysis. Based on tables 4 and 5, this input seems to present the limitations of the RRT\*-algorithm the strongest and we believe that diving deeper into the performance of

RRT\* only on this particular input is sufficient to gain the necessary insights to identify the algorithm's limitations. The validity of this approach is further supported by the invariance of the distribution of pipeline slots for the different inputs (see Figure 5), which shows that the limitations for the different inputs are essentially the same.

## 5.5. Analysis of Subfunctions

To gain a better insight into RRT\* and to determine its limitations, we performed a more thorough analysis of the six previously mentioned subfunctions on input *d*, determining the distribution of pipeline slots across the five categories Retiring (which is the desired category), Front-End Bound, Bad Speculation, and Back-End Bound. The results are found in Table 6.

Function	Retiring	Front-End Bound		Bad Speculation (Branch Mispredictions)	Back-End Bound	
		Latency	Bandwidth		Memory Bound	Core Bound
<i>Pop_Heap_Hole</i>	32.3%	9.1%	14.7%	36.8%	1.4%	5.7%
<i>NearestNeighbor</i>	20.4%	10.2%	7.9%	47.6%	4.4%	9.5%
<i>isStateValid</i>	8.6%	2.1%	2.4%	35.4%	46.2%	5.3%
<i>Distance</i>	24.5%	4.1%	8.4%	45.0%	7.1%	10.8%
<i>isCostBetterThan</i>	38.5%	5.3%	17.6%	35.7%	0.6%	2.3%
<i>less</i>	25.7%	10.6%	12.6%	41.6%	1.2%	8.4%

Table 6. This table contains the distribution of pipeline slots - in percentages - for the six different subfunctions. A pipeline slot represents hardware resources needed to process one instruction. The pipeline slots are categorized as retired (an algorithm with 100% retired pipeline slots would have no inefficiencies), Front-End Bound, which is further subdivided into Latency and Bandwidth issues, Bad Speculation (mostly Branch Mispredictions), and Back-End Bound, which is further subdivided into Memory Bound and Core Bound.

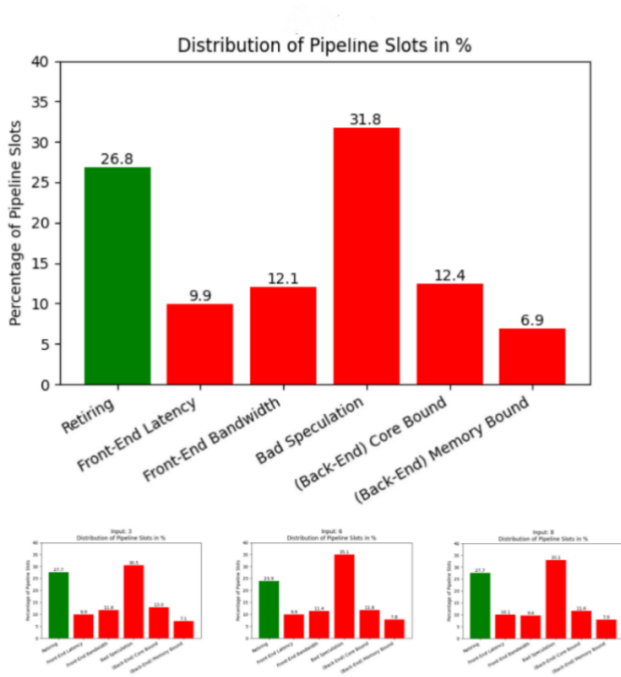


Figure 5. This figure shows four different bar plots, one for each input (Input *d* at the top, input *a*, *b*, *c*, in this order, at the bottom). The green bar on the left represents the percentage of retired instructions (i.e. the ones that successfully completed). The red bars represent inefficiencies in the CPU usage of the RRT\* algorithm. It is clearly visible that the general distribution of pipeline slots does not change for the different inputs.

### 5.5.1 *isStateValid*

The *isStateValid* subfunction very clearly demonstrates a limitation to the RRT\* algorithm, as only 8.6% of its pipeline slots retire, meaning that the other 91.4% are not used efficiently. This can also be seen in Figure 6: the *isStateValid* function has an extremely low Instructions Retired to Number of Cycles ratio. From Table 5, we know that RRT\* has an average IPC of 1.277, but only 0.318 in-

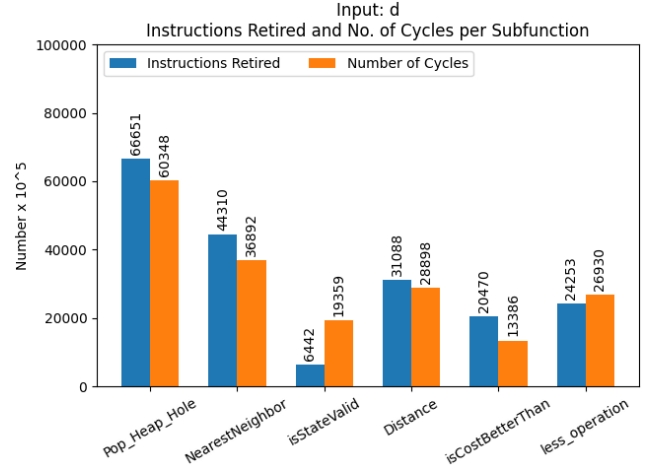


Figure 6. Instructions retired and number of cycles for *isStateValid*.

structions retire per cycle for *isStateValid*. This indicates that there is a lot of room for improvement for this function in particular. As *isStateValid* also is one of the three most expensive subfunctions in RRT\* for input *d* (as seen in Figure 4, the orange bar), fixing some of the issues of this subfunction should lead to noticeable improvements in RRT\*.

Moreover, from Table 6, we can see that *isStateValid* specifically suffers from being memory bound (46.2% of pipeline slots) and from having a lot of bad speculation (35.4% of pipeline slots). The former issue is also demonstrated by Figure 7: *isStateValid* has a significant amount of Last-Level Cache Miss Counts, meaning that these memory requests must be served by the main memory. This introduces a lot of extra latency and is, thus, a considerable limitation to RRT\*.

The latter issue of 35.4% of the pipeline slots being lost due to bad speculation is a result of branch mispredictions. This, too, is a significant amount of pipeline slots and limits the efficiency of RRT\* considerably.



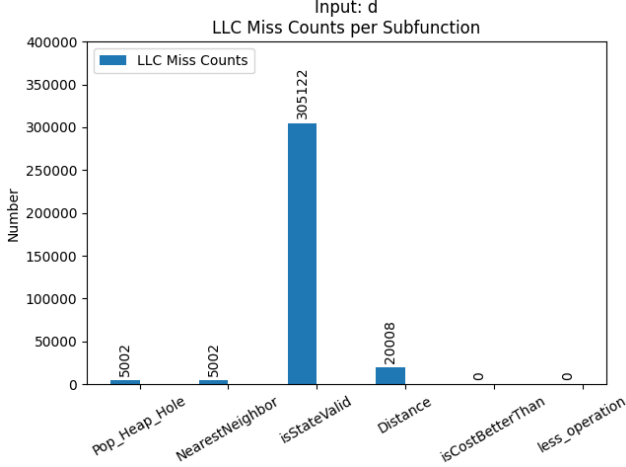


Figure 7. LLC Miss Counts

### 5.5.2 Distance

The *Distance* subfunction presents another limitation to RRT\*. As seen from Figure 4, it takes the most amount of CPU-Self time out of the six subfunctions, yet only 24.5% of its pipeline slots retire, meaning that 75.5% of pipeline slots are used inefficiently. Especially since *Distance* takes such a considerable amount of time in the execution of RRT\*, this is a limitation that should be addressed. Most of the inefficiently used pipeline slots, 45%, stem from branch mispredictions.

### 5.5.3 NearestNeighbor

While *NearestNeighbor* takes the least amount of time from the six considered subfunctions (see Figure 4), it also has the second lowest percentage of retired instructions (only 20.4%), meaning that there is a lot of opportunity to make this subfunction more efficient. Specifically, *NearestNeighbor* suffers from 47.6% of its pipeline slots being lost due to branch mispredictions, as well.

### 5.5.4 Pop\_Heap\_Hole, isCostBetterThan, and less

As seen in Table 6, the remaining three subfunctions also mainly suffer from Branch Mispredictions.

## 6. Discussion

In our experiments, we have found that the RRT\* motion planning algorithm is quite powerful, only taking about 1.2 seconds to compute a path in a static, obstacle-filled environment. This amount of time is not unreasonable for such an algorithm, making it a very feasible option for motion planning in static environments.

However, we begin to see the limitations of RRT\* once moving objects are introduced. In a real-world scenario, it can easily happen that objects move at a pace at which RRT\* cannot keep up its calculations anymore, causing any system that employs this algorithm to not seamlessly operate anymore.

As we can clearly see from Figure 5 and from Table 6, the probably most effective way to address this limitation would be to reduce the number of branch mispredictions the RRT\* algorithm experiences. 31.8% of its pipeline slots get completely lost, due to the algorithm predicting wrong branches, resulting in already finished work getting canceled and thrown away. Minimizing these branch mispredictions would lead to a significant improvement in RRT\*'s efficiency, making it up to almost a third faster. Additionally, most of the Front-End Latency issues are due to Branch Resteers, meaning that an approximately additional 10% of pipeline slots would retire.

Particularly for the *isStateValid* function, it would also be interesting to look a bit further into the memory issue, as the amount of LLC Misses signals that a lot of DRAM needs to be accessed, thereby greatly increasing RRT\*'s latency, especially when considering the fact that state collision checks are a huge element of motion planning.

## 7. Future Work

As all this gathered data was gathered in purely simulated scenarios, it would be a lot more insightful if this data could be gathered with an actual robot with the specific hardware a robot has access to. Moreover, while we have determined that RRT\* struggles a lot in dynamic environments, we have not at all considered the fact that almost all robots first execute a localization algorithm before starting motion planning when faced with a dynamic environment. Testing RRT\* with a preceding localization step was out of the scope of this paper, but it would - we assume - add quite a bit more latency. Thus, gathering this data with a robot that has access to the full computational pipeline would give a lot more useful insights into RRT\*'s limitations in dynamic environments.

Additionally, it would also be interesting to further test our proposed predictive version of RRT\* in dynamic environments with an actual robot. Perhaps this could be a good way to address this limitation, too. We have only performed a simulation, so a real-world test would definitely be necessary.

Last but not least, we believe it would be very interesting to test out some proposed solutions to the branch misprediction problem. According to Chou et al. [3], for example, it is not always the case that a branch misprediction must lead to all the affected instructions being useless. Rather, it can often be the case that some instructions can be reused and, thus, do not need to be re-executed, thereby saving

valuable time. Such instructions that do not need to be re-executed despite a branch misprediction are called *control-* and *data-independent*. Employing this strategy might lead to a considerable improvement in RRT\*'s branch misprediction problem.

## References

- [1] Ompl rrt star. [https://ompl.kavrakilab.org/classompl\\_1\\_1geometric\\_1\\_1RRTstar.html](https://ompl.kavrakilab.org/classompl_1_1geometric_1_1RRTstar.html). Accessed: 2023-12-04. [3](#)
- [2] Mohammad Bakhshalipour, Seyed Borna Ehsani, Mohamad Qadri, Dominic Guri, Maxim Likhachev, and Phillip B. Gibbons. Racod: Algorithm/hardware co-design for mobile robot path planning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture, ISCA '22*, page 597–609, New York, NY, USA, 2022. Association for Computing Machinery. [2](#)
- [3] Yuan Chou, Jason Fung, and John Paul Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th International Conference on Supercomputing, ICS '99*, page 109–118, New York, NY, USA, 1999. Association for Computing Machinery. [9](#)
- [4] Jun Ding, Yinxuan Zhou, Xia Huang, Kun Song, Shiqing Lu, and Lusheng Wang. An improved rrt\* algorithm for robot path planning based on path expansion heuristic sampling. *Journal of Computational Science*, 67:101937, 2023. [2](#), [3](#)
- [5] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning, 2011. [2](#)
- [6] F. Lamiroux and J.P. Laumond. Flatness and small-time controllability of multibody mobile robots: Application to motion planning. In *1997 European Control Conference (ECC)*, pages 3246–3251, 1997. [1](#)
- [7] Steven M. LaValle and Jr. James J. Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001. [1](#)
- [8] Ahmed Qureshi and Yasar Ayaz. Potential functions based sampling heuristic for optimal path planning. *Autonomous Robots*, 40, 08 2016. [2](#), [3](#)

## 8. Individual Contribution

### 8.1. Jonas Raedler

- Collected all the data of the RRT\* algorithm and its subfunctions on VTune
- Created Figures 4, 5, 6, and 7, as well as Tables 4, 5, and 6.
- Report sections: Data Analysis - 5.3, 5.4, 5.5, Discussion, and Future Work. Edited everything for flow.

### 8.2. Prithviraj Khelkar

- Implemented RRT\* using the algorithm provided in the OMPL library

- Implemented a simulator to test the RRT\* algorithm with moving objects.
- Implemented RRT\* for multiple DOF Scenes.
- Created inputs to conduct further data analysis on.
- Report sections: Introduction, Background, Motivation, Methodology, Implementation, Data Analysis - Stationary Scenes, Dynamic Scenes
- Tables 1, 2, and 3. Figures 1, 2, 3.

### 8.3. Shiven Sharma

- Data Analysis - isStateValid function