





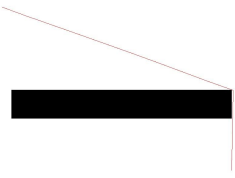
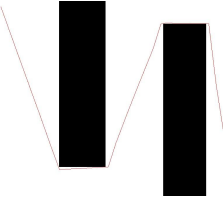
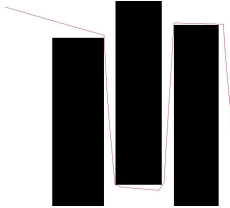
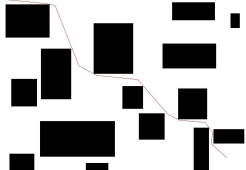
## Milestone 4

### Project Data Analysis

#### 1. Introduction

Our goal is to quantitatively analyze the OMPL implementation of the RRT\* algorithm. To achieve this, we ran the algorithm on a local Intel Machine<sup>1</sup> and used the VTune profiler in order to obtain precise measurements that would help us identify RRT\*'s bottlenecks and limitations.

In accordance with our second milestone, we decided to run the RRT\* algorithm on multiple inputs, increasing in complexity. The inputs we used (with the starting point being the top-left corner and the end-point being the bottom-right corner) can be found below.

Input: A	Input: B	Input: C	Input: Map
			
			

**Table 1:** The different inputs we used. The starting point of the path is the top-left corner and the end point is the right-bottom corner. The first row contains the original inputs and the second row contains the path returned by RRT\*.

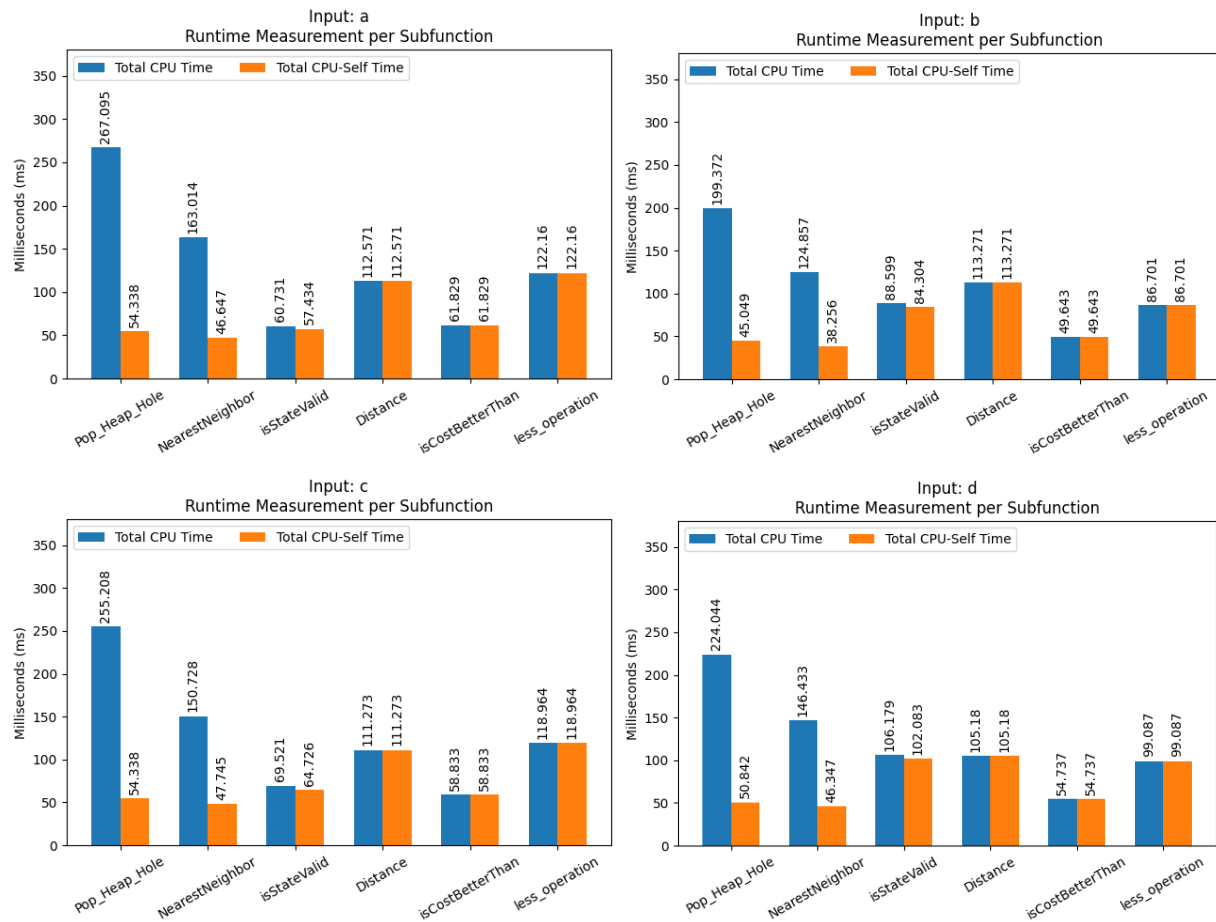
#### 2. Measurement Data

Using these inputs, we used VTune to get basic information about the algorithm's execution, such as the time during which the CPU is actively executing RRT\*, the number of actual 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, which we wanted to examine further in milestone 2. We are looking at the *less* operation, as it takes up a significant amount of CPU usage time.

After testing with the aforementioned four different inputs, each characterized with varying degrees of complexity, it becomes evident that the execution time, as depicted in Figure 1 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.

<sup>1</sup> see Appendix for more details about the machine

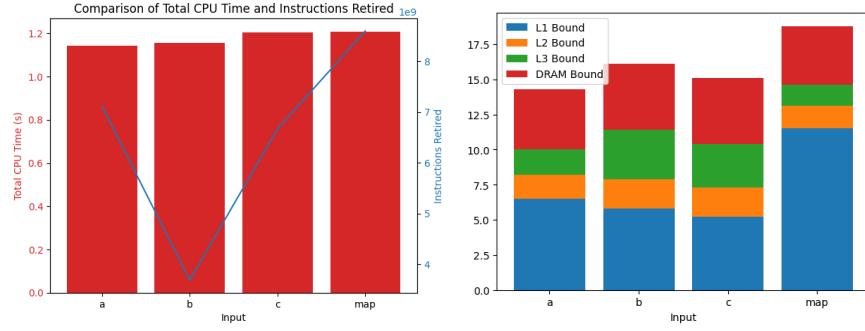


**Figure 1:** Each plot contains the measurements for Total CPU Time (blue) and Total CPU-Self Time (orange) in ms for the 6 different subfunctions for a different input. Total CPU Time refers to the total amount of time the subfunction is executing on the CPU (including all the subfunctions those subfunctions call), whereas the Total CPU-Self Time only refers to the time without any subfunctions that are called within the subfunction in question.

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 Map', the execution time increases from 57.4ms to 102.0ms. This is most likely due to the fact that the subfunction 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.

### 3. Comparison of Inputs

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



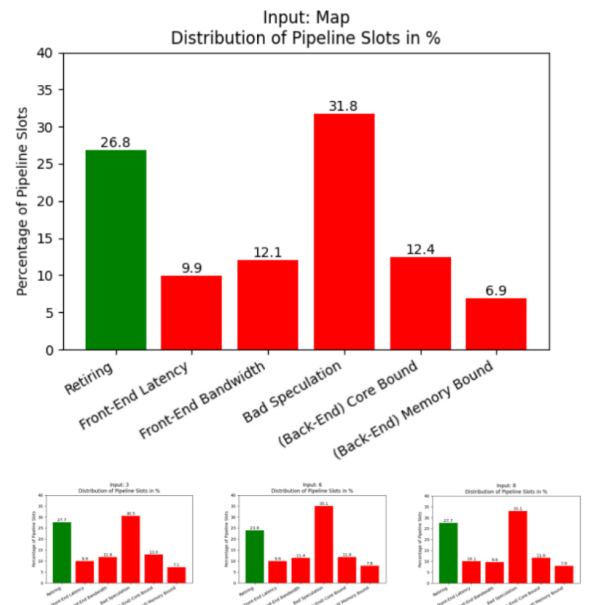
Further exploration into the cache hierarchy reveals more intriguing insights: the L1 Bound sees substantial variation, with input "Map" reaching the highest percentage at 11.5%, indicating a potentially higher demand for L1 cache. Meanwhile, the L2 Bound experiences the lowest values in Input "Map" 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 "Map" highest and input "a" lowest.

Input	Total CPU Time (s)	Instructions Retired	IPC	Stores	LLC Miss Count	L1 Bound	L2 Bound	L3 Bound	DRAM Bound
A	1.144	7,116,200,000	1.276	339,800,000	490,196	6.5%	1.7%	1.8%	4.3%
B	1.156	3,685,080,000	1.199	97,300,000	460,184	5.8%	2.1%	3.5%	4.7%
C	1.204	6,670,160,000	1.244	290,800,000	605,242	5.2%	2.1%	3.1%	4.7%
Map	1.208	8,595,720,000	1.277	367,700,000	515,206	11.5%	1.6%	1.5%	4.2%

**Table 2:** This table showcases different data metrics for each of the four inputs

Based on these findings, we have decided to only focus on the input *Map* for the rest of our analysis. Since it is - evidently - the most complex input and since it seems to present the limitations of the RRT\* algorithm the strongest, we believe that diving deeper into the performance of RRT\* on this particular input only 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 Fig. 2).

**Figure 2 (to the right):** This figure shows four different bar plots, one for each input (Map 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 complete). 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.



## 4. Analyses of Subfunctions

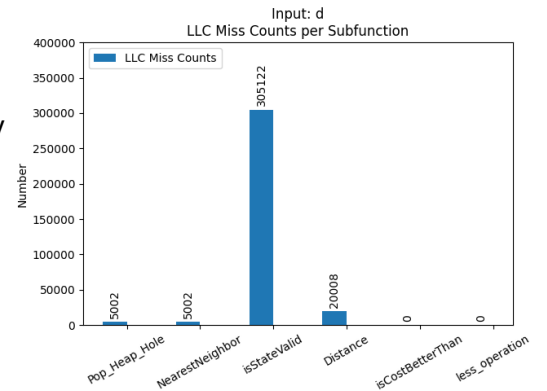
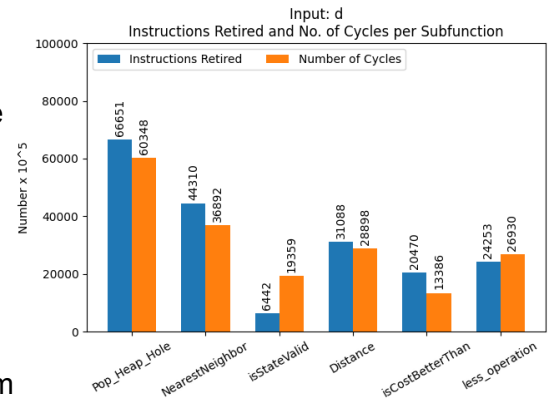
Function	Retiring	Front-End Bound Latency   Bandwidth		Bad Speculation	Back-End Bound Memory Bound   Core Bound	
<i>Pop_Heap_Hole</i>	32.3 %	9.1 %	14.7 %	36.8 %	1.4 %	5.7 %
<i>NearestNeighbor</i>	20.4 %	10.2 %	7.9 %	47.6 %	4.4 %	9.5 %
<i>isStateValid</i>	8.6 %	2.1 %	2.4 %	35.4 %	46.2 %	5.3 %
<i>Distance</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_operation</i>	25.7 %	10.6 %	12.6 %	41.6 %	1.2 %	8.4 %

**Table 3:** 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.

### 4.1 *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 the figure to the right: the *isStateValid* function has an extremely low amount of Instructions Retired, especially in comparison to the other subfunctions. Notably, *isStateValid* also is the only subfunction that has a significant negative discrepancy between retired instructions and the number of cycles. From table 2, we know that RRT\* has an average IPC of 1.277, but only 0.318 instructions 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 *Map* (as seen in Figure 1), fixing some of the issues of this subfunction should lead to noticeable improvements in RRT\*.

From table 3, 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 clearly demonstrated by the figure to the right: *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.

#### **4.2 Distance**

The *Distance* subfunction presents another limitation to RRT\*. As seen from figure 1, 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.

#### **4.3 NearestNeighbor**

While *NearestNeighbor* takes the least amount of time from the six considered subfunctions (see figure 1), 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. Conclusions**

As we can clearly see from figure 2 and from table 3, the RRT\* algorithm experiences a lot of limitations due to bad speculation. 31.8% of its pipeline slots get completely lost, as the algorithm seems to often predict wrong branches, meaning that useful work gets canceled. Minimizing these branch mispredictions would lead to a significant improvement of RRT\*'s efficiency. This could be done either by reducing the number of if-statements (by, for instance, adding more work to individual if-statements, thus hopefully reducing the amount of total if-statements) or by moving them higher up in the code, so that they execute earlier. One could also dive deeper into the source code and determine which branches exactly are often mispredicted. This information could then be used to make the algorithm more predictable, aiming towards fewer overall mispredictions.

### **6. Next Steps**

For the final report, we need to look into the three remaining subfunctions. RRT\* also has some Front-End Bound limitations, specifically in the *isCostBetterThan*, *less\_operation*, and *Pop\_Heap\_Hole* subfunction. We are planning on being more detailed in this area, in order to address the limitations of RRT\* as thoroughly as possible.

On top of that, it would also be beneficial to look into the source code and potentially identify some of the specific branches that cause the mispredictions. If we can manage this, we could even suggest some changes to improve the efficiency of RRT\*.

Other than that, we still need to put all of our pieces together, so that we have the introduction to the algorithm, its context in robotics, and the data and its analysis together in a coherent form.

## **Appendix**

### **Machine Details:**

The machine used is an HP ENVY Laptop 13-ba1xxx model with an 11th Gen Intel(R) Core(TM) i7-1165G7 processor, running with a base speed of 2.80 GHz. The CPU has 4 cores and 8 Logical Processors. It has an L1 cache with a capacity of 320 KB, an L2 cache with 5.0 MB, and an L3 cache with 12.0 MB. The machine further has 15.8 GB of total physical memory and a memory speed of 3200 MHz. Last but not least, it has an Intel(R) Iris(R) Xe Graphics GPU with 7.9 GB of shared GPU memory.

The data has been gathered using the Intel VTune Profiler

(<https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>)

### **Individual Contributions**

#### **Jonas:**

- Got VTune running on my machine
- Ran the RRT\* algorithm and collected all the data.
- Analyses of the subfunctions
- Conclusion
- Next Steps

#### **Prithvi:**

- Implemented RRT\* using the algorithm provided in the OMPL library
- Created the different inputs and generated the outputs paths.
- Comparison analysis for CPU time and Instructions retired, and memory.

#### **Shiven:**