
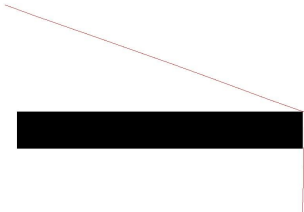
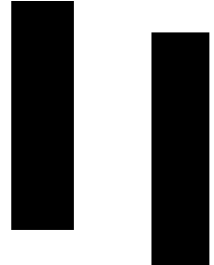
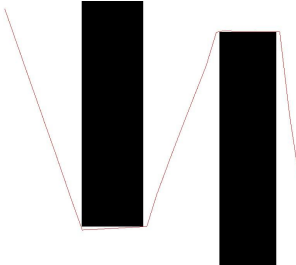
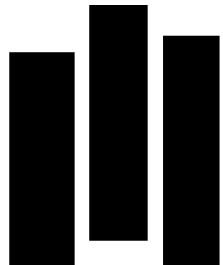
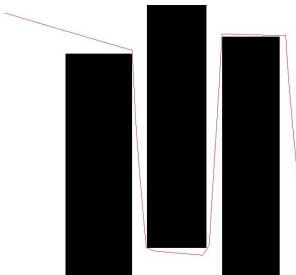

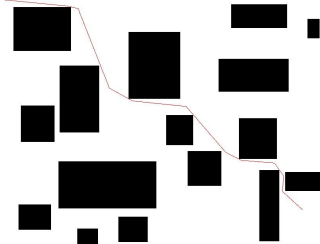


Milestone 3 - Project Data

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 Machine¹ 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), along with some information about the execution and RRT*'s parameters, can be found below.

	Input Scenes	Output paths by RRT Star	
a)			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
b)			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
c)			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
d)			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

¹ see Appendix for more details about the machine

Milestone 3 - Project Data

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 percentage of clock ticks that are L1, L2, L3, and DRAM bound, the number of LLC Miss Counts, 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 *lesser* operation. We took an interest in the *Pop_heap_hole_by_index* function, as it is part of the rewiring process, which we wanted to examine further in milestone 2, and we are looking at the *lesser* operation, as it takes up a significant amount of CPU usage time.

The following tables and plots showcase our current measurement data:

Input	Total CPU Time (s)	Instructions Retired	IPC
a	1.144	7,116,200,000	1.276
b	1.156	3,685,080,000	1.199
c	1.204	6,670,160,00	1.244
d	1.208	8,595,720,000	1.277

Table 1: This showcases the total CPU time taken as well as the Instructions Retired, as well as the Instructions Retired Per Cycle (IPC).

Input	L1 Bound	L2 Bound	L3 Bound	DRAM Bound
a	6.5%	1.7%	1.8%	4.3%
b	5.8%	2.1%	3.5%	4.7%
c	5.2%	2.1%	3.1%	4.7%
d	11.5%	1.6%	1.5%	4.2%

Table 2: The percentages of clockticks that are L1, L2, L3, and DRAM Bound for each input.

Input	Stores	LLC Miss Count
a	339,800,000	490,196
b	97,300,000	460,184
c	290,800,000	605,242
d	367,700,000	515,206

Table 3: The total number of stores and LLC Miss Counts for each input.

Milestone 3 - Project Data

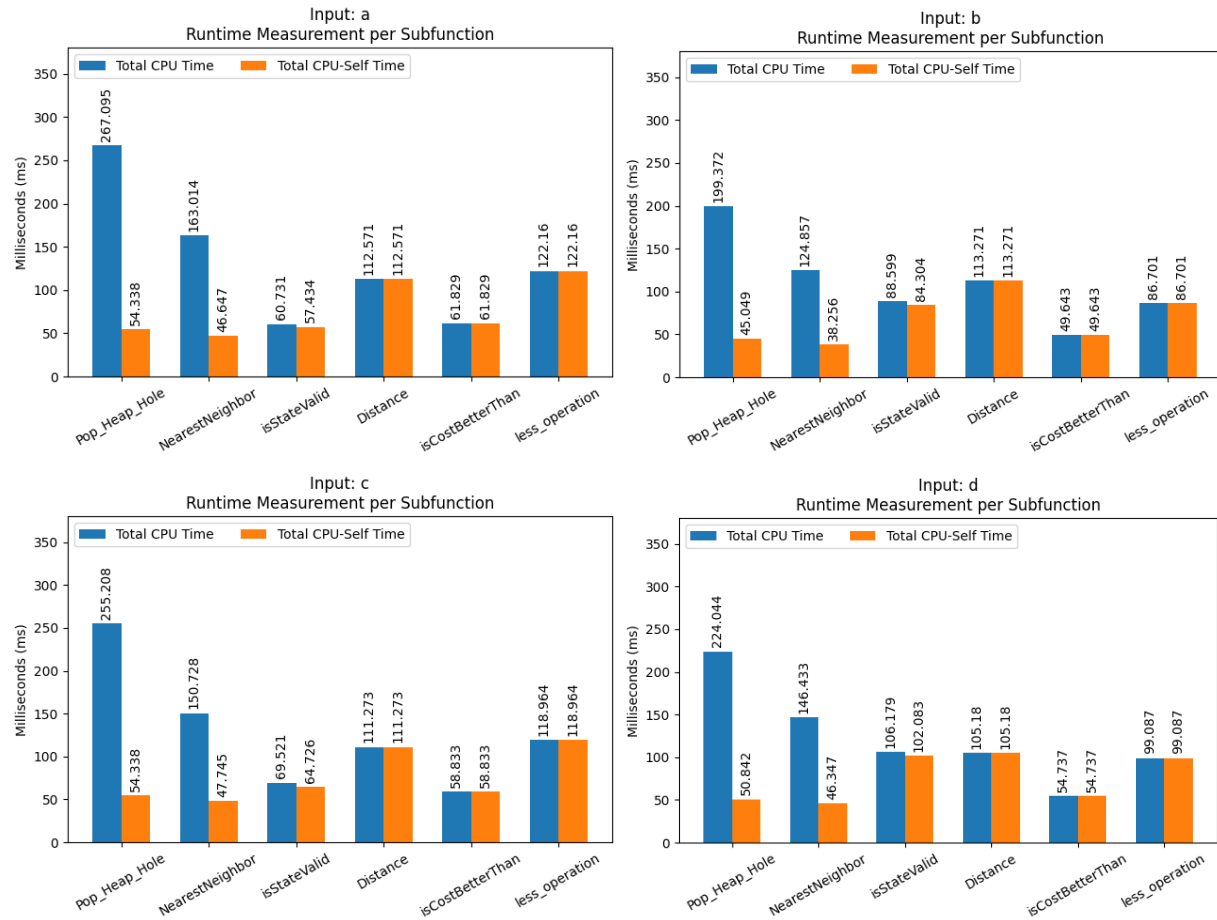
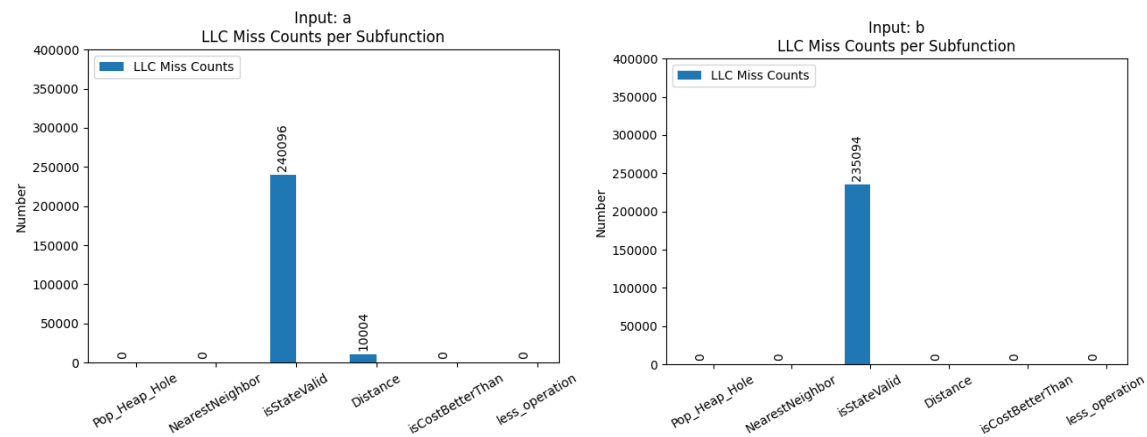


Figure 1: Each plot contains the measurements for Total CPU Time (blue) and Total CPU-Self Time (orange) in ms for the 6 different subfunctions for a different input. Total CPU Time refers to the total amount of time the subfunction is executing on the CPU (including all the subfunctions those subfunctions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



Milestone 3 - Project Data

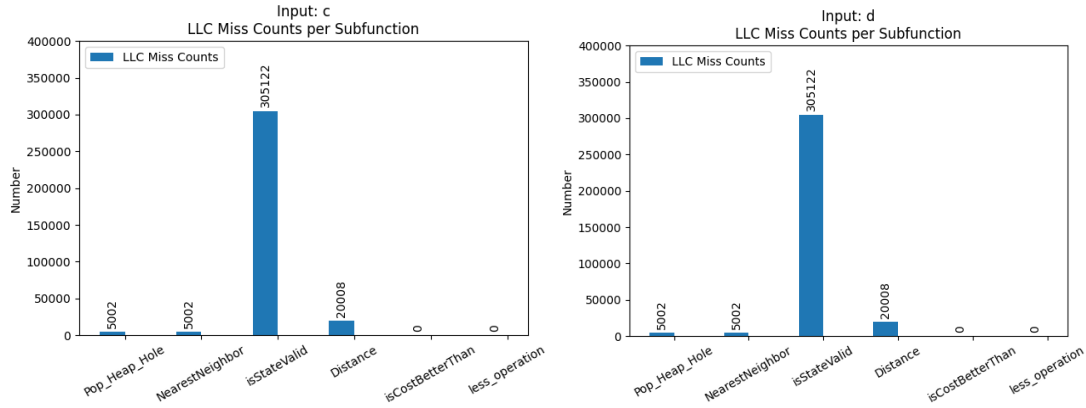


Figure 2: Each plot contains the number of LLC Miss Counts per subfunctions for a different input.

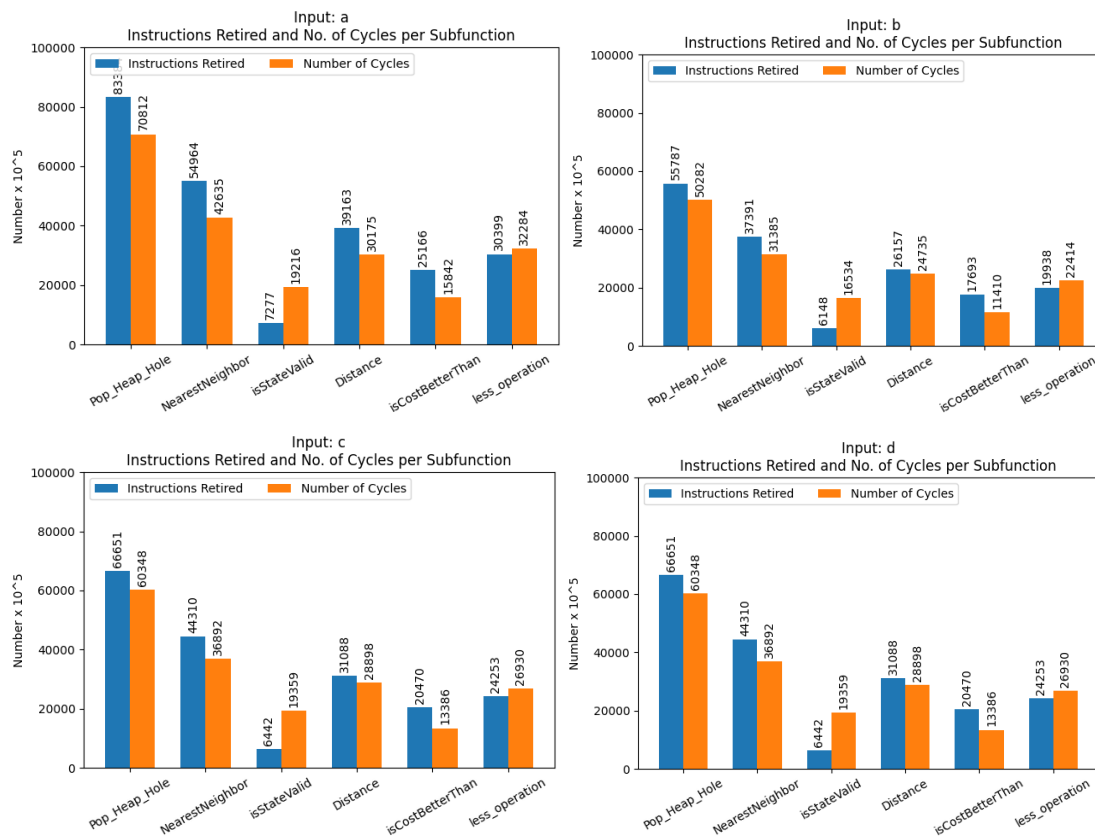


Figure 3: Each plot contains the measurements for the Number of Instructions Retired (blue) and the Number of Cycles (orange) ($\times 10^5$) for the 6 different subfunctions for a different input.

Notes

We are, unfortunately, still in the process of getting the AMD profiler to work properly, so this milestone only includes measurement data from the Intel machine.

We are also having some issues with the Microarchitecture Exploration result, as VTune does not - currently - provide us with the distribution of pipeline slots that fall under the categories of retired, bad speculation, front-end bound, and back-end bound. We will try our best to collect this data until the next milestone.

Milestone 3 - Project Data

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

Prithvi:

- Implemented RRT* using the algorithm provided in the OMPL library
- Created the different inputs and generated the outputs
- Gathered the basic information about the execution of the algorithm

Jonas:

- Got VTune running on my machine
- Ran the RRT* algorithm and collected all the data from the tables and figures in this milestone on VTune
- Created Figure 1, 2, and 3

Shiven:

- Got VTune
- Created Table 1, 2, and 3