CS 599: Milestone 2

What specific software implementation(s) of the algorithm will you study, and where will you get the code?

The Optimal Rapidly-exploring Random Trees (Star) algorithm is included as one of the geometric planners as part of the Motion class in the Open Motion Planning Library (OMPL). OMPL provides the source code of the RRT* algorithm in the C++ programming language. The algorithm inherits from ompl::base::Planner and requires a Space Information Pointer (base::SpaceInformationPtr) for initialization. In our proposed experiment plan, we plan to study the behavior of this algorithm under varying input parameters to observe it's limitations. The primary focus of our studies is observing the performance of the `solve(const base::PlannerTerminationCondition &ptc)` function, which, as the name suggests, is the heart of the algorithm. We will observe the performance of this function under varying parameters like different space information (RealVectorStateSpace, SE3StateSpace), varying goals, ranges, nearest neighbor factor (K), etc. Some algorithm parameters that we'll work with are - setGoalBias(), setRange(), setRewireFactor().

We also plan to use a variant of this algorithm provided in the RTRBench Suite, for the sake of a thorough comparative study of two different implementations under similar conditions.

What machine(s) will you run the code on, and what will be your experimental setup methodology?

We will run the RRT* algorithm on two machines. The first machine is a Lenovo Legion 5 with an AMD Ryzen 7 6700H processor, running with a base speed of 3.20 GHz. The CPU has 8 cores and 16 Logical Processors, as well as an L1 cache with a capacity of 512 KB, an L2 cache with 4.0 MB, and an L3 cache with 16.0 MB. The machine further has 15.2 GB of total physical memory and a memory speed of 4800MHz. Last but not least, it has an NVIDIA GeForce RTX 2050 T GPU with 4.0 GB dedicated and 7.6 GB shared GPU memory.

The second machine we will use 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.

In order to effectively quantitively analyze the RRT* algorithm, we intend to use the μProf profiler for the AMD machine and the Intel VTune profiler for the Intel machine.

Furthermore, in order to properly test the algorithm, we are going to use a variety of different inputs. Our current plan is to use already existing input data that we have found in other open-source motion-planning projects, but we are also going to try generating inputs using some procedural map generator algorithm. This, we hope, will increase the randomness and complexity of our tests. On top of that, we plan to experiment with various algorithm parameters, such as setting different goal bias values, range values, and rewire factors, to see how adjusting these parameters impacts the algorithm's performance.

Regarding the verification of the output of RRT*: while the primary output of the algorithm is a path, we are mostly interested in whether or not the algorithm comes up with a valid path. After all, the focus of this project is not the quality of the output, but rather the performance and the bottlenecks of RRT*. Therefore, we are simply going to rely on the Planner Status output that the OMPL implementation provides, which indicates whether the algorithm successfully found a solution or if the search was terminated due to a given termination condition. This, we believe, is a sufficient verification of the output.

What measurements will you take, and what data will you collect?

As mentioned earlier, we will use the VTune (Intel) and µProf (AMD) profiler to identify time and space bottlenecks for the RRT* algorithm. We have already set up the VTune profiler on the Intel machine (we are still working on µProf), so the following layout of our strategy will be based on VTune's terminology. µProf has the same functionality as VTune, so we will simply use the respective methods from µProf.

After running the initial analysis from the profiler on the RRT* algorithm to create a baseline runtime, we are going to run a Hotspot analysis. The Hotspot analysis will be performed with the Hardware Event-Based Sampling mode, as this mode enables the analysis of algorithms that run in less than a few seconds (we assume RRT* falls under this category) and it allows us to gain a more detailed insight into the bottlenecks of the subfunctions of the algorithm. Through the Hotspot Analysis, we will be able to determine which parts of the algorithm take the most time. This amount of time per subfunction of RRT* will be collected.

After that, we will perform a Microarchitecture Exploration analysis in order to assess how efficiently (or inefficiently) RRT* is running in more detail. This analysis provides us with the percentage of pipeline slots that fall under the category of retired, bad speculation, front-end bound, and back-end bound (which further divides into memory-bound and compute-bound). This breakdown of percentages will also be collected. The Microarchitecture Exploration analysis further offers an insight into which subfunctions of the RRT* algorithm contribute to these percentages. We will pay particular attention to subfunctions that are responsible for significant increases in the usage of pipeline slots. We expect to look at the Nearest-Neighbour Search, the Collision Check, the selection of the best parent node, and the rewiring subfunctions, specifically.

In general, we are going to measure the intensity of how the varying parts of the RRT* algorithm contribute to the different bounds. These measurements will be taken for different inputs of varying complexity level.

We can also use the System Overview Analysis to analyze the power usage and potential throttling reasons of the algorithm.

Lay out a strategy to answer these questions

What pressures does the state-of-the-art implementation of this algorithm place on what parts of the computing system stack?

With the data we plan to collect, it should be fairly straightforward to determine which parts of the computing system stack are mostly pressured by the RRT* algorithm. Since the profiler allows us to measure the percentage of pipeline slots that fall under the category of retired, bad speculation, front-end bound, memory-bound, and core-bound, we should be able to easily identify the categories with the highest percentages (and, thus, the most pressured ones, unless they fall under the category of retired).

Based on our hypotheses from milestone 1, we expect RRT* to be mostly memory- and/or compute-bound. If it is compute-bound, we can use the results from the Hotspot analysis to determine which part(s) of the algorithm take a significant amount of time and dive deeper into which specific lines of code contribute to this time usage. If it is memory-bound, we can run the Memory Access Analysis to gain more insight into what exactly the problems are (cache misses, general latency, and where that latency occurs (L1, L2, L3 cache, DRAM)).

These steps should enable us to evaluate what pressures this algorithm puts onto the parts of the computing stack.

In the best of circumstances, how much benefit would you expect from addressing these limitations?

The Microarchitecture Exploration Analysis provides a diagram that depicts pipe efficiency. This means that it will depict the bounds that limit the performance efficiency and it will state how much of the whole algorithm's efficiency suffers from this limitation (in percentage) - see the references for an example image.

This limitation percentage, taken together with the baseline runtime of RRT*, should allow us to identify how much performance boost we could achieve if we address the existing limitations. This would then allow us to determine whether this performance boost would be worth the required effort to address the limitations.

Additionally, give a timeline for your project, with specific intermediate goals and dates

Week 1 & 2 (10/16 - 10/29):
Study the algorithm implementation. Gather a (small) variety of valid inputs. Run RRT* on both systems. Collect the specified data using VTune and µProf using simple test cases. Start drafting the project data report.

Week 3 (10/30 - 11/5): (Project Data due 10/30)
Continue collecting and analyzing performance data. Experiment with varied input parameters, to identify potential bottlenecks in the RRT* algorithm. Finish the project data report with the obtained data.

Week 4 (11/6 - 11/12): (Project Data Analysis due 11/9)
Continue with analyzing the performance data to identify potential bottlenecks.
Begin developing scripts or code instrumentation for more in-depth data collection. Start drafting the final project report.

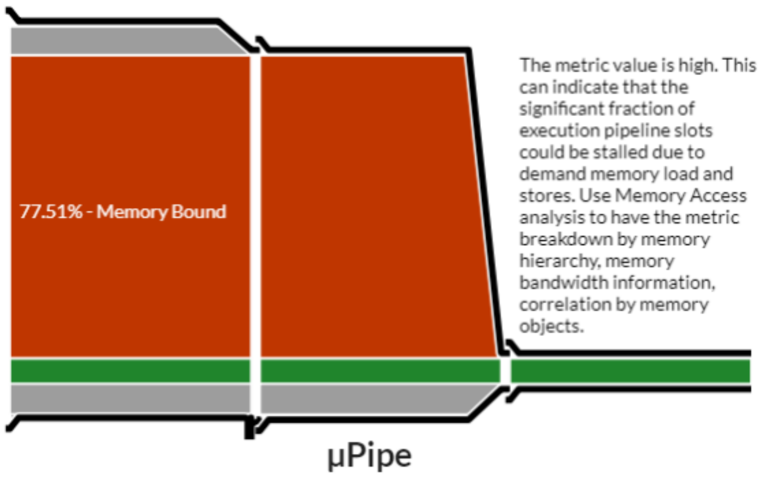
Week 5 & 6   (11/13 - 11/26):
Deepen the analysis of collected data to gain insights into the algorithm's behavior.
Refine the experimental setup by introducing real-world or complex test cases, simulating challenging environments. Investigate the sensitivity of the algorithm to specific input parameter variations in more detail. Finish the final project report. Begin carefully curating content for the project presentation.

Week 7 (11/27 - 12/3) : (Final Project Report due 11/30)
Finish the final report and continue working on the presentation.

Week 8+(12/4): (Project Presentations due)
Hold the final presentation

References

1. ompl::geometric::RRTstar Class Reference (kavrakilab.org)
2. RTRBench | A Benchmark Suite for Real-Time Robotics (cmu-roboarch.github.io)
3. https://www.amd.com/en/developer/uprof.html
4. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
5. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/memory-access-analysis.html
6. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/microarchitecture-pipe.html



The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

Individual contributions

We divided the work done into separate sections, as follows:

Jonas worked on:
- Section 2: What machine(s) will you run the code on, and what will be your experimental setup methodology?
- Section 3: What measurements will you take, and what data will you collect?
- Section 4: Lay out a strategy to answer these questions.

Prithvi worked on:
- Section 1: What specific software implementation(s) of the algorithm will you study, and where will you get the code?
- Section 2: What machine(s) will you run the code on, and what will be your experimental setup methodology?
- Section 3: What measurements will you take, and what data will you collect?