**D. Motion Planning**

For any type of autonomous robot, motion planning adds the essential piece in the autonomy. It helps to find a sequence of valid configurations that moves the vehicle from a source location to a destination. Motion planning algorithms help to plan the shortest obstacle-free path to the goal.

We can classify motion planning into these categories:

* Global Planning: Path planning algorithms
* Behavior Planning: High-level decision making
* Local Re-Planning: Trajectory generation and optimization
* Velocity Planning: Velocity profile generation

**1) Global Path Planner**

In a typical autonomous vehicle workflow, the perception model provides information on how the environment around the vehicle looks like, that includes the obstacles that the robot should avoid bumping into. We get the location of the robot at each movement or state from the localization algorithms.

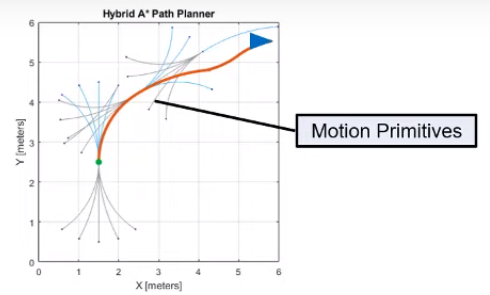
1. **Options Assessment**

For global path planning in the case of non-holonomic vehicles, we can classify the types of path planning algorithms into:

* Search-based algorithms such as Hybrid A\*
* Sampling-based algorithms such as: RRT, RRT\*,

**Hybrid A\* Algorithm**

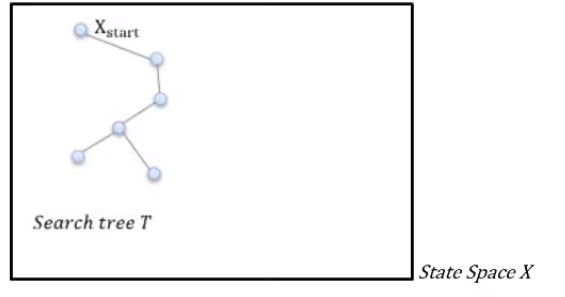
Hybrid A\* is similar to the traditional A\* algorithm (which can only be used for holonomic robots) in how the search space, that is (x,y,θ), is discretized. But unlike A\*, hybrid A\* associates a continues 3-d state of the vehicle, or as we call them motion primitives with each grid cell. These motion primitives are checked for collisions in the map using state validators which generates a smooth obstacle-free path.



Hybrid A\* also guarantees kinematic feasibility and takes differential constraints (orientation and velocity) into account. We can give the orientation of the vehicle as an input along with the start and goal locations and we can see how we get different drivable paths based on different goal orientations.

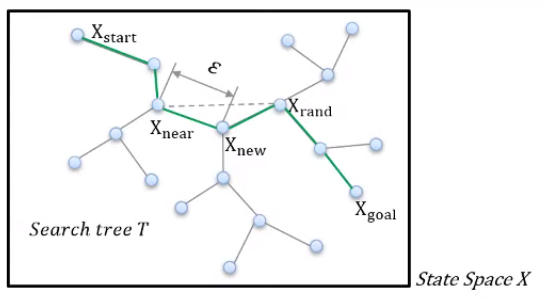
A big problem with grids and search-based algorithms is the exponential explosion in the number of nodes when the dimensionality of the problem increases and the time to find a solution quickly becomes intractable. Search-based algorithms are not suitable for applications that have high degrees of freedoms or when the map size is very large. Storing the grid information for a large map becomes computationally expensive and this is where the sampling-based planning algorithms are helpful.

**Rapidly-exploring Random Tree (RRT)**

RRT creates a tree in the state space with randomly sampled states or nodes. The RRT algorithm is designed for efficiently searching non-convex high dimensional spaces. RRTs are constructed incrementally in a way that it quickly reduces the expected distance of a randomly chosen node in the tree.

We first define the state space with an initial tree T that has start point as one of the nodes in it to represent the map in a way that the planning algorithm or the planner can understand it we represent it in the form of a state space.

A state of our vehicle is its position (x,y), its orientation (theta), and its steering angle. A state space is a set of many possible vehicle states.

For example, we randomly sample a state Xrand in the state space X, and we find a node Xnear nearest to Xrand that already exists in the tree. This Xrand can be anywhere in the state space, so we need another node Xnew between Xrand and Xnear to expand the tree. And we repeat this process until we reach the Xgoal. Every time we sample a new load node like Xnew we also check for collision between the nodes Xrand and Xnew.

We can use motion primitives or motion models like Dubins-curve to generate local motion from one node to another RRTs are particularly suitable for path planning problems that involve obstacles and differential constraints.

The RRT algorithm gives a valid path but not necessarily the shortest path which brings us to the following alogrithms.

**RRT-A\***

RRT-A\* is an improved heuristic of RRT where the cost function of A\* is introduced into the RRT algorithm to optimize the performance. Meanwhile, several metric functions are used as the heuristic information

functions respectively to measure the performance of different metric function. Research and simulation results have shown\* that the Manhattan heuristic information function based RRT-A\* planning algorithm is better than the other improved RRT algorithms in optimization path and computational cost.

When choosing the nearest neighbor Xnear as the following best input variable, the original RRT adopts the principle of nearest neighbor accomplished by the metric function. And there are several metric functions used in RRT algorithm as follows, only in the space of two dimensions.

The Manhattan distance between two points can be expressed as:



* **Artificial Potential Fields**
* **Sampling-based algorithms**
* **Global Potential Functions: NF1, NF2, Harmonic Potentials**

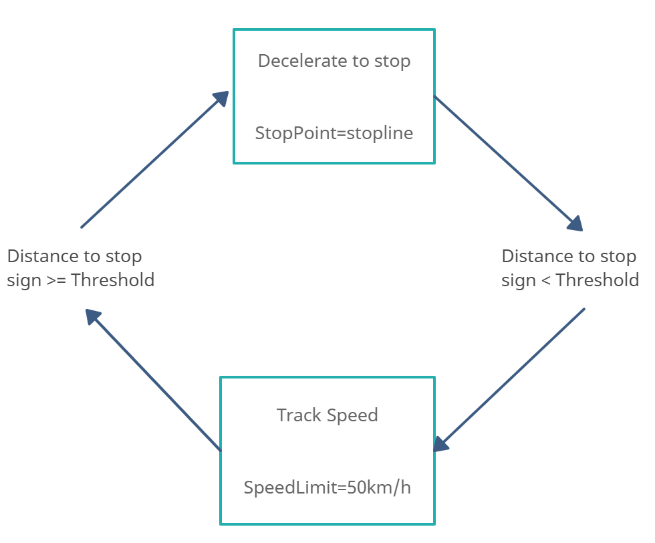
**2) Behavioral Planner**

**i. Options Assessment**

* Finite State Machines (FSMs)

One approach traditionally used to represent the set of rules required to solve behavior selection is a finite state machine.

The first set of components of a finite state machine, is the set of states. For behavior planning system, the states will represent each of the possible driving maneuvers, which can be encountered. In the example of handling a stop sign intersection with no traffic, we will only need two possible maneuvers or states, track speed and decelerate to stop. The maneuver decision defined by the behavior planner is set by the state of the finite state machine. Each state has associated with it an entry action, which is the action that is to be taken when a state is first entered.

For our behavior planner, these entry actions involve setting the necessary constraint outputs to accompany the behavior decision.

For instance, as soon as we enter the decelerate to stop state, we must also define the stopping point along the path. Similarly, the entry condition for the track speed state, sets the speed limit to track.

The second set of components of the finite state machine, are the transitions, which define the movement from one state to another. In our two-state example, we can transition from track speed to decelerate to stop, and from decelerate to stop back to track speed. Note that there can also be transitions which return us to the current state, triggering the entry action to repeat for that state. Each transition is accompanied with a set of transition conditions that need to be met before changing to the next state. These transition conditions are monitored while in a state to determine when a transition should occur. For our simple example, the transition conditions going from track speed to decelerate to stop involved checking if a stop point is within a threshold distance in our current lane.

Similarly, if we have reached zero velocity at the stop point, we can transition from decelerate to stop back to track speed. These two-state example highlights the most important aspects of the finite state machine-based behavior planner. As the number of scenarios and behaviors increases, the finite state machine that is needed becomes significantly more complex, with many more states and conditions for transition.

As a result of the decomposition of behavior planning into a set of states with transitions between them, the individual rules required remain relatively simple. This leads to straightforward implementations with clear divisions between separate behaviors. However, as the number of states increases, the complexity of defining all possible transitions and their conditions explodes. There is also no explicit way to handle uncertainty and errors in the input data. These challenges mean that the finite-state machine approach, as we approach full level 5 autonomy, tends to run into difficulties such as:

*Rule-explosion:* As we develop a more complete set of scenarios and maneuvers to handle, the number of rules required grows very quickly. This limitation means that while it is possible to develop a finite state machine behavior planner to handle a limited operational design domain, developing a full level 4 or 5 capable vehicle is almost impossible.

*Noisy environment:* While the addition of hyperparameters can be used to deal with some noise, this type of noise-handling is only able to deal with some very limited situations.

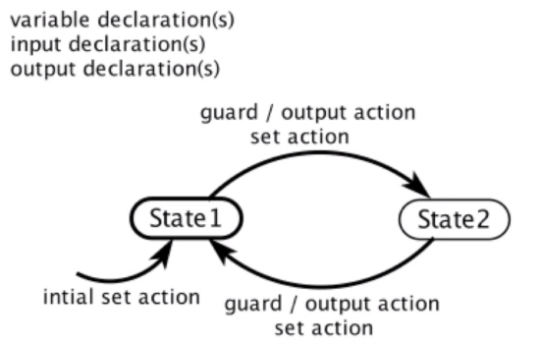
*Hyperparameter tuning:* As the behaviors required get more complex, the number of hyperparameters to both discretize the environment and handle some of the low-level noise grow rapidly. All these hyperparameters have to be tuned very carefully and this can be a lengthy process.

*Unencountered situations:* Due to the program nature of this approach, it is very likely that there will arise a situation in which the programmed logic of the system will react in an incorrect or unintended manner.

*Maintainability*: when adding or removing a state, it is necessary to change the conditions of all other states that have transition to the new or old one. Big changes are more susceptible to errors that may pass unnoticed.

*Scalability*: FSMs with many states lose the advantage of graphical readability, becoming a nightmare of boxes and arrows.

*Reusability*: as the conditions are inside the states, the coupling between the states is strong, being practically impossible to use the same behavior in multiple scenarios.

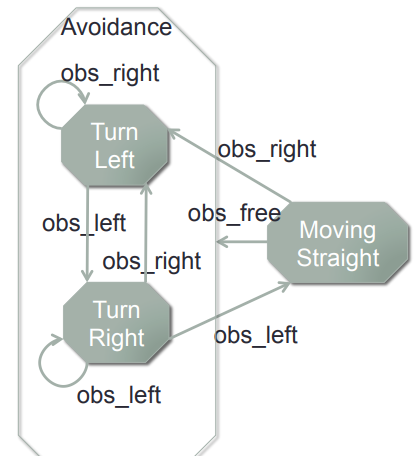
* Extended Finite State Machines (EFSMs)

Extended finite-state machines (EFSM) are similar to conventional finite-state machines, but augmented with updates associated to the transitions; formulas constructed from variables, integer constants, the Boolean literals true and false, and the usual arithmetic and logic connectives. With an EFSM, we extend to state machines with variables that may be either read or written.

EFSMs suffer from the same problems as described for FSMs. Thus, we continued our research for alternatives.

* Hierarchical Finite State Machines (HFSMs): Harel’s StateCharts

Harel's state machines are a mixture of mealy and Moore machines with three extra concepts:

*Hierarchy*: in this example of an obstacle avoidance, moving straight can each be thought of as part of a parent state. Organizing through hierarchy further compartmentalizes the design and can reduce the number of transition lines required to represent a system.

*Orthogonality*: the system might be comprised of multiple state machines operating simultaneously. While it's certainly possible to have two separate diagrams, in some cases, it may be convenient to display things together. We can illustrate this by including a Turn Right signal light in our state machine. The light turns on or off independent of the actions of the obstacle avoidance system.

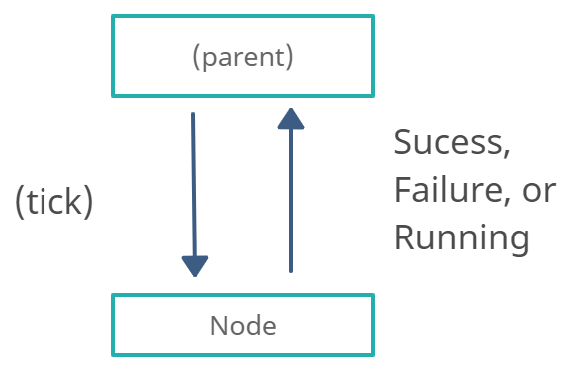
*Broadcasting*: these mostly autonomous state machines could exchange information with one another. Let's say we want the Turn Right signal light to turn off every time we’re in the Moving Straight state. So, what we will do is broadcast the light off event from the Moving Straight state, thus the light goes out when we want it to.

HFSM certainly provide a way to reuse transitions, but it’s still not an ideal solution. The problem is that:

* Reusing transitions isn’t trivial to achieve, and requires a lot of thought when you have to create logic for many different contexts.
* Editing transitions manually is rather tedious in the first place.

Another solution is to focus on making individual states modular so they can be easily reused as-is different parts of the logic. Behavior trees take this approach.

* Behavior Trees (BTs)

One of main problems with FSMs is that they’re full of GOTO statements, which both harms modularity and increases complexity, making it very hard to debug in the case of complex problems. Another drawback of FSMs, is that the optimal transition structure is identical for all FSM-states. This is justified by the fact that the next action should not depend on the previous action, but on only on the current world state.

|  |  |  |
| --- | --- | --- |
|  | **Finite State Machines (FSMs)** | **Behavior Trees (BTs)** |
| **Pros** | Easy to implement | Modular: week dependence between subtrees |
| Function call analogy  (Instead of GOTO analogy) |
| Easy to understand |
| Generalizes architectures: *Decision trees, Subsumption architecture, Teleo-reactive approach* |
| **Cons** | Does not scale well (n² possible transitions) | Harder to implement |
| GOTO analogy problem (harms modularity) |
| State feedback is optimal  *> Next action should only depend on current world state, not on current action* |
| Harder to understand |
| **Verdict** | => Good for smaller problems | => Good for larger problems |

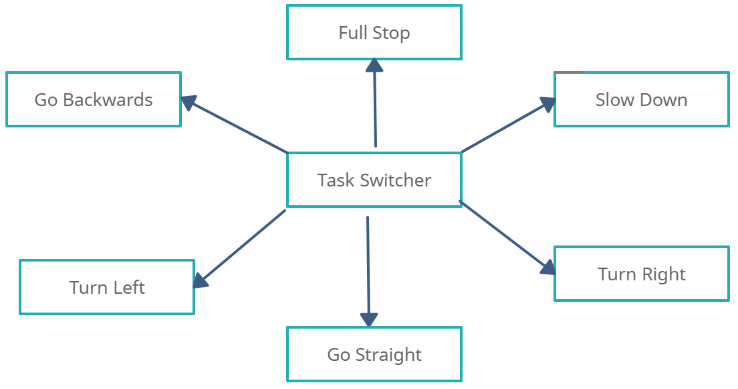
Behavior trees work by having a parent that “ticks” an action and then the action responds with Success, Failure, and Running. Thus, each action only needs to know if it succeeded, failed, or if its still trying. This creates a much more modular and weak dependence between subtrees. You don’t need to know what actions to do next. This ticking and returning state are like a function call, instead of a GOTO statement.

Other advantages of BTs include:

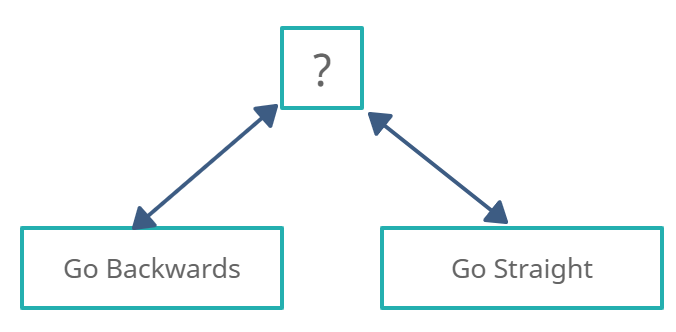
*Maintainability*: transitions in BT are defined by the structure, not by conditions inside the states. Because of this, nodes can be designed independent from each other, thus, when adding or removing new nodes (or even subtrees) in a small part of the tree, it is not necessary to change other parts of the model.

*Scalability*: when a BT have many nodes, it can be decomposed into small sub-trees saving the readability of the graphical model.

*Reusability*: due to the independence of nodes in BT, the subtrees are also independent. This allows the reuse of nodes or subtrees among other trees or projects.

The whole structure of a BT can be seen as a central task switcher, with all the actions at the leaves of the behavior tree, and interior nodes of a tree as a task switcher, deciding what to do next. And this task switcher is only depending on the world state, and not on an interior state of the tree.

There are 2 fundamental compositions of actions that help to answer the question “What to do next?”:

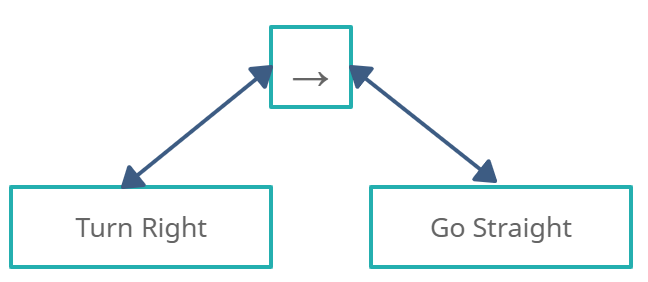
*Fallback:* Denoted by a question mark (?) and can be thought of as an OR function.

For example, let us consider the action of a simple parking scenario:

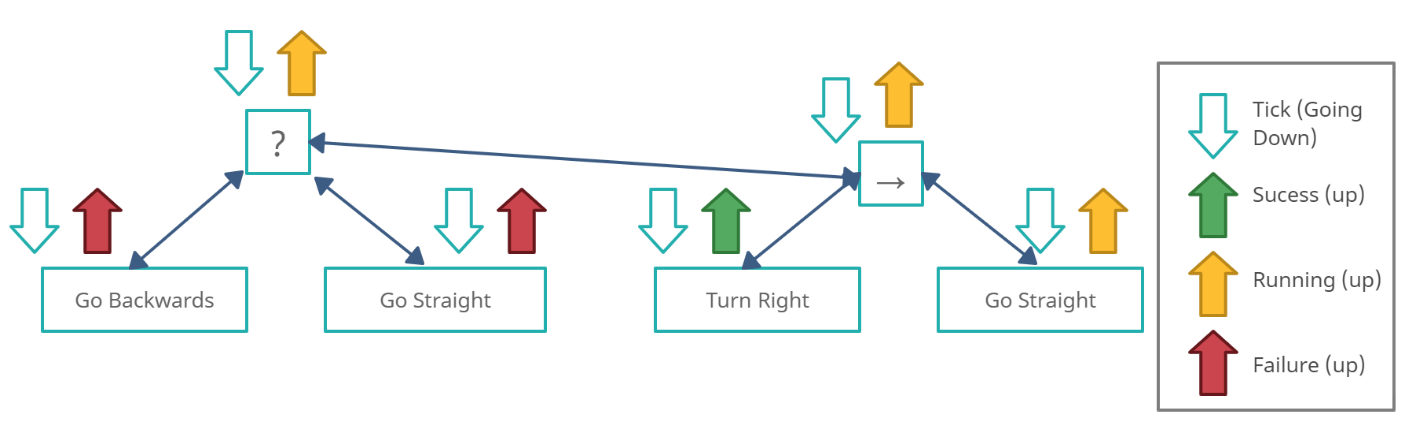
(Go Backwards ? Go Straight)

The rule is that we try to park by going backwards, and if we fail, then we try going straight (to adjust). But if “Go Backwards” succeeds, then we are done parking.

*Sequence:* Denoted by an arrow → and be thought of as an AND function.

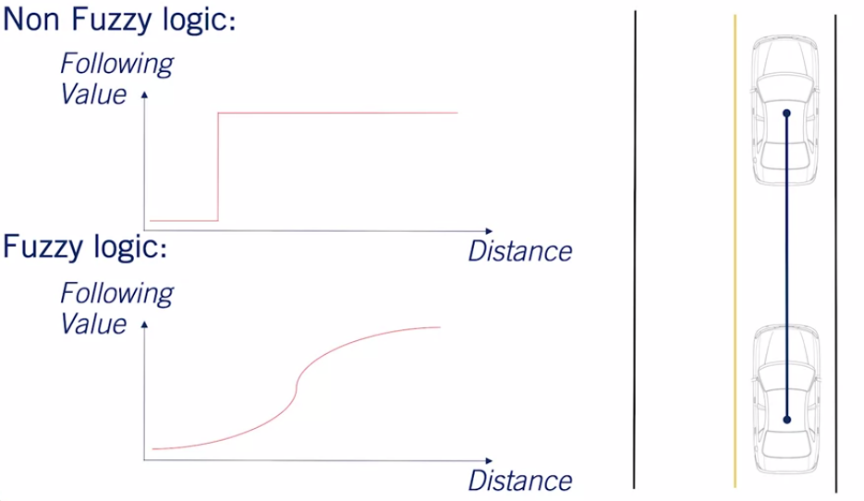
For example, let us consider the action of changing lanes to the right. We first tick the “Turn Right” action, then if it succeeds, then we tick the “Go Straight” action.

We can also connect these subtrees into a small behavior tree.

Behavior trees are run by a tick (Going down) that has a frequency dependent on the dynamics of the system.

Let us imagine a scenario where we tick the root (?), then tick “Go Backwards”, and this action fails, so the fallback takes the next child of the root which is “Go Straight”, imagine this returns Failure. So, the fallback takes the next child which is a sequence node, which itself takes its first child “Turn Right”, and if it succeeds, the sequence ticks “Go Straight”, and if this node returns Running, then this state goes all the way up to the OR root.

* Fuzzy Logic

Fuzzy logic is a system by which a set of crisp, well-defined values are used to create a more continuous set of fuzzy states. For a simple example of how a Fuzzy based system works, let's take a look at the example of the vehicle following behavior.

Usually, we set a parametrized distance which divided the space into follow the vehicle or do not follow the vehicle. With a Fuzzy system we're able to have a continuous space over which different rules can be applied. For example, a Fuzzy system might react strongly to a lead vehicle when very close to it. But be less concerned with the speed matching or distance following requirements when it's further away.

While Fuzzy based rule systems are able to deal with the environmental noise of a system to a greater degree than traditional discreet systems, both rule-explosion and hyperparameter tuning remain issues with Fuzzy systems. In fact, Fuzzy systems can result in rule-explosion to an even greater degree, as even more logic is required to handle the fuzzy set of inputs. Accompanying the rule-explosion is a large challenge in hyperparameter tuning as well.

* Reinforcement Learning

Reinforcement learning is a form of machine learning in which an agent learns how to interact with a given environment by taking action and receiving a continuous reward. By learning in a simulated environment, no real-world repercussions occurred during the many failures experienced during learning.

Because of the extremely large variety of scenarios and inputs that an autonomous vehicle can encounter, direct reinforcement learning for behavior planning is unlikely to succeed. Instead, some further adaptations are usually applied:

*Hierarchical reinforcement learning:* Here we divide the problem into low level policies in the maneuver space and high-level policies in the scenarios. This is similar to the hierarchical finite state machine. Then, each low-level policy is learned independently and only once successfully learned a high level policy can be learned to complete a scenario.

*Model-based reinforcement learning:* Not only is the agent attempting to learn a policy, but also a model of the current environment around the agent. An example of how this approach would be to include a model of the movement of dynamic objects. If the agent understands the movement patterns of the dynamic object it can create more effective plans through the environment.

While reinforcement learning is a very exciting and highly promising area of research it too is not without its own limitations. Many simulation environments used to learn the policies required for autonomous driving are overly simplified. And due to their simplicity, the policies learned may not be transferable to real world environments. Overly realistic simulators lead to the issue of severe computational requirements. Especially when running thousands of repetitions of widely varying scenarios for self-driving learning.

The second is an issue concerning safety. While there are techniques within reinforcement learning that attempt to ensure safety constraints along the trajectories created by the reinforcement learner, there is still no way to perform rigorous safety assessment of a learned system, as they are mostly black boxes in terms of the way in which decisions are made.

*Inverse reinforcement learning:* Rather than trying to obtain a policy given a reward function, the approach is to use human driving data as the policy and attempt to learn the reward function used by humans. Then the algorithm can execute driving maneuvers similarly to a human driver.

*End-to-end approaches:* These take as an input raw sensor data and attempt to output throttle, break, and steering commands. By learning, once again, from human driving commands in an imitation learning approach. This approach was pioneered by researchers at NVIDIA. While this is not explicitly classified as behavior planning, the end-to-end approach still implicitly performs the task of behavior selection as part of its output selection process.

While these reinforcement learning methods might produce outstanding behavioral planners, in our case, the competition only includes a fully-autonomous round. So we’re not going to pursue Reinforcement Learning for now. However, we plan on making our vehicle Mars-ready after the competition ends.

**ii. Implementation**

1. **Competition State Machine States**
2. **TUNMSR Behavioral Planner State Diagram**
3. **Path Generation: Conformal Lattice Planner**

**III.3.2.3) Velocity Planner**

1. **Evaluating our options**
2. **Velocity Generation: Trapezoidal Profile**

**Originality Aspects:**

**References:**

Yuanfei Lin, Sebastian Maierhofer, & Matthias Althoff, “Sampling-Based Trajectory Repairing for Autonomous Vehicles”

* RRT-A\* Motion Planning Algorithm for Non-holonomic Mobile Robot, Jiadong Li1,2, Shirong Liu1 \*, Botao Zhang1 , Xiaodan Zhao, School of Automation, Hangzhou Dianzi University, Hangzhou, Zhejiang 310018, China