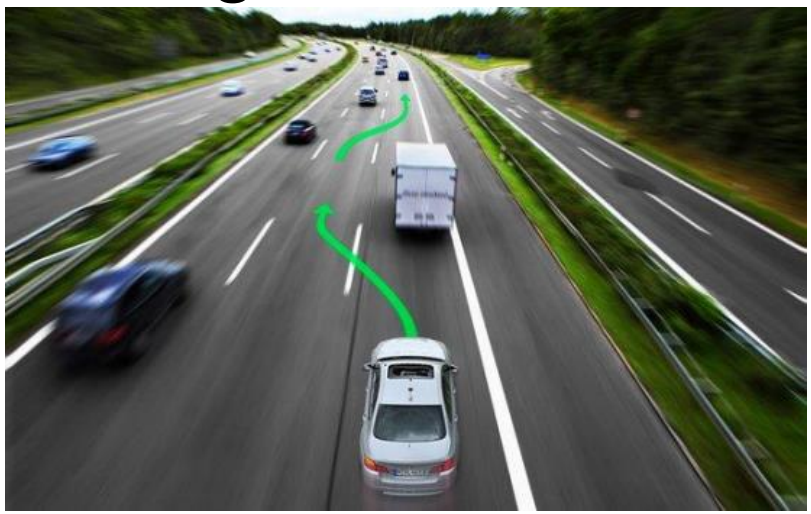
L5 Planning

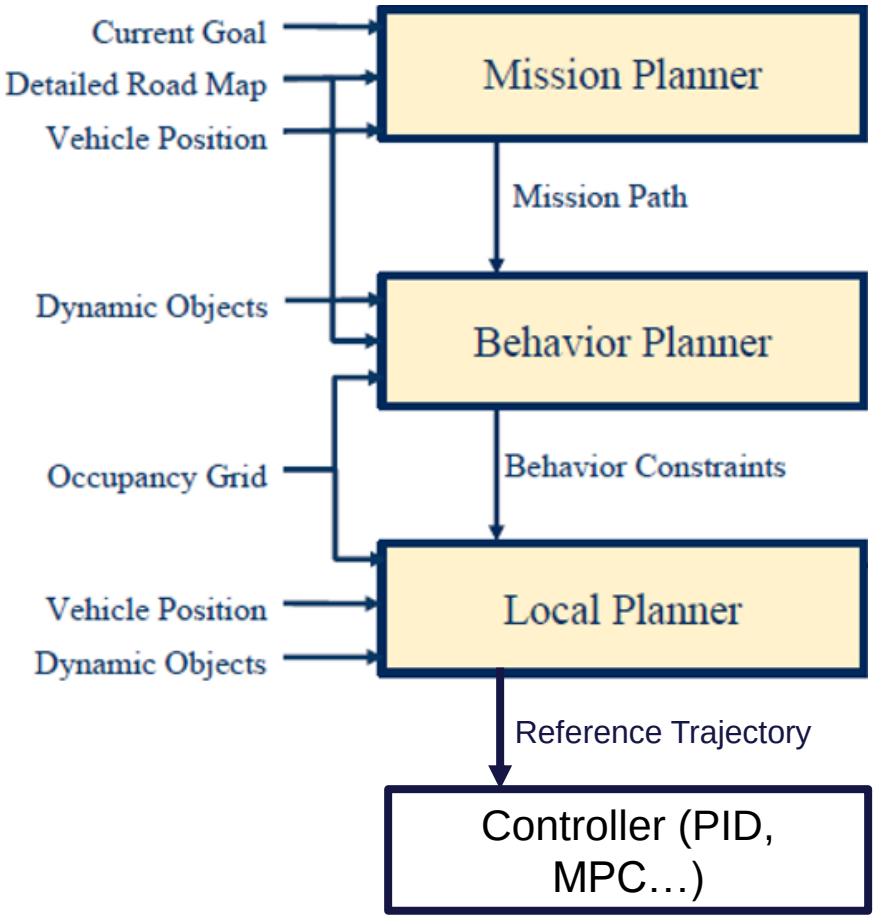
Zonghua Gu 2021



Acknowledgement: slides partly based on https://www.coursera.org/learn/motion-planning-self-driving-cars

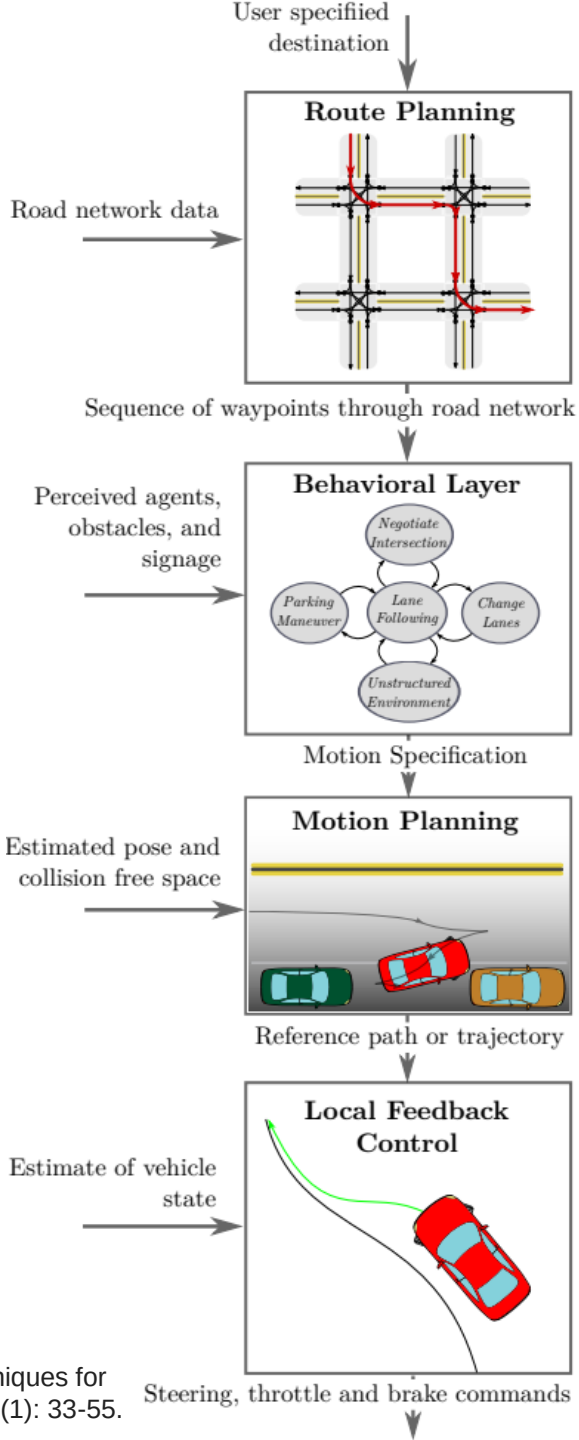
A Hierarchy of Planners

- Mission Planning: map-level navigation
- Behavior Planning: choosing a behavior based on environmental conditions
- Motion Planning: includes path planning and speed profile generation, generates a reference trajectory to be tracked by controller



Outline

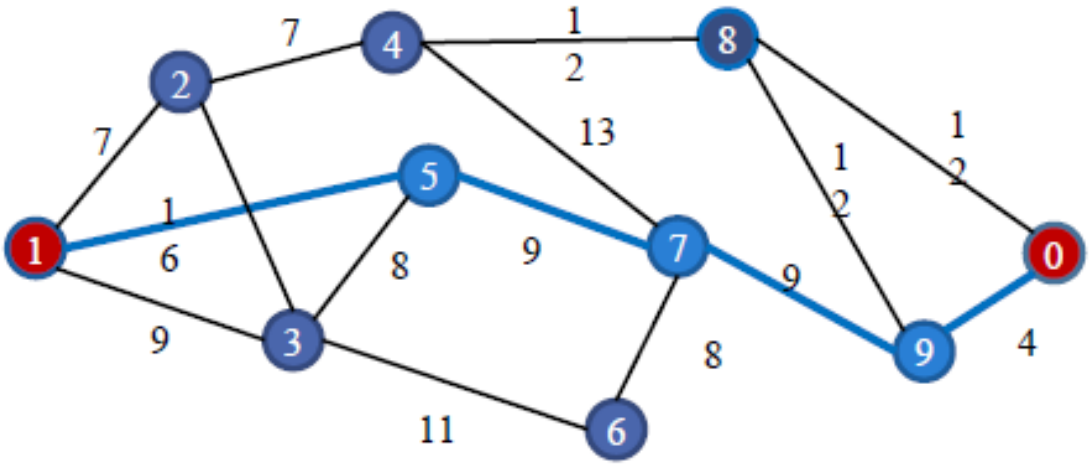
- Route planning
- Behavior planning
- Motion Planning
- Responsibility-Sensitive Safety (RSS)



Paden B, Čáp M, Yong S Z, et al. A survey of motion planning and control techniques for self-driving urban vehicles[J]. IEEE Transactions on intelligent vehicles, 2016, 1(1): 33-55.

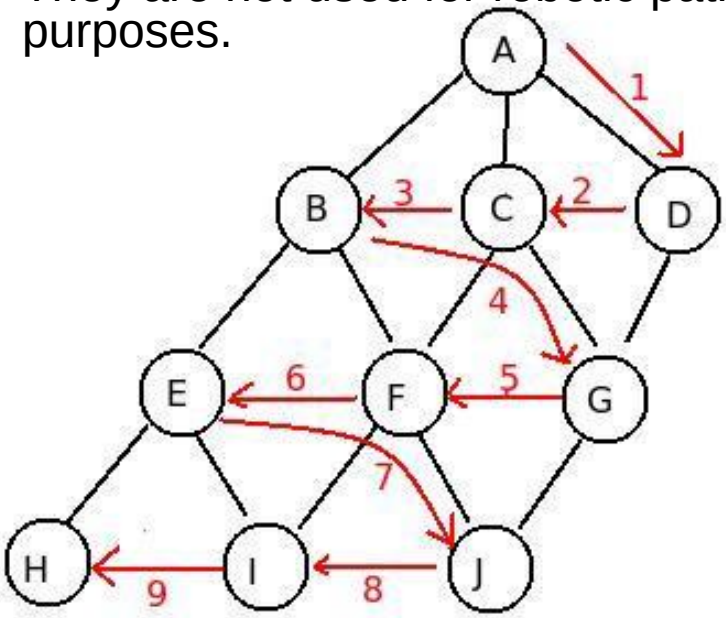
Route Planning

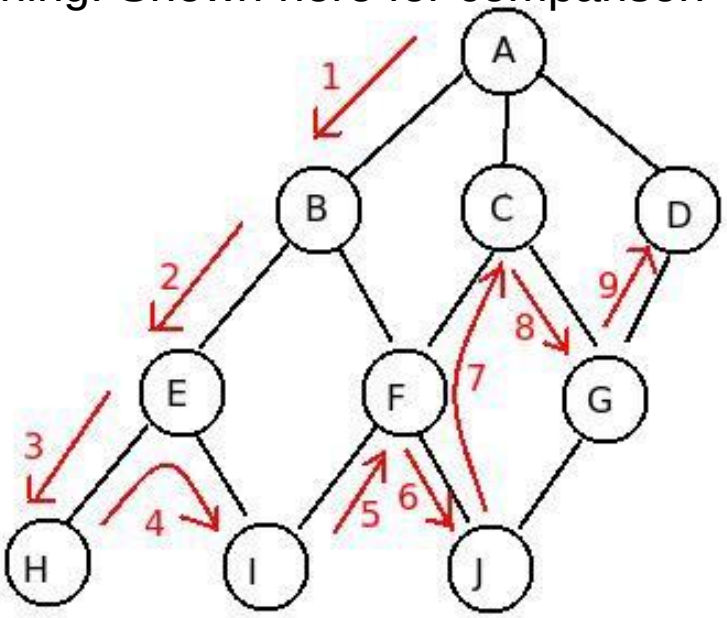
- Find the optimal path for navigation from Start to Goal on a weighted graph
- Graph search algorithms
  - Dijkstra's, A*,…



BFS and DFS

- Breadth-First Search (BFS): at each node, expand all neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. (Left figure)
- Depth-First Search (DFS): at each node, expand as far as possible along each branch before backtracking. (Right figure)
- BFS and DFS are inefficient since they do not consider edge costs for selecting the next node to expand. BFS is better than DFS for path planning:
  - BFS can finish after finding a path to the goal with total cost ≤ cost of all other partial paths.
  - DFS must expand all nodes of the graph.
- They are not used for robotic path planning. Shown here for comparison purposes.

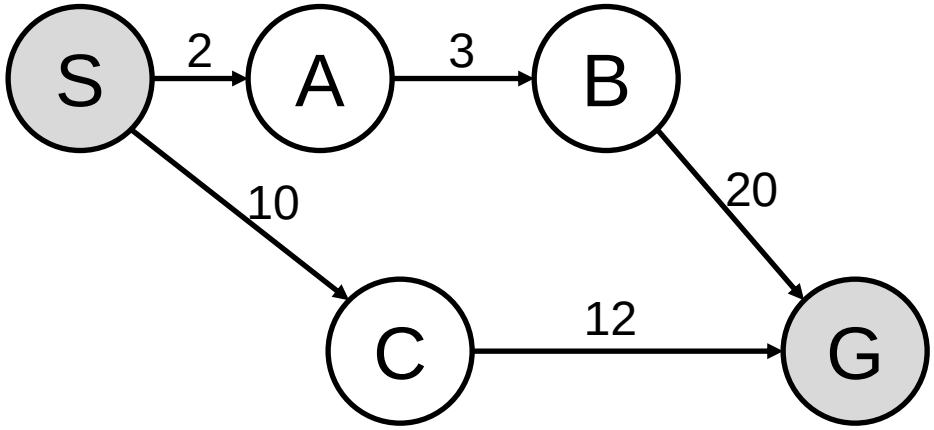




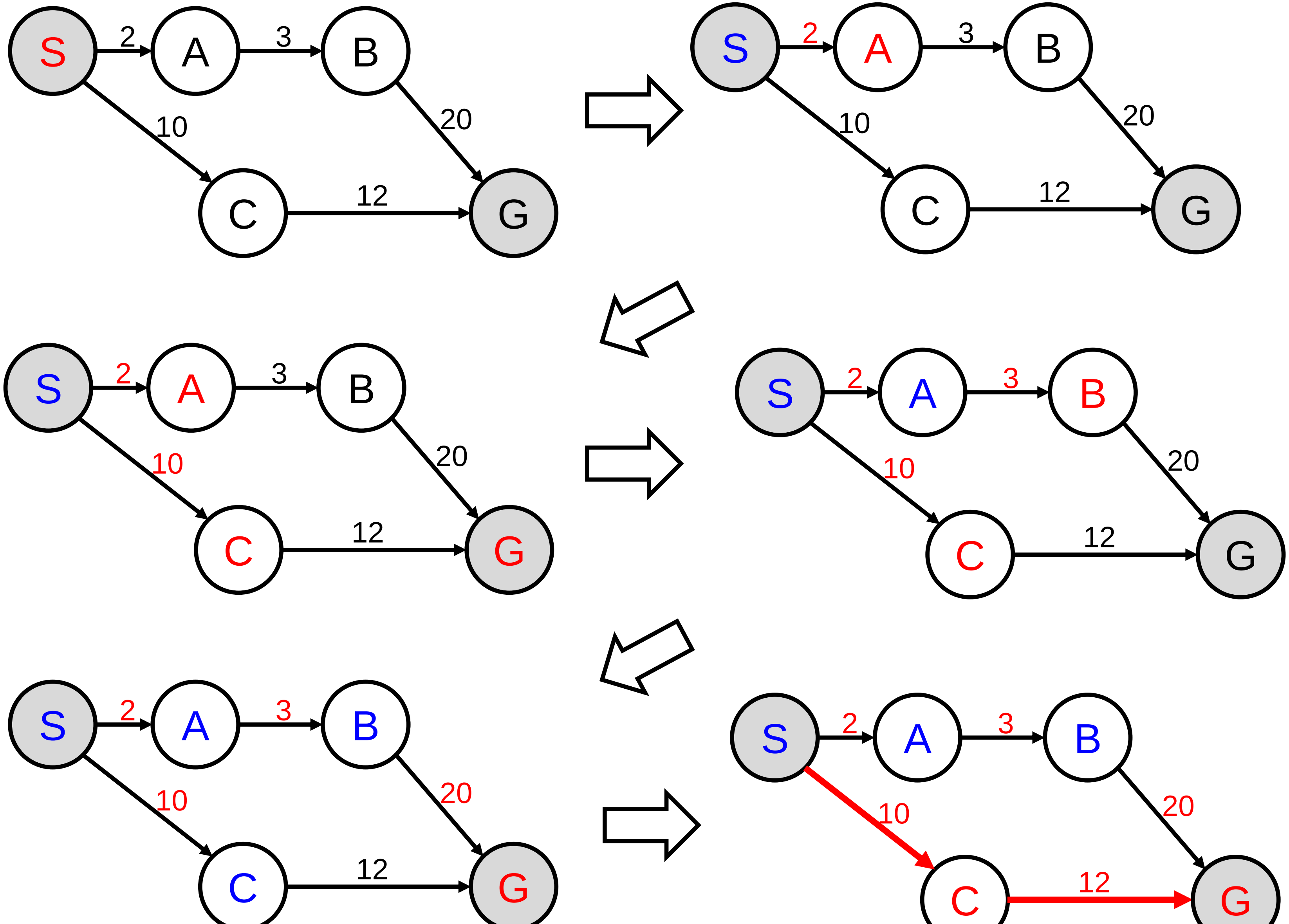
https://medium.com/@nihatkoc/dfs-vs-bfs-6b632c381ed6

Running Example

- Shortest path is $S \rightarrow C \rightarrow E$ with length $10 + 12 = 22$

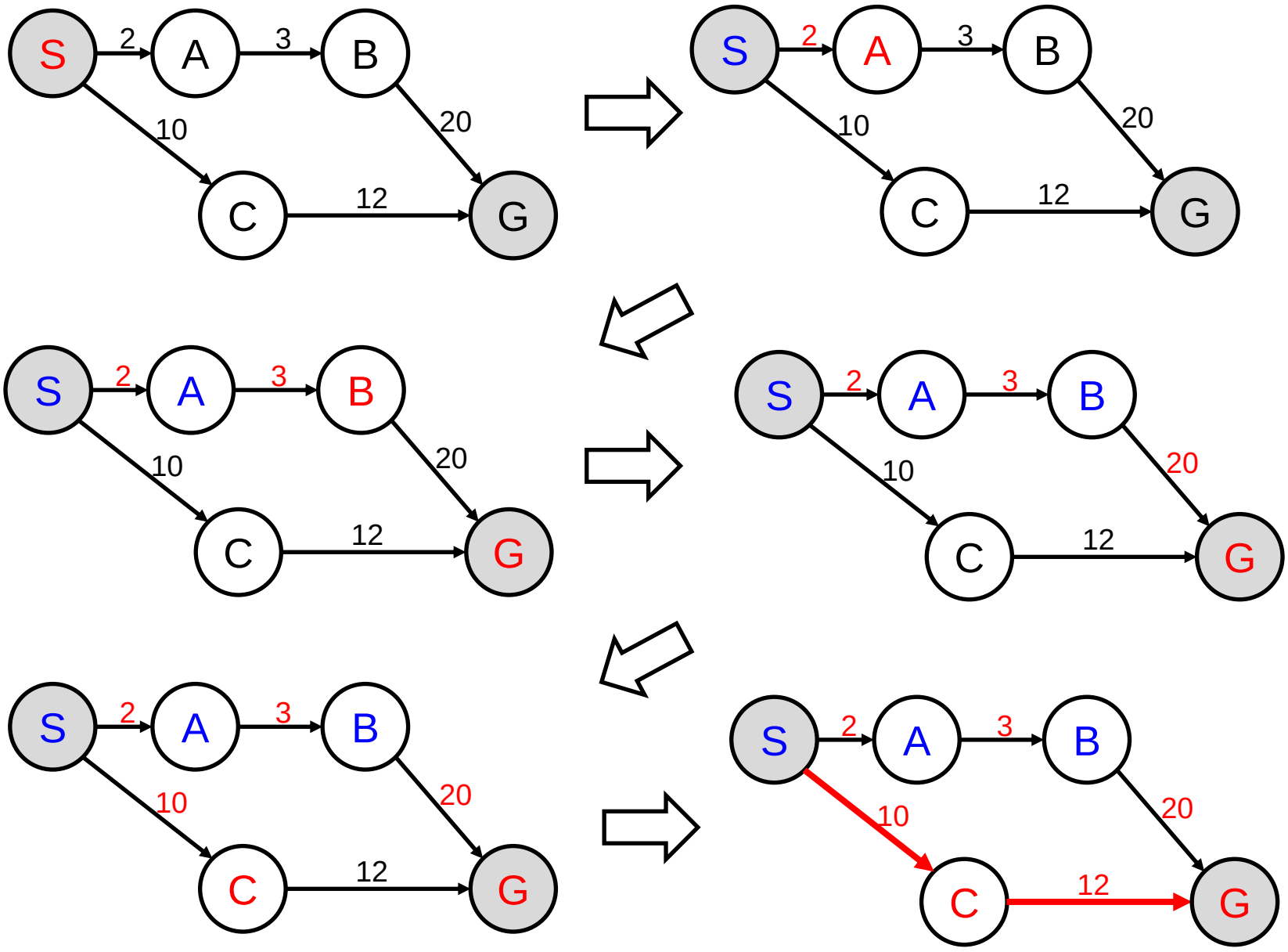


BFS



Cannot finish here, must expand node C, since total cost $25 > 10$, cost of the other partial path through C.

DFS



Must always expand all nodes.

Dijkstra's Algorithm

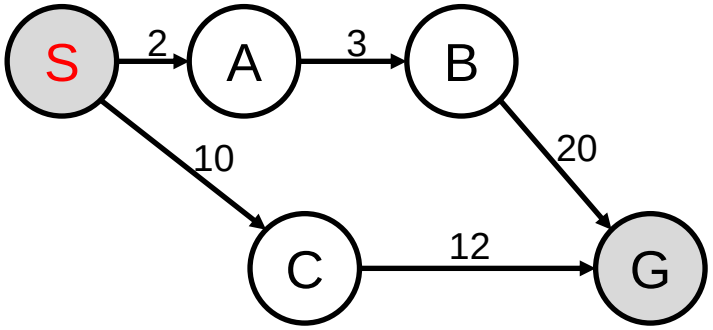
Algorithm Dijkstra's(G,s,t)

```
1.  open ← MinHeap()
2.  closed ← Set()
3.  predecessors ← Dict()
4.  open.push(s, 0)
5.  while !open.isEmpty() do
6.    u, uCost ← open.pop()
7.    if isGoal(u) then
8.      return extractPath(u, predecessors)
9.    for all v ∈ u.successors()
10.     if v ∈ closed then
11.       continue
12.     uvCost ← edgeCost(G, u, v)
13.     if v ∈ open then
14.       if uCost + uvCost < open[v] then
15.         open[v] ← uCost + uvCost
16.         predecessors[v] ← u
17.     else
18.       open.push(v, uCost + uvCost)
19.       predecessors[v] ← u
20.   closed.add(u)
```

- open[v] is sorted and popped in decreasing order of $uCost + uvCost$, cost of partial path to the source node s.

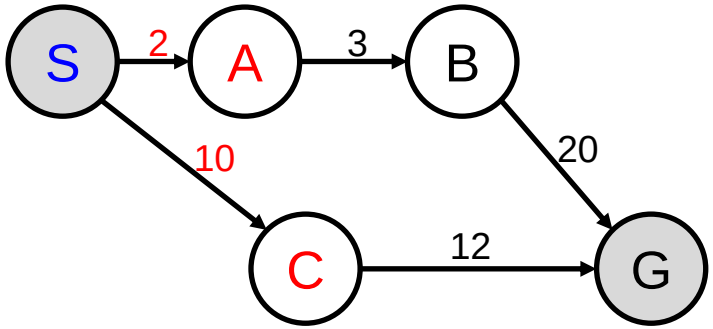
Dijkstra Example

Node	Cost to S
S	0



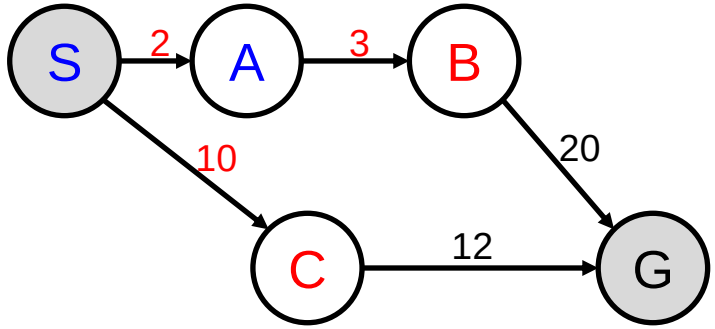
Node	Cost to S
A	2
C	10

Closed set: S



Node	Cost to S
B	5
C	10

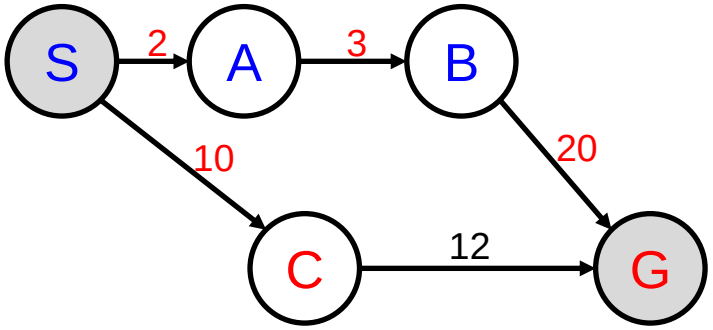
Closed set: S, A



Dijkstra Example cont'

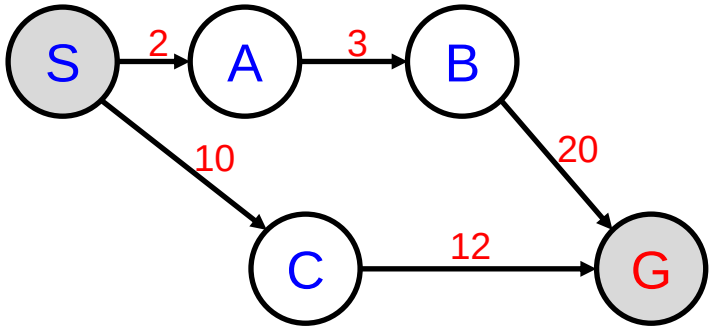
Node	Cost to S
C	10
G	25

Closed set: S, A, B

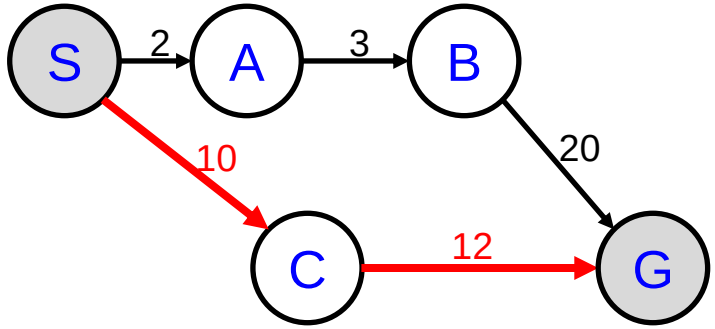


Node	Cost to S
G	22

Closed set: S, A, B, C



extractPath() yields the shortest path $S \rightarrow C \rightarrow G$ with length $10 + 12 = 22$



A* Algorithm (Compare to Dijkstra)

Algorithm Dijkstra's(G,s,t)

```
1.  open ← MinHeap()
2.  closed ← Set()
3.  predecessors ← Dict()
4.  open.push(s, 0)
5.  while !open.isEmpty() do
6.      u, uCost ← open.pop()
7.      if isGoal(u) then
8.          return extractPath(u, predecessors)
9.      for all v ∈ u.successors()
10.         if v ∈ closed then
11.             continue
12.         uvCost ← edgeCost(G, u, v)
13.         if v ∈ open then
14.             if uCost + uvCost < open[v] then
15.                 open[v] ← uCost + uvCost
16.                 predecessors[v] ← u
17.         else
18.             open.push(v, uCost + uvCost)
19.             predecessors[v] ← u
20.     closed.add(u)
```

open[v] is sorted and popped in decreasing order of $uCost + uvCost$, cost of partial path to the source node s.

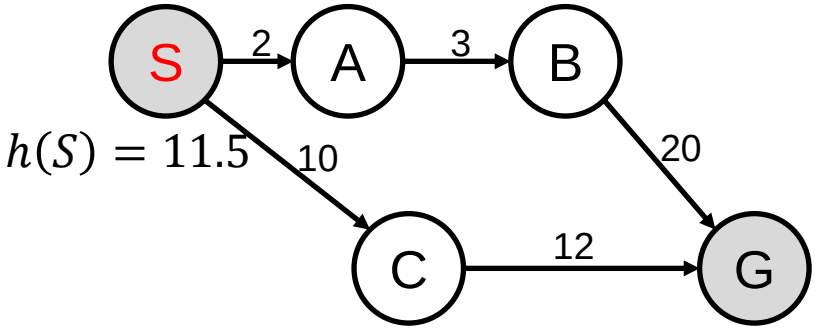
Algorithm A*(G,s,t)

```
1.  open ← MinHeap()
2.  closed ← Set()
3.  predecessors ← Dict()
4.  open.push(s, 0)
5.  while !open.isEmpty() do
6.      u, uCost ← open.pop()
7.      if isGoal(u) then
8.          return extractPath(u, predecessors)
9.      for all v ∈ u.successors()
10.         if v ∈ closed then
11.             continue
12.         uvCost ← edgeCost(G, u, v)
13.         if v ∈ open then
14.             if uCost + uvCost + h(v) < open[v] then
15.                 open[v] ← uCost + uvCost + h(v)
16.                 costs[v] ← uCost + uvCost
17.                 predecessors[v] ← u
18.         else
19.             open.push(v, uCost + uvCost)
20.             costs[v] ← uCost + uvCost
21.             predecessors[v] ← u
22.     closed.add(u)
```

open[v] is sorted and popped in decreasing order of $uCost + uvCost + h(v)$, cost of partial path to the source node s plus estimated cost-to-go $h(v) \leq$ actual cost-to-go (called an admissible heuristic). A* with $h(v) = 0$ becomes Dijkstra's algo.

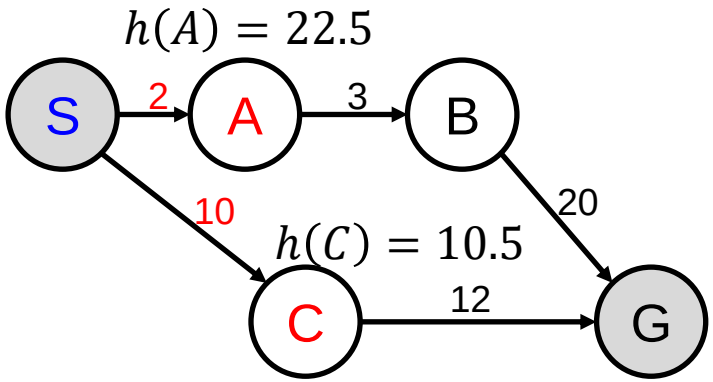
A* Example

Node	Cost to S	Est. cost-to-go $h(v)$	Est. total cost
S	0	11.5 (<22 actual)	11.5



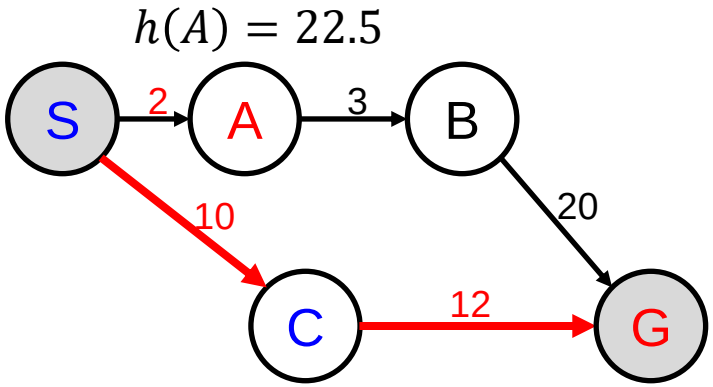
Node	Cost to S	Est. cost-to-go $h(v)$	Est. total cost
C	10	10.5 (<12 actual)	20.5
A	2	22.5 (<23 actual)	24.5

Closed set: S



Node	Cost to S	Est. cost-to-go $h(v)$	Est. total cost
E	22	0 (exact)	22
A	2	22.5 (<23 actual)	24.5

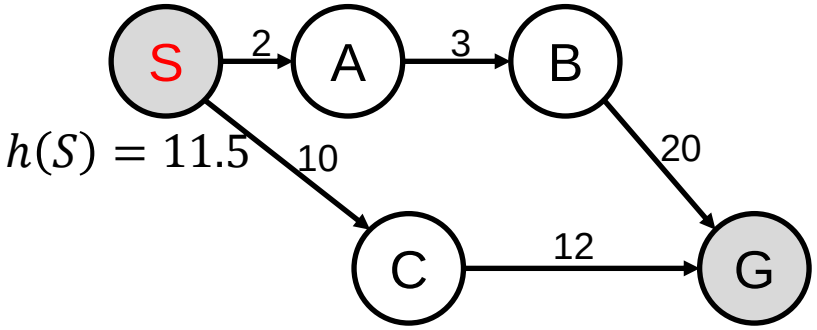
Closed set: S, C



extractPath() yields the shortest path $S \rightarrow C \rightarrow G$ with length $10 + 12 = 22$

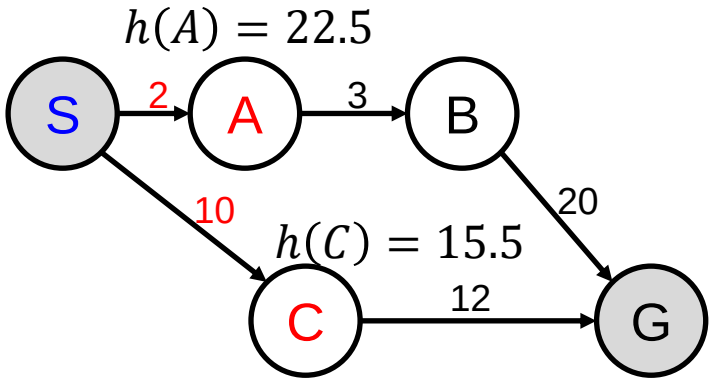
A* Example w Inadmissible Heuristic

Node	Cost to S	Est. cost-to-go $h(v)$	Est. total cost
S	0	11.5 (<22 actual)	11.5



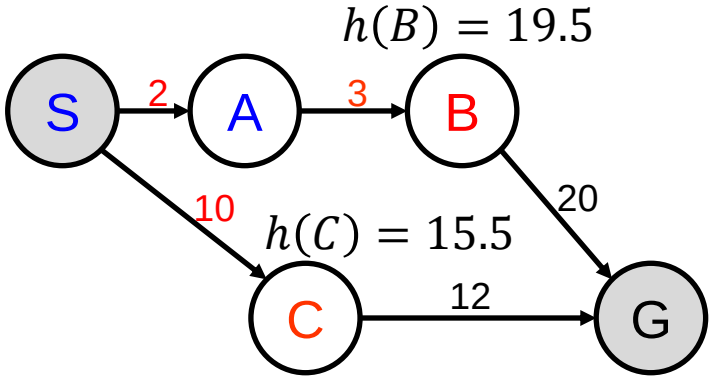
Node	Cost to S	Est. cost-to-go $h(v)$	Est. total cost
C	10	15.5 (>12 actual)	25.5
A	2	22.5 (<23 actual)	24.5

Closed set: S



Node	Cost to S	Est. cost-to-go $h(v)$	Est. total cost
B	5	19.5	24.5
C	10	15.5 (>12 actual)	25.5

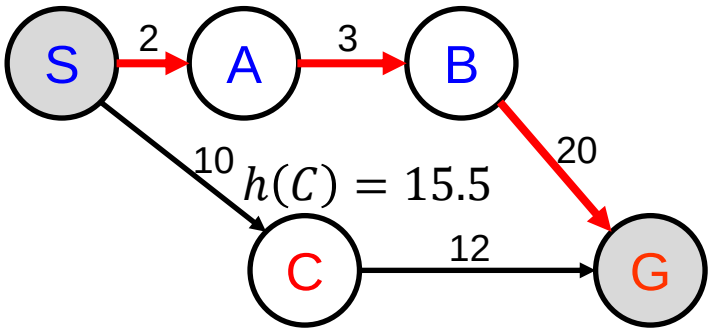
Closed set: S, C



A* Example w Inadmissible Heuristic

Node	Cost to S	Est. cost-to-go $h(v)$	Est. total cost
E	25	0 (exact)	25
C	10	15.5 (>12 actual)	25.5

Closed set: S, A, B



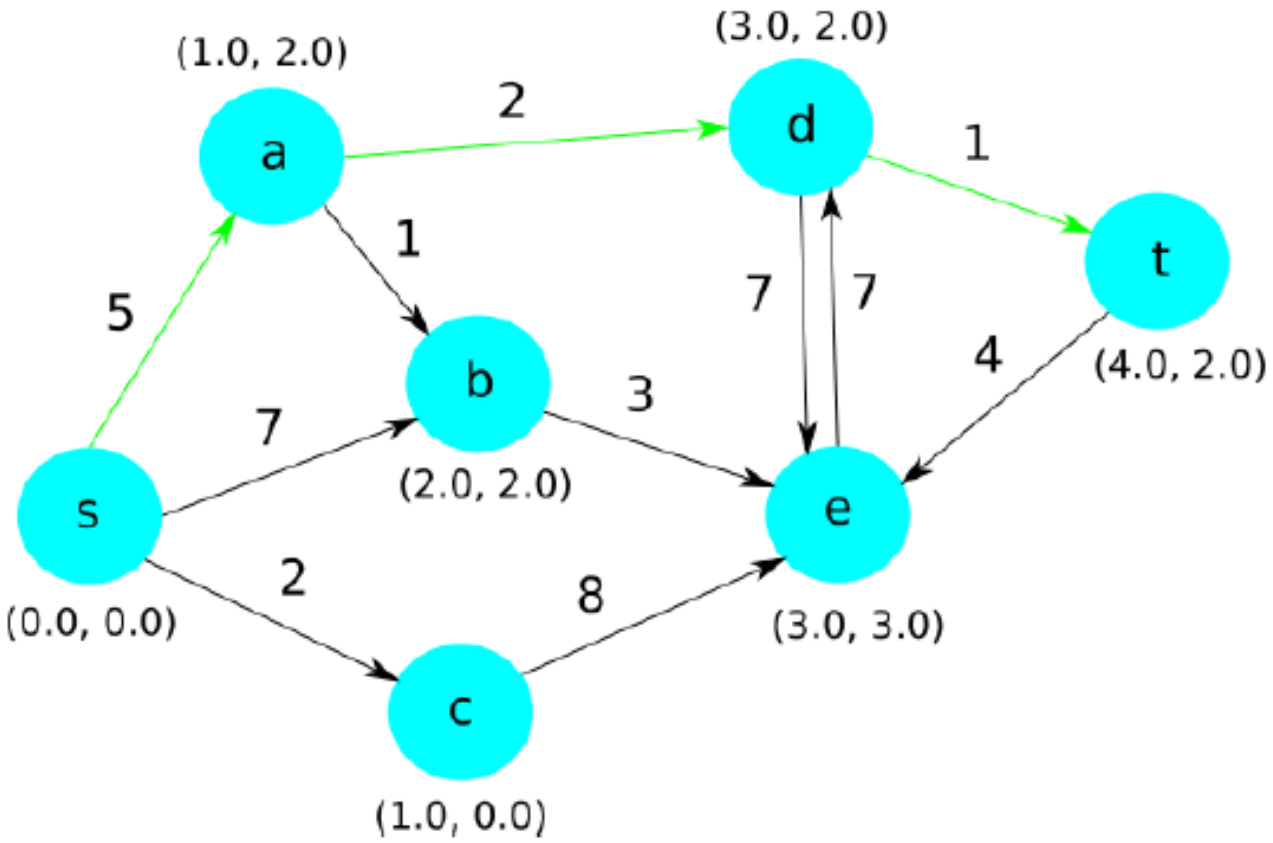
- extractPath() yields the shortest path to goal $S \rightarrow A \rightarrow B \rightarrow G$ with length $2 + 3 + 20 = 25$. It is incorrect, due to overestimation of the cost-to-go from node C $h(C) = 15.5 > 12$.
  - After finding the path $S \rightarrow A \rightarrow B \rightarrow G$ with length 25, we INCORRECTLY ignore the other path through C since its est. total cost $10 + h(C) = 25.5$ is more than 25.
- With an admissible heuristic like $h(C) = 10.5 < 12$.
  - We either directly find the correct shortest path $S \rightarrow C \rightarrow G$ with length $10 + 12 = 22$, and CORRECLY ignore the other path through A, B (as shown earlier),
  - Or we find the path $S \rightarrow A \rightarrow B \rightarrow G$ with length 25 (due to even stronger underestimation of $h(A)$ and $h(B)$), but then we must continue to expand node C to find the correct shortest path $S \rightarrow C \rightarrow G$, since its est. total cost $10 + h(C) = 20.5$ is less than 25, so there is a chance of shorter path through C.
- With an admissible heuristic $h(v) \leq$ actual cost-to-go:
  - If we find a path to goal with exact cost $V \leq$ estimated total cost of all other paths, we CAN finish and safely ignore all the other paths, since their actual cost must be higher than V.
- With an inadmissible heuristic $h(v) \geq$ or $\leq$ actual cost-to-go:
  - If we find a path to goal with exact cost $V \leq$ estimated total cost of all other paths, we CANNOT finish and safely ignore all the other paths, since their actual cost may be lower or higher than V.

How to Design an Admissible Heuristic?

- In the weighted graph representing the road network, edge cost may be travel distance, or travel time taking into account factors such as speed limit, traffic congestion level, etc.
  - If objective is to minimize total travel distance from start to goal, then edge cost is travel distance between two locations. We can set $h(v)$ to be the straight-line distance to goal.
  - If objective is to minimize total travel time from start to goal, then edge cost is travel time between two locations. We can set $h(v)$ to be the straight-line distance to goal divided by the maximum speed limit.

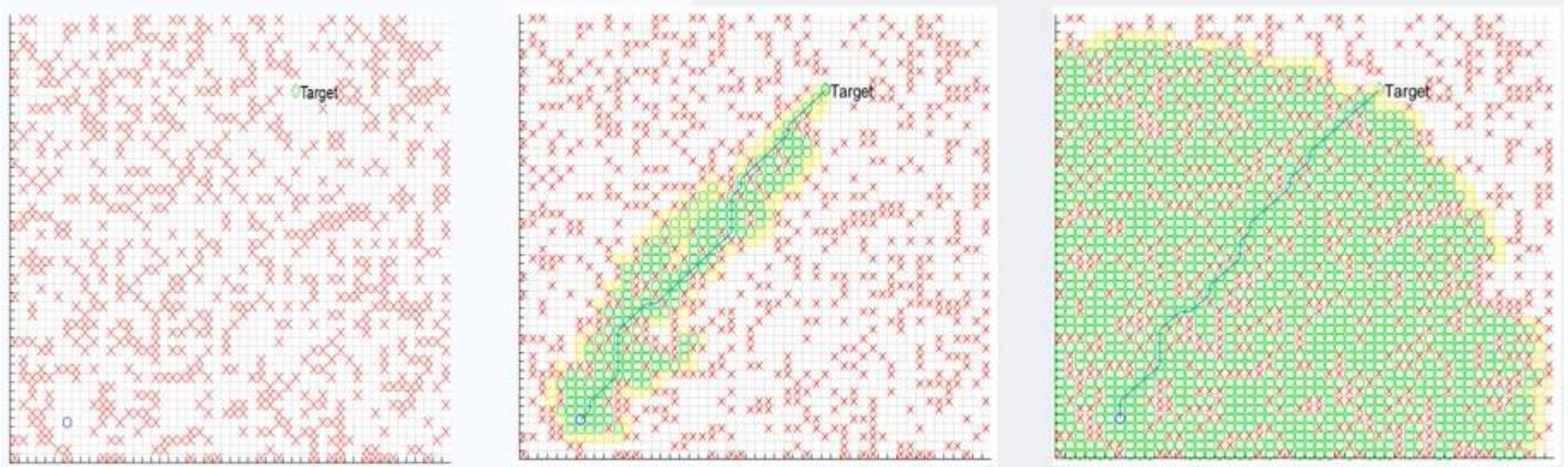
Exercise

- Consider the following graph. Number pairs in parenthesis denote Cartesian coordinates of the node. Find the shortest path (green) from s (start) to t (goal) with:
  - Dijkstra's algo
  - A* with heuristic function $h(v)$ as the straight-line distance to goal.



A* vs. Dijkstra

- Red crosses denote obstacles.
- Green circles denote explored positions.
- A* search is more efficient, as it is directed towards the target; Dijkstra's algorithm explores in every direction and less efficient.

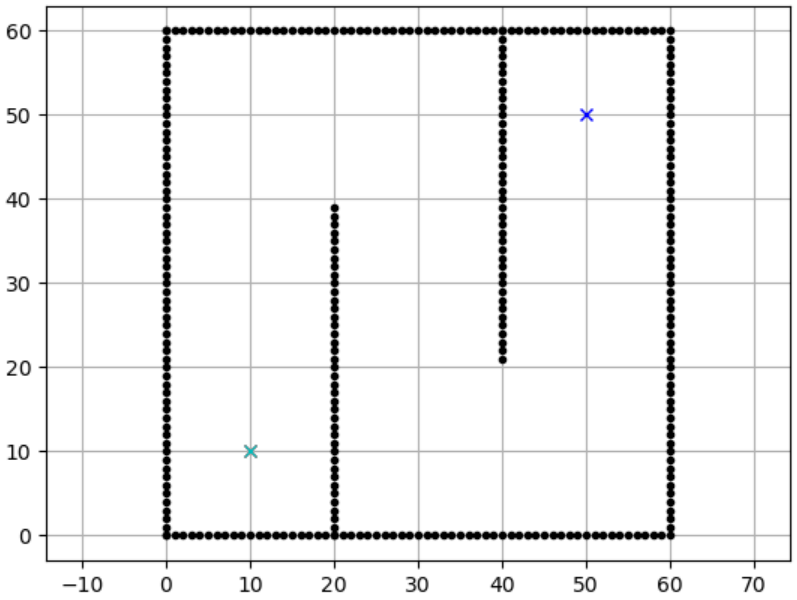


Grid Map

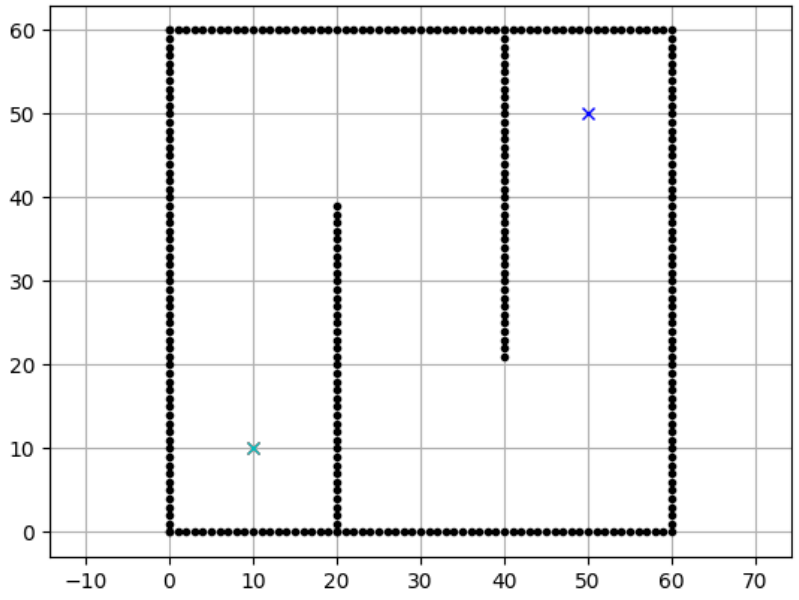
A* Algorithm

Dijkstra's Algorithm

A* vs. Dijkstra (Animation)



Dijkstra



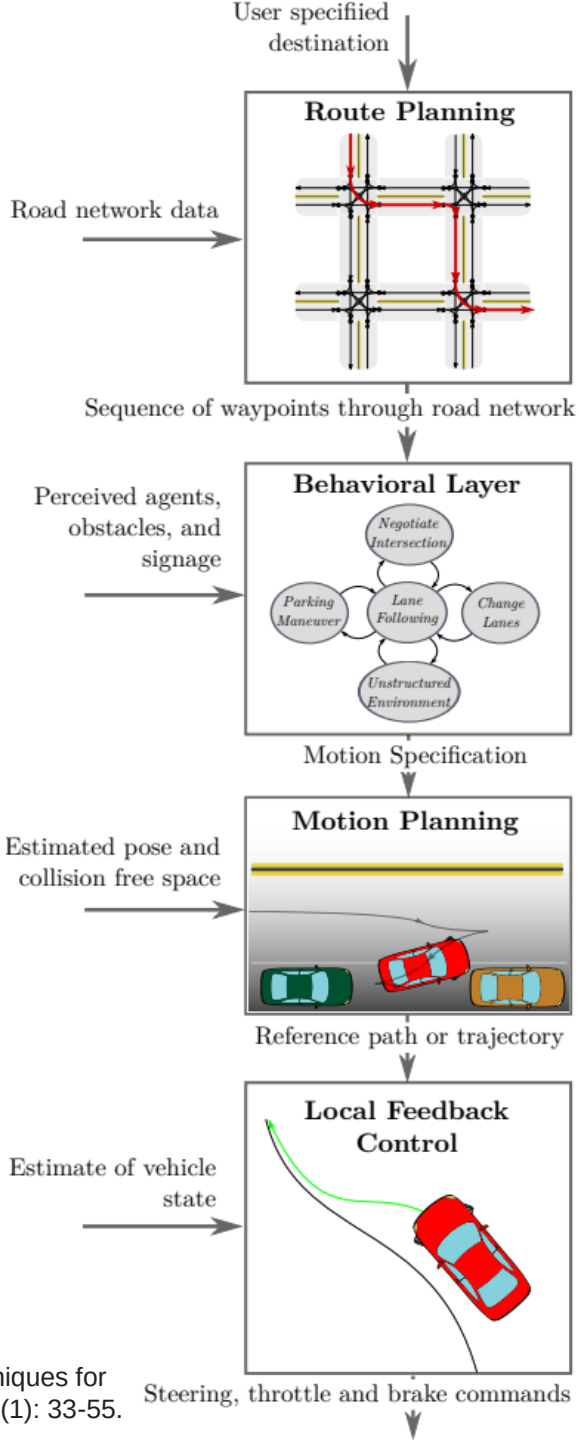
A*

More animations https://qiao.github.io/PathFinding.js/visual/

https://github.com/AtsushiSakai/PythonRobotics

Outline

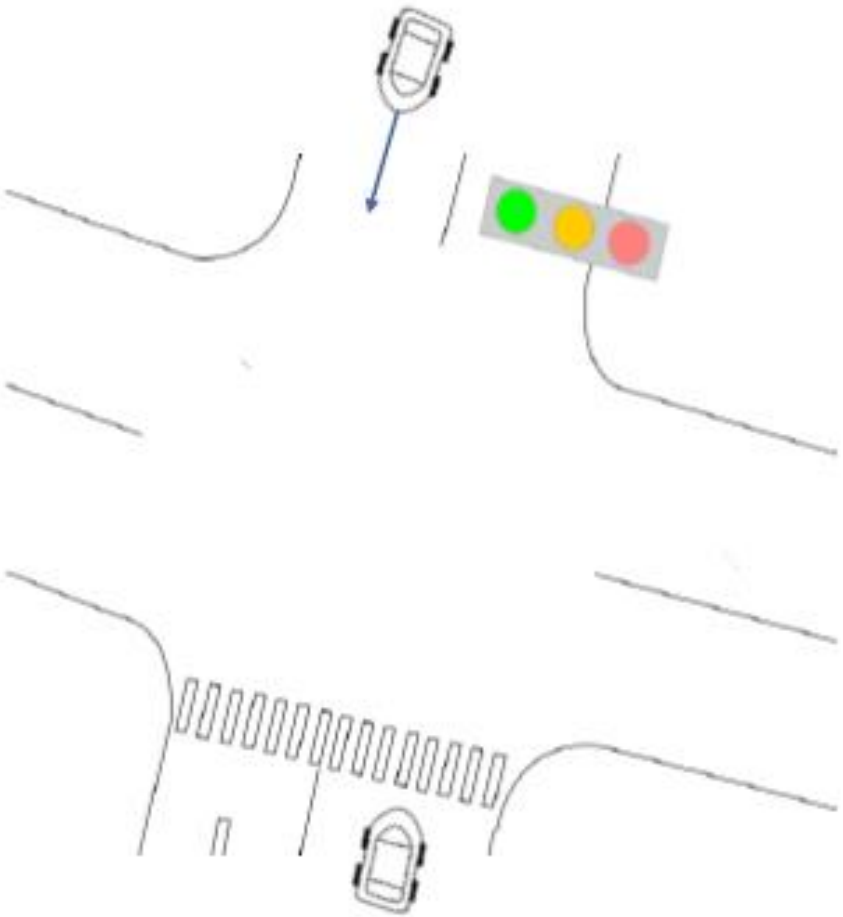
- Route planning
- Behavior planning
- Motion Planning
- Responsibility-Sensitive Safety (RSS)



Paden B, Čáp M, Yong S Z, et al. A survey of motion planning and control techniques for self-driving urban vehicles[J]. IEEE Transactions on intelligent vehicles, 2016, 1(1): 33-55.

Behavior Planner

- Plan the set of high-level driving actions, or maneuvers to safely achieve the driving mission under various driving situations based on:
  - Rules of the road (traffic lights, stop signs…)
  - Static objects (parked vehicles…)
  - Dynamic objects (moving vehicles, cyclists, pedestrians…). Need behavior prediction.

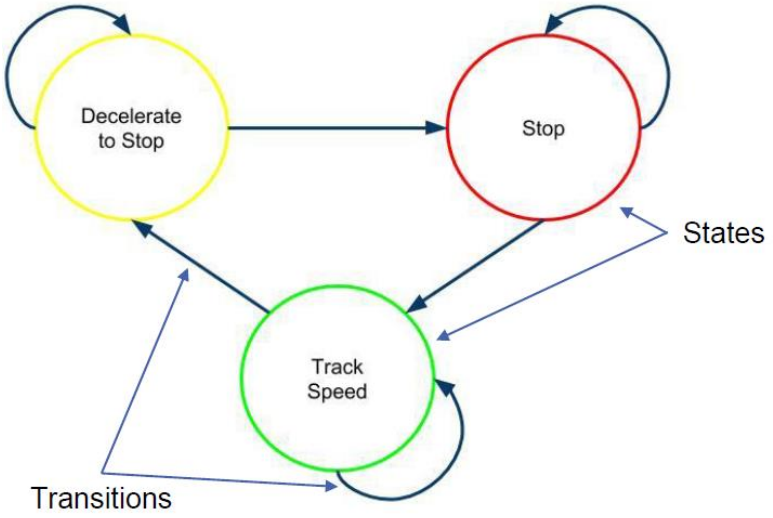


Driving Maneuvers

- Track speed
  - Maintain current speed consistent with other vehicles on the road.
- Follow leader
  - Match speed of the leading vehicle and maintain a safe distance.
- Decelerate to stop
  - Begin decelerating and stop before a given space.
- Stop
  - Remain stopped in the current position.
- Change lanes
- Turn
  - Left, right, U-turn…
- Pass
  - Complex maneuver involving changing lanes, passing, (optional) changing back to ego-lane.

Behavior Planning Approaches

- Finite State Machine-based
  - Transitions triggered by inputs
- Rule-based: Logical statements with priorities
- Reinforcement Learning-based
  - Learned from trial-and-error to maximize cumulative reward.



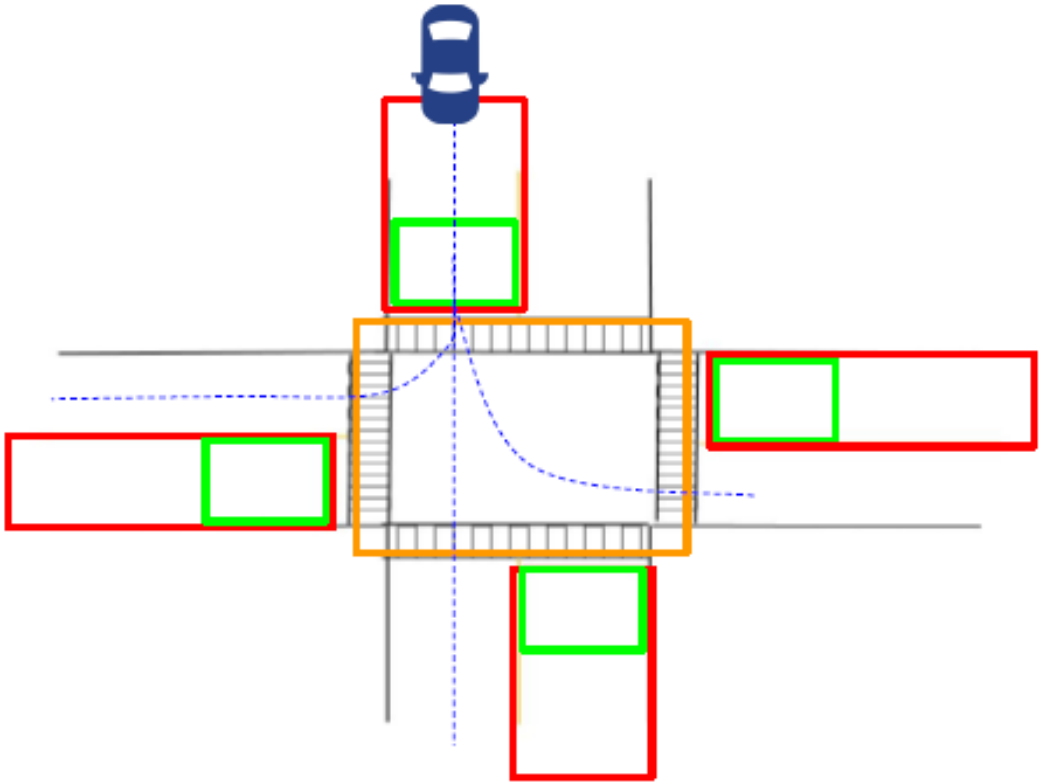
FSM-based example: traffic light changing color

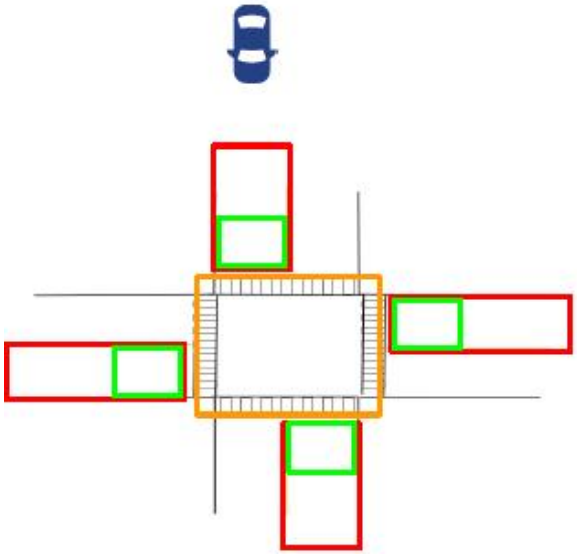
Rule1: green light at intersection → drive straight
Rule 2: pedestrian in intersection → stop

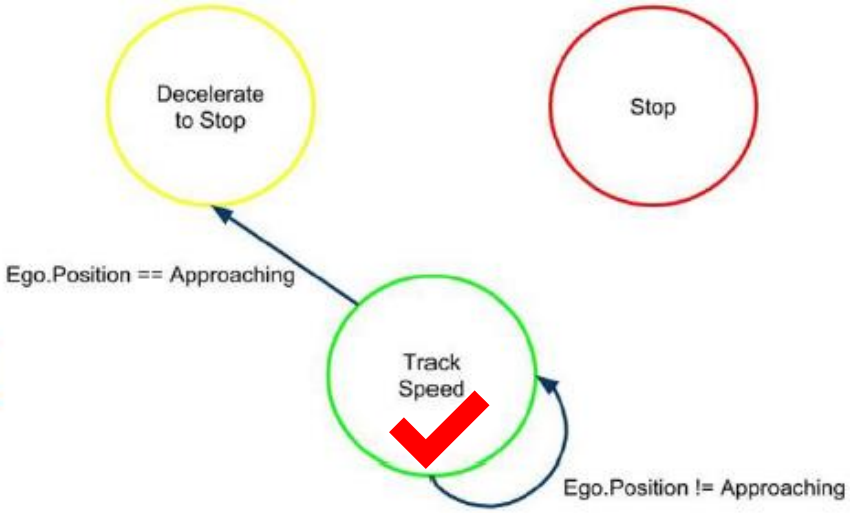
Rule-based planning example: Rule 2 has higher priority and overrides Rule 1

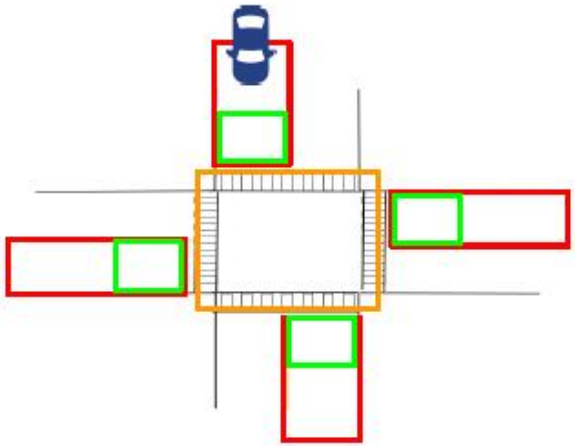
An Intersection Scenario without Dynamic Objects

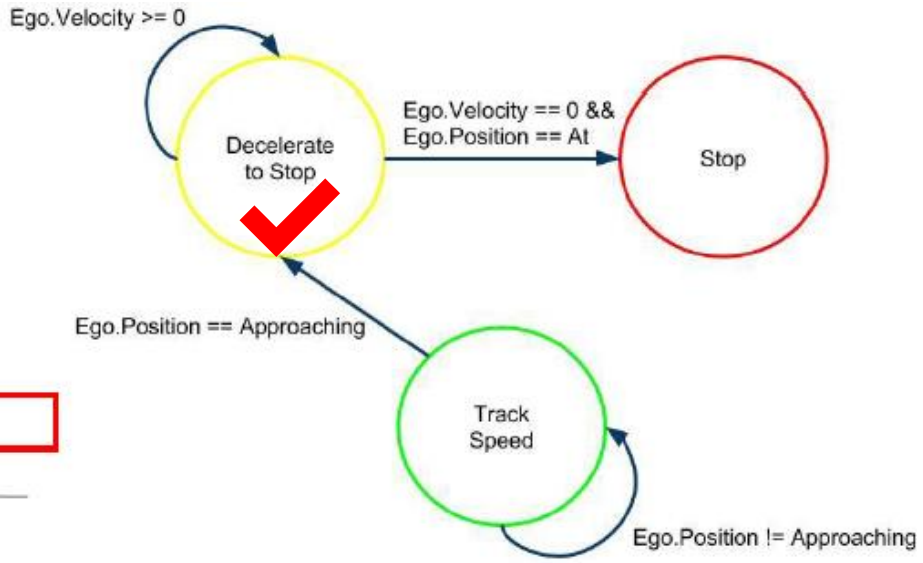
- 4-way intersection
- Two-lane road
- Stop sign in every direction
- No other dynamic objects
- Discretized into
  - Approaching zone (red box)
  - At zone (green box)
  - On zone (orange box)
- Size of each zone determined by
  - Ego vehicle speed
  - Size of the intersection

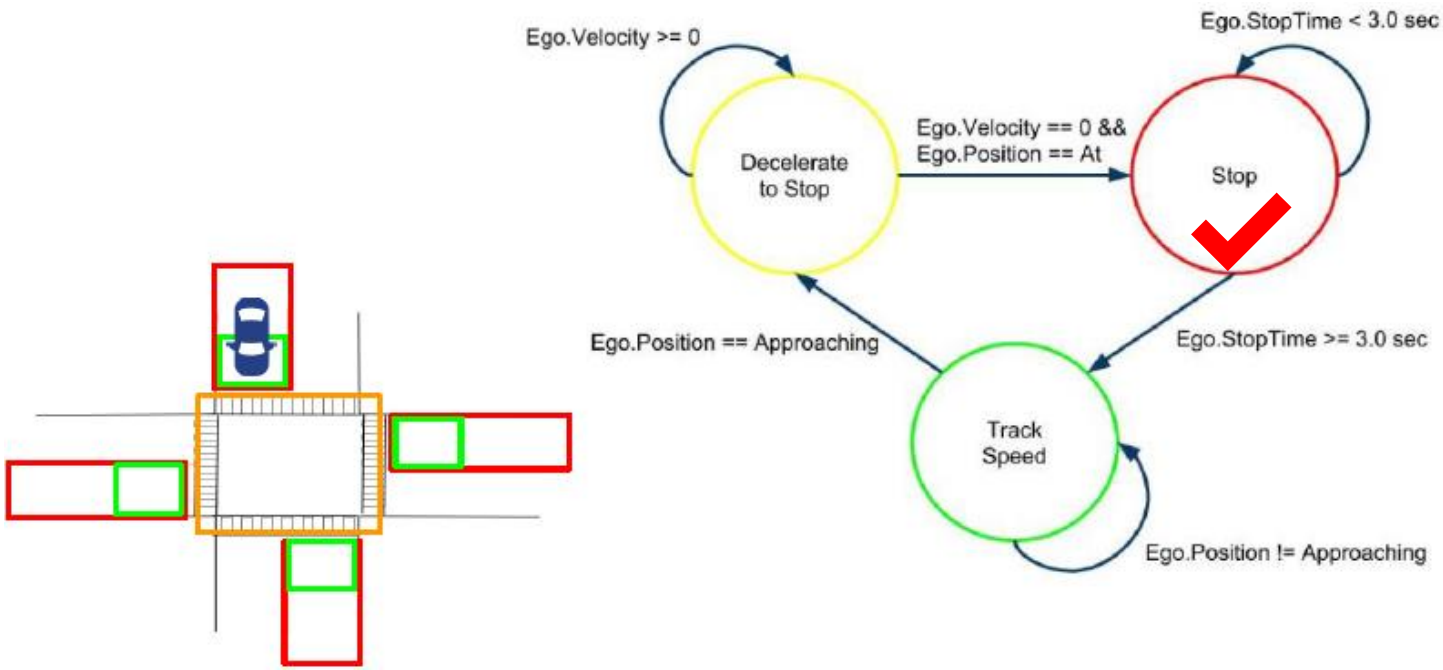





Decelerate
to Stop
Stop
Ego.Position == Approaching
Track
Speed
Ego.Position != Approaching



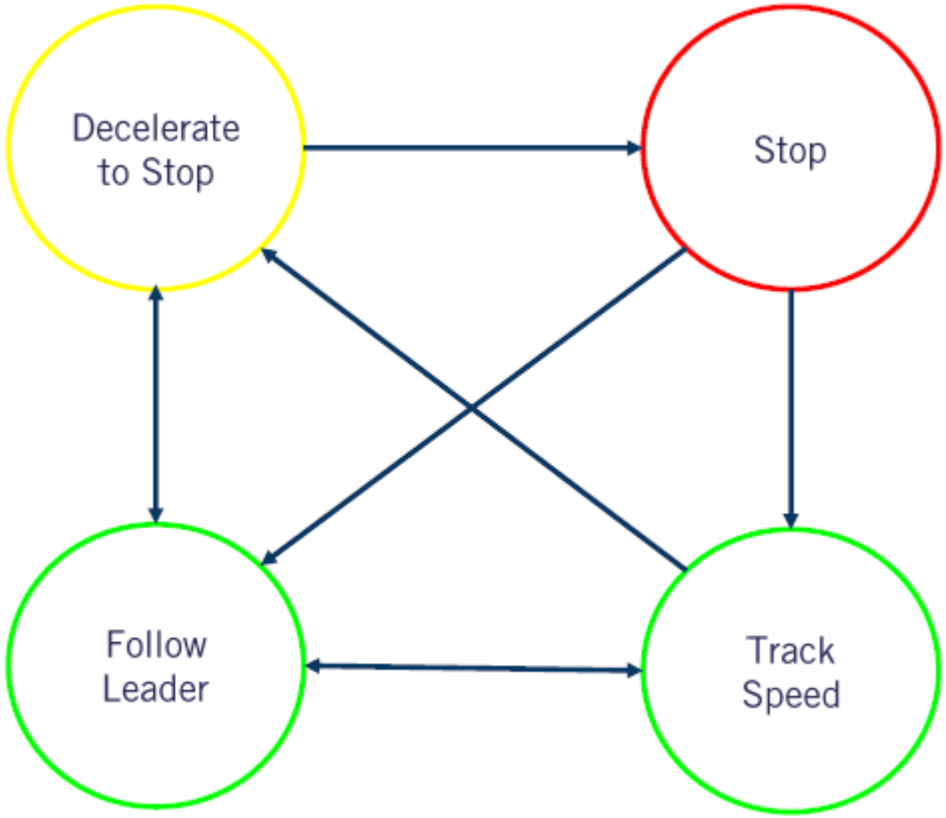

Ego.Velocity >= 0
Decelerate
to Stop
Ego.Velocity == 0 &&
Ego.Position == At
Stop
Ego.Position == Approaching
Track
Speed
Ego.Position != Approaching



- To make the state machine more robust to noise in state estimation, may change Ego.Velocity==0 to Ego.Velocity<=StopThreshold
- For an intersection with stop signs, there is no transition from “Decelerate to Stop” to “Track Speed”, since vehicle must come to a complete stop at a Stop sign
- For an intersection with traffic lights, there should be such a transition, as the light may turn green while vehicle is decelerating.

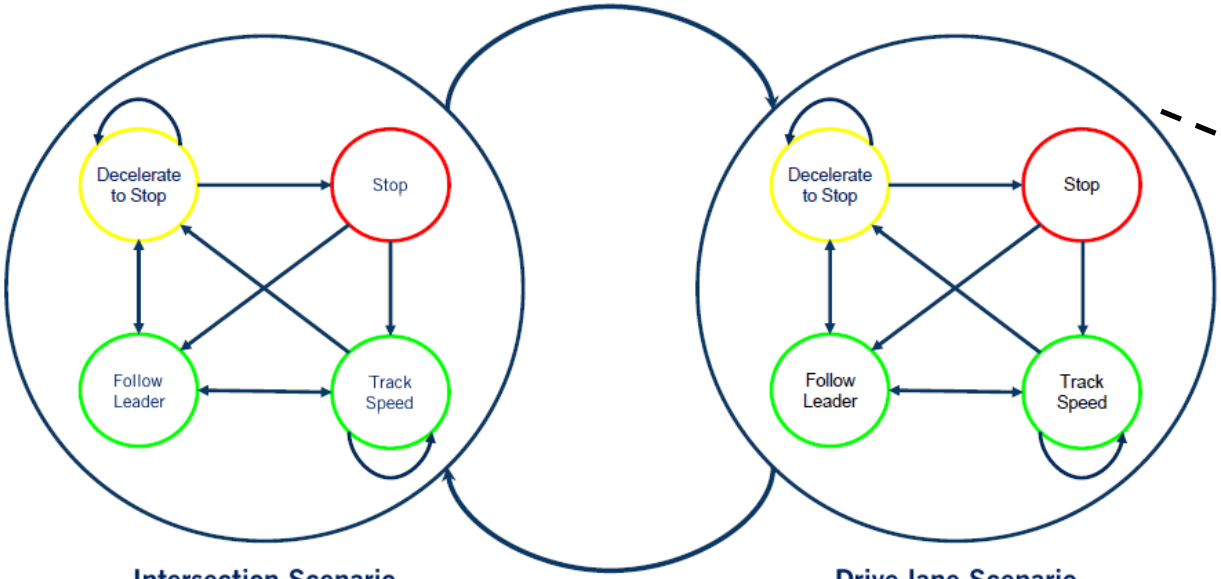
An Intersection Scenario with Dynamic Objects

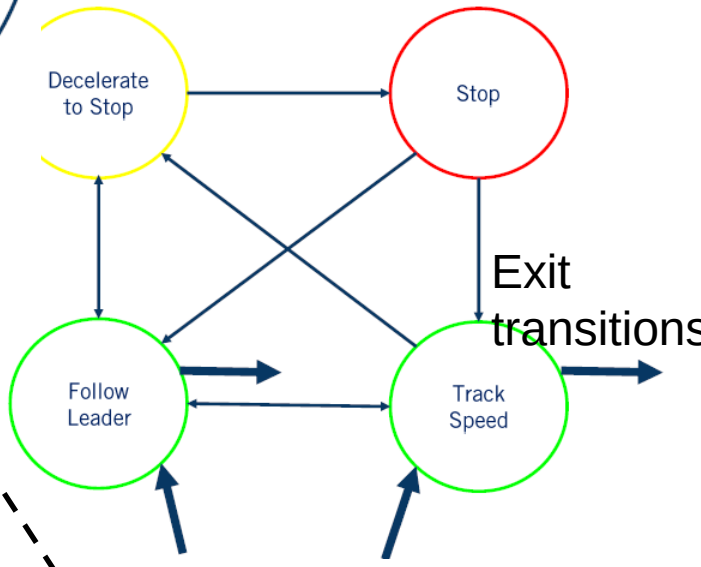
- Additional state “Follow Leader” to handle situation when another vehicle is in front of ego-vehicle.
- Again, no transition from “Decelerate to Stop” to “Track Speed”, since vehicle must come to a complete stop at a Stop sign



Handling Multiple Scenarios with Hierarchical FSM

- Each driving scenario is modeled as a super-state, which contains a low-level FSM for the scenario.



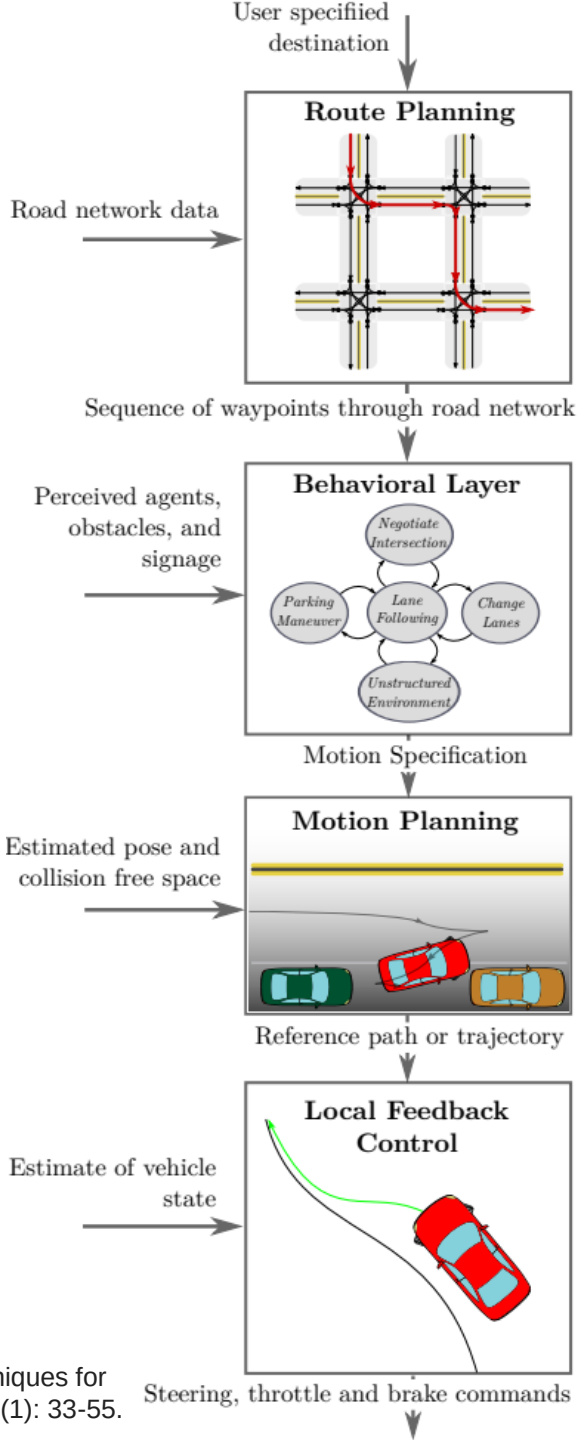


Limitations of FSM-based Behavior Planning

- State explosion for many and complex scenarios
- Dealing with measurement noise and uncertainty of the environment
  - Hyperparameters can be used to deal with some noise, but only in some limited situations, e.g., Ego.Velocity<=StopThreshold instead of Ego.Velocity==0
- Incapable of dealing with unencountered scenarios
- Reinforcement Learning is a promising alternative

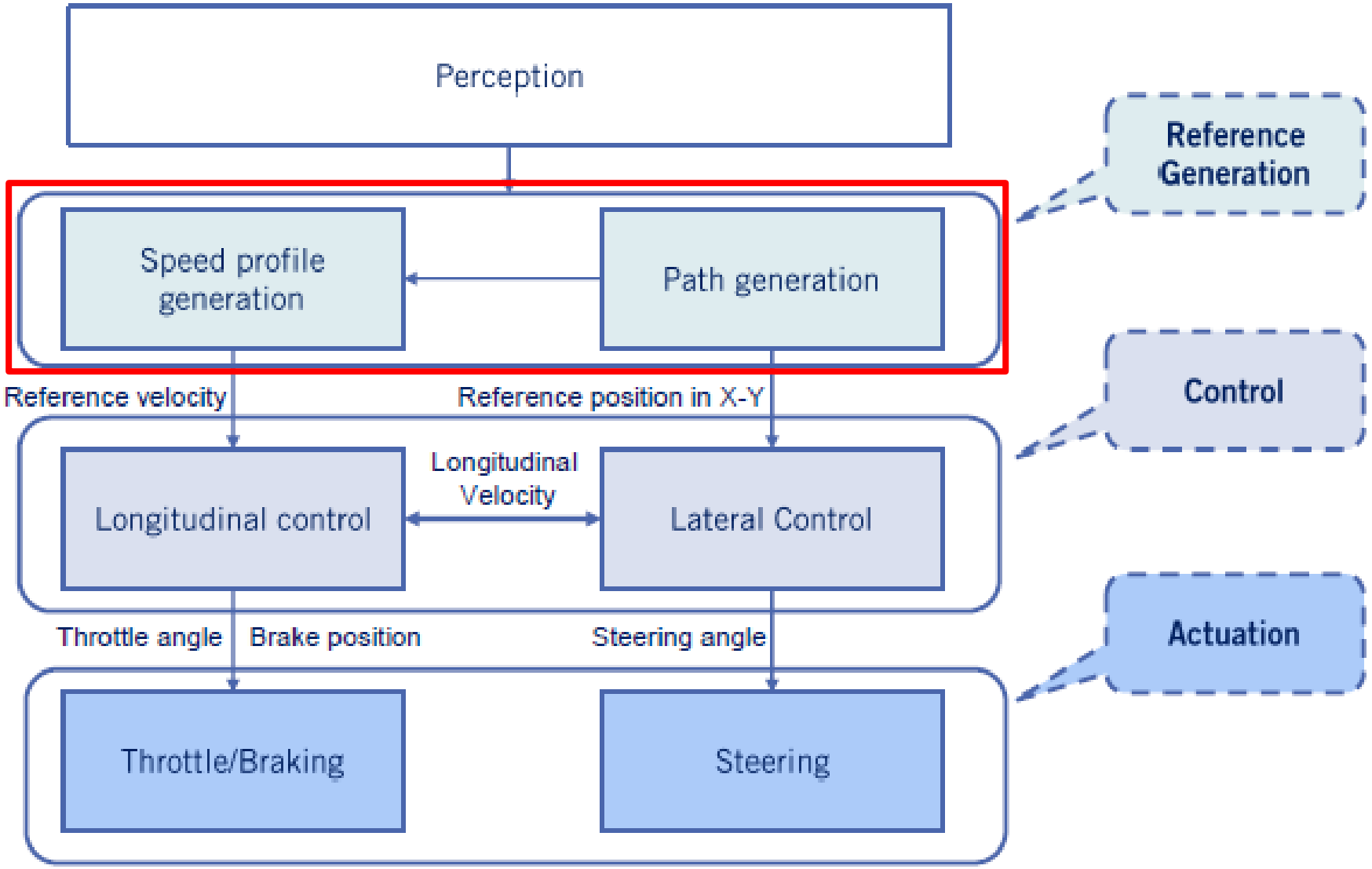
Outline

- Route planning
- Behavior planning
- Motion Planning
- Responsibility-Sensitive Safety (RSS)



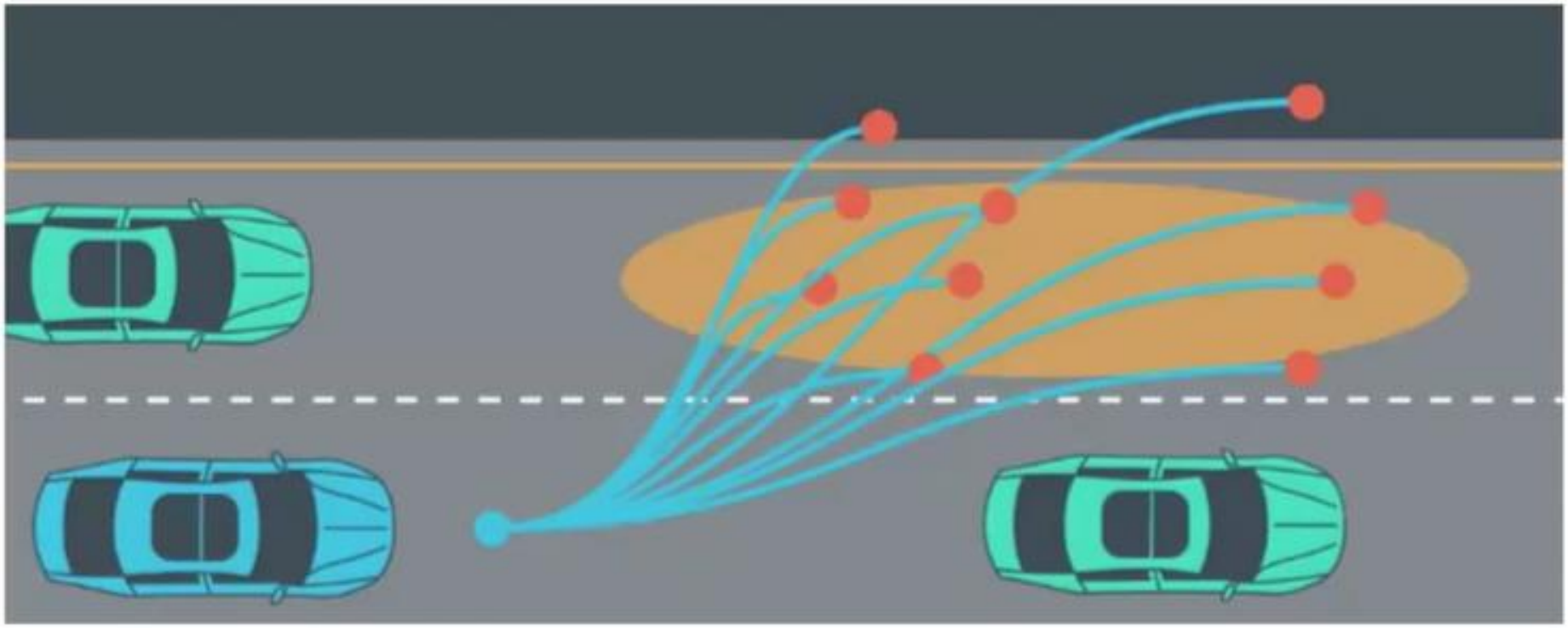
Paden B, Čáp M, Yong S Z, et al. A survey of motion planning and control techniques for self-driving urban vehicles[J]. IEEE Transactions on intelligent vehicles, 2016, 1(1): 33-55.

Vehicle Control Architecture



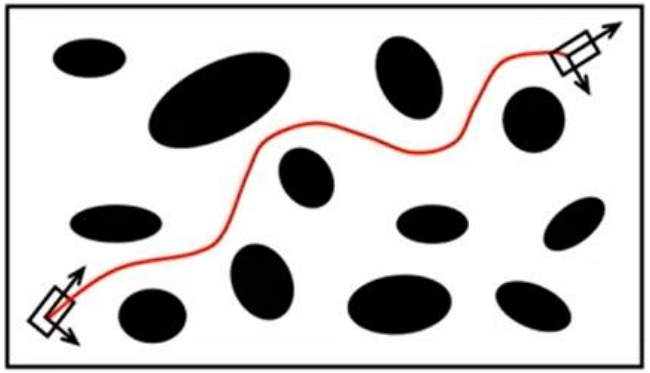
Motion Planning

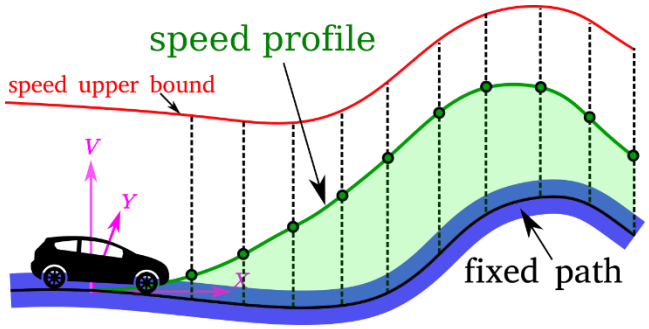
- Generate a feasible (smooth and collision-free) trajectory
  - A trajectory is a path parametrized by time



Motion Planning Methods

- Also called trajectory planning or local planning, includes path planning and speed profile generation:
  - Path planning:
    - Plan a path from point A to point B
  - Speed profile generation
    - Assign a speed value to every point on a given path
  - Variational planning: combine both path planning and velocity planning to generate a trajectory in a single-step.

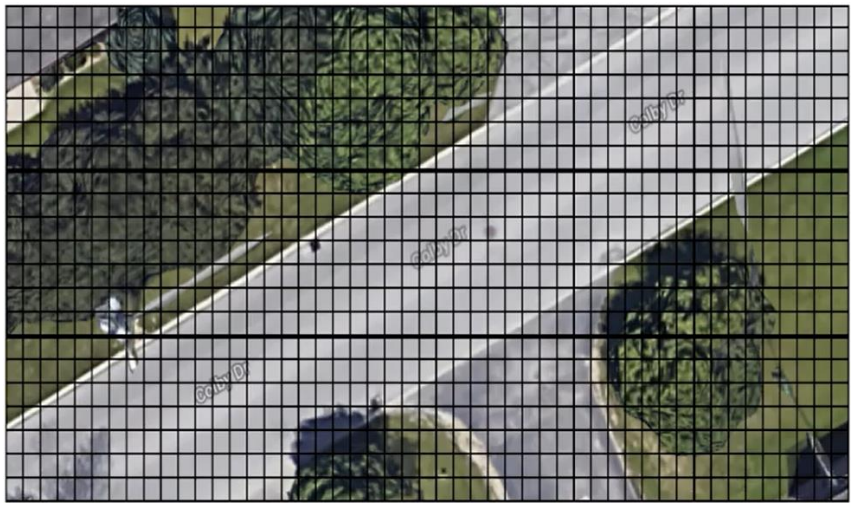


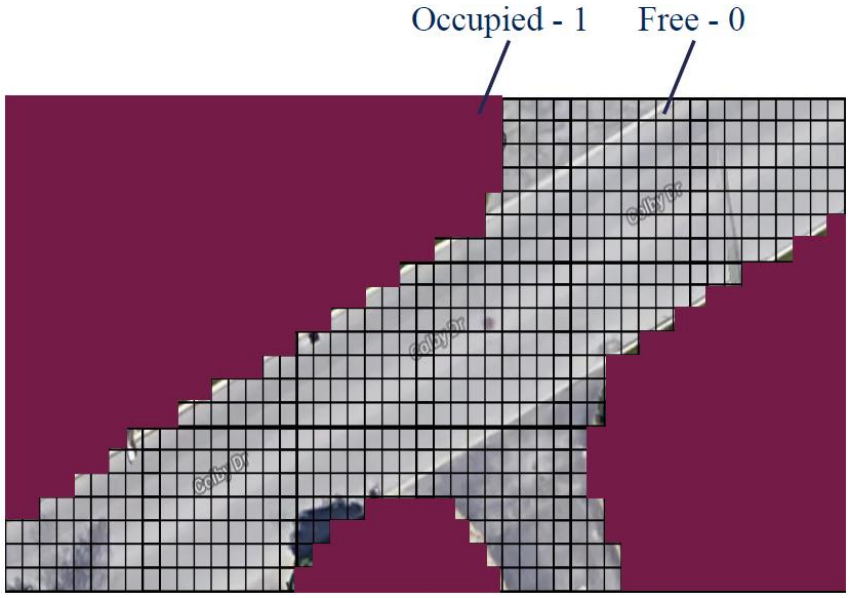


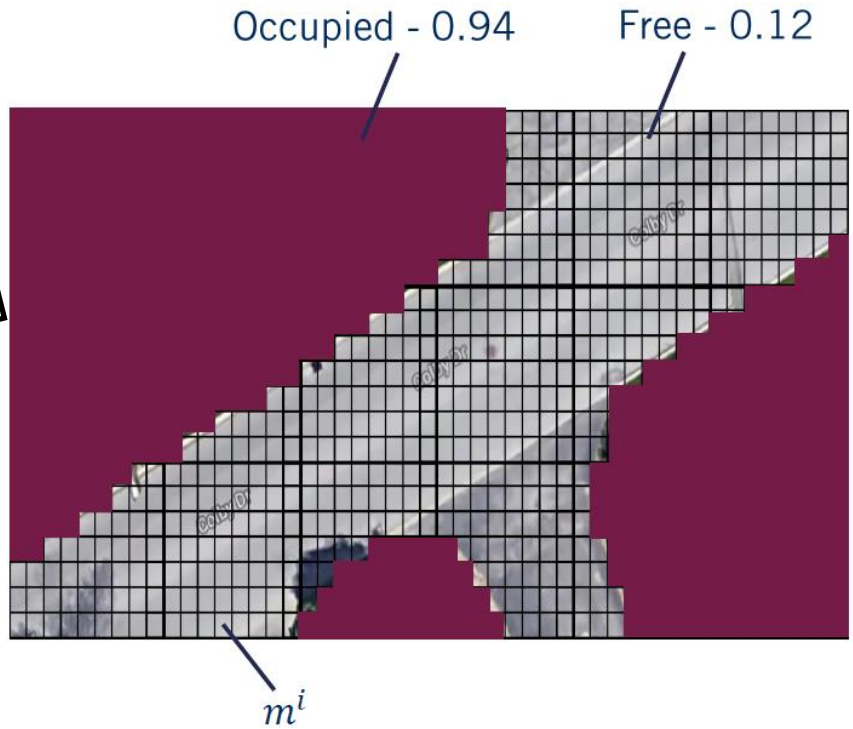
Path Planning Methods

	Model assumption	Completeness	Optimality	Time complexity	AnyTime
Geometric methods					
Parametric curve based planning (cubic/B spline, Bezier curve etc.)	-	Yes	No	O(n)	No
Variational methods					
Variational methods (direct collocation etc.)	Lipschitz-c Jacobian	No	Locally	O(1/e)	No
Graph-Search methods					
Visibility graph + Djiksta	-	No	No	O(n + m log m)	No
Lattice/Tree motion primitive + Djikstra	-	No	No	O(n + m log m)	No
PRM + Djikstra	Exact steering procedure available	Probabilistically complete	Asymptotically optimal	O(n log n)	No
PRM* + Djikstra	Exact steering procedure available	Probabilistically complete	Asymptotically optimal	O(n log n)	No
RRG + Djikstra	Exact steering procedure available	Probabilistically complete	Asymptotically optimal	O(n log n)	Yes
Incremental-Search methods					
RRT	-	Probabilistically complete	suboptimal	O(n log n)	Yes
RRT*	Exact steering procedure available	Probabilistically complete	Asymptotically optimal	O(n log n)	Yes

Occupancy Grid



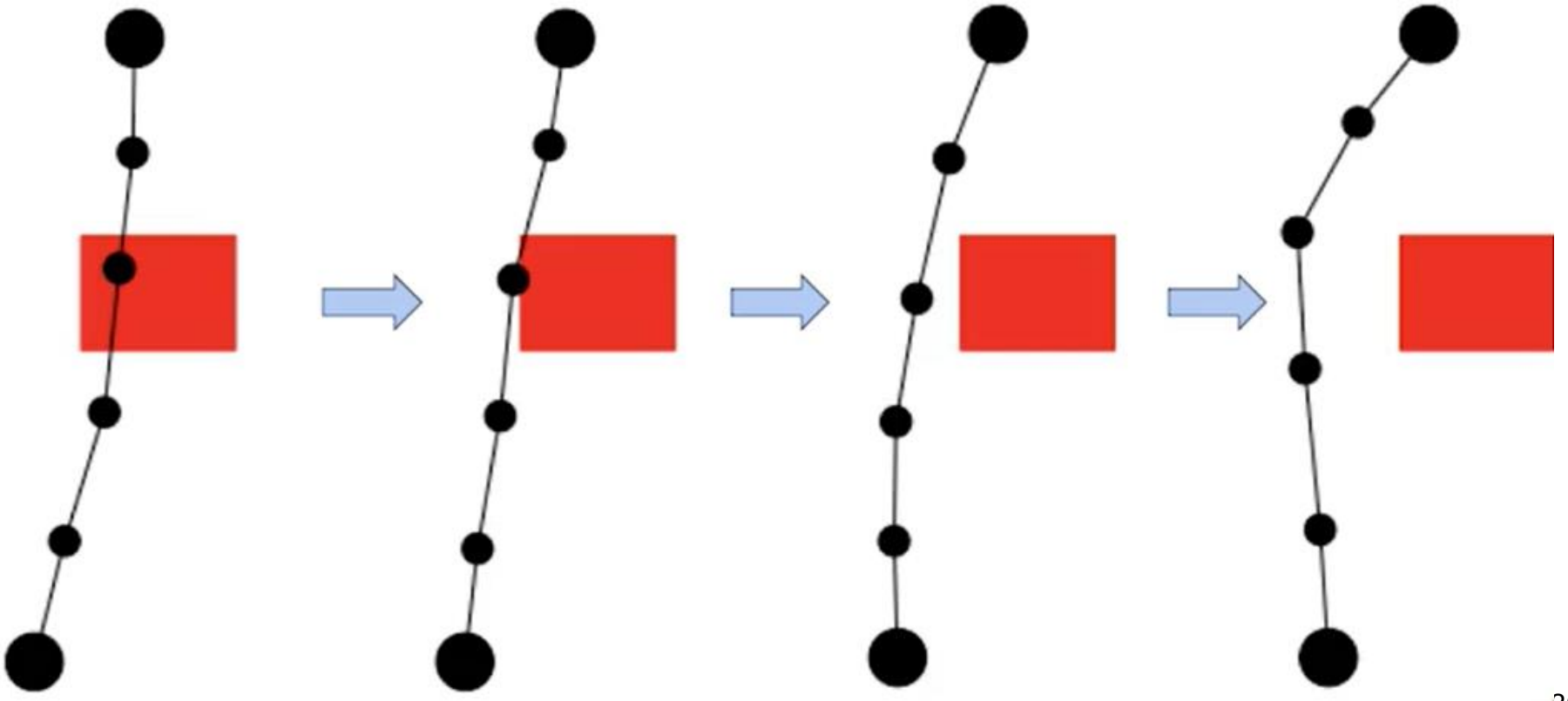




- Discretized grid map encoding occupancy of static objects (e.g., all non-drivable surfaces are occupied, incl. trees, buildings, sidewalks, lawns…), obtained with Lidar or camera
- Can be binary encoding (upper left), or probabilistic encoding (upper right), using a belief threshold to convert to binary encoding

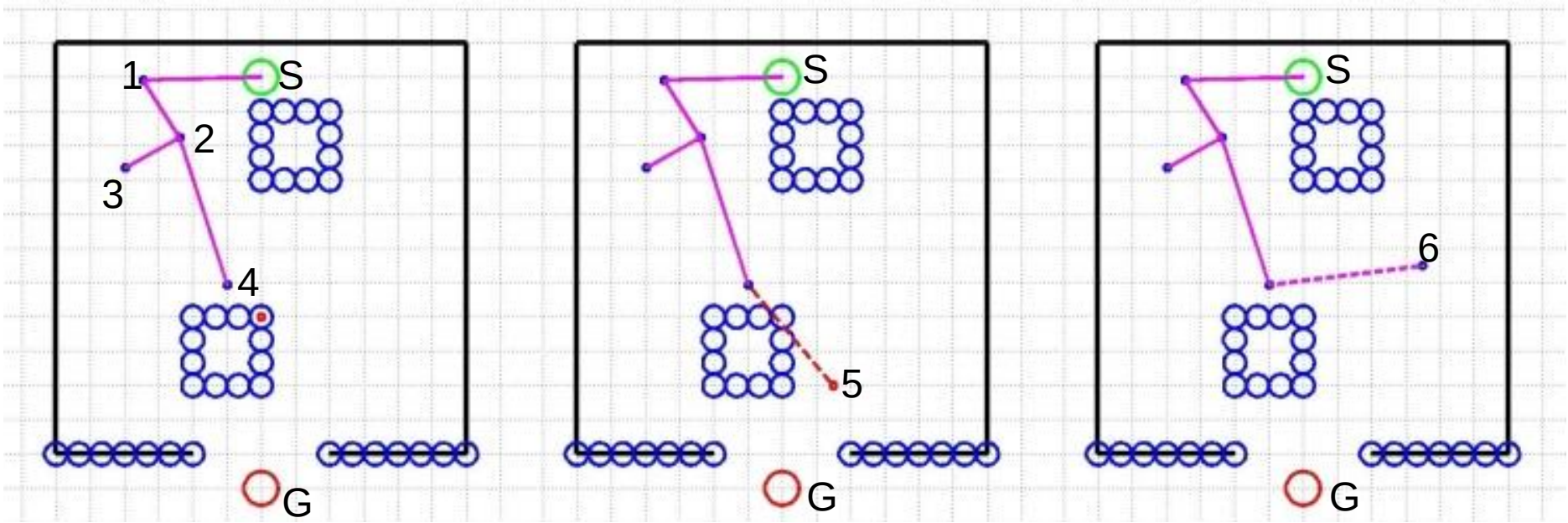
Variational Methods

- Optimize trajectory by minimizing a functional, $\min_{\delta x} J(x + \delta x)$ which contains penalties for collision avoidance and robot dynamics
- Can be slow, and may not converge to a feasible solution
- (Detailed omitted)



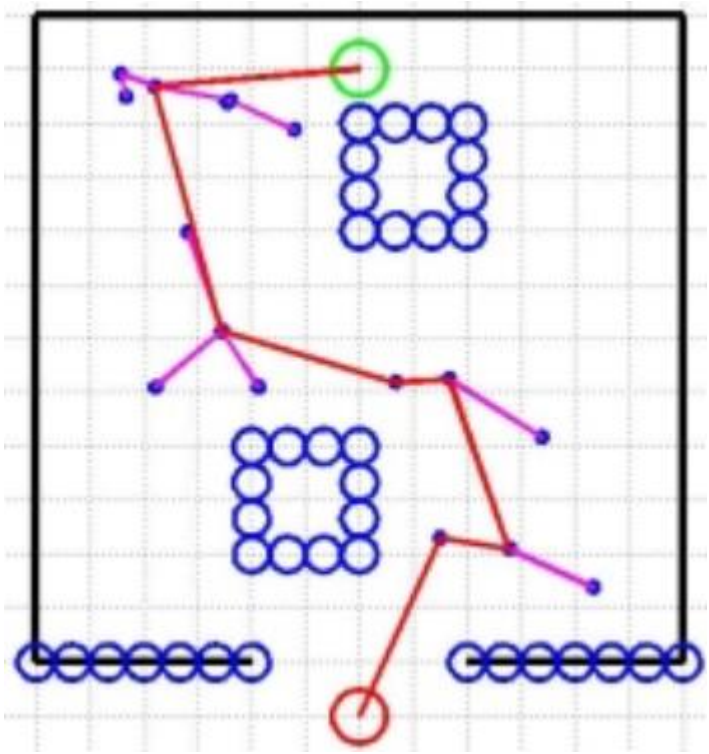
Rapidly-exploring Random Tree (RRT)

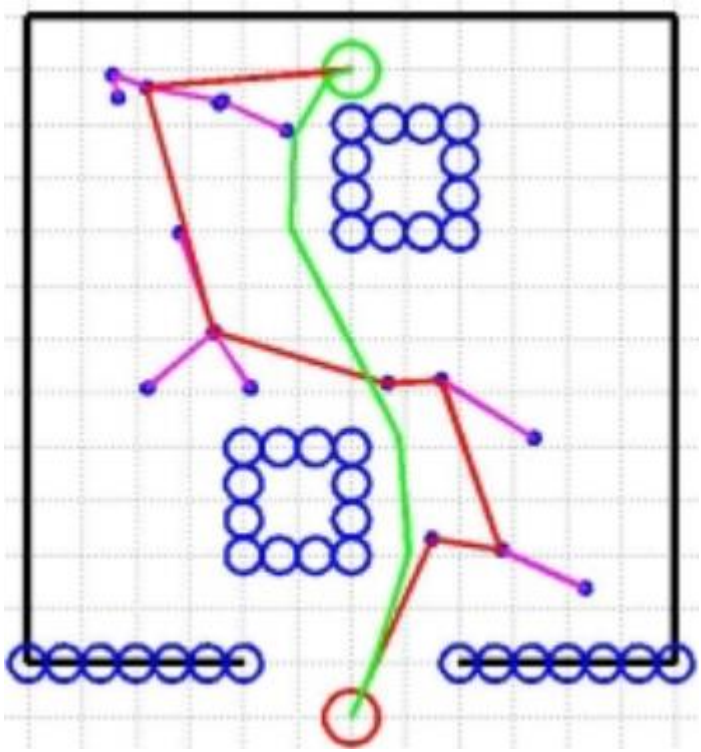
- Insert start location S in tree
- While tree cannot connect to goal
  - Sample random point r
  - Find point p in tree that is closest to r
  - Add branch between p and r
    - Another variant is to add branch of predefined length from p in direction of r
  - If new branch intersects obstacle:
    - Discard new branch (or shorten)
- Compute path from start (S) to goal (G) through tree
- Shortcut path: for any two points in path, add direct line unless direct line intersect an obstacle (collision checking)



Path Smoothing

- Left: an example completed path
- Right: smoothed path (green), obtained by iteratively sampling points between nodes on the overall path and then checking if two points could be connected to reduce path length

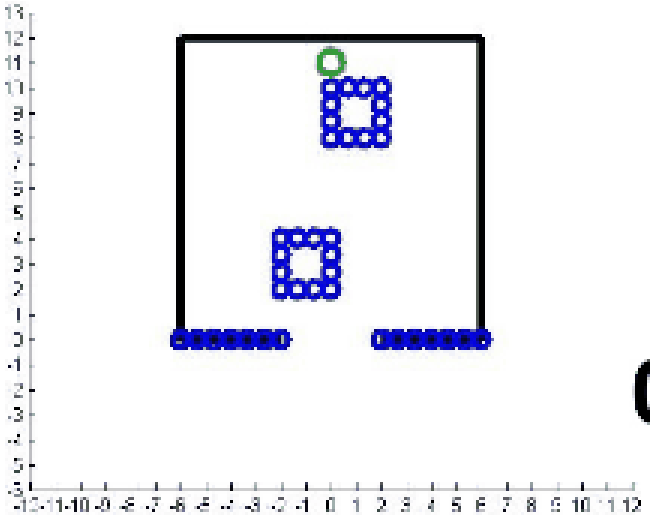




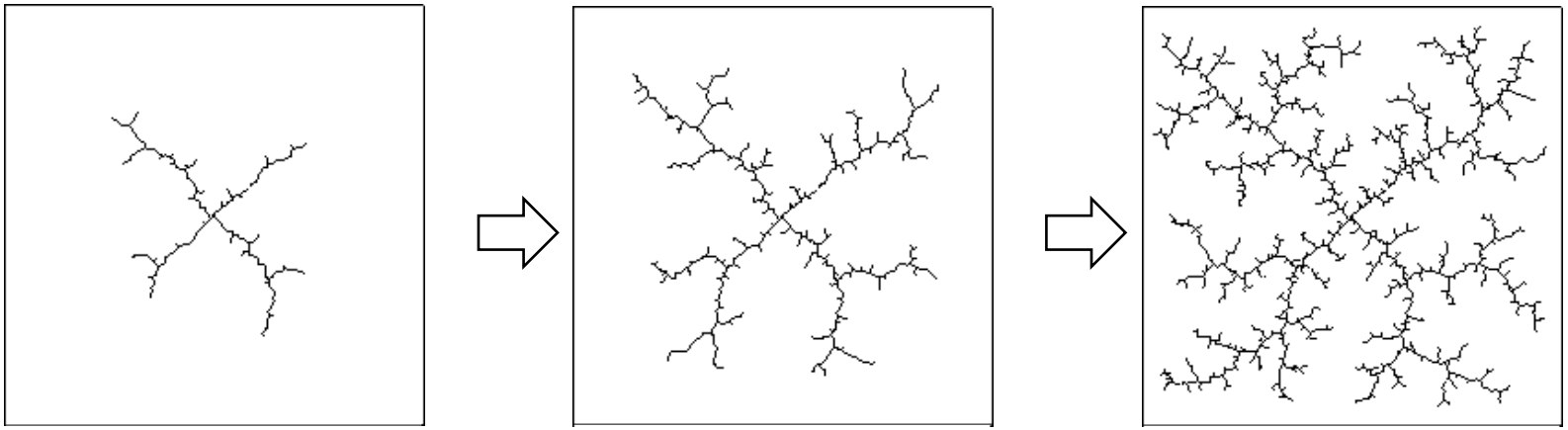
http://coecsl.ece.illinois.edu/ge423/spring13/RickRekoskeAvoid/rrt.html

RRT Animation

- RRT growth process (with start location but no goal location)



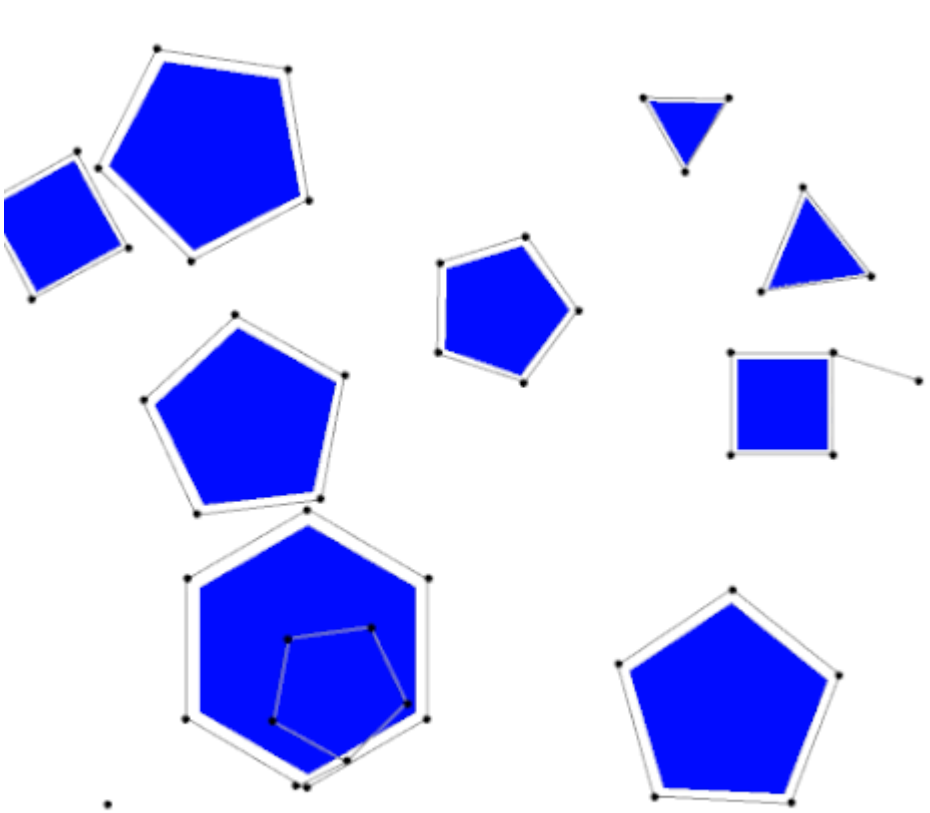
0



More animations https://www.youtube.com/watch?v=pOFtvZ_qVsA

Probabilistic Roadmap (PRM)

- **Map construction**:
- While # points in roadmap lower than threshold
  - Sample random point
  - If new point is not in collision:
    - Connect new point to all other points in the roadmap with lines, as long as they do not intersect obstacles.
- **Path finding**:
  - Connect start location to nearest point in roadmap s.t. connecting line does not intersect obstacle
  - Connect goal location to nearest point in roadmap s.t. connecting line does not intersect obstacle
  - Find a path between start and goal traversing the roadmap, with algorithms such as as A* or Dijkstra's algo.



RRT vs. PRM

- With RRT, each time new start/goal locations are given, an entirely new graph must be generated, which may be inefficient.
- PRM builds a roadmap graph for traveling through a region, but relies on A* or Dijkstra's algo to find the shortest path from start to goal traversing the roadmap. Hence the roadmap is reused each time new start/goal locations are given.

RRT & PRM

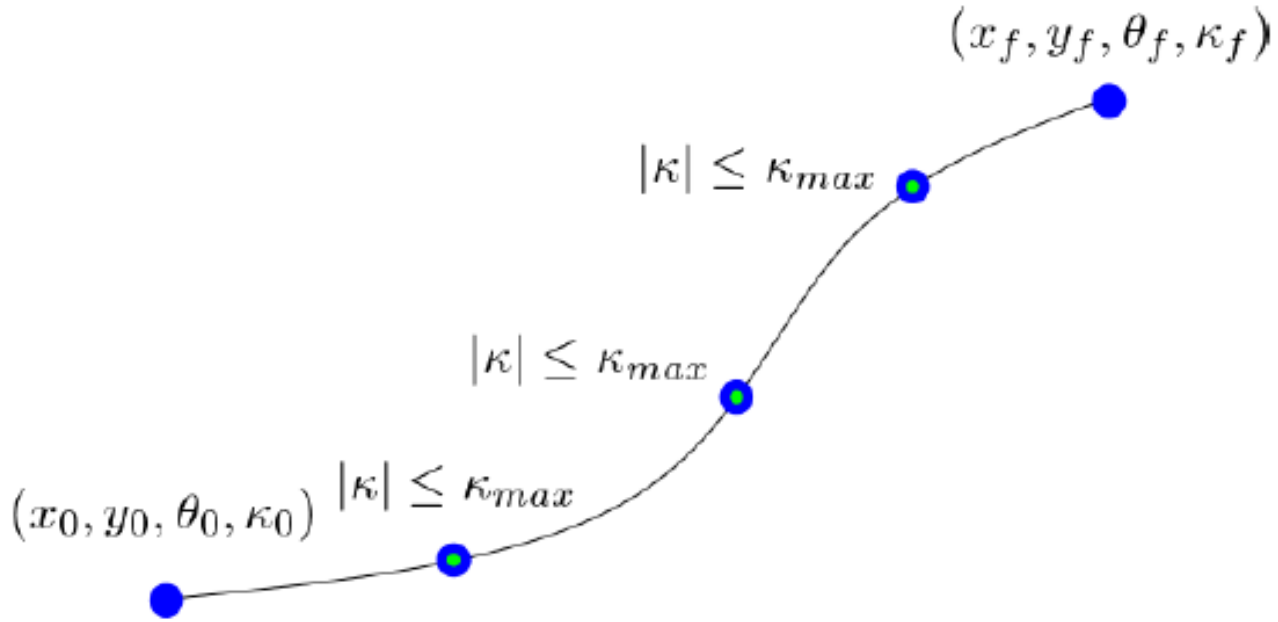
- Many variants of RRT and PRM are probabilistically complete:
  - If a path exists, it will be found in finite time
  - But in the worst-case, solution time can be very long (even longer than exhaustive search)
- No guarantees on solution quality
  - May not find the shortest path
    - With PRM, A*/Dijkstra is guaranteed to find the shortest path for *the given roadmap*, but the roadmap may not be optimal.
  - Often needs post-processing (e.g., smoothing)

Quiz

- Which statement is true for RRT and PRM motion planning algorithms?
  - A. Both algorithms will produce the shortest path.
  - B. Both algorithms can be used for high dimensional spaces
    - Robotic motion planning may be high-dimensional; mobile robot path planning is low-dimensional (3 for ground robot, 6 for aerial robot).
  - C. PRM always results in a better path than RRT.
  - D. The time for computation is consistent and predictable.
- ANS: B

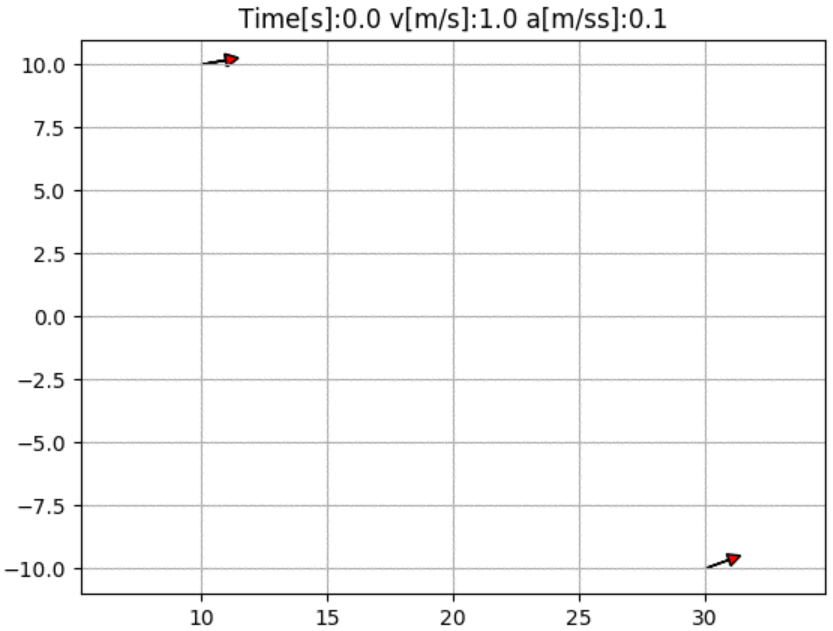
Parametric Curve-based Planning

- Non-parametric path: a path is represented as a sequence of points (as in A*, Dijkstra's algo, RRT/PRM)
- Parametric path: a smooth parametric curve that satisfies boundary conditions at 2 end points, constraints on curvature, and optimizes an objective function.
  - $(x_0, y_0, \theta_0, \kappa_0), (x_f, y_f, \theta_f, \kappa_f)$: pose (position and heading angle) and curvature ($\kappa = \frac{1}{R}$, inverse of radius of rotation) of starting and final points
  - $|\kappa| \leq \kappa_{max}$: curvature has an upper bound for every point on the curve, to help satisfy kinematic/dynamic constraints, and improve rider comfort.
- Example: Quintic splines and polynomial spirals



Quintic Spline

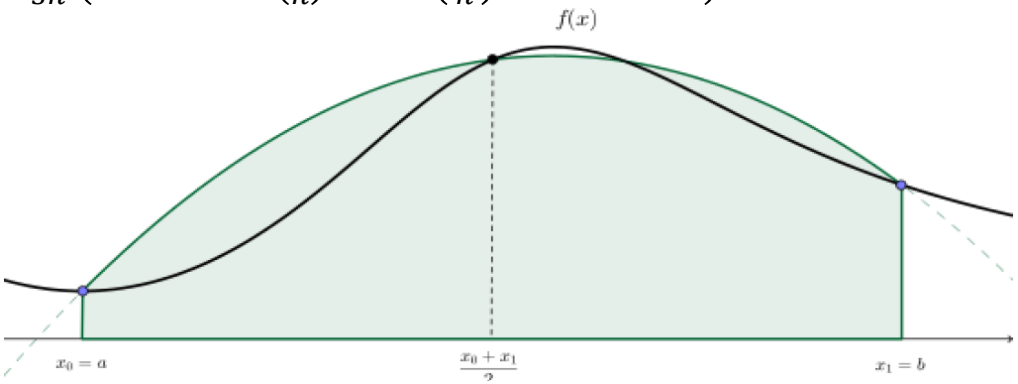
- Path modeled as 5th-order polynomials, with closed-form solutions for boundary conditions (x, y, θ, κ)
  - $x(u) = \alpha_5 u^5 + \alpha_4 u^4 + \alpha_3 u^3 + \alpha_2 u^2 + \alpha_1 u + \alpha_0$
  - $y(u) = \beta_5 u^5 + \beta_4 u^4 + \beta_3 u^3 + \beta_2 u^2 + \beta_1 u + \beta_0$
  - $u \in [0,1]$
  - The curve connects from start point $(x(0), y(0))$ to end point $\big(x(1), y(1)\big)$
- Pro: path has closed-form equation for given start and end points
- Con: curvature function and its derivatives may be discontinuous



https://pythonrobotics.readthedocs.io/en/latest/modules/path_planning.html

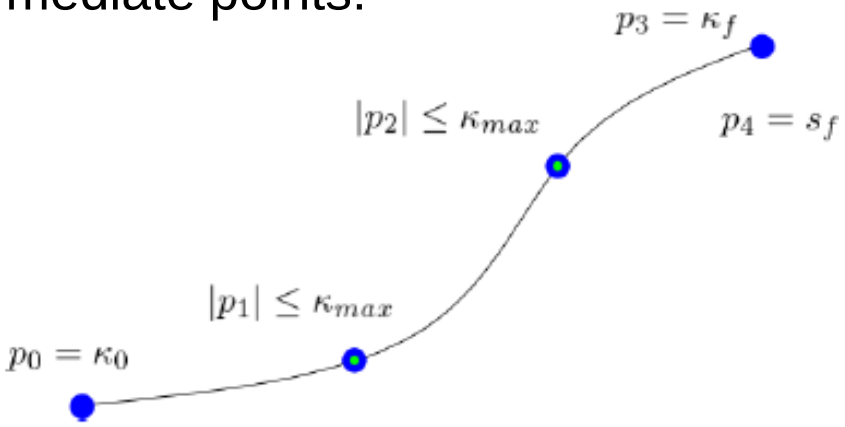
Polynomial Spirals

- Curvature defined as polynomial function (cubic or higher):
  - $\kappa(s) = a_3 s^3 + a_2 s^2 + a_1 s + a_0$
  - It is smooth and never discontinuous, so constraining a few points often enough to constrain the entire curve.
- Path modeled based on curvature function:
  - $\theta(s) = \theta_0 + \int_0^s (a_3 s'^3 + a_2 s'^2 + a_1 s'^1 + a_0) ds' = \theta_0 + a_3 \frac{s^4}{4} + a_2 \frac{s^3}{3} + a_1 \frac{s^2}{2} + a_0 s$
  - $x(s) = x_0 + \int_0^s \cos\theta(s') ds'$
  - $y(s) = y_0 + \int_0^s \sin\theta(s') ds'$
- Pro: curvature function is smooth, so curvature can be given upper bound constraint.
- Con: path has no closed-form equation; Fesnel integrals need to be computed with numerical integration using Simpson's rule, with high computation overhead:
  - $\int_0^s f(s') ds' \approx \frac{s}{3n}\left(f(0) + 4f\left(\frac{s}{n}\right) + 2f\left(\frac{2s}{n}\right) + \cdots + f(s)\right)$ (fig below shows example w. $n = 2$)

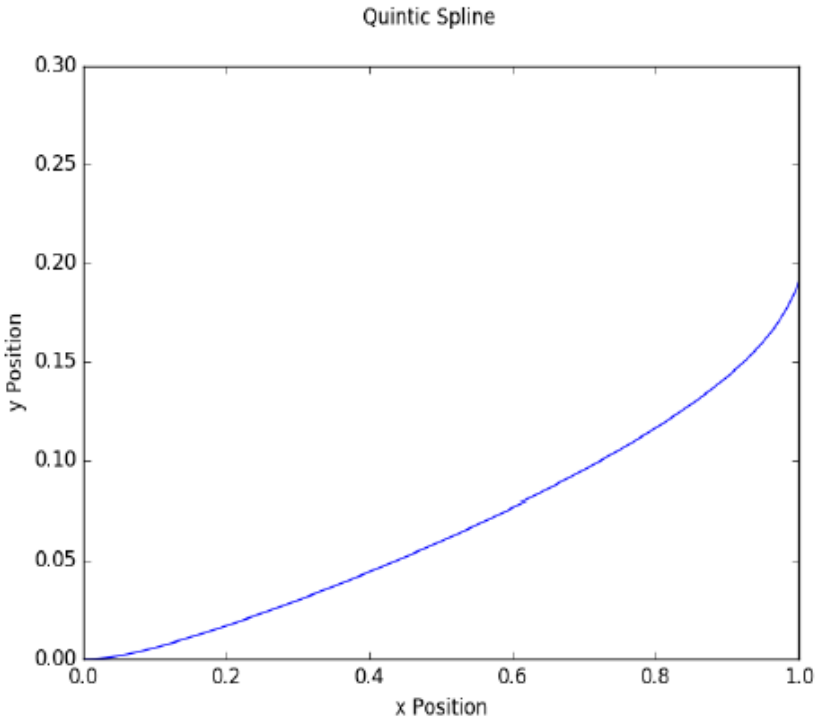


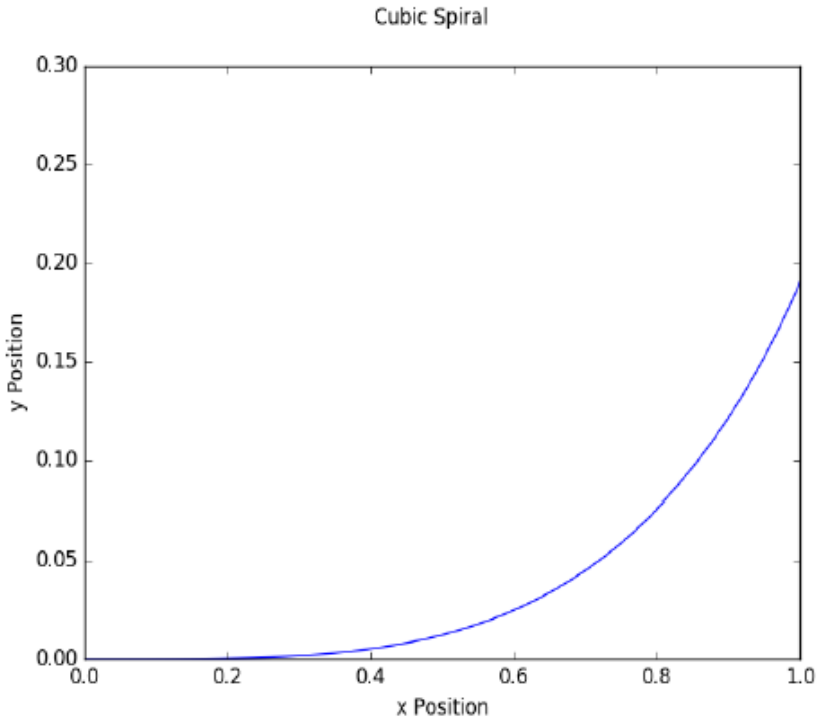
Optimization Objective Function

- Consider curvature modeled as Cubic spiral.
- Minimize *Bending Energy*, defined as integral of squared curvature along the path, in order to distribute curvature evenly along spiral:
  - Minimize $f_{be}(a_0, a_1, a_2, a_3, s_f) = \int_0^{s_f} (a_3 s^3 + a_2 s^2 + a_1 s + a_0)^2 \, ds$
  - It and its gradient have closed-form solutions.
- Subject to constraints:
  - End point constraints:
    - $(x_s(0), y_s(0), \theta(0), \kappa(0)) = (x_0, y_0, \theta_0, \kappa_0)$
    - $(x_s(s_f), y_s(s_f), \theta(s_f), \kappa(s_f)) = (x_f, y_f, \theta_f, \kappa_f)$
  - Curvature constraints for two intermediate points:
    - $\left|\kappa(\frac{s_f}{3})\right| \leq \kappa_{max}, \left|\kappa(\frac{2s_f}{3})\right| \leq \kappa_{max}$



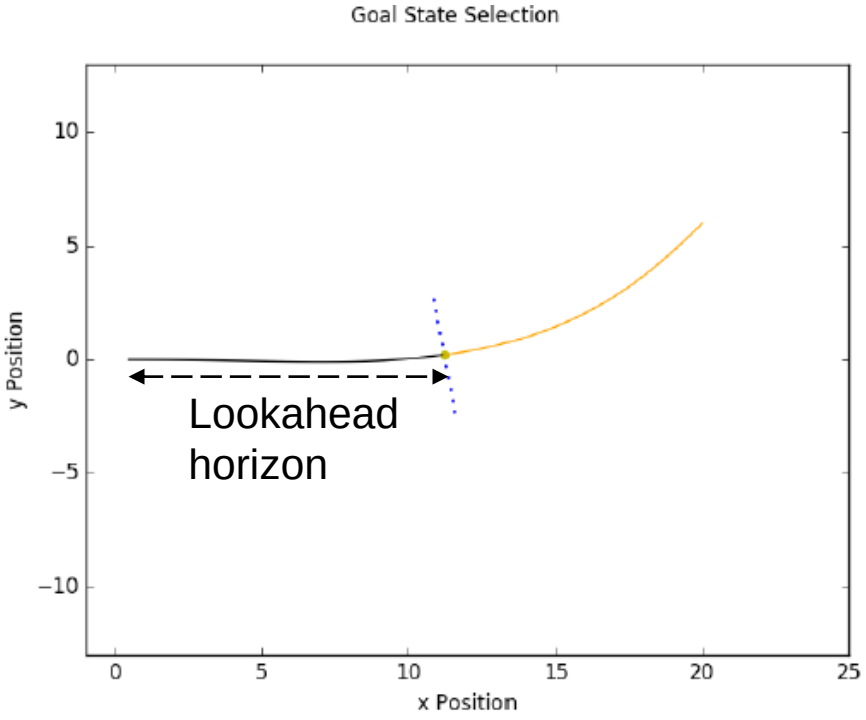
Example Spline vs. Spiral

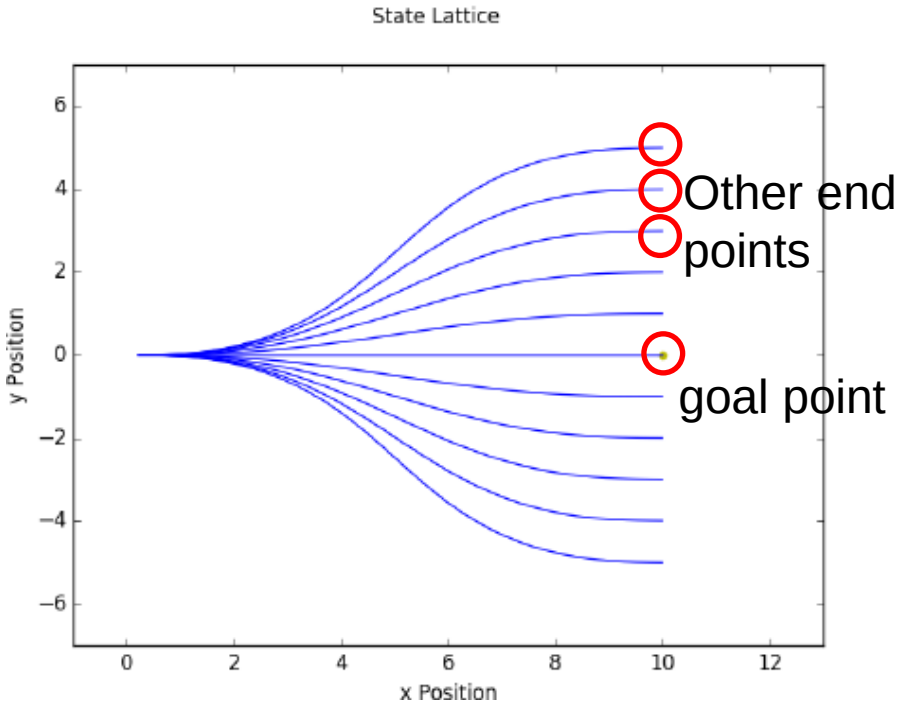




Conformal Lattice Planning

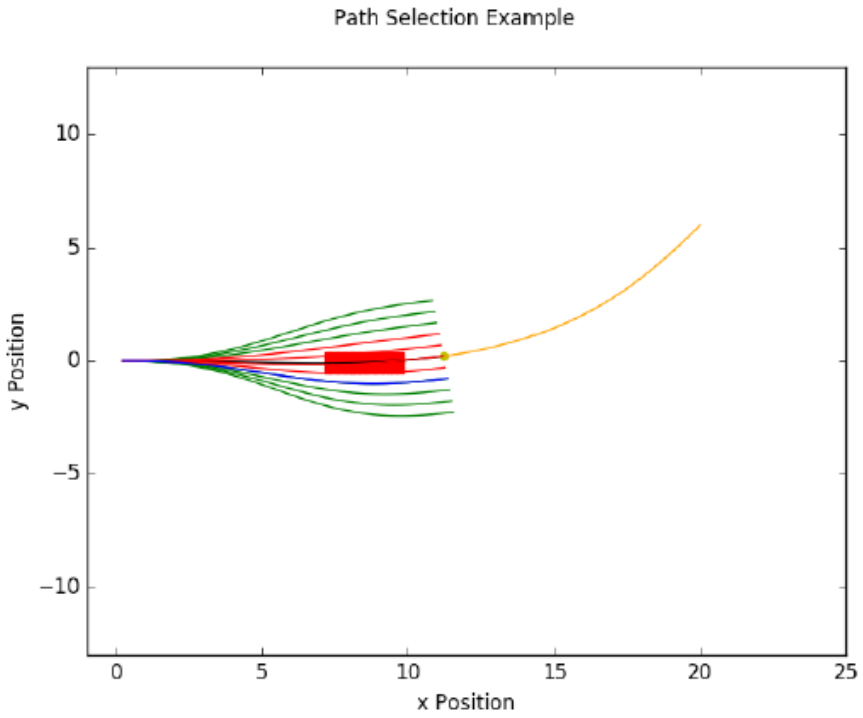
- To plan a feasible (collision-free) path to goal, exploit road structure, and generate a set of lattice paths laterally offset from a goal point along the road
- Goal point is dynamically calculated based on speed and other factors
  - Longer lookahead horizon improves ability of collision avoidance, but increases computation time.
- Other end points are sampled with lateral offsets from goal according to vehicle heading.
- For each end point (incl. goal point), solve the opt problem to get a spiral, then use numerical integration (Trapezoidal rule) to generate points along the paths.





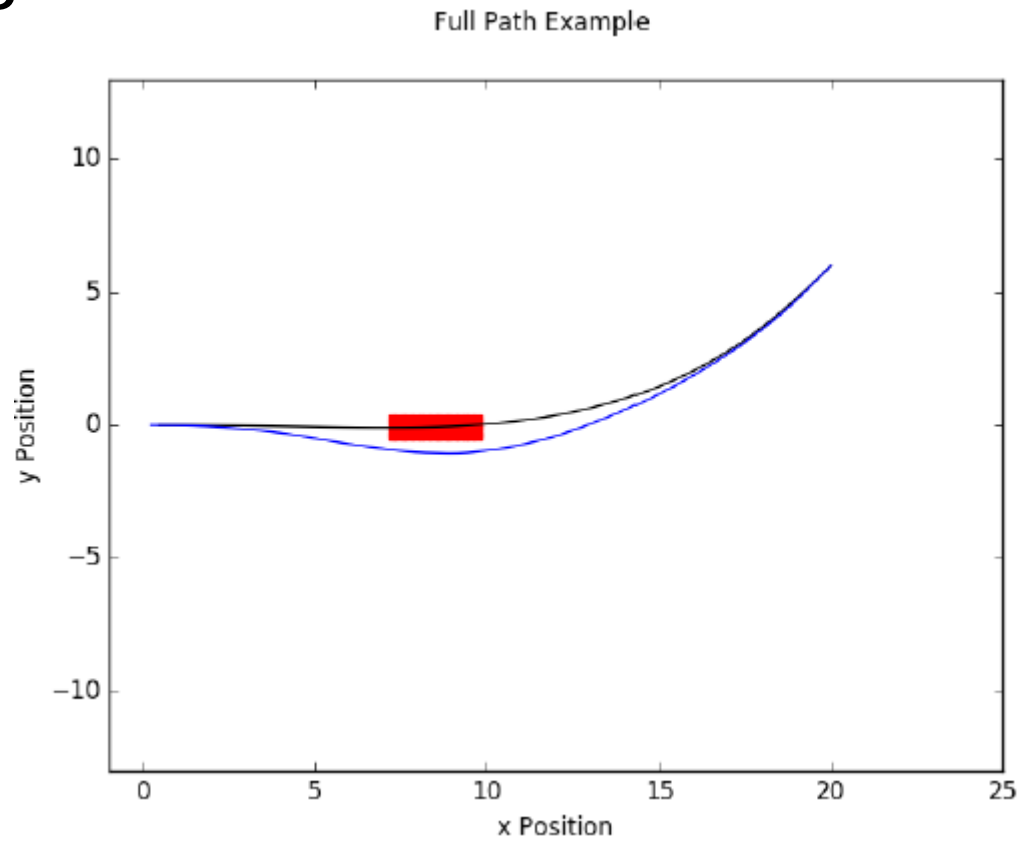
Collision Checking and Path Selection

- Perform collision checking for each point along each path
  - Red box denotes an obstacle (a parked car), and red lines denote paths in collision with it.
- Among collision free paths, select one based on custom obj function (blue path)
  - May prefer paths close to center of road, while penalizing paths that come too close to obstacle.



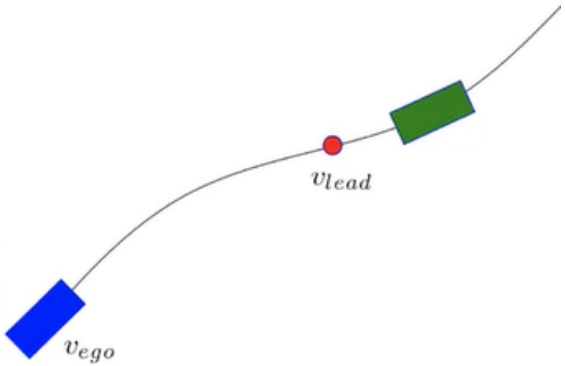
Full Path

- Repeat this process for each planning period in a receding horizon manner
- Full path will converge to center of road, after passing the obstacle.



Speed Profile Generation

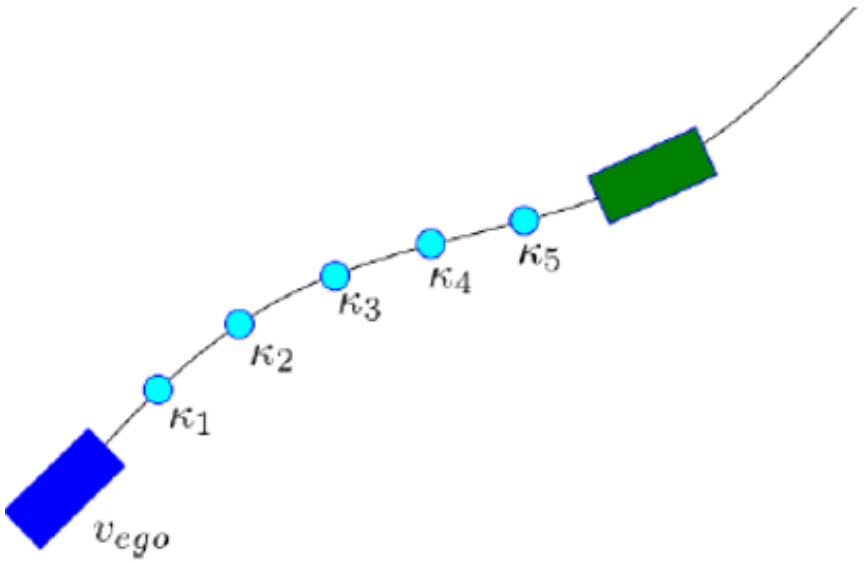
- Behavior planner provides reference speed to local planner
  - May be current road speed limit
  - Or current maneuver (stopping due to Stop sign or red light, following a lead vehicle…)
- Time-To-Collision (TTC) can be computed by difference in speeds $v_{ego} - v_{lead}$ divided by distance (arc length s).
    - Need to reach the red point at lead vehicle speed to avoid collision.



$$TTC = \frac{v_{ego} - v_{lead}}{s}$$

Curvature and Lateral Acceleration

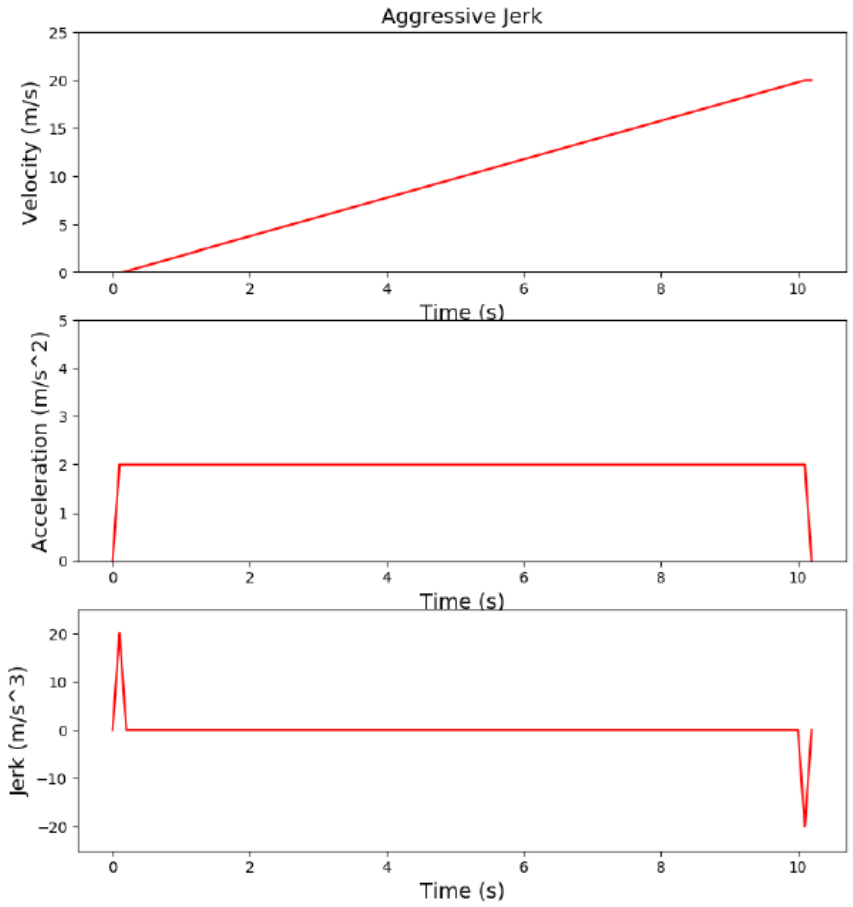
- Speed also bounded by max lateral acceleration and curvature
  - $v_k \leq \sqrt{\frac{a_{lat}}{\kappa_i}}$
- Final reference speed selected as minimum of all upper bounds

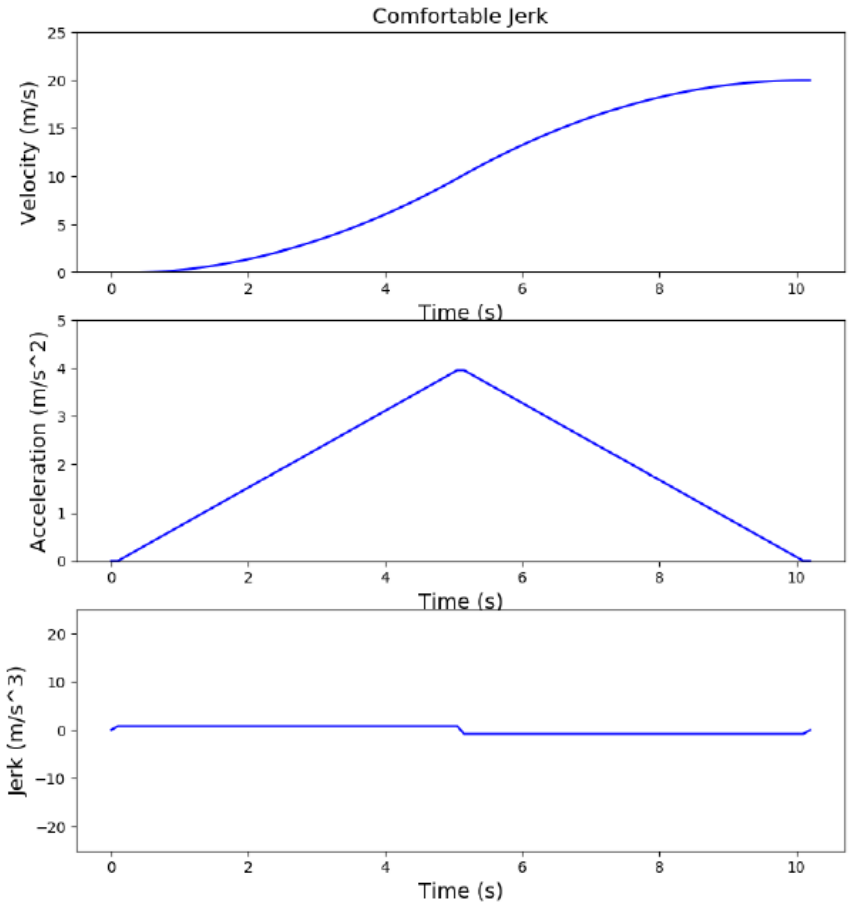


Speed Profile Examples

$$\int_0^{s_f} \|\dddot{x}(s)\|^2 ds$$

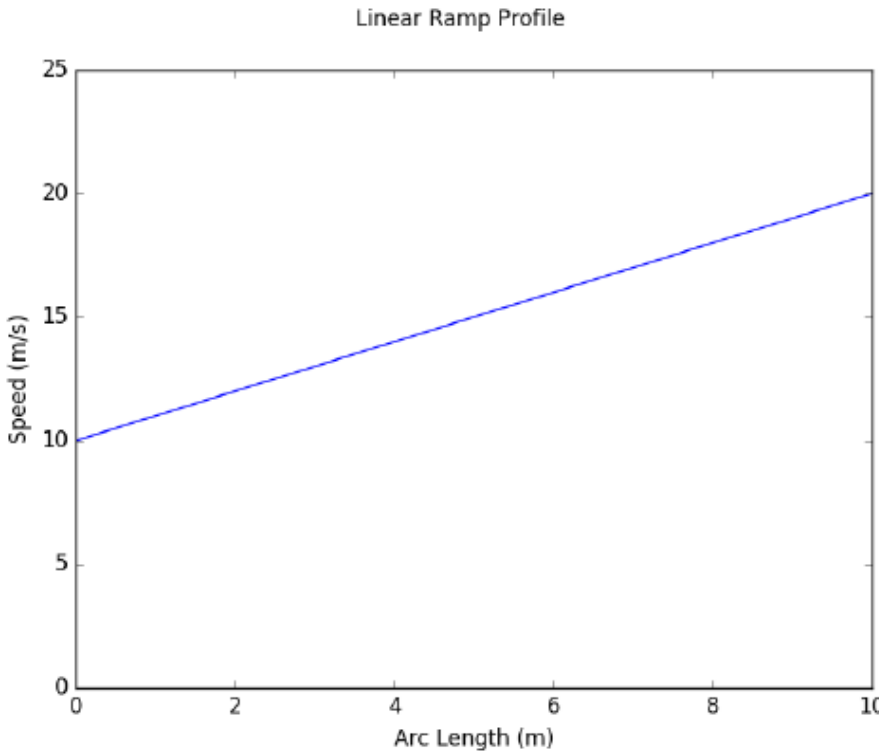
Jerk

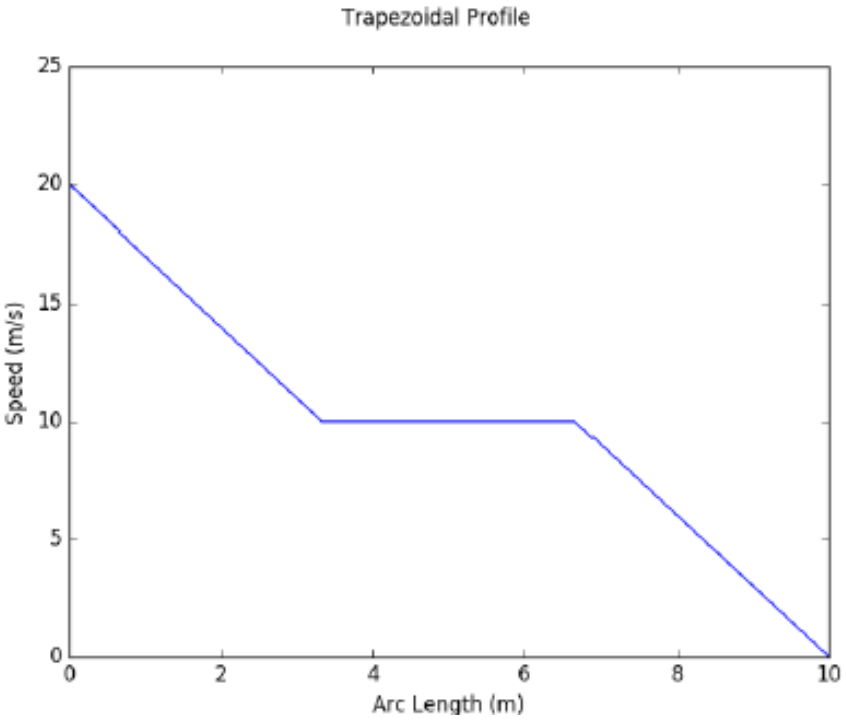




Linear Ramp vs. Trapezoidal Profile

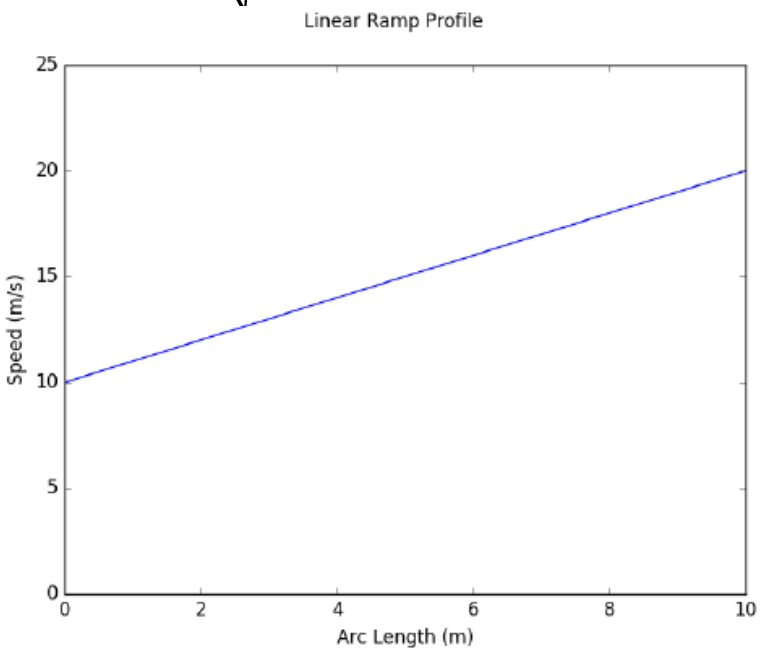
- Linear ramp profile: constant acceleration to reach reference speed
- Trapezoidal profile: constant-0-constant deceleration (e.g., for stopping at red light)

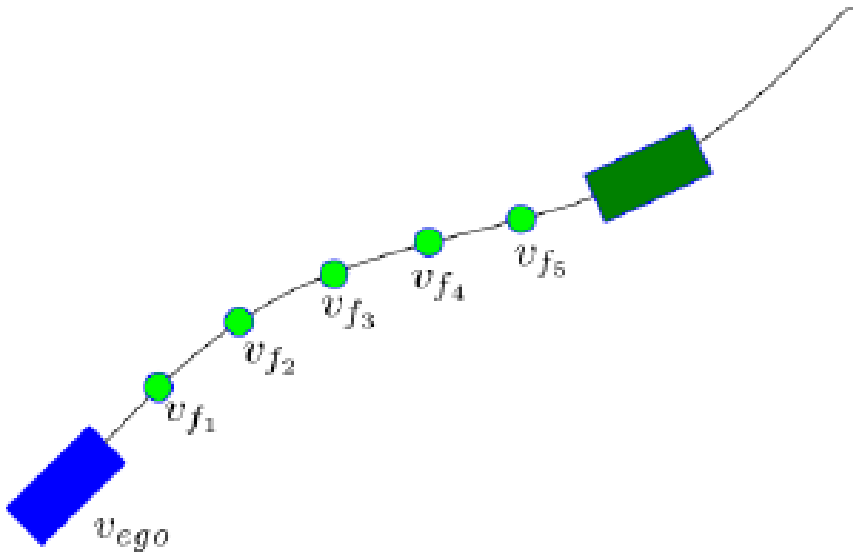




Linear Ramp Profile

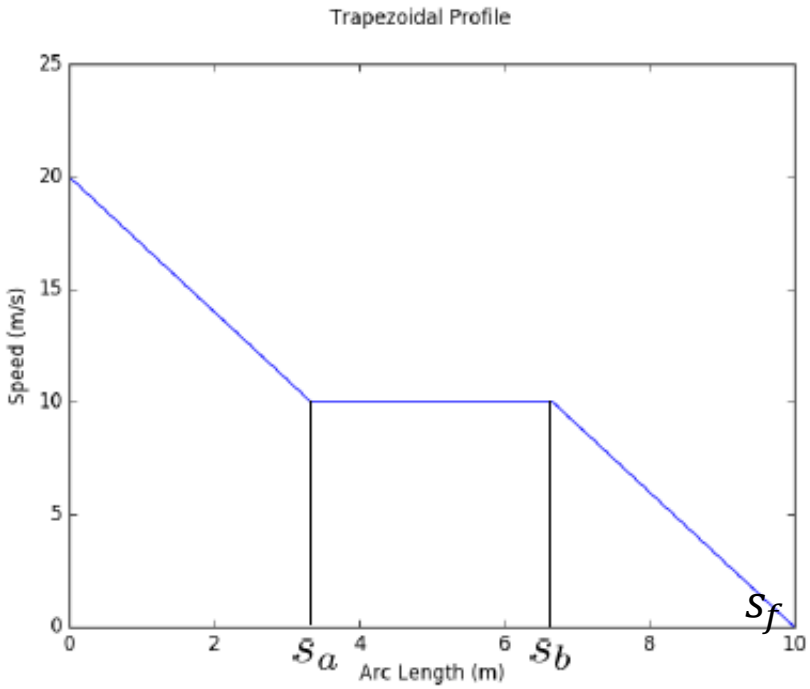
- Given path length s and initial/final speeds v_0, v_f, compute required acceleration a:
  - $a = \frac{v_f^2 - v_0^2}{2s}$
  - Proof: consider a robot with starting speed v_0 and constant acceleration a. After time t, it reaches speed of $v_0 + at = v_t$, and travels a distance of $s = \frac{(v_0 + v_t)}{2} t = \frac{v_t^2 - v_0^2}{2a}$.
- May clamp acceleration a to improve rider comfort.
- For each path segment, compute speed using accumulated path arc length s_i up to that point, to generate the speed profile:
  - $v_{f_i} = \sqrt{2as_i + v_0^2}$





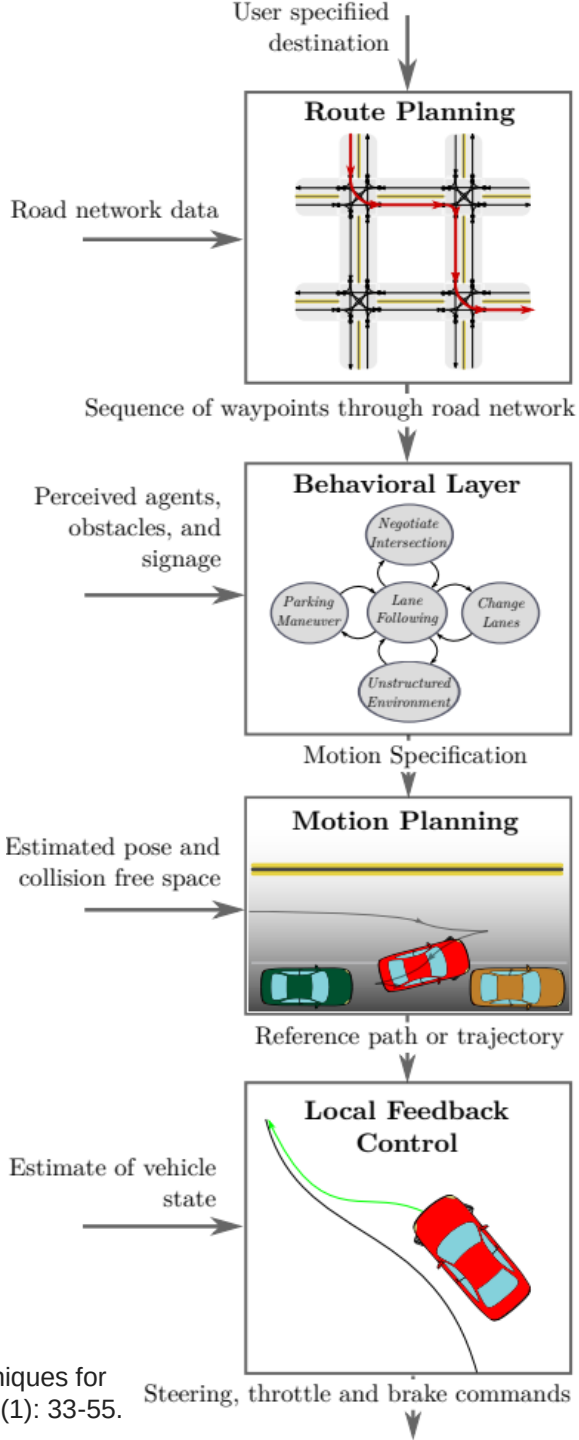
Trapezoidal Profile

- Given total path length s, initial/final speeds v_0, v_f, and transit speed v_t.
- For 1st segment $s_i \leq s_a$:
  - Compute path arc length: $s_a = \frac{v_t^2 - v_0^2}{2a_0}$
  - Then compute speed profile: $v_{fi} = \sqrt{2a_0 s_i + v_0^2}$
- For 2nd segment $s_a \leq s_i \leq s_b$:
  - Speed is constant $v_{fi} = v_t$
- For 3rd segment $s_b \leq s_i \leq s_f$,
  - Compute path arc length: $s_f - s_b = \frac{0 - v_t^2}{2a_0}$
  - Then compute speed profile: $v_{fi} = \sqrt{2a_0(s_i - s_b) + v_t^2}$
- (Coursera MOOC contains typos here.)



Outline

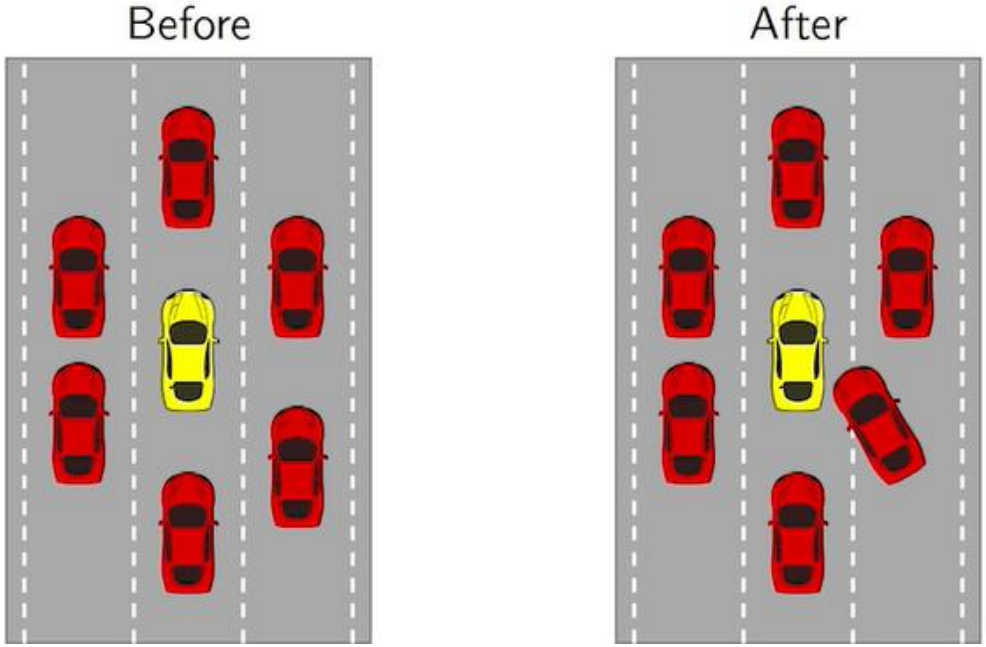
- Route planning
- Behavior planning
- Motion Planning
- Responsibility-Sensitive Safety (RSS)



Paden B, Čáp M, Yong S Z, et al. A survey of motion planning and control techniques for self-driving urban vehicles[J]. IEEE Transactions on intelligent vehicles, 2016, 1(1): 33-55.

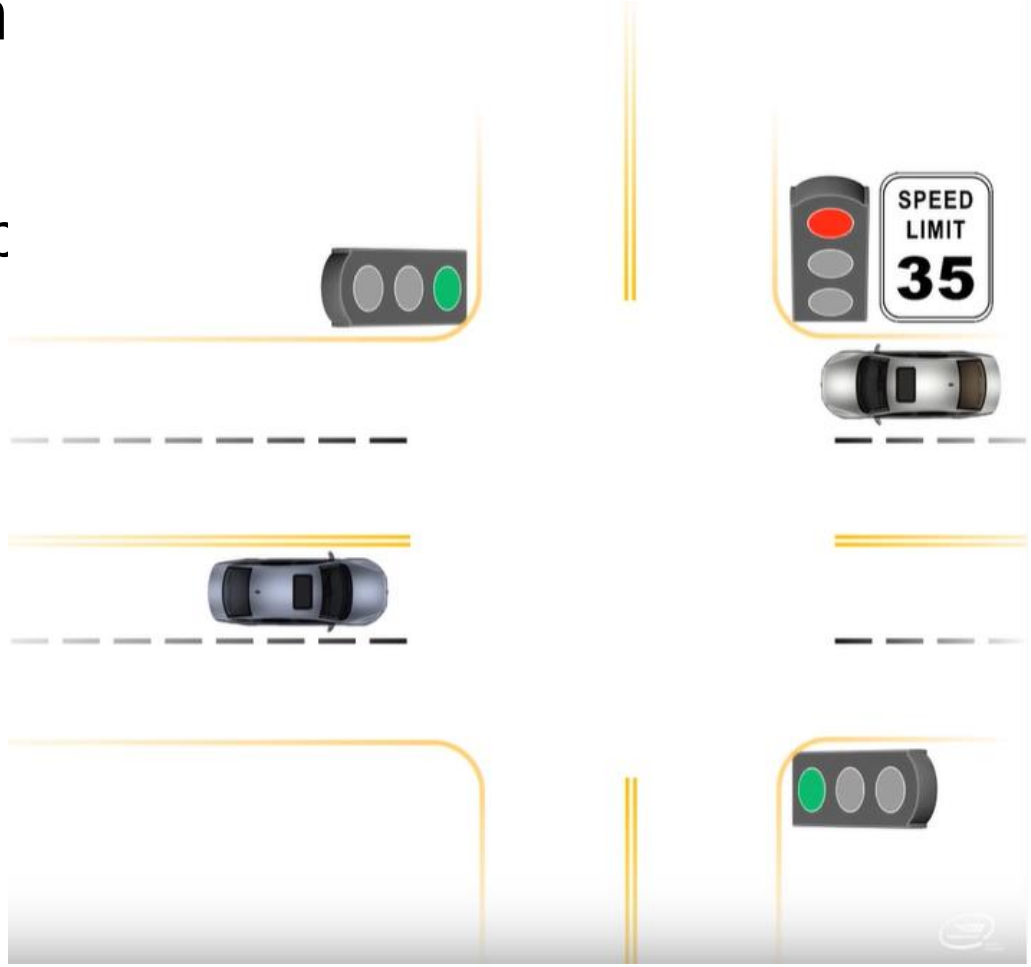
Avoid collisions at all cost?

- But absolute safety is impossible.
- In the scenario below, the AV (yellow car in the center lane) can do nothing to ensure absolute safety.
  - If the red car swerves into the AV, collision cannot be prevented
- To avoid this scenario, should the AV never drive in the center lane? Or should it never leave the garage?
  - Avoiding collisions at all cost leads to a useless system.



Explicit Traffic Rules

- Hard rules set limits on vehicle operation. Examples:
  - Come to complete stop at red lights
  - Don’t cross a double-yellow line
  - Obey posted speed limits
  - Yield to other road users when signaled
  - …
- These rules can be programmed into the AV.



Implicit Traffic Rules

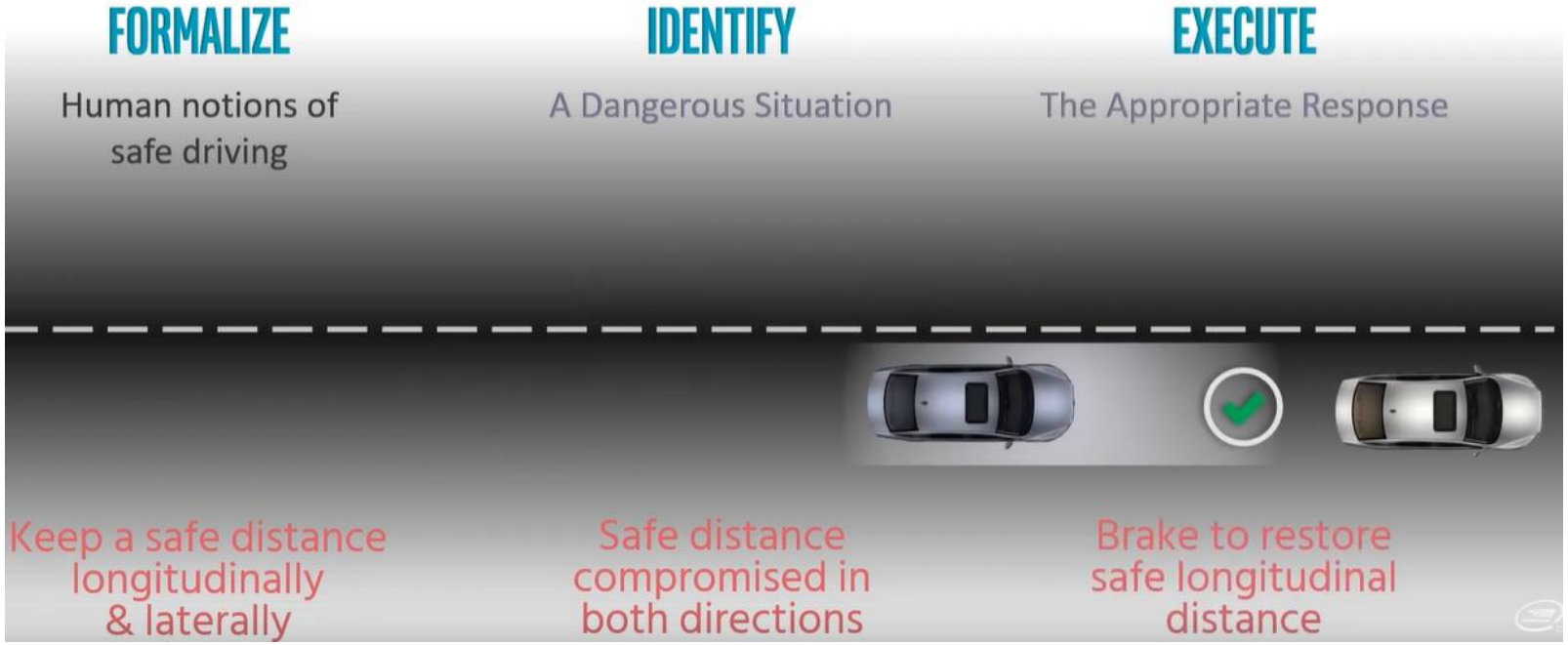
- A general set of principles applied by the human driver. They are flexible and culturally dependent.
  - Keep a safe distance from the car in front of you.
  - Drive cautiously under limited visibility.
  - Don’t drive slowly in the fast lane.
  - Don’t cut off other drivers.
  - …
- How to formalize and program these rules into the AV?

Responsibility-Sensitive Safety (RSS)

- RSS formalizes human notions of safe driving into a verifiable model with logically provable rules, defines appropriate responses, and ensures that only safe decisions are made by the AV, and clearly assigns blame/responsibility in case of accidents.
- Goal: An AV should never be responsible for accidents, meaning:
  - It should never cause accidents
  - It should properly respond to mistakes of other drivers
- RSS is a mathematical, interpretable model, formalizing the implicit rules (common sense) of
  - What is a dangerous situation?
  - What is the proper response to a dangerous situation?
  - Who is responsible for an accident?
- RSS: Safety Assurance for Automated Vehicles https://www.youtube.com/watch?v=EceAB6TUYzo
  - Shalev-Shwartz S, Shammah S, Shashua A. On a formal model of safe and scalable self-driving cars[J]. arXiv preprint arXiv:1708.06374, 2017.

Five Rules of RSS

- 1. Do not hit the car in front (safe longitudinal distance. See figure below for example)
- 2. Do not cut-in recklessly (safe lateral distance)
- 3. Right-of-Way is given, not taken.
- 4. Be careful in areas with limited visibility
- 5. If you can avoid a crash without causing another, you must

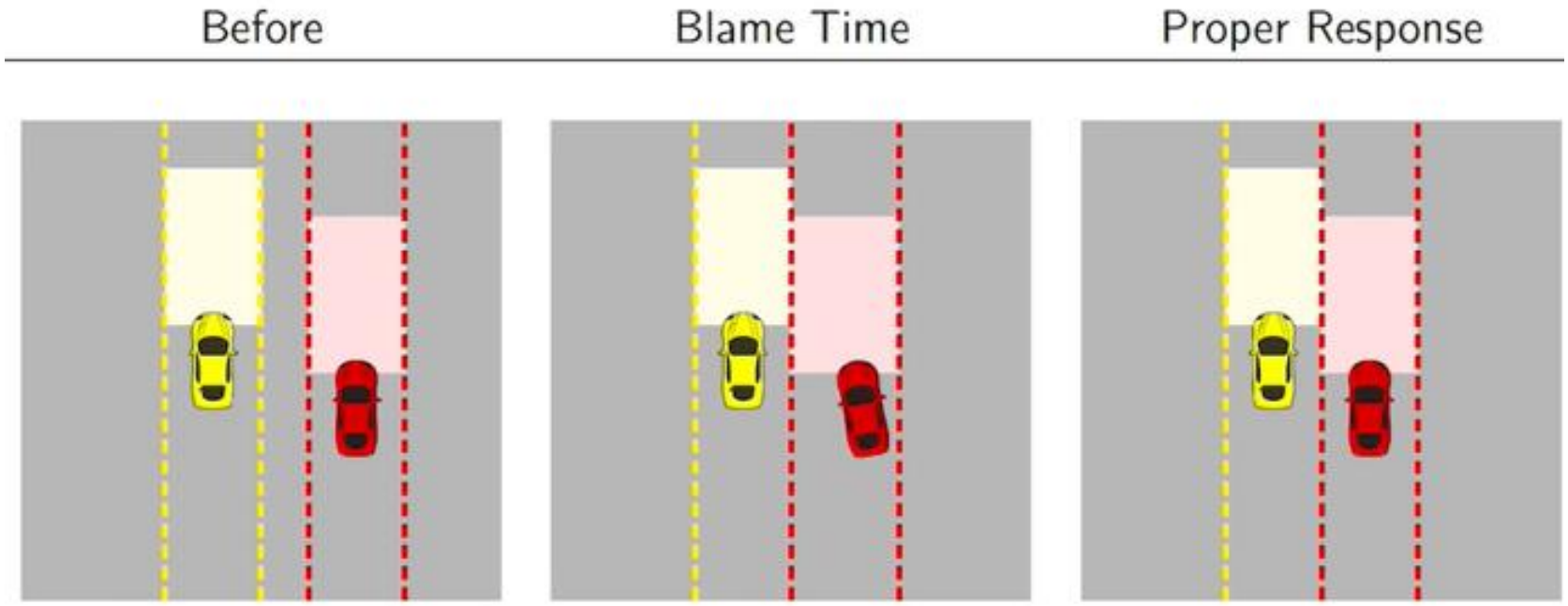


RSS: High-Level Description

- Define safe distance (longitudinal).
- A situation is dangerous if it is non-safe longitudinally.
- Blame time: the first moment in which the situation becomes dangerous
- Proper response:
  - If the longitudinal distance becomes non-safe, brake longitudinally.
- Responsibility:
  - We prove that a collision can only occur if one of the agents did not respond properly.
  - The responsibility is on the agent(s) that did not respond properly.

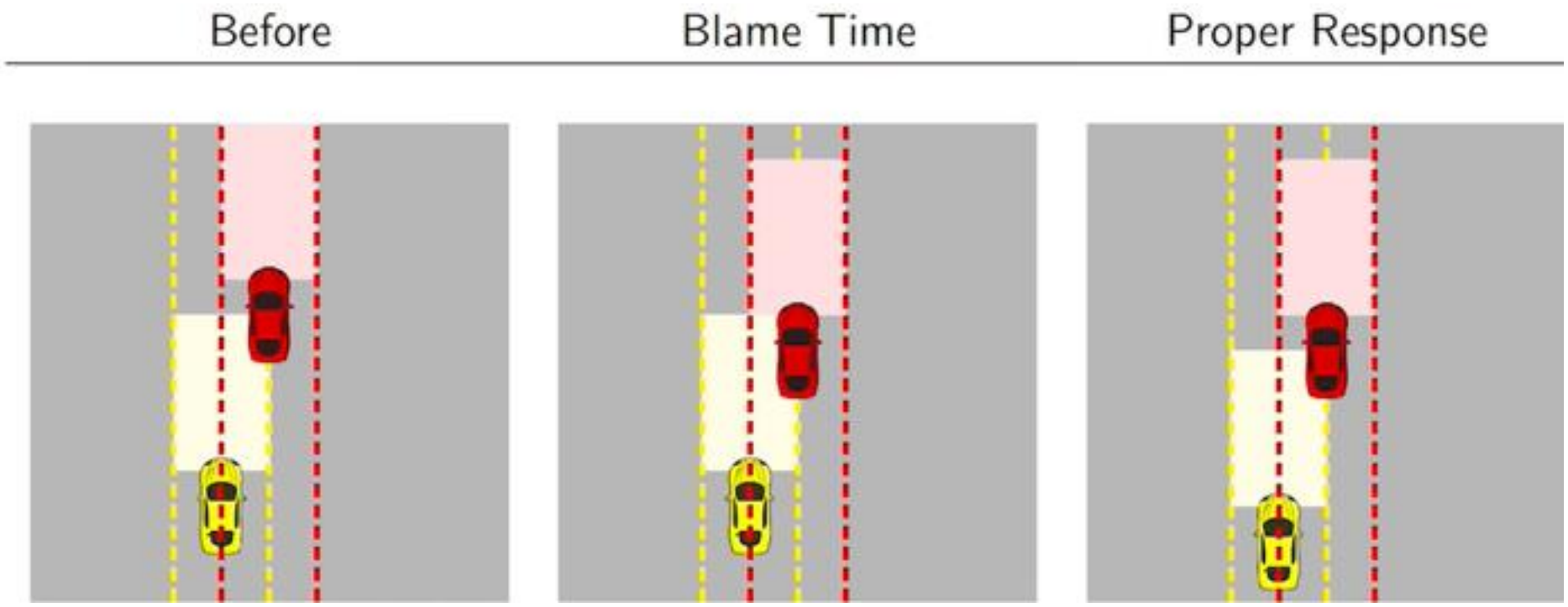
Non-safe Lateral Maneuver

- Proper response: red should brake laterally,
  - since red is cutting into yellow's lane.
  - "Brake laterally" means "brake while steer to the right direction".



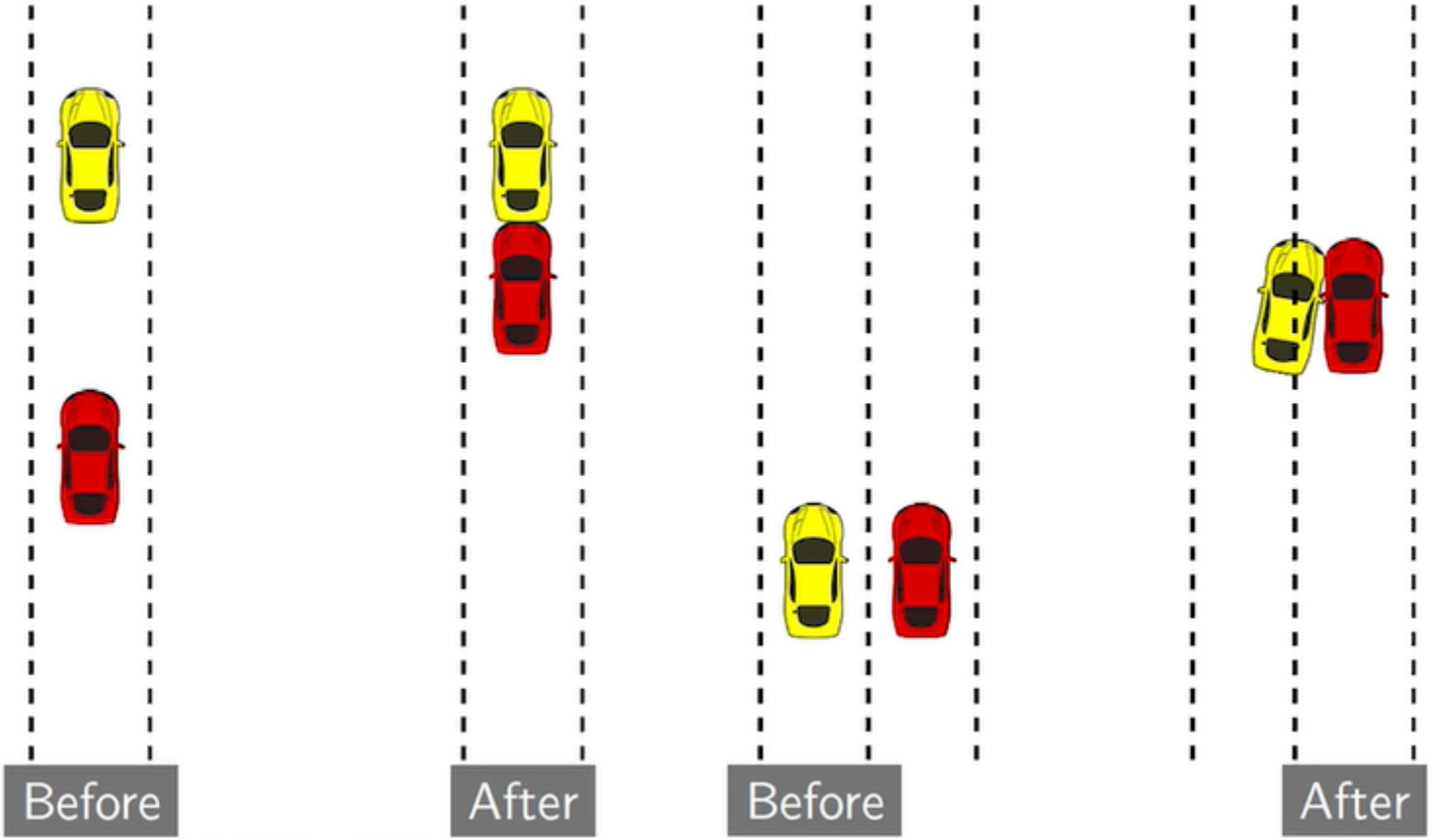
Non-Safe Longitudinal Maneuver

- Proper response: yellow should brake,
  - Since yellow is driving too fast and getting too close.



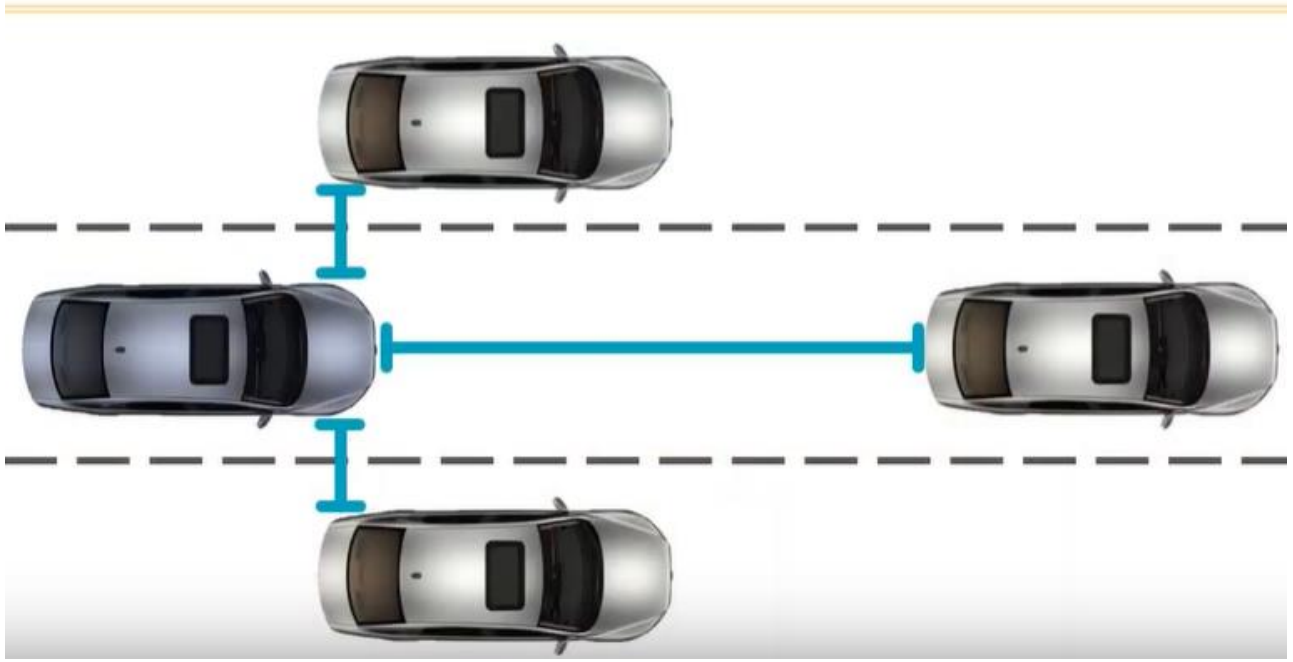
Responsibility

- Hit from behind
  - Yellow is not responsible for the accident.
- Side hit
  - Red is not responsible for the accident.



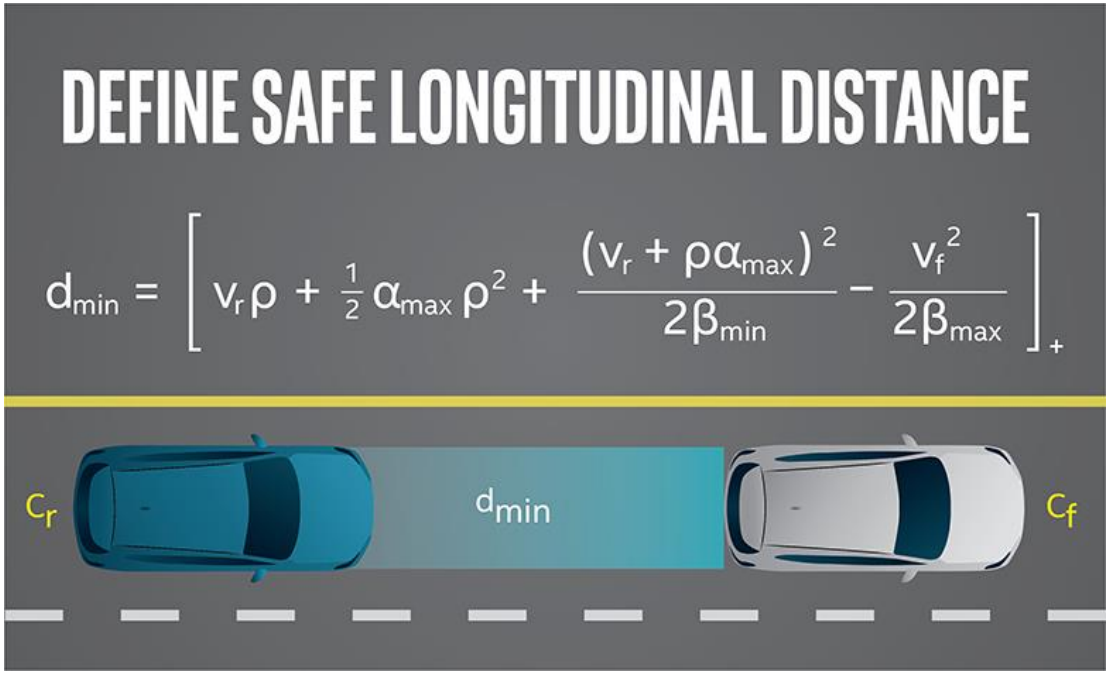
Keep Safe Distances

- The AV (blue car) should keep safe longitudinal and lateral distances from other cars
  - If the front car slams on the brakes, how much longitudinal distance does it need to avoid a rear-end collision?
  - If the left or right car suddenly swerves, how much lateral distance does it need to avoid a side collision?



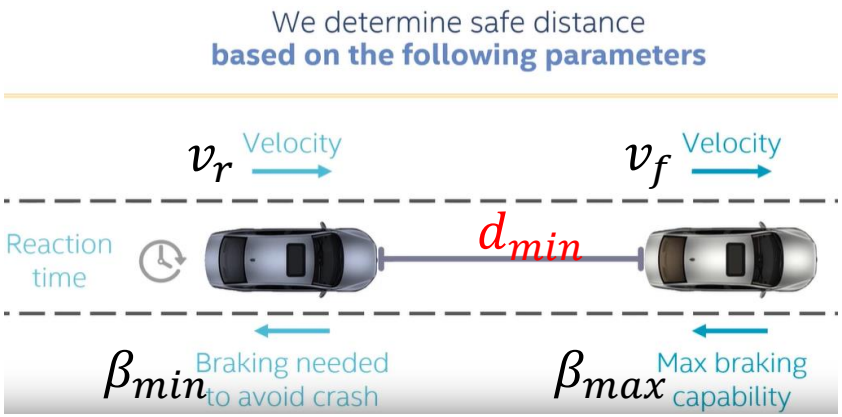
Rule 1: Safe Longitudinal Distance

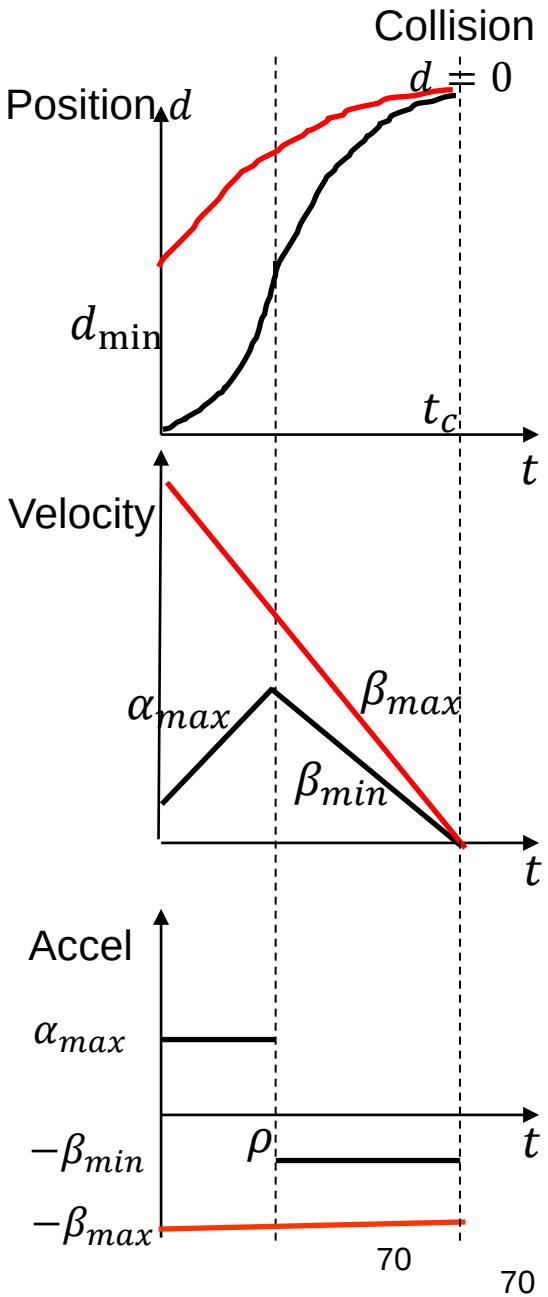
- Consider two cars travelling in the same direction: front car c_f and rear car c_r. In order to ensure that c_r will never hit c_f from behind, it is the responsibility of c_r to keep a *safe longitudinal distance* d_{min} from c_f, which should be large enough so that c_r will not hit c_f. The worst-case situation is that c_f will suddenly brake hard, it will take c_r some response time ρ (sensing and reaction delay) to figure this out and to start braking as well, and then both cars will decelerate. Assuming known parameters of β_{max}: maximum deceleration rate due to braking for c_f; β_{min}:minimum deceleration rate for c_r; α_{max}:maximum acceleration rate for c_r, we can derive d_{min} as:



Safe Longitudinal Distance: the Proof

- Red (black) lines indicate accel/velocity/position of front (rear) car.
- The initial distance between front car c_f and rear car c_r is d_0. Assuming $\beta_{\min} \le \beta_{\max}$, their distance will decrease monotonically with time. To avoid collision, the minimum initial distance d_{min} can be obtained by assuming both cars will come to full stop at some time t_c with distance $d = 0$.
- Time 0: front car and rear car have initial speeds v_f and v_r, respectively
- Time interval $[0, \rho]$: Front car applies brake with constant deceleration $\beta_{\max}$; rear car accelerates with constant acceleration α_{max}.
- Time ρ: rear car reaches speed of $v_{\rho,max} = v_r + \rho\alpha_{max}$, and travels a distance of $(v_r + \frac{1}{2}\rho\alpha_{max})\rho = v_r\rho + \frac{1}{2}\alpha_{max}\rho^2$.
- Time interval $[\rho, t_c]$: Rear car applies brake with constant deceleration $\beta_{\min}$, travels a distance of $\frac{v_{\rho,max}^2}{2\beta_{min}} = \frac{(v_r+\rho\alpha_{max})^2}{2\beta_{min}}$.
- Time interval $[0, t_c]$: Rear car travels a distance of $\frac{v_f^2}{2\beta_{max}}$ (average speed $\frac{v_f}{2}$ times time-to-stop $\frac{v_f}{\beta_{max}}$).
- Time t_c: both cars come to full stop with distance $d = 0$.
- Min initial distance $d_{\min}$ =distance traveled by rear car before full stop - distance traveled by front car before full stop= $\left[v_r\rho + \frac{1}{2}\alpha_{max}\rho^2 + \frac{(v_r+\rho\alpha_{max})^2}{2\beta_{min}} - \frac{v_f^2}{2\beta_{max}}\right]_+$

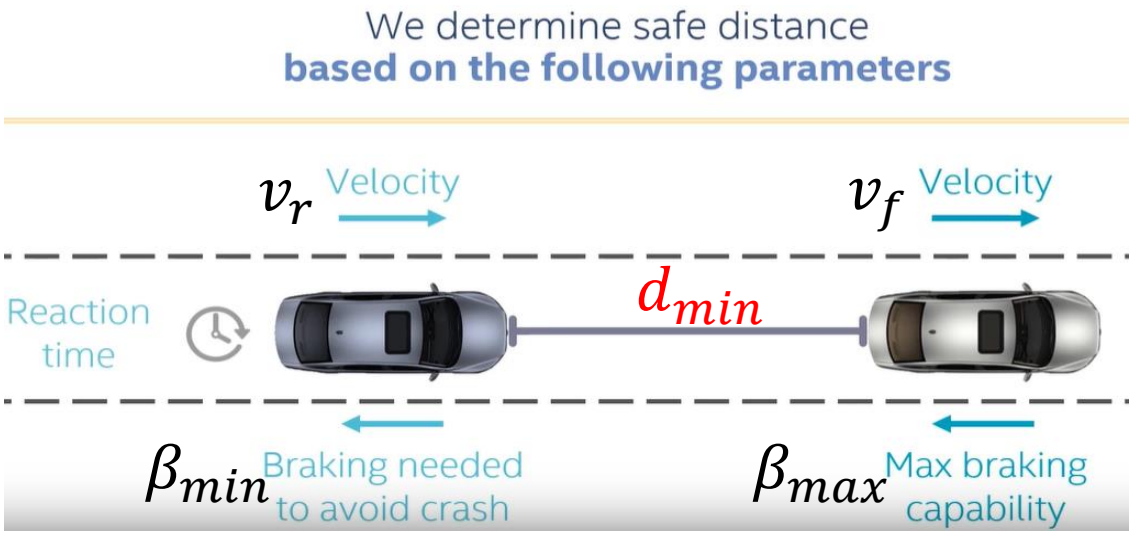




70

Safety vs. Efficiency

- RSS allows to formally define the desired balance of safety and efficiency of AVs on the road.
  - e.g., d_{min} is larger if we assume a large deceleration rate β_{max} for the front car; or larger acceleration rate α_{max} or smaller deceleration rate β_{min} for the rear car. So the AV will drive more conservatively, resulting in lower efficiency, and vice versa
- These parameters should be determined by regulatory authorities working with the car makers.
  - E.g., NHTSA (National Highway Traffic Safety Administration) in USA.



$$d_{min} = \left[v_r \rho + \frac{1}{2} \alpha_{max} \rho^2 + \frac{(v_r + \rho \alpha_{max})^2}{2\beta_{min}} - \frac{v_f^2}{2\beta_{max}} \right]_+$$

Proper Response

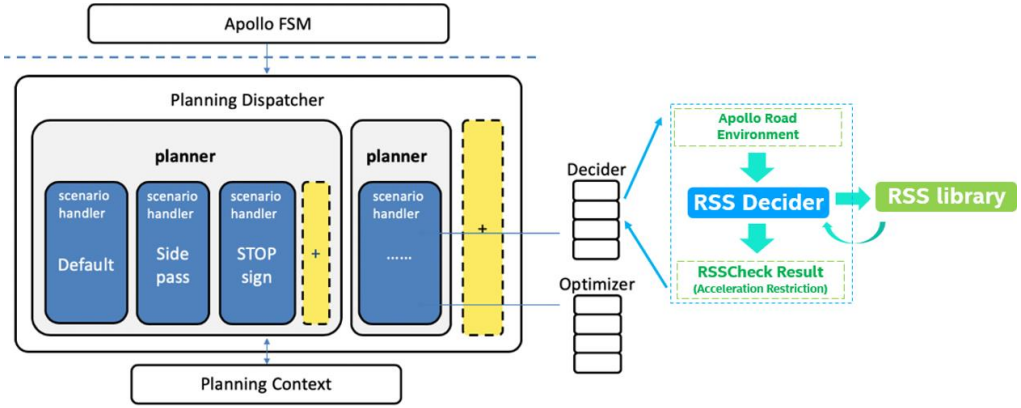
- The moment the distance between the two cars is less than d_{min}, the AV will perform the proper response: after a response time ρ, apply braking of at least β_{min} until a safe following distance is restored or until the vehicle comes to a complete stop.
- RSS can be a proactive safety mechanism that improves Automatic Emergency Braking (AEB). Called Automatic Preventive Braking (APB), it determines the moment when a vehicle enters a dangerous situation, then uses comfortable, subtle braking (β_{min}) to help return the vehicle to a safer position without waiting for an imminent collision to engage maximum braking force. This preventive approach would provide a stopping distance buffer that could prevent a chain reaction of braking and swerving should an emergency stop occur.

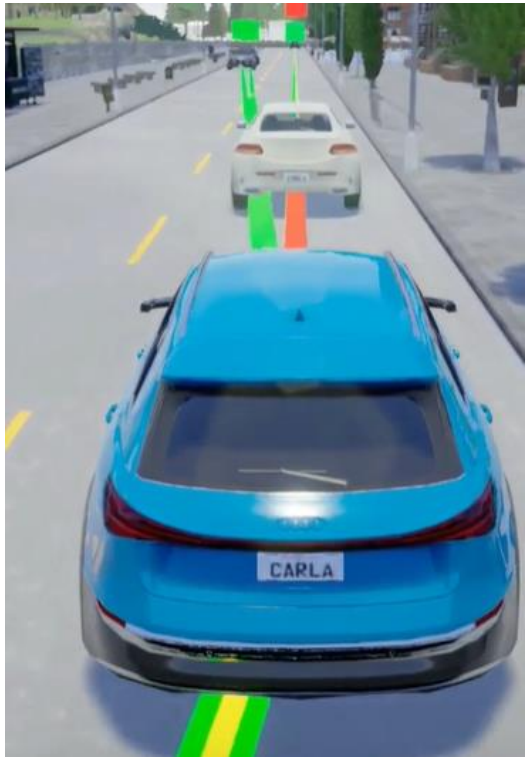
Other RSS Rules

- Rule 2: Safe Lateral Distance
  - Safe lateral distance enables AVs to be aware when their lateral safety may be compromised by unsafe drivers turning into their lanes.
- Rule 3. Right of way is given, not taken
  - On well-marked roads, the right of way is clear. Lane lines, signs, and traffic lights establish priorities for routes as they intersect one another. However, there are other times when the right of way is less clear, and human drivers must negotiate with one another.
- Rule 4. Be cautious in areas with limited visibility
  - Drivers must proceed cautiously, especially as they approach crosswalks or pass cars parked along the street
- RULE 5. If the vehicle can avoid a crash without causing another one, it must
  - e.g., if boxes fall off the front car, and the next lane is free, the following car can take evasive action to avoid the accident.

Integration into Apollo and CARLA

- RSS is open-source as a C++ library. It has been integrated into Baidu Apollo's planner module, and the driving simulator CARLA
- RSS safety sensor in CARLA
  - Use the opensource library to evaluate if a driving situations is safe or unsafe according to RSS. In that regard, a driving situation is composed of the ego vehicle (here in this video the ego vehicle is the one focused by the camera), and another traffic participant (e.g. a leading vehicle). The sensor evaluates longitudinal and lateral conflicts, but does not yet cover intersection conflicts.
  - The results are highlighted via green (all safe), yellow (only lateral or longitudinal unsafe) and red lines (dangerous situation that requires to a counter measure according to RSS).
  - https://www.youtube.com/watch?v=UxKPXPT2T8Q





NVIDIA Safety Force Field (SFF)

- NVIDIA proposed SFF in 2019:
  - “SFF is a robust driving policy that analyzes and predicts the vehicle’s environment. It determines a set of acceptable actions to protect the vehicle, as well as others on the road. These actions won’t create, escalate, or contribute to an unsafe situation, and include the measures necessary to mitigate harmful scenarios.”
- Criticized by Mobileye to be a close replica of RSS
- Introducing NVIDIA Safety Force Field
  - https://www.youtube.com/watch?v=R0H77yZSEUk&feature=emb_logo

https://www.nvidia.com/en-us/self-driving-cars/safety-force-field/
https://newsroom.intel.com/editorials/innovation-requires-originality

Summary

- RSS formalizes what is dangerous, what is the proper response to danger, and who is responsible for accidents.
- Soundness: it complies with the common sense of human judgement.
- Usefulness: we give 100% guarantees to never cause accidents and always properly response to dangerous situations.