Robotic Navigation and Exploration

Unit 2: Motion Planning

Min-Chun Hu anitahu@cs.nthu.edu.tw
CS, NTHU

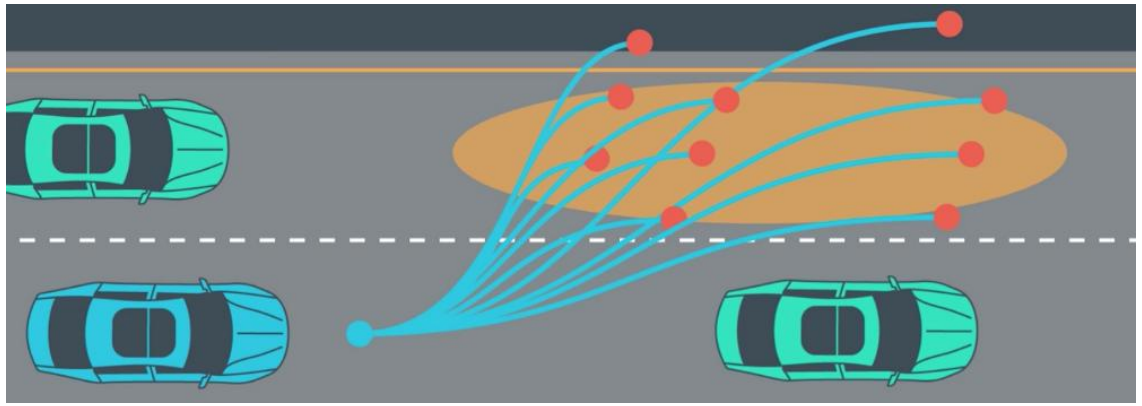
直播連結 https://www.youtube.com/@NTHURNE-l9v

Outline

- Introduction to Motion Planning
- Path Planning
  - Graph Search Based Methods
  - Sampling Based Methods
- Curve Interpolation
- Trajectory Planning
  - State Lattices

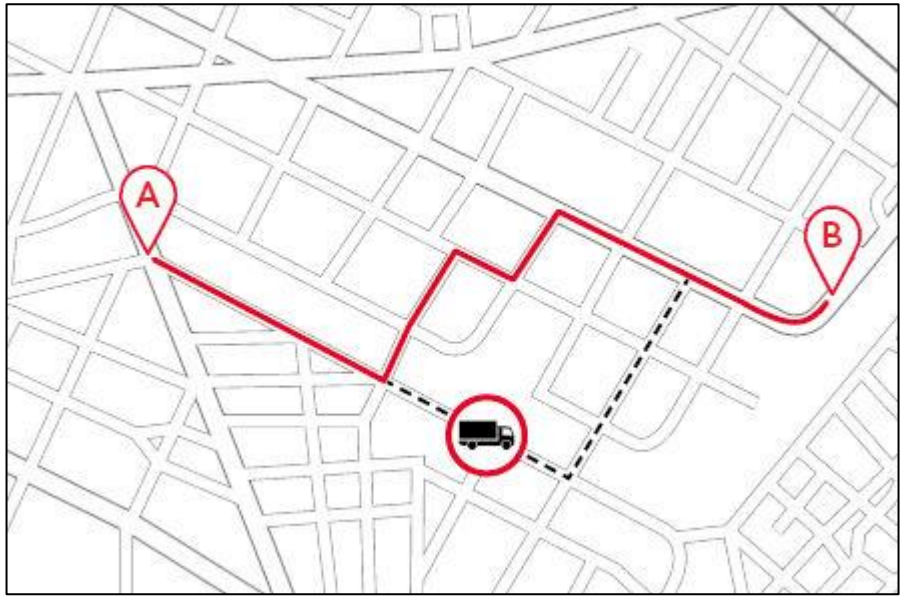
Motion Planning

- Motion planning aims to find the motion steps that moves the robot from the source to destination.
- The planned motion sequence needs to satisfy the following requirements:
  - Avoiding collision with known obstacles
  - Smooth path
  - Minimal length
  - Motion constraints (e.g. maximum speed)

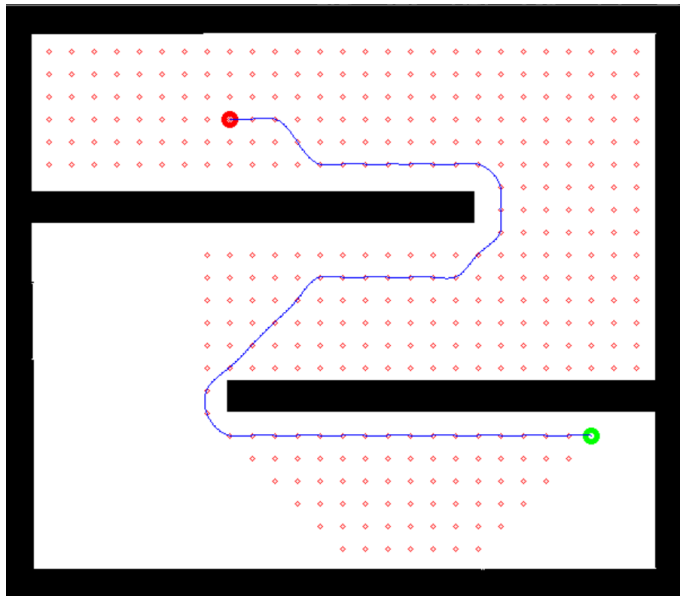


Path Planning

- Find the way-points from the source position to the target position.

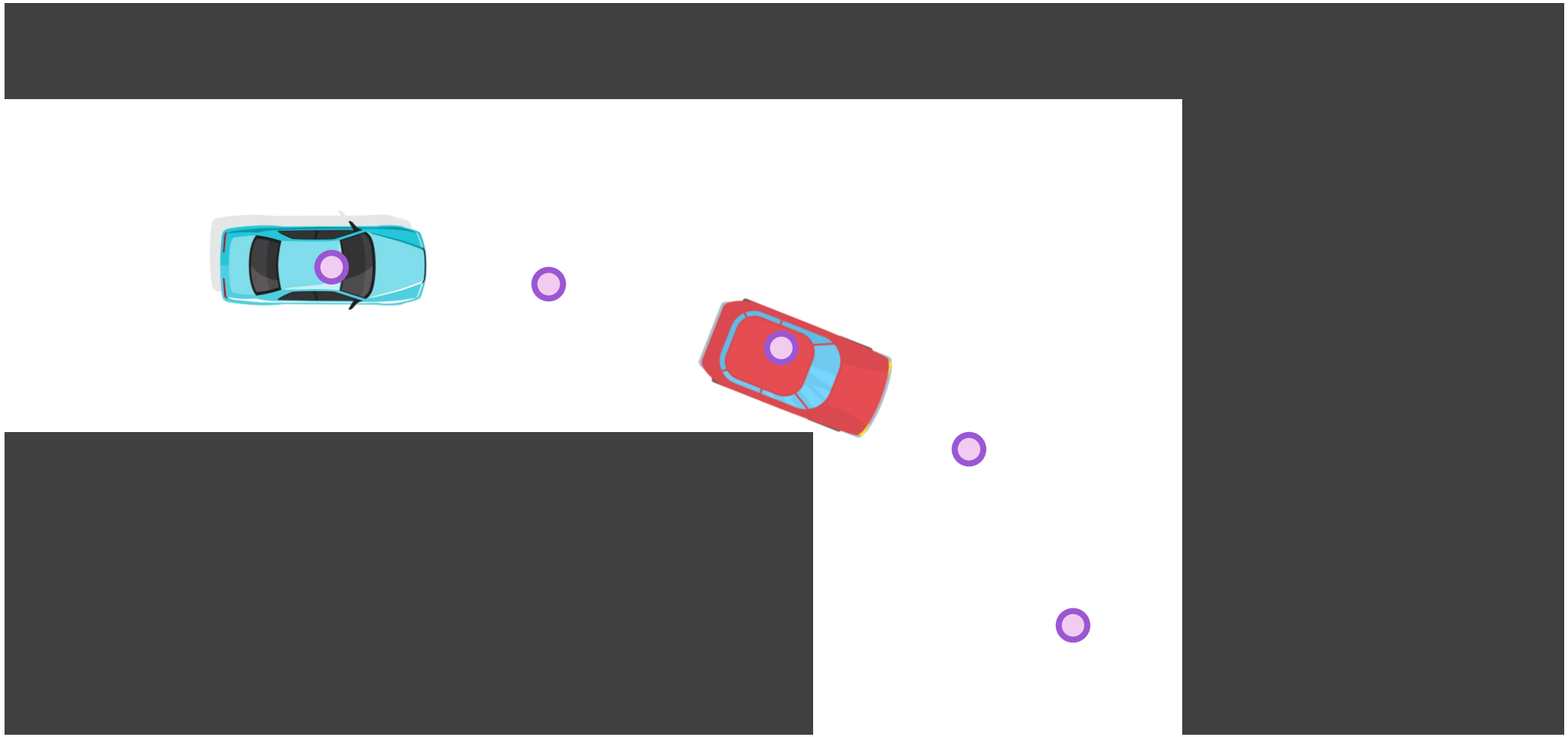


Route Planning



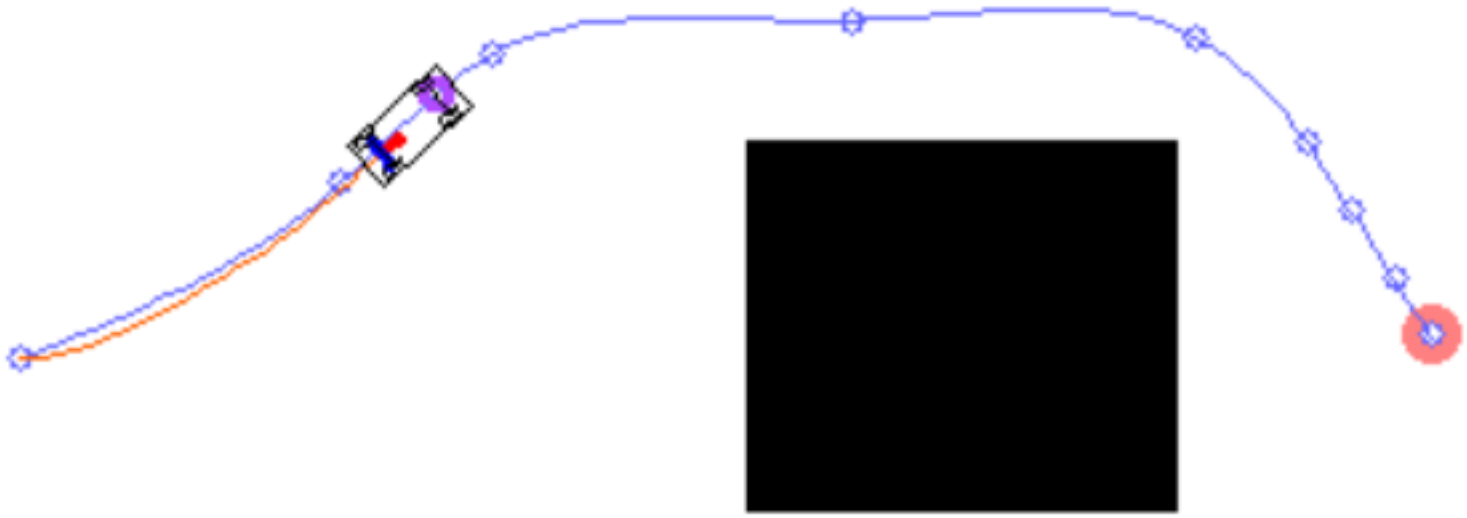
Planning in Grid Space

Path Planning

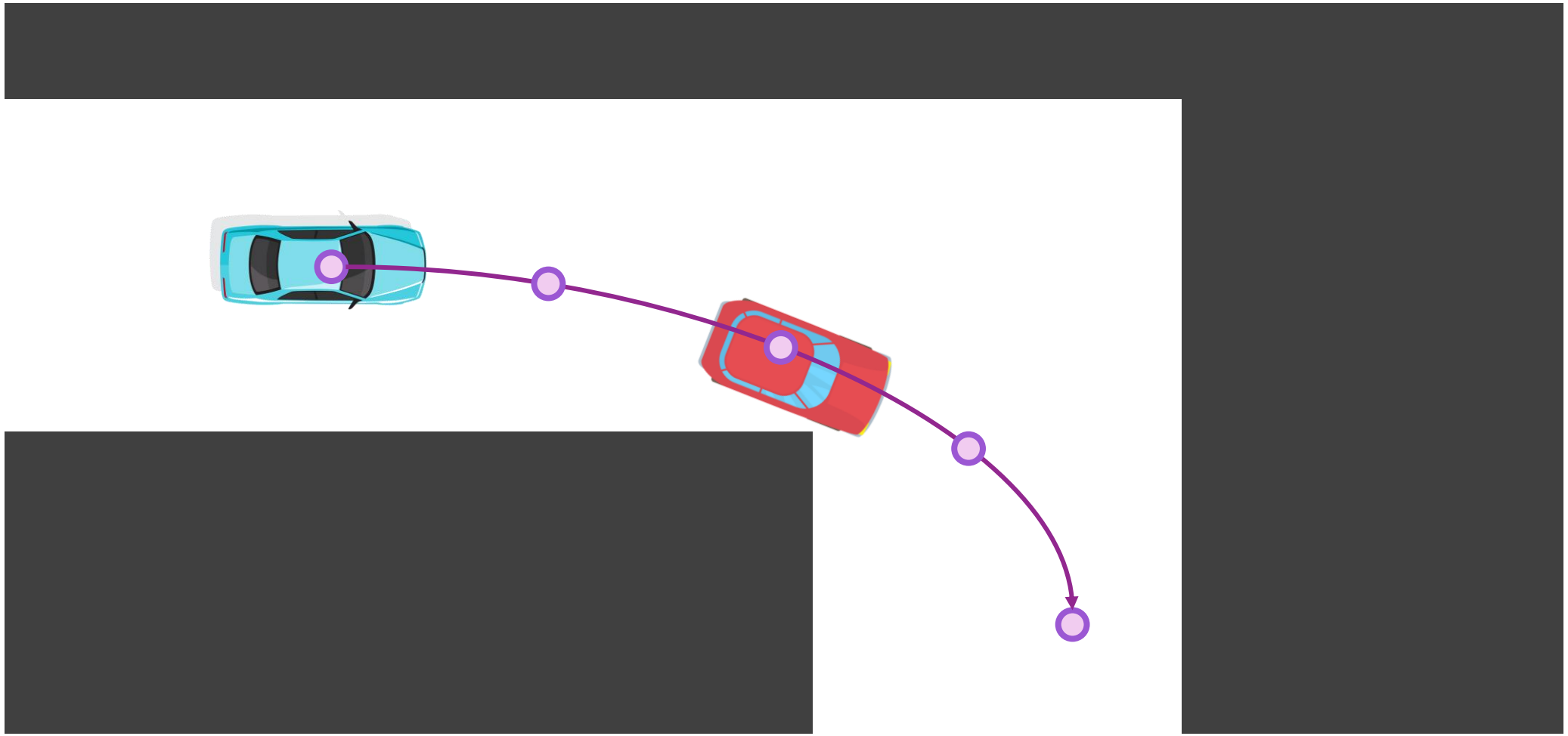


Curve Interpolation

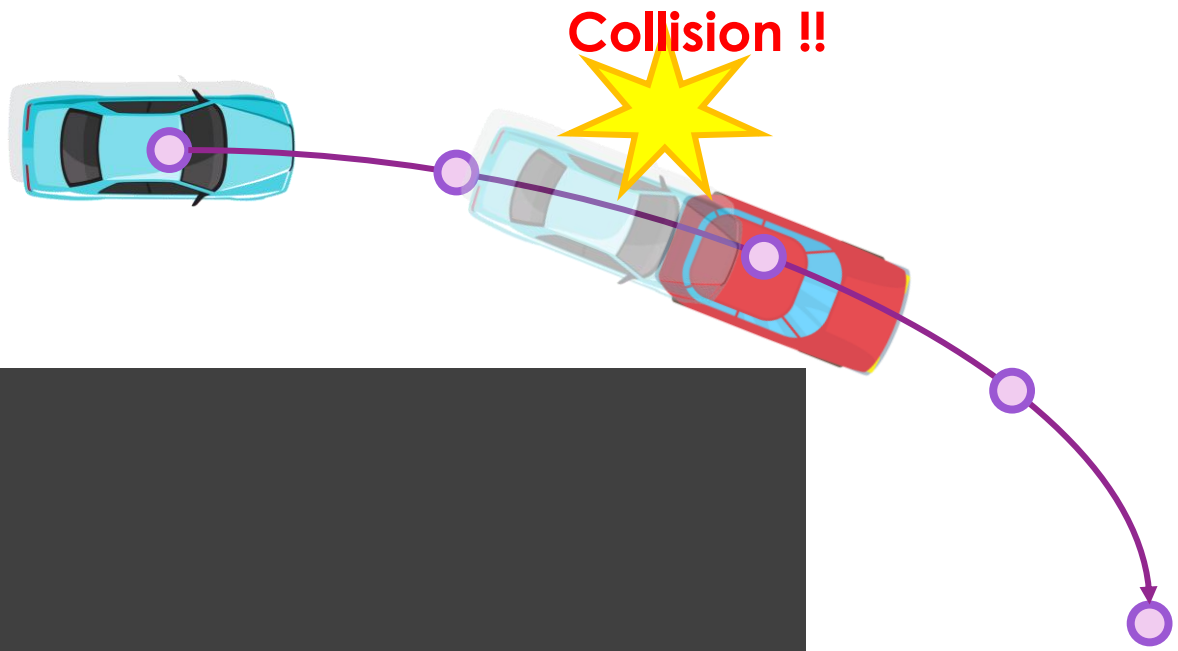
- Given the way points and a curve function, interpolate the middle point to make the path smooth.
- We can also utilize the curve function to compute the derivative information of the path.



Curve Interpolation

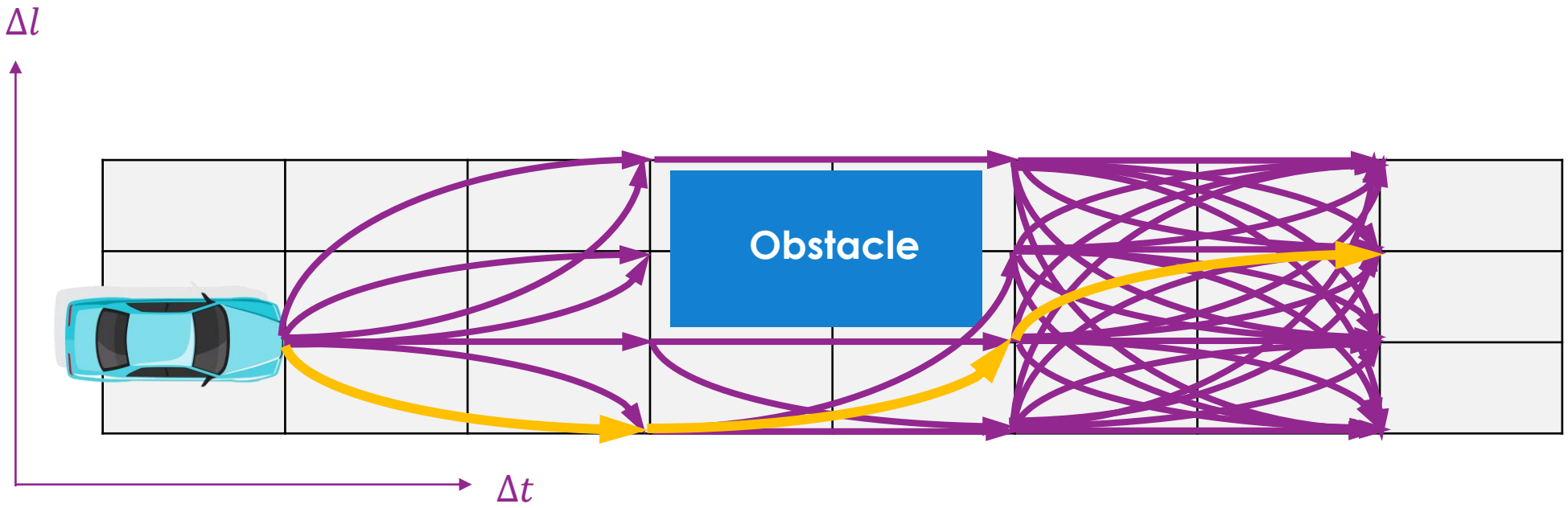


Dynamic Environment Problem



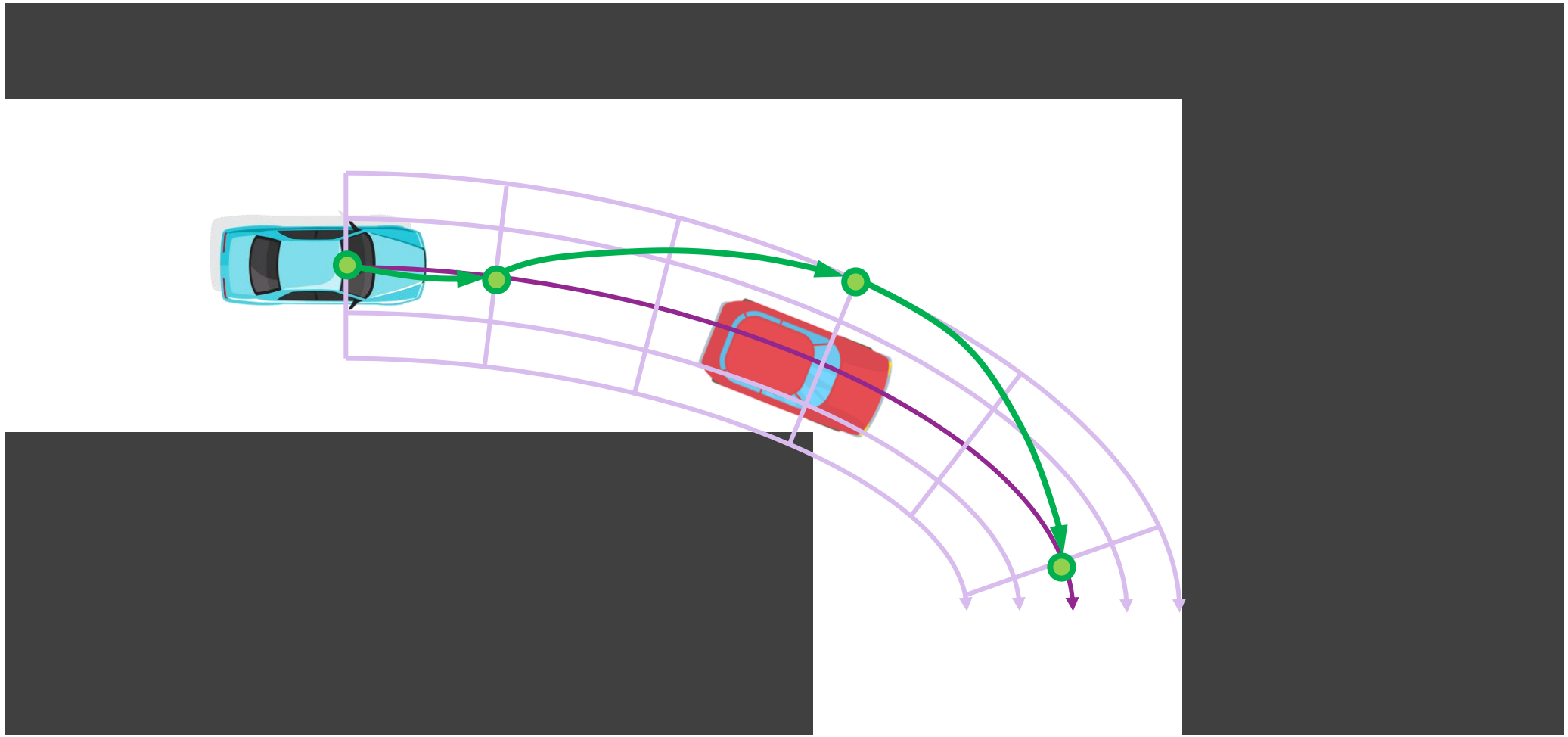
Trajectory Planning

- Plan the motion sequence that contains speed and acceleration information to handle the dynamic environment.

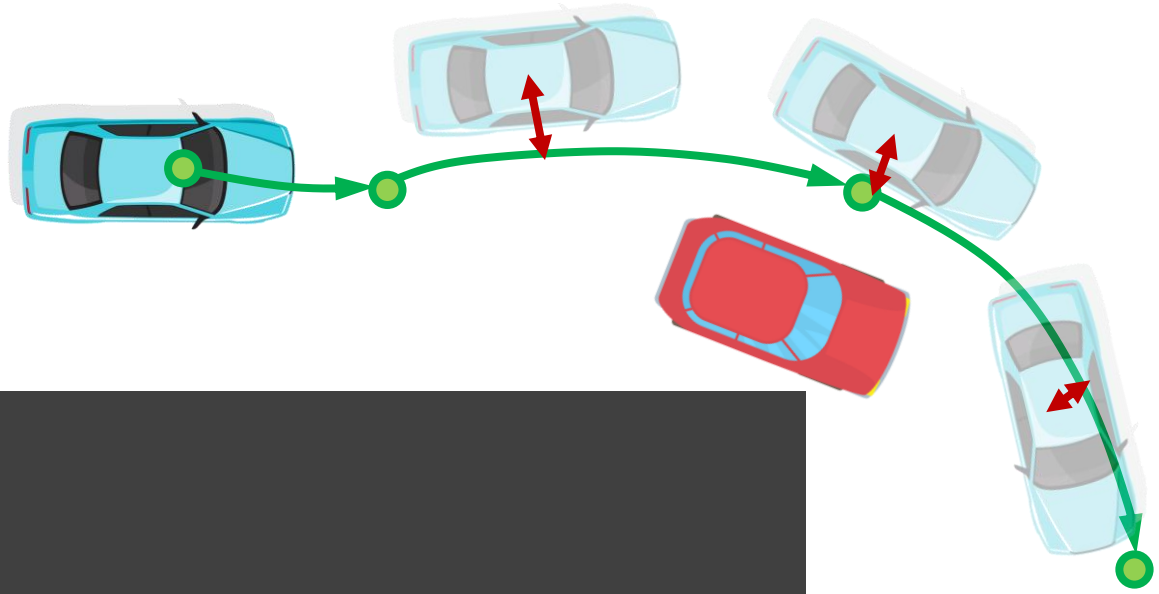


State Lattices Planning

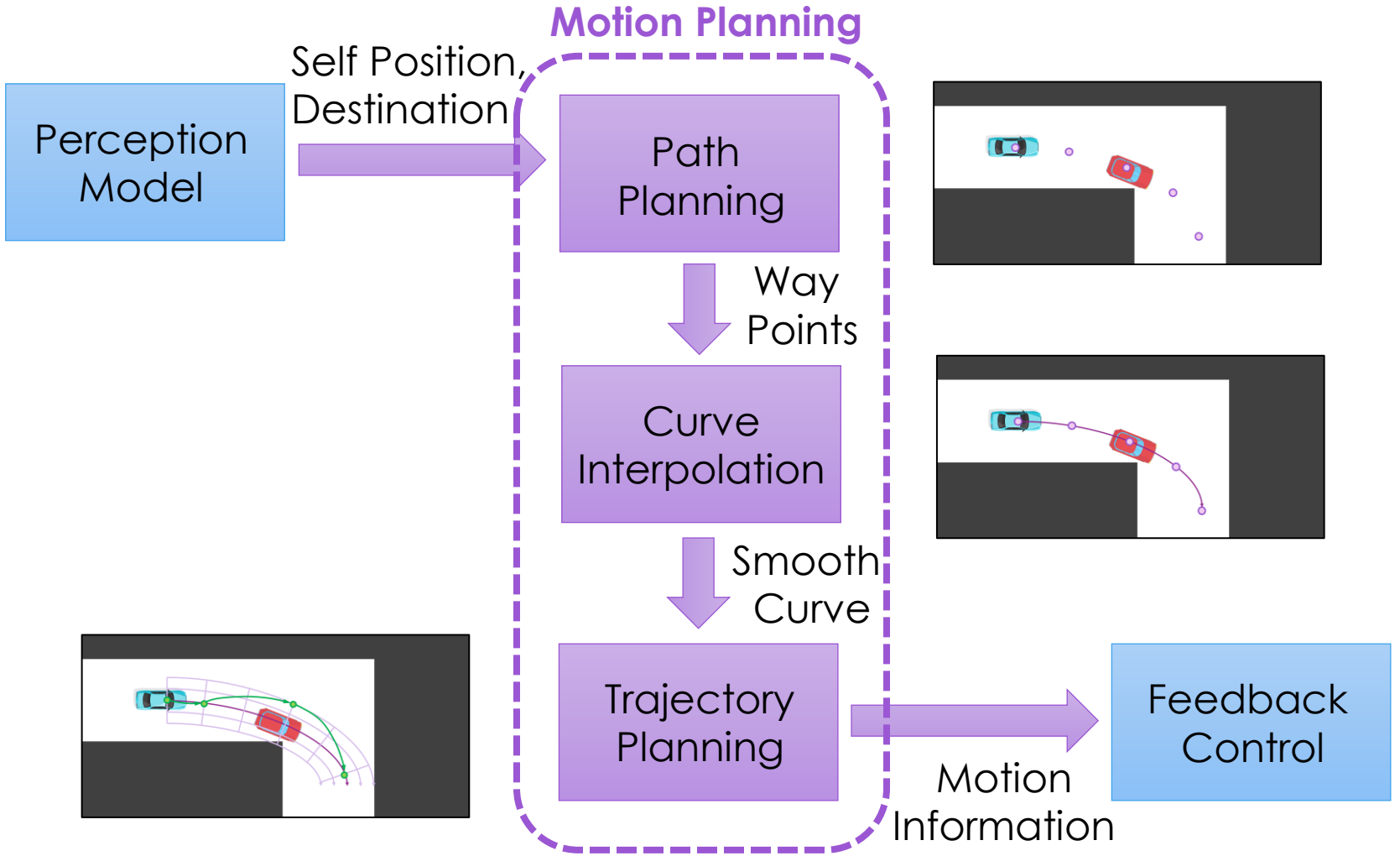
Trajectory Planning



Feedback Control



Motion Planning Flow



Path vs. Trajectory

- **Motion Planning = Path Planning + Trajectory Planning**

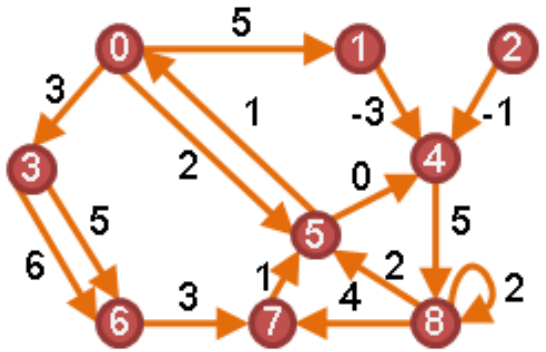
- Path Planning (Only consider the spatial information)
  - **Input:** Start/End Position $\{[x_s, y_s], [x_e, y_e]\}$
  - **Output:** Way Points $\{[x_s, y_s], [x_1, y_1], [x_2, y_2], \dots, [x_e, y_e]\}$

- Trajectory Planning (Consider both the spatial and temporal information)
  - **Inputs:** Start/End Position $\{[x_s, y_s], [x_e, y_e]\}$, Curve Function $f(x)$, Time Density Δt
  - **Outputs:** Motion information with time
    $\{[x_0, y_0, v_0, a_0, t_0], [x_1, y_1, v_1, a_1, t_1], [x_2, y_2, v_2, a_2, t_2], \dots, [x_N, y_N, v_N, a_N, t_N]\}$

Path Planning

Graph Representation

- A graph G Consists of two sets, V and E
  - G = (V, E)
  - V : a set of Vertices
  - E : a set of Edges (pair of vertices)
- Direction
  - Undirected : (v1, v2) = (v2, v1)
  - Directed : <v1, v2> ≠ <v2, v1>
- Weight
  - Graph may have weight on edges



Directed Graph with weight

Shortest Path Problem

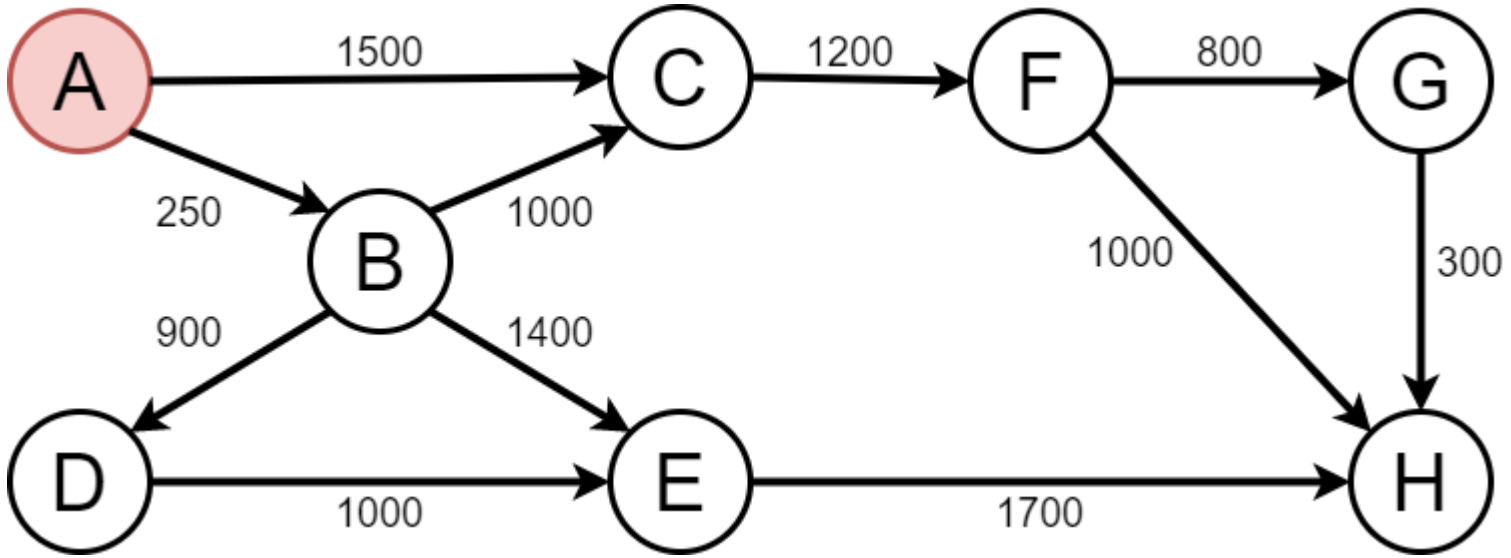
- Find a path between two vertices such that
  - Sum of the weights of its constituent edges is minimized
- Single-source shortest paths
  - Dijkstra : Non-negative weights graphs (high complexity)
  - Best-First Search (BFS) : Heuristic Greedy Search (fast but non-optimal)
  - A* : BFS with Dijkstra

Dijkstra's Algorithm

- Solves the single-source shortest path problem
  - Only with non-negative edge weight

- From starting point v
  - Select a minimal distance and unfinished vertex u
  - Update other unfinished vertices v' using the equation:
    - d(v, v') = min(d(v, u) + <u, v'>, d(v, v'))

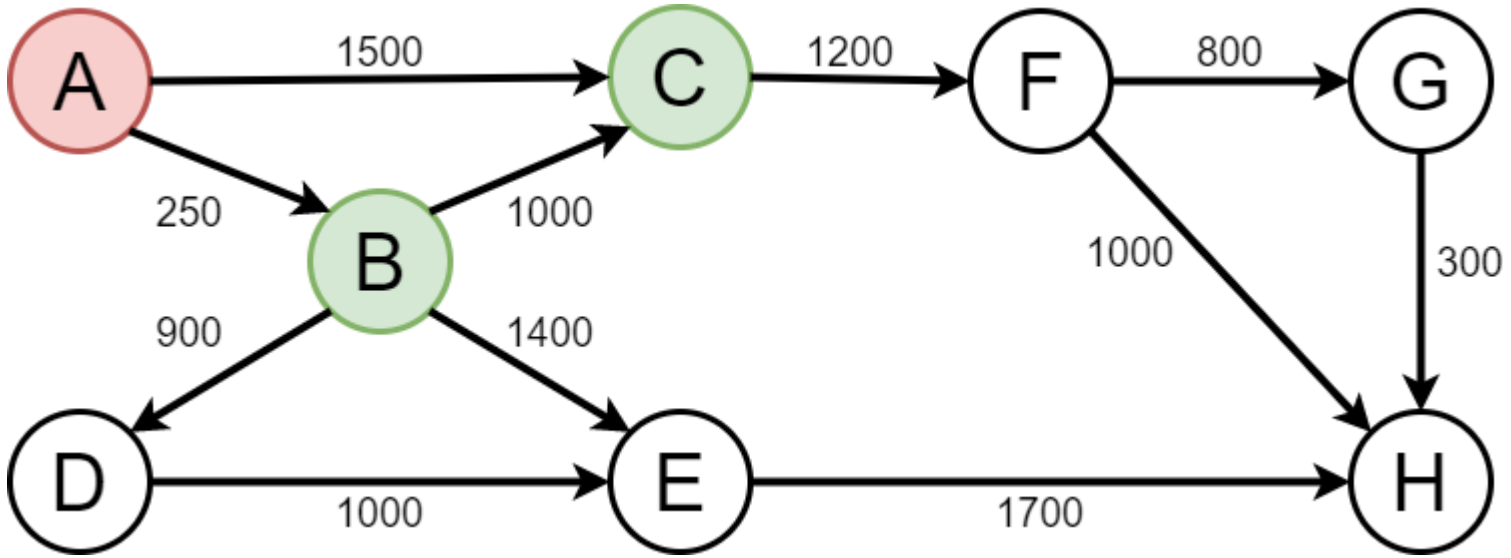
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	F	F	F	F	F	F	F
Cost	0	∞	∞	∞	∞	∞	∞	∞
Path	-1	-1	-1	-1	-1	-1	-1	-1

Finished = {A}

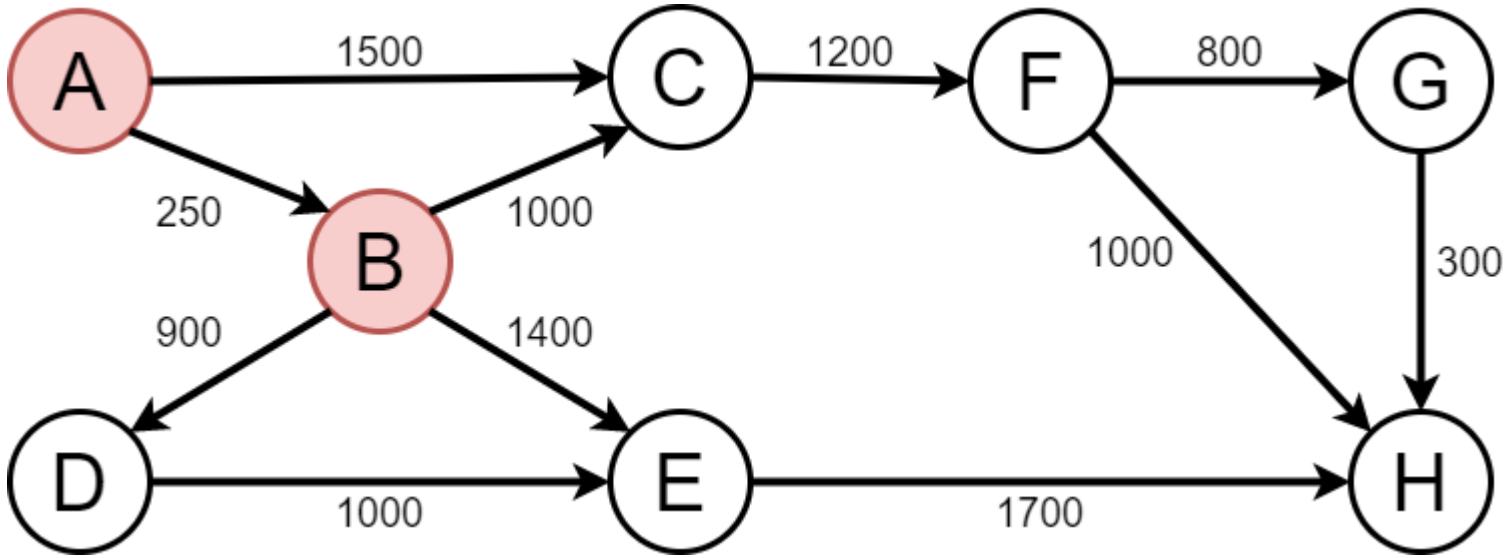
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	F	F	F	F	F	F	F
Cost	0	250	1500	∞	∞	∞	∞	∞
Path	-1	A	A	-1	-1	-1	-1	-1

Finished = {A}

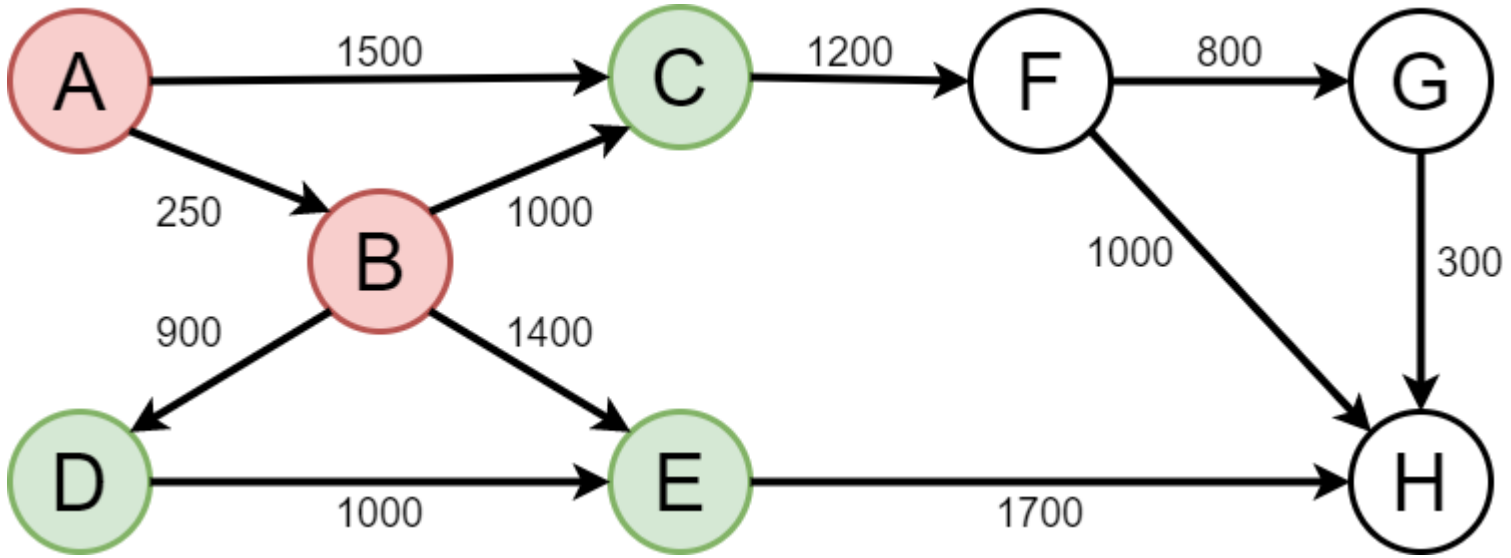
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	F	F	F	F	F	F
Cost	0	250	1500	∞	∞	∞	∞	∞
Path	-1	A	A	-1	-1	-1	-1	-1

Finished = {A, B}

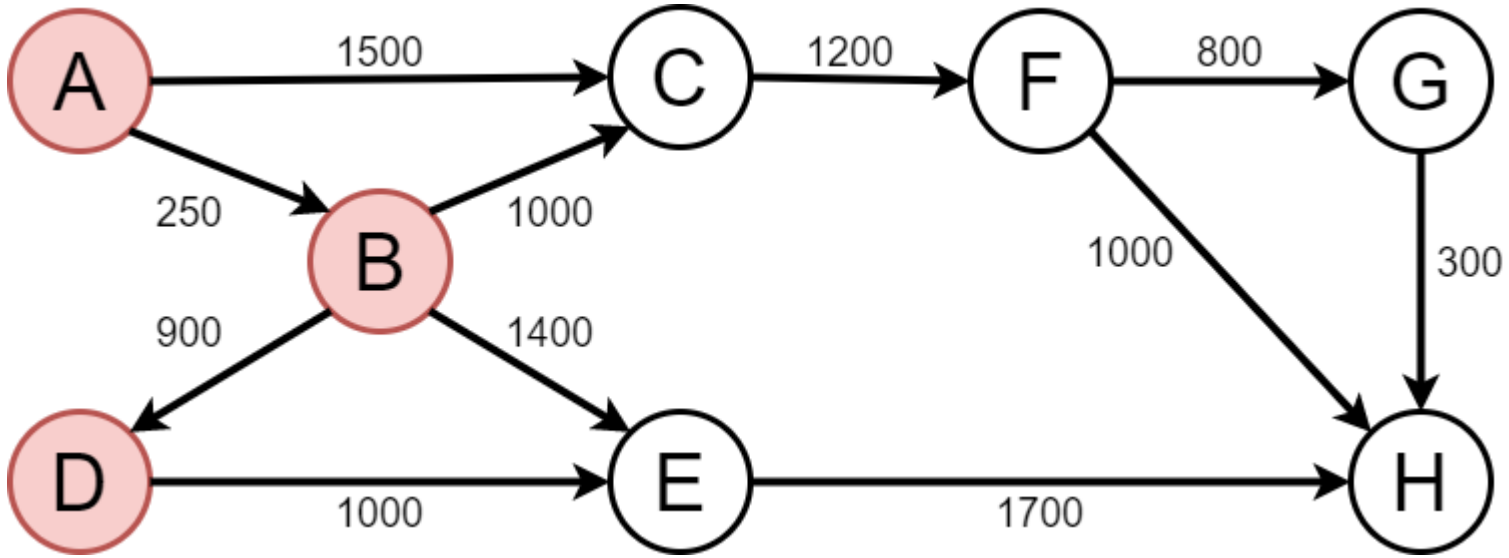
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	F	F	F	F	F	F
Cost	0	250	1250	1150	1650	∞	∞	∞
Path	-1	A	B	B	B	-1	-1	-1

Finished = {A, B}

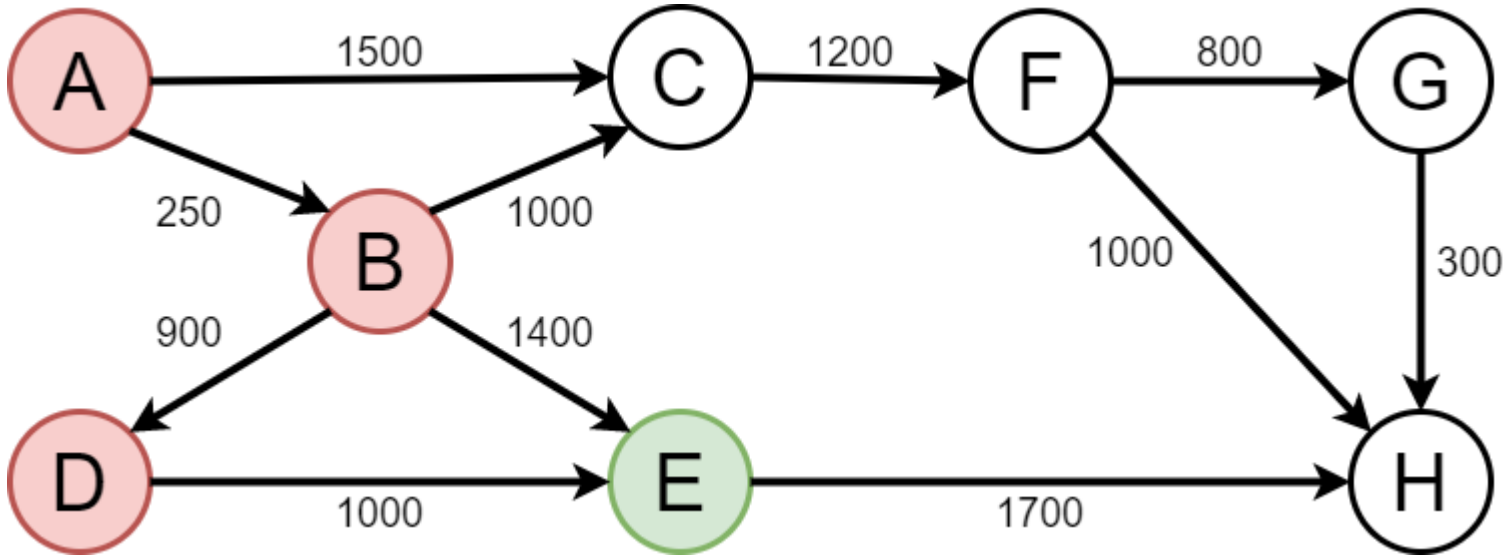
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	F	T	F	F	F	F
Cost	0	250	1250	1150	1650	∞	∞	∞
Path	-1	A	B	B	B	-1	-1	-1

Finished = {A, B, D}

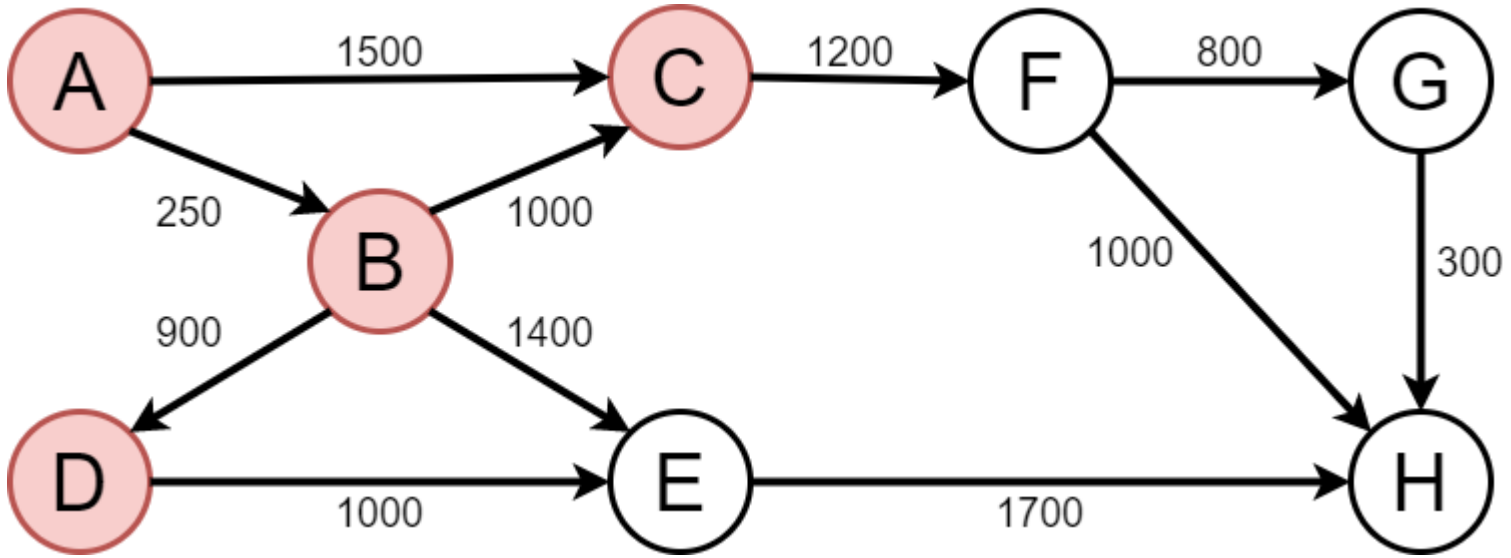
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	F	T	F	F	F	F
Cost	0	250	1250	1150	1650	∞	∞	∞
Path	-1	A	B	B	B	-1	-1	-1

Finished = {A, B, D}

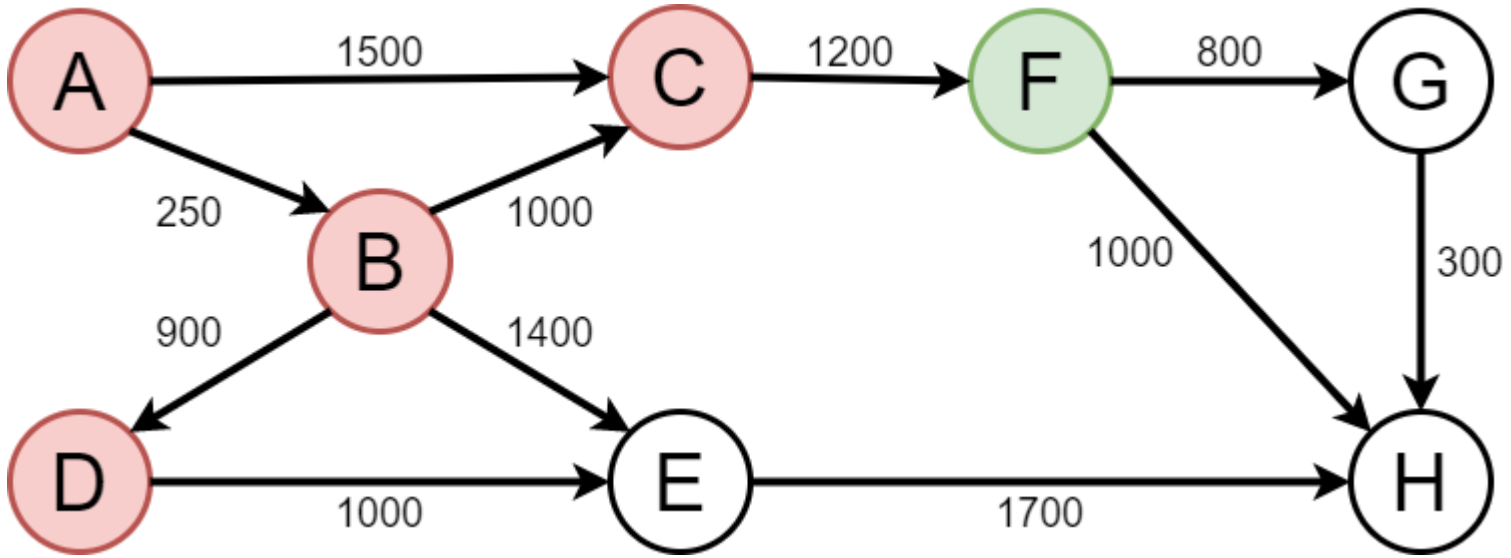
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	F	F	F	F
Cost	0	250	1250	1150	1650	∞	∞	∞
Path	-1	A	B	B	B	-1	-1	-1

Finished = {A, B, D, C}

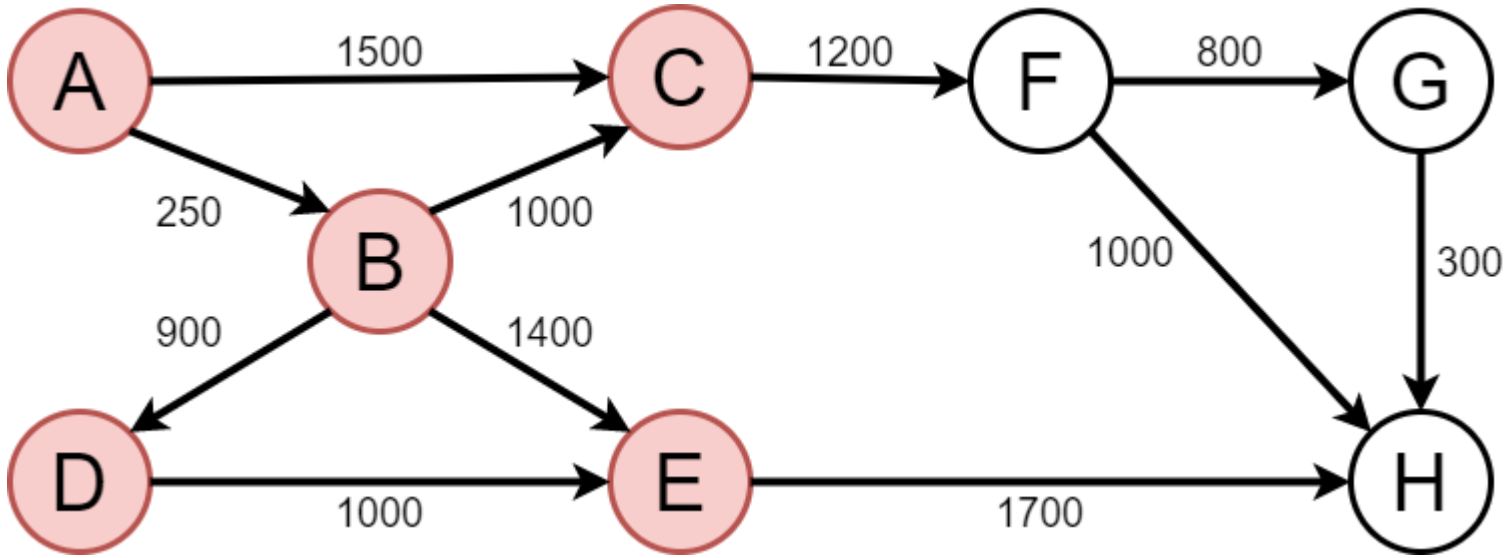
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	F	F	F	F
Cost	0	250	1250	1150	1650	2450	∞	∞
Path	-1	A	B	B	B	C	-1	-1

Finished = {A, B, D, C}

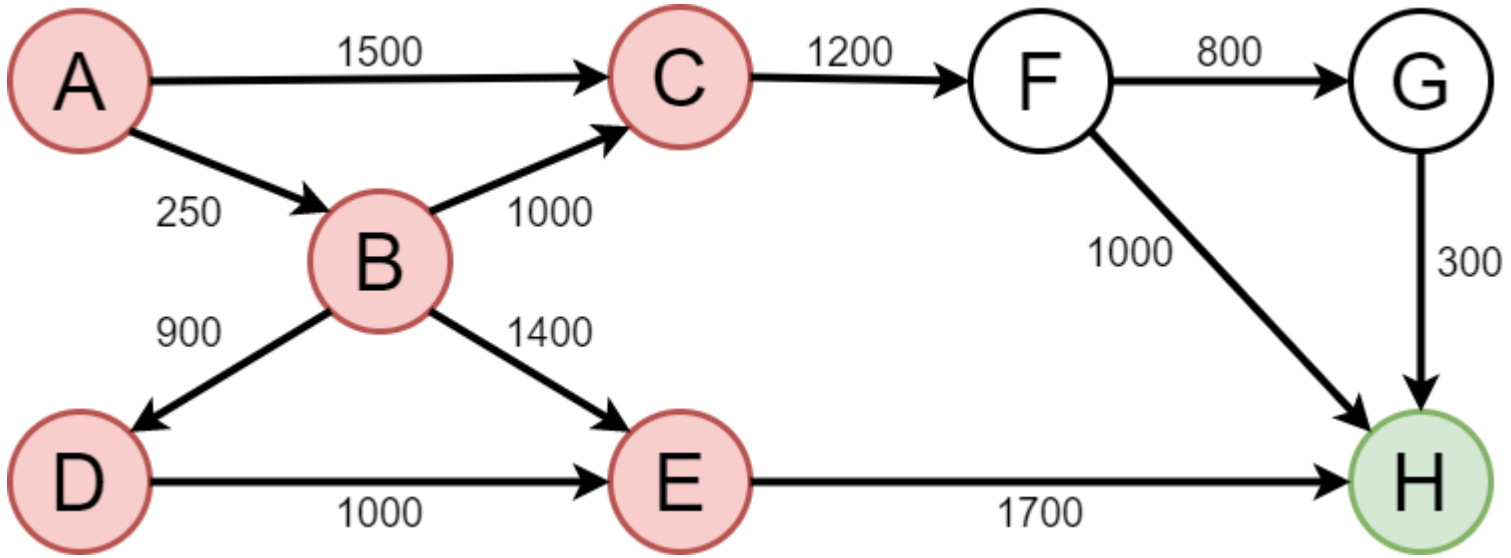
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	F	F	F
Cost	0	250	1250	1150	1650	2450	∞	∞
Path	-1	A	B	B	B	C	-1	-1

Finished = {A, B, D, C, E}

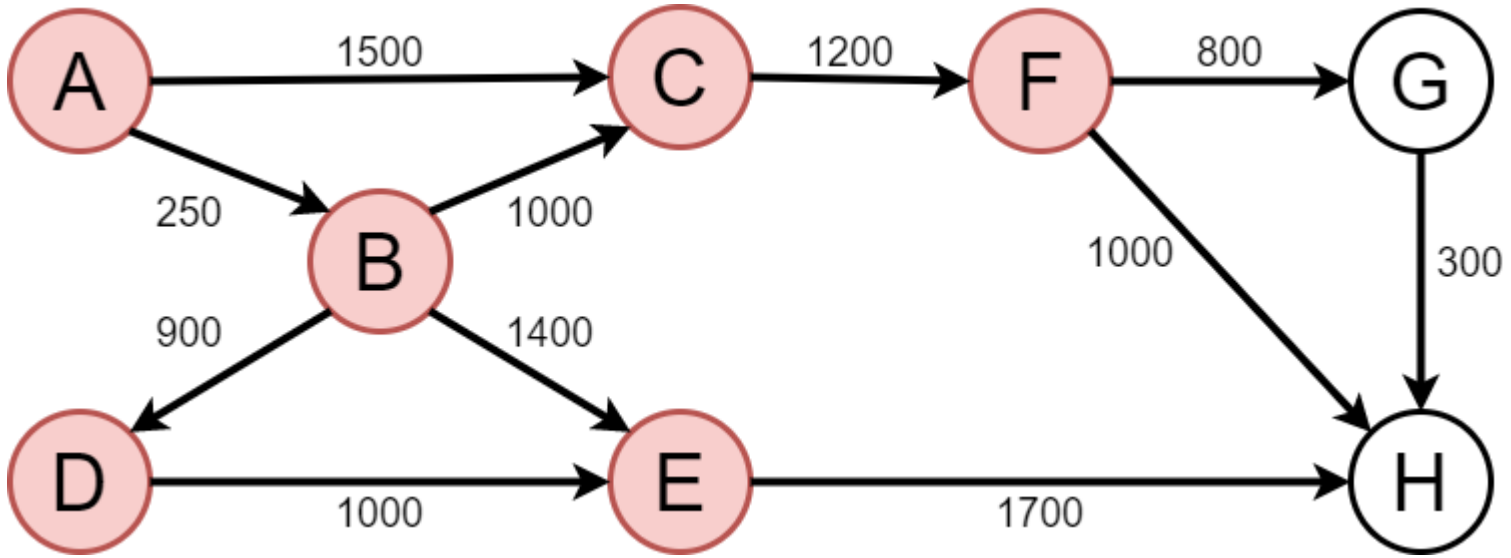
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	F	F	F
Cost	0	250	1250	1150	1650	2450	∞	3350
Path	-1	A	B	B	B	C	-1	E

Finished = {A, B, D, C, E}

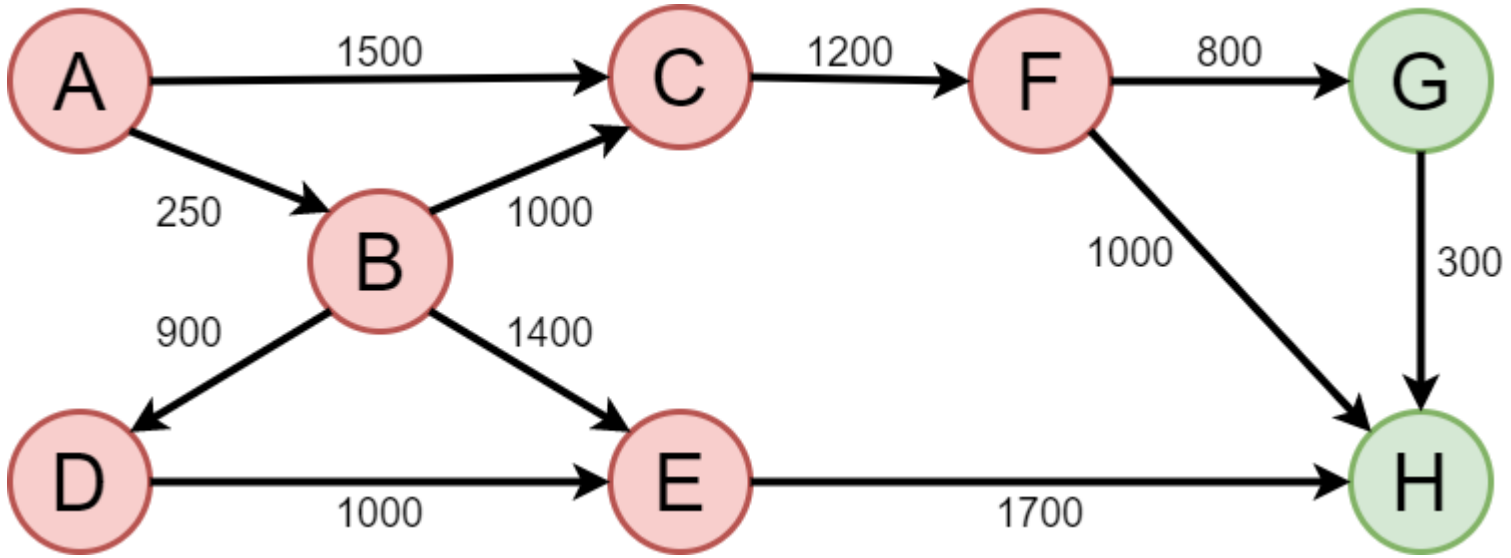
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	T	F	F
Cost	0	250	1250	1150	1650	2450	∞	3350
Path	-1	A	B	B	B	C	-1	E

Finished = {A, B, D, C, E, F}

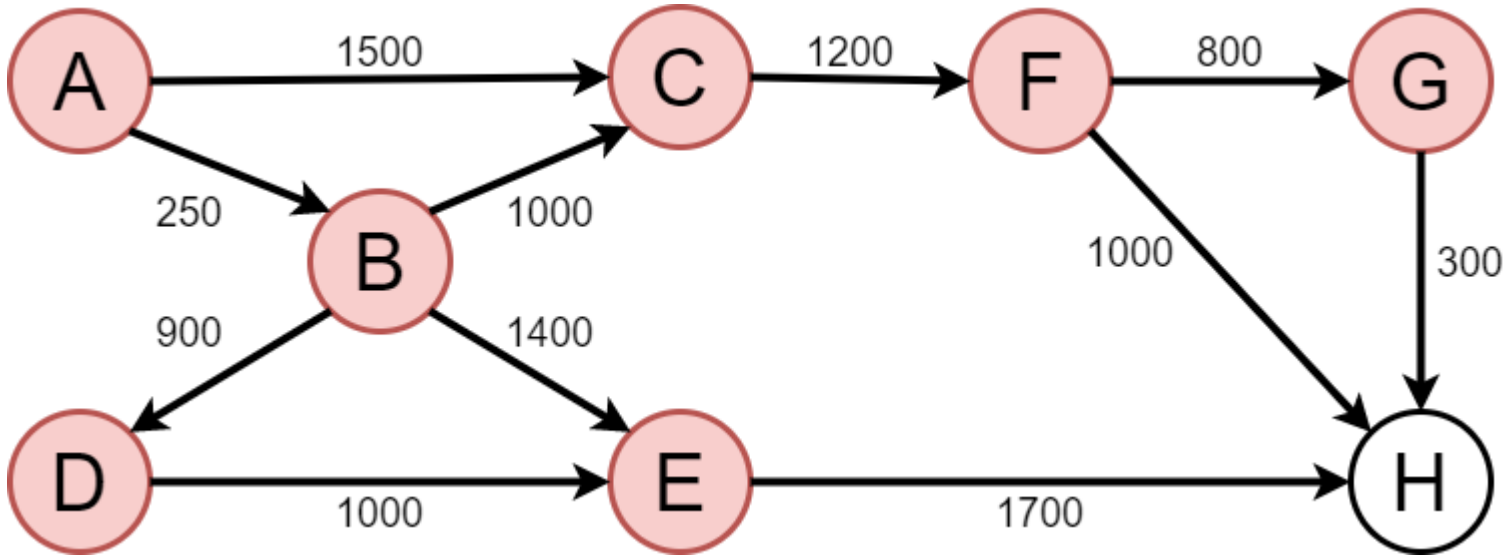
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	T	F	F
Cost	0	250	1250	1150	1650	2450	3250	3350
Path	-1	A	B	B	B	C	F	E

Finished = {A, B, D, C, E, F}

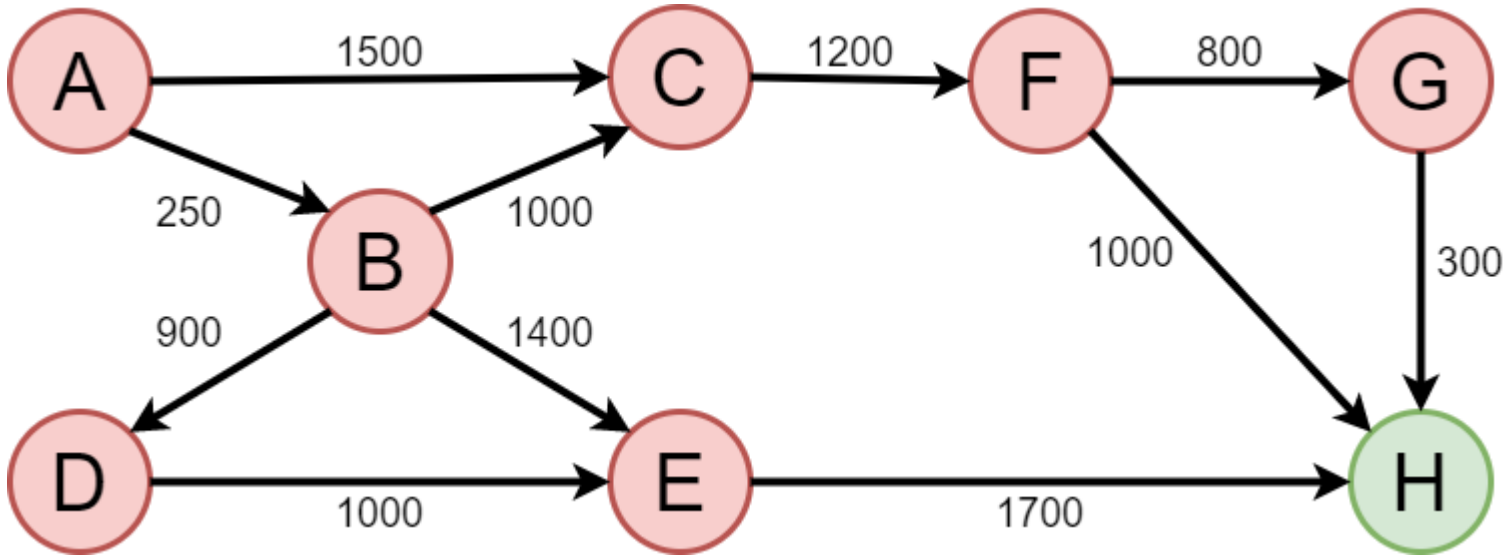
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	T	T	F
Cost	0	250	1250	1150	1650	2450	3250	3350
Path	-1	A	B	B	B	C	F	E

Finished = {A, B, D, C, E, F, G}

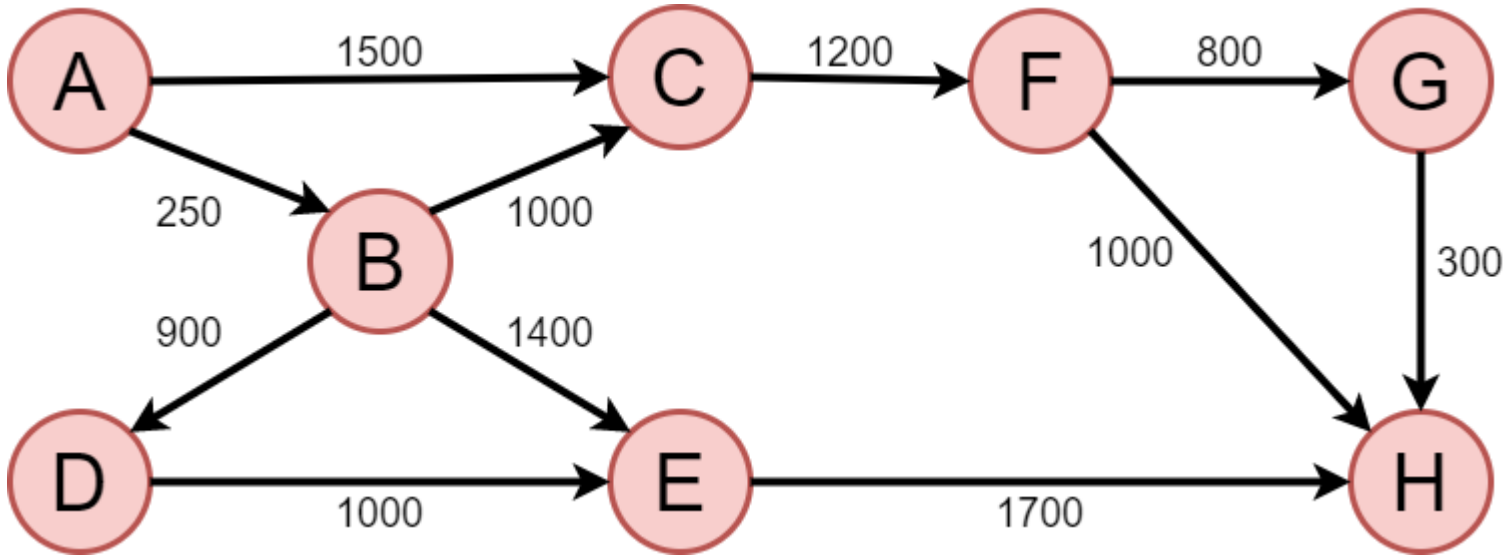
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	T	T	F
Cost	0	250	1250	1150	1650	2450	3250	3350
Path	-1	A	B	B	B	C	F	E

Finished = {A, B, D, C, E, F, G}

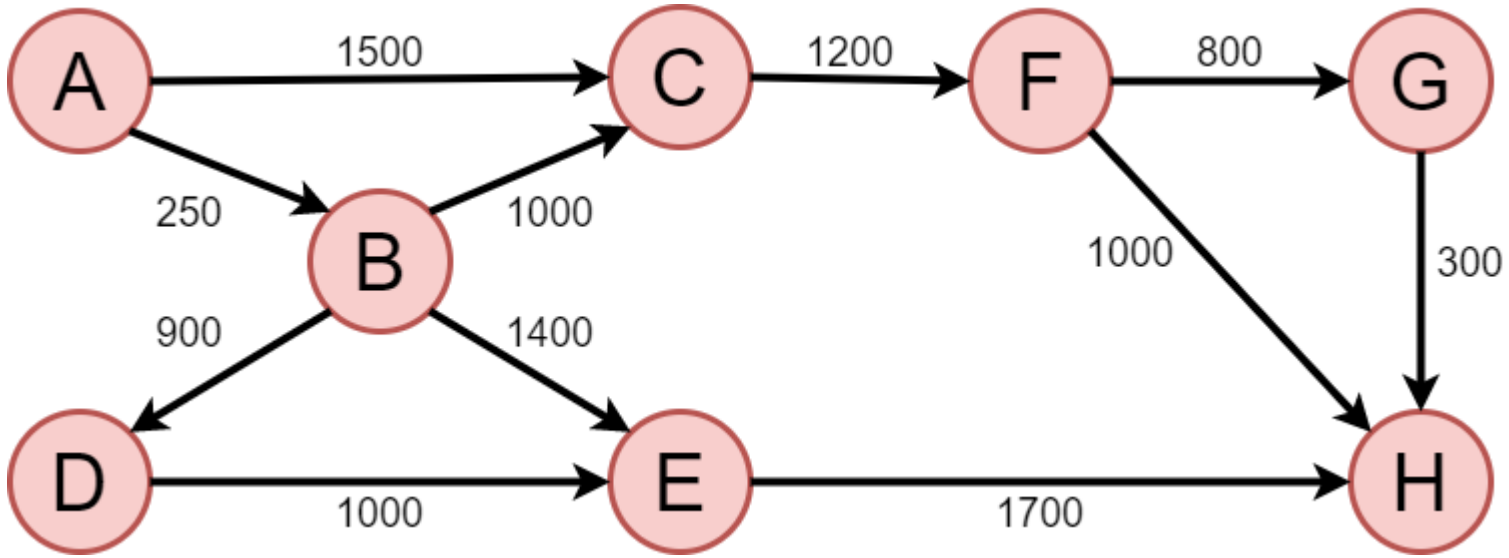
Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	T	T	T
Cost	0	250	1250	1150	1650	2450	3250	3350
Path	-1	A	B	B	B	C	F	E

Finished = {A, B, D, C, E, F, G, H}

Dijkstra's Algorithm : Example



	A	B	C	D	E	F	G	H
Visited	T	T	T	T	T	T	T	T
Cost	0	250	1250	1150	1650	2450	3250	3350
Path	-1	A	B	B	B	C	F	E

Finished = {A, B, D, C, E, F, G, H}

Dijkstra's Algorithm : Pseudo Code

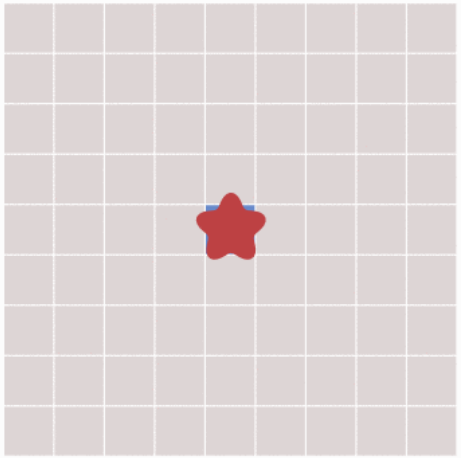
```
1  def Dijkstra(G, weight, v_start):
2
3      for each vertex v in G.vertices:
4          v.distance = INF
5          v.predecessor = None
6      v_start.distance = 0
7
8      finished = set()
9      Q = set(G.vertices)
10
11     while Q is not empty:
12         u = extract_min(Q)
13         finished.union(u)
14         for each vertex v in G.Adj[u]:
15             if v.distance > u.distance + weight[u][v]:
16                 v.distance = u.distance + weight[u][v]
17                 v.predecessor = u
```

Dijkstra's Algorithm : Analysis

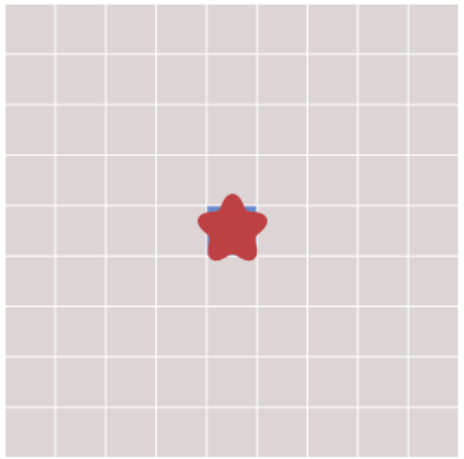
- Time Complexity
  - Original Algorithm : $O(V^2)$
  - Optimized (Fibonacci-Heap): $O(E + V\log V)$

- Pros : Guarantees to find an optimal path
- Cons : Very slow when there are numerous vertices

Best-First Search



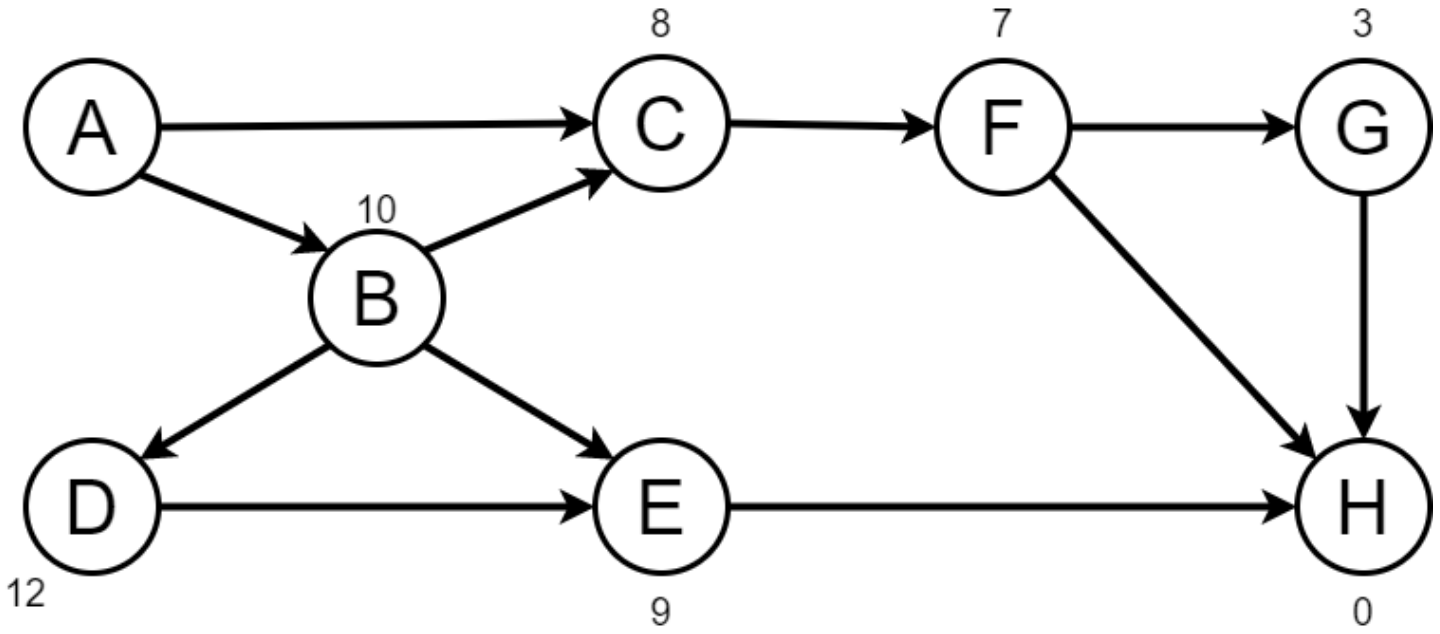
Breadth First Search(BFS)



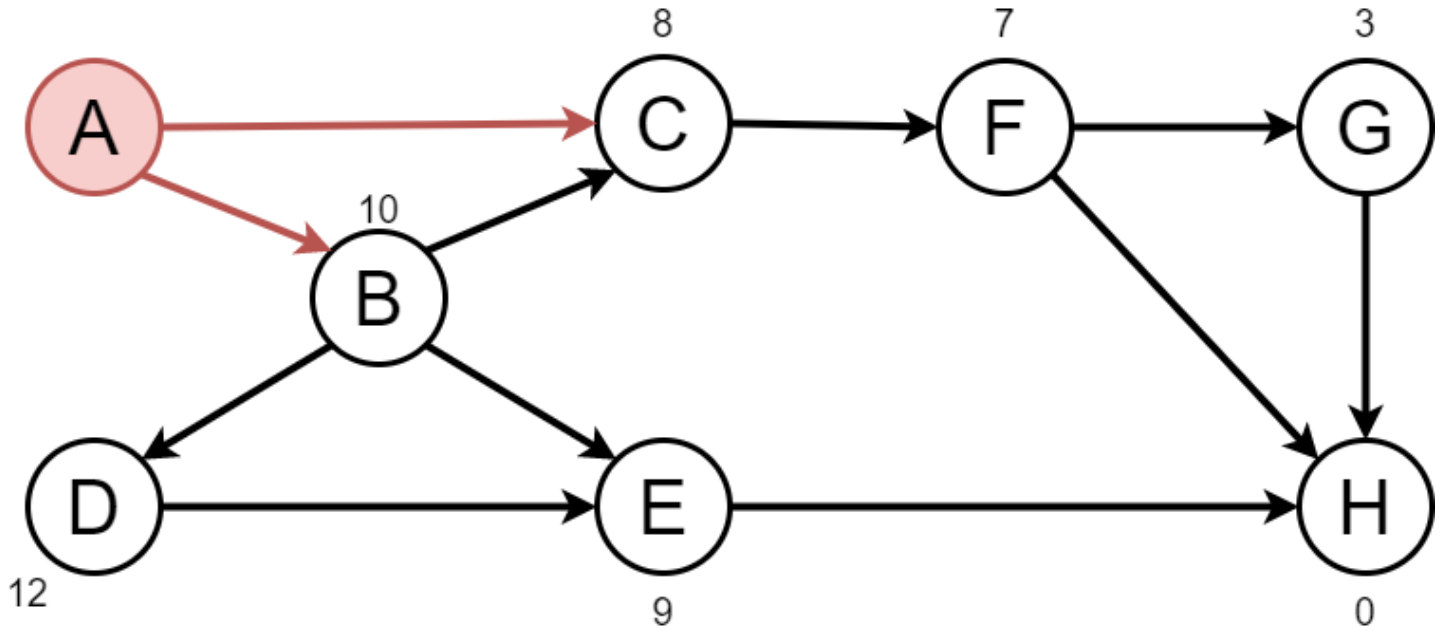
Depth First Search(DFS)

- Heuristic Search - explores a graph by expanding the most promising vertex
- Promise of a vertex - Heuristic Function f(v)
  - Estimation of the cheapest cost from v to the goal
  - e.g. Euclidean Distance, Manhattan Distance

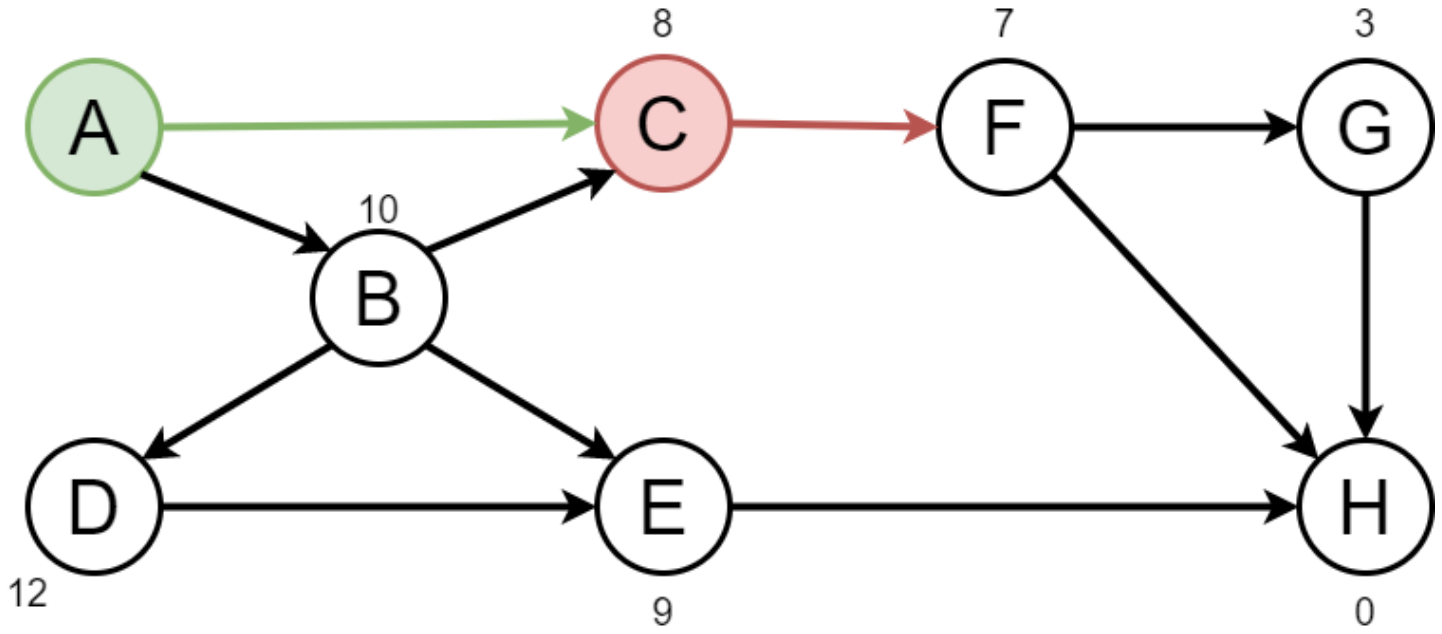
Best-First Search : Example



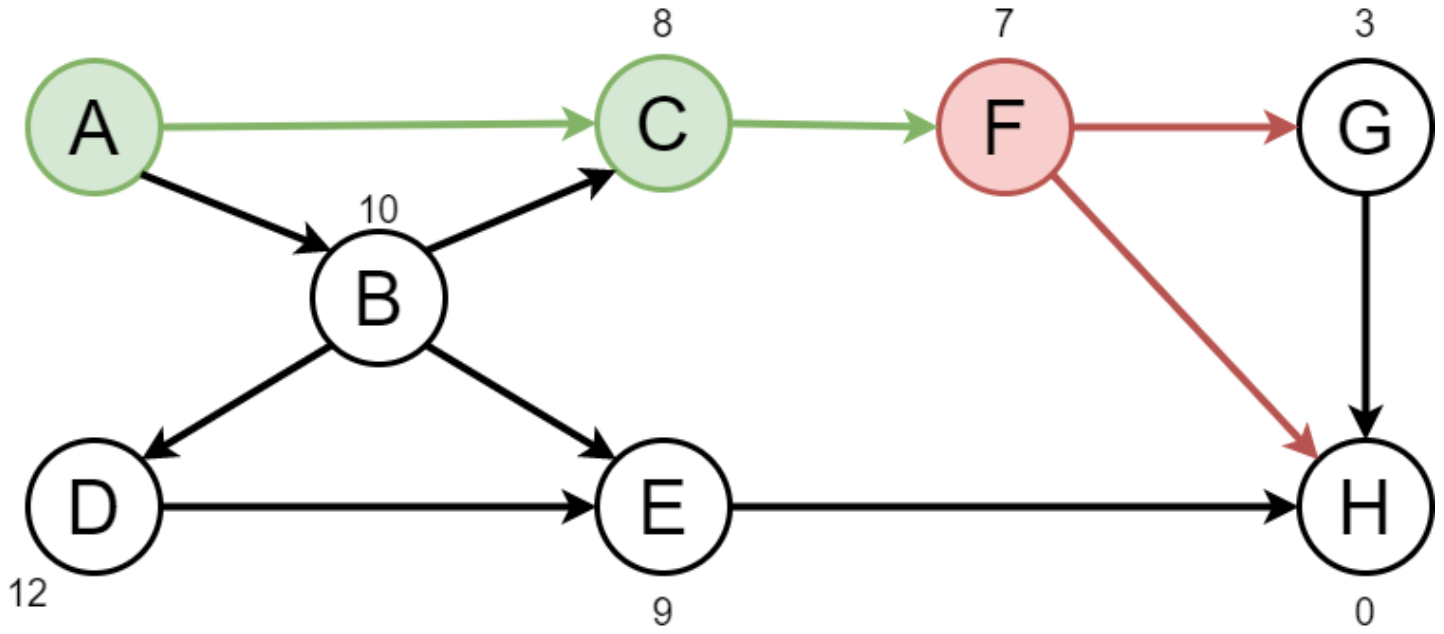
Best-First Search : Example



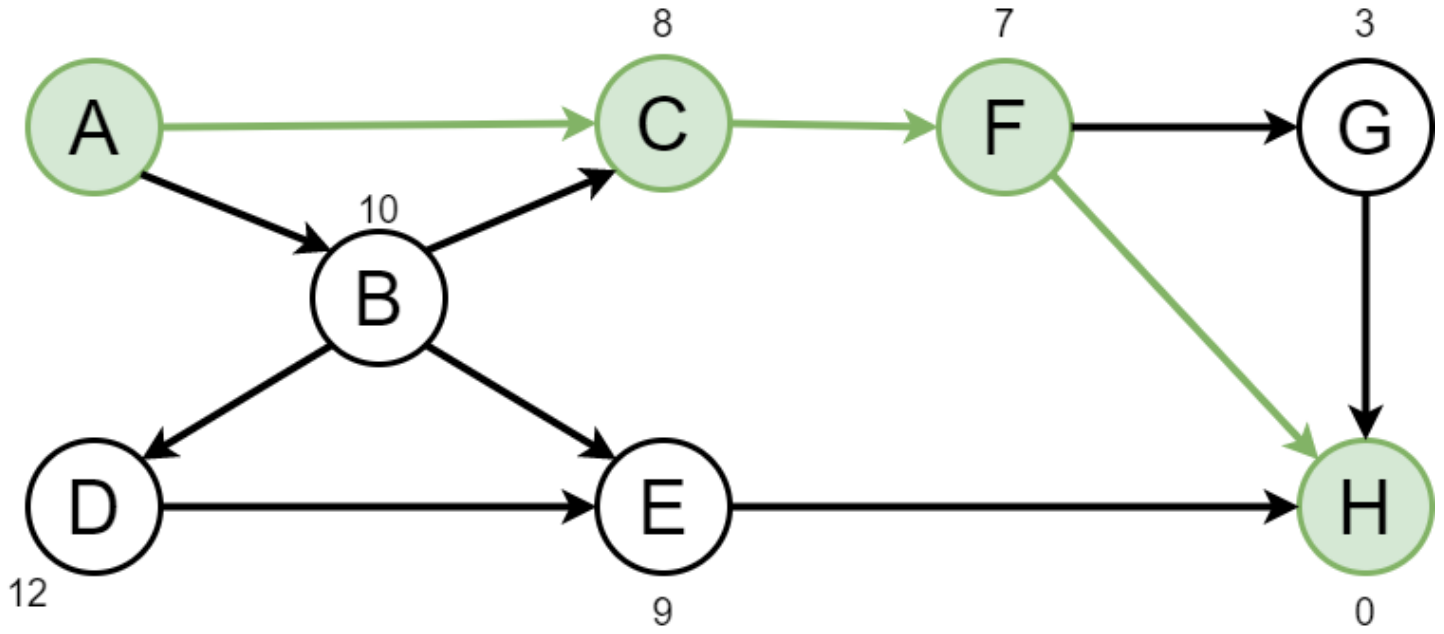
Best-First Search : Example



Best-First Search : Example

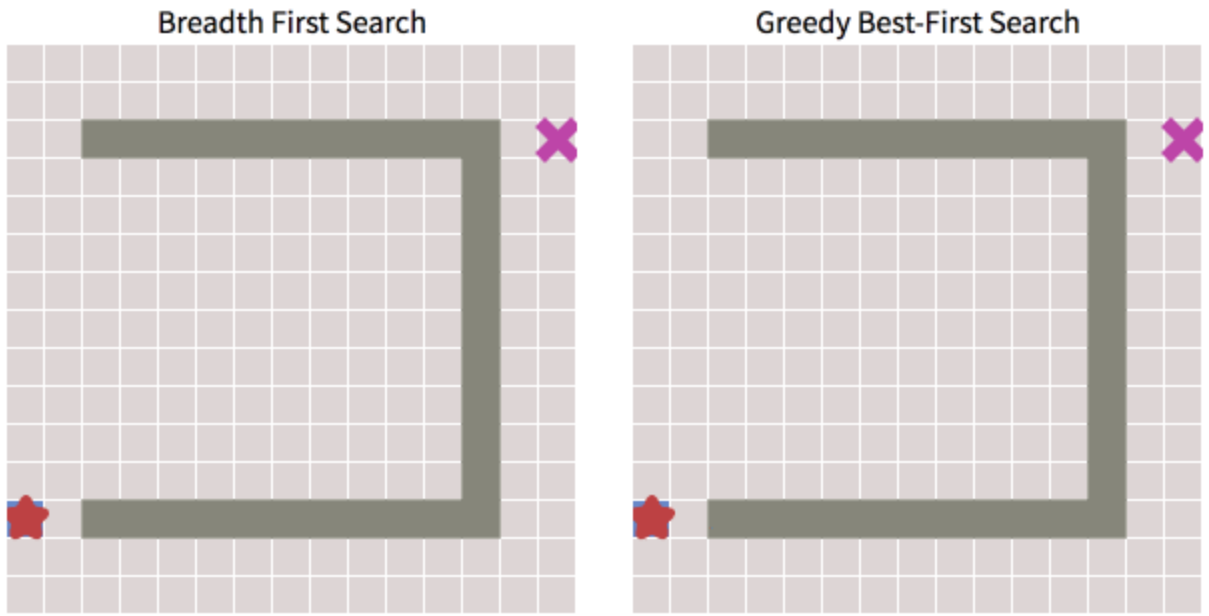


Best-First Search : Example



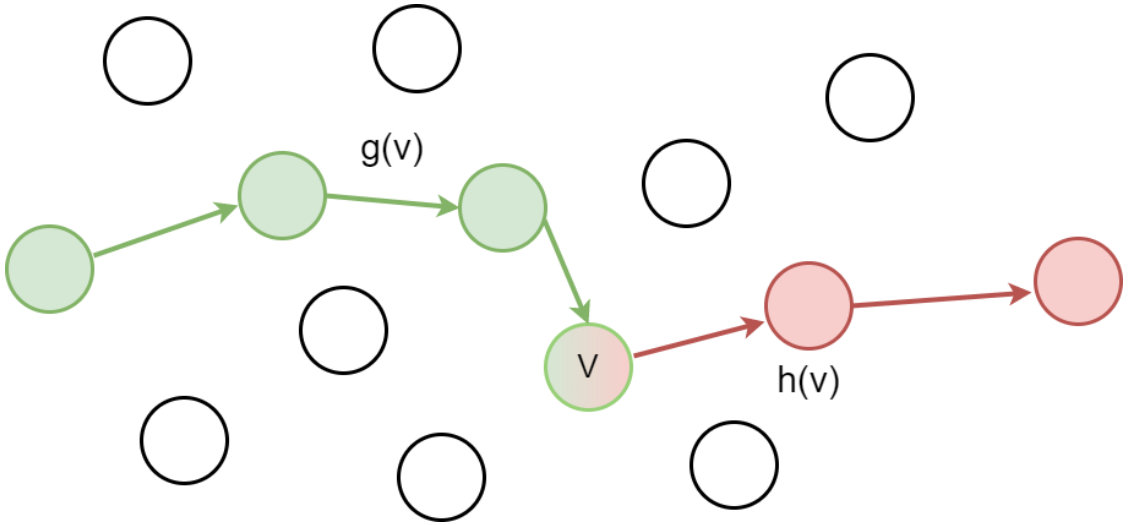
Best-First Search : Analysis

- Pros : Fast
- Cons : Heuristic estimation doesn't guarantee the best path



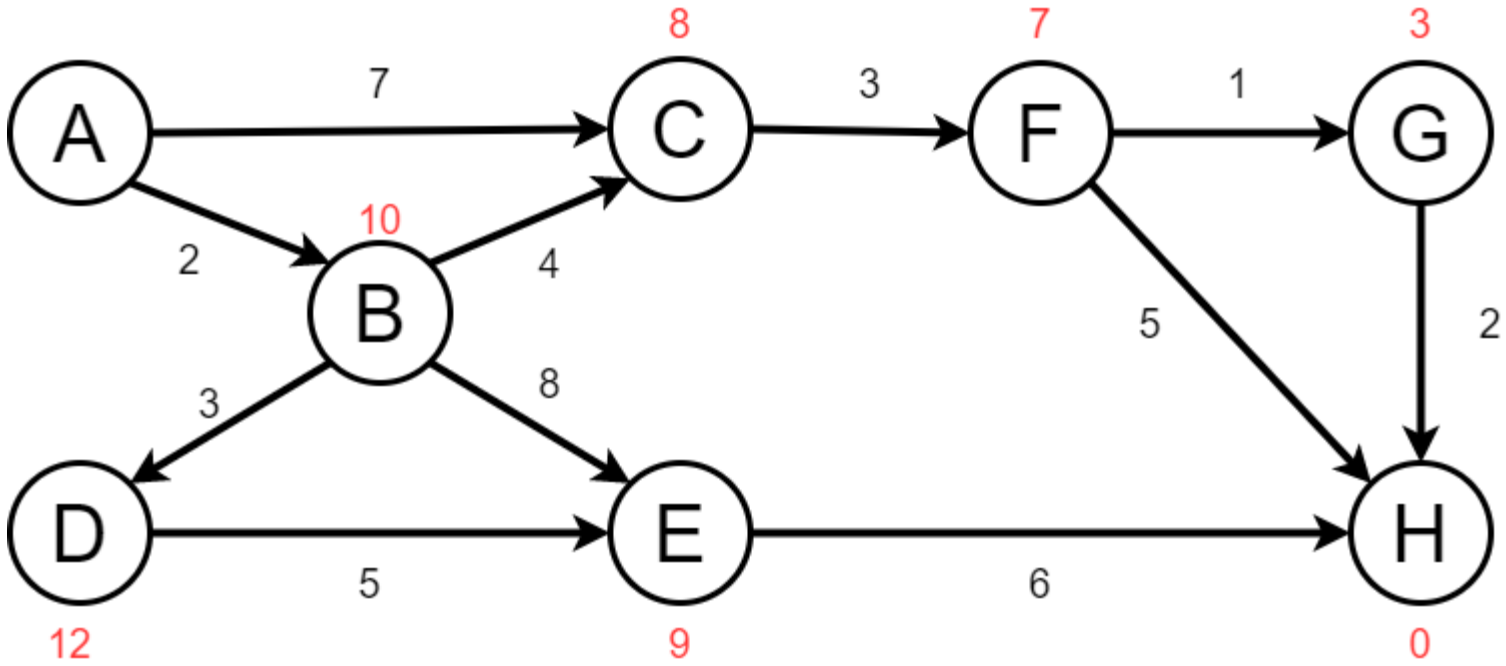
A* Algorithm

- Combine the advantages of both Dijkstra's and Best First Search

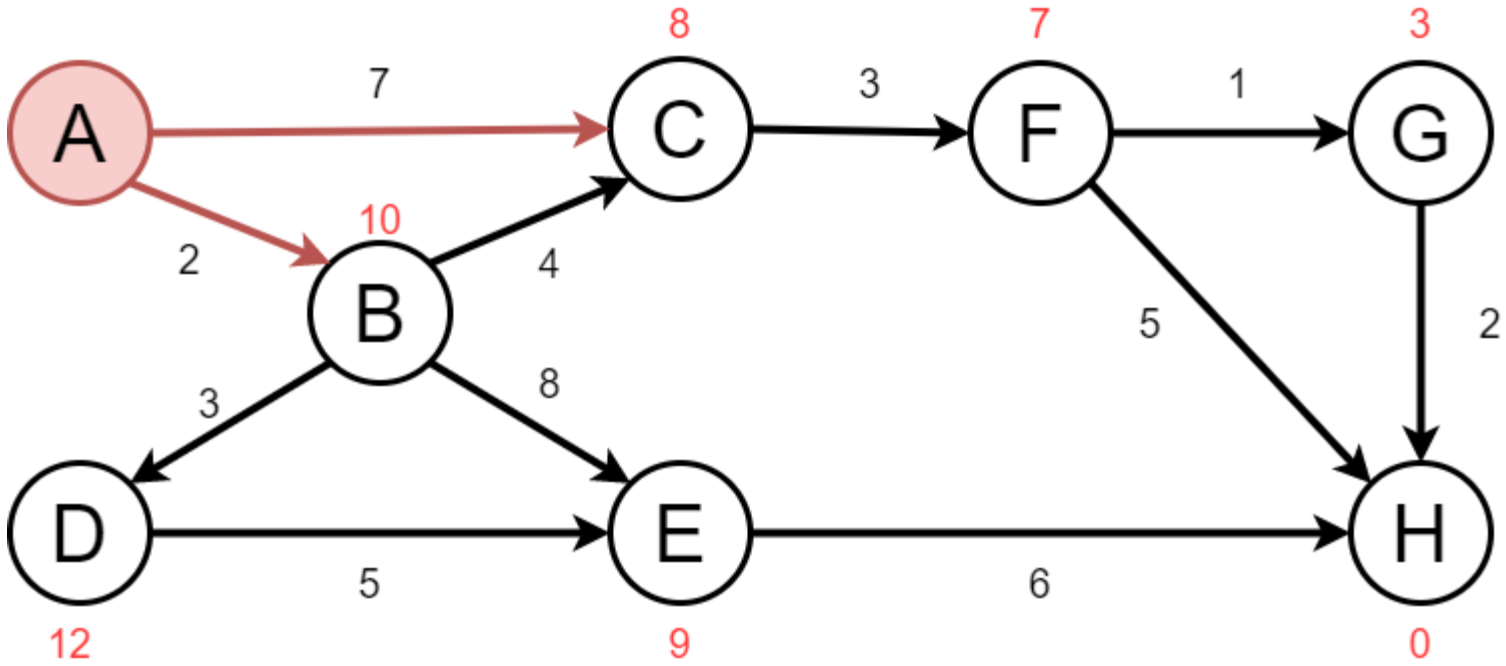


- Consider both previous and future parameters:
  - g(v) : calculated path cost from start to v (Dijkstra)
  - h(v) : heuristic path estimation from v to target (Best First Search)

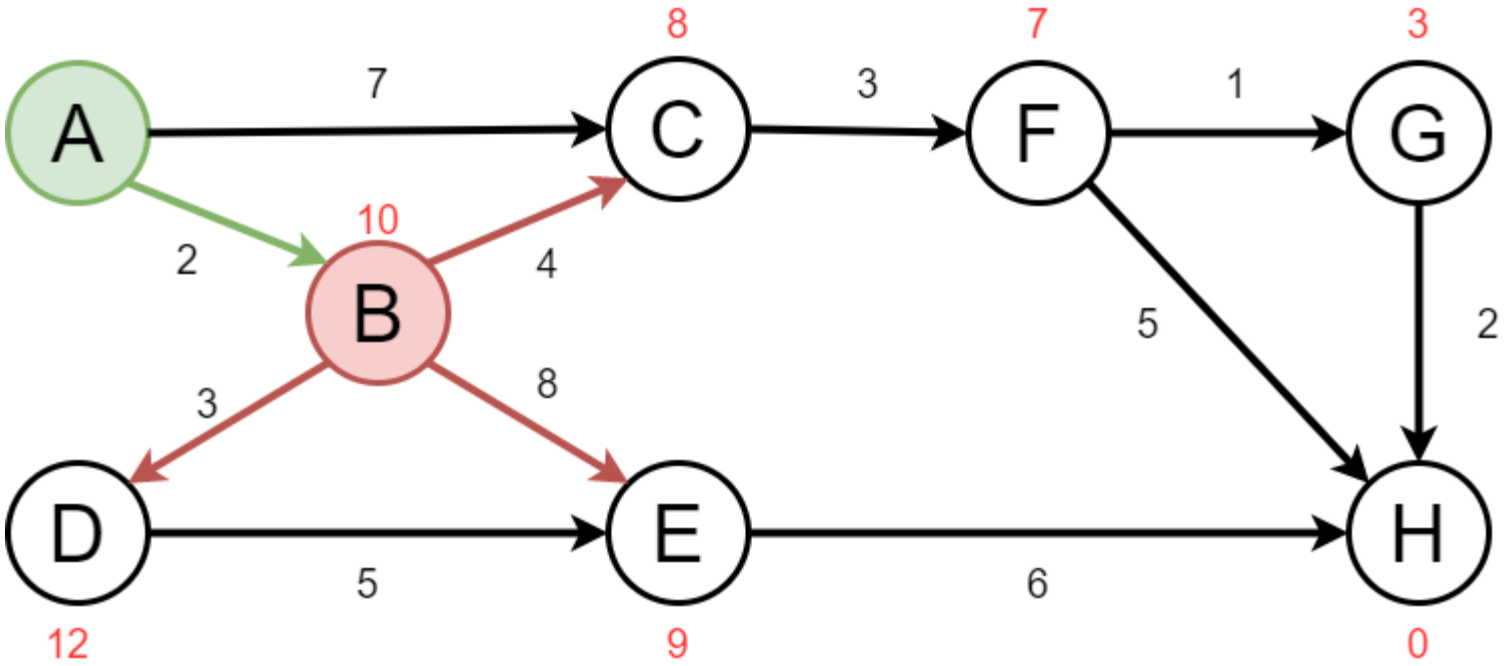
A* Algorithm : Example



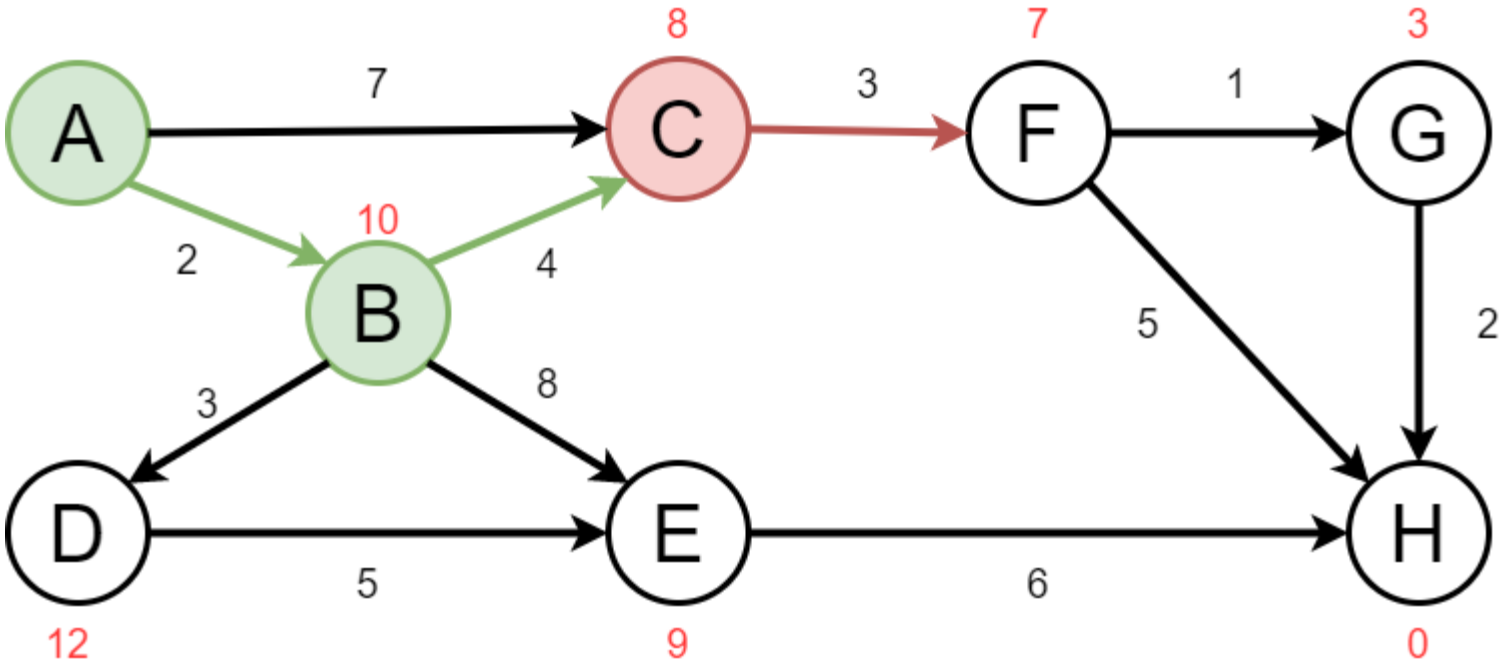
A* Algorithm : Example



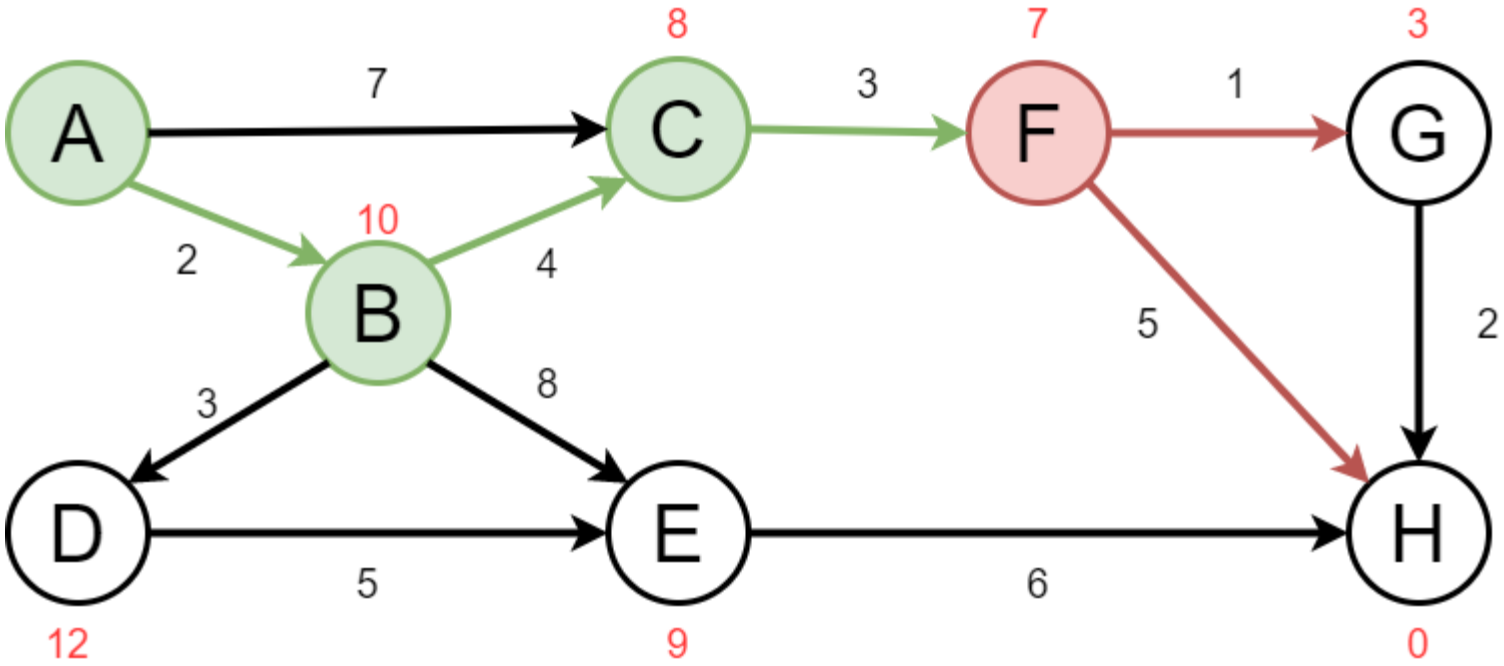
A* Algorithm : Example



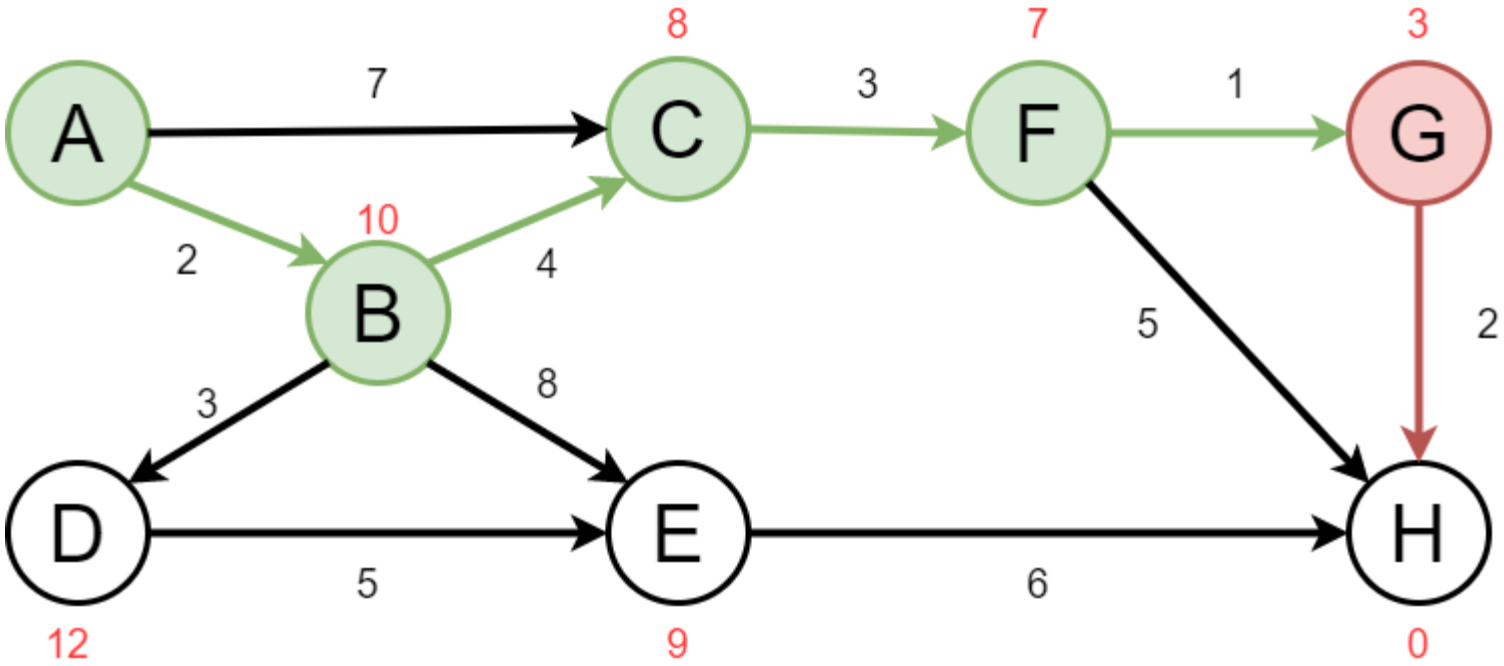
A* Algorithm : Example



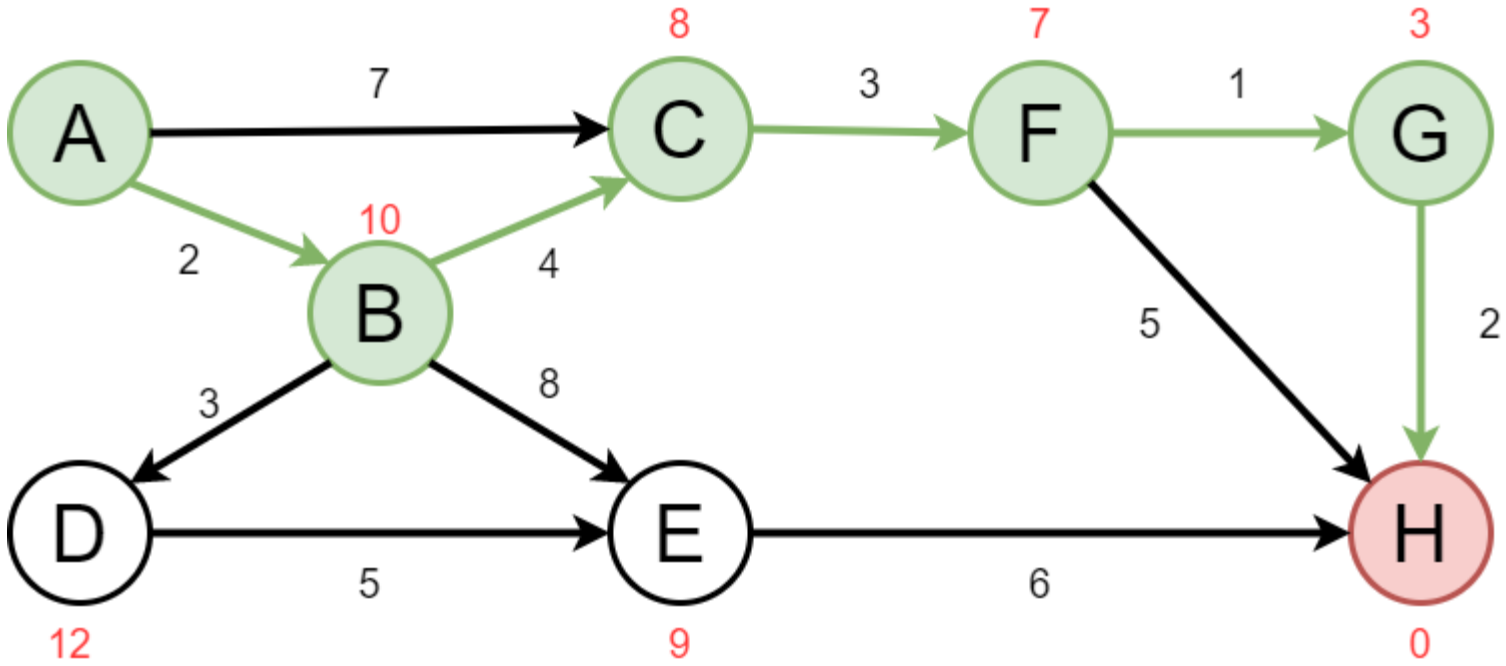
A* Algorithm : Example



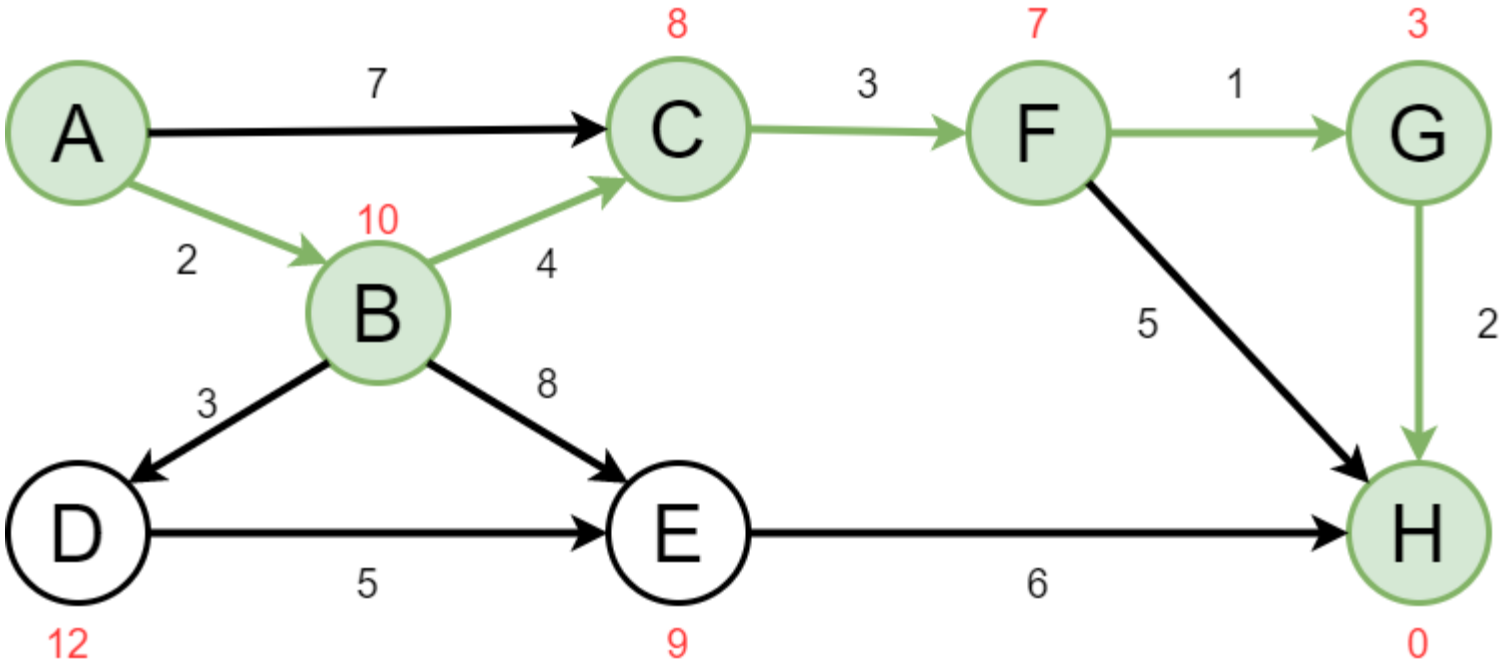
A* Algorithm : Example



A* Algorithm : Example

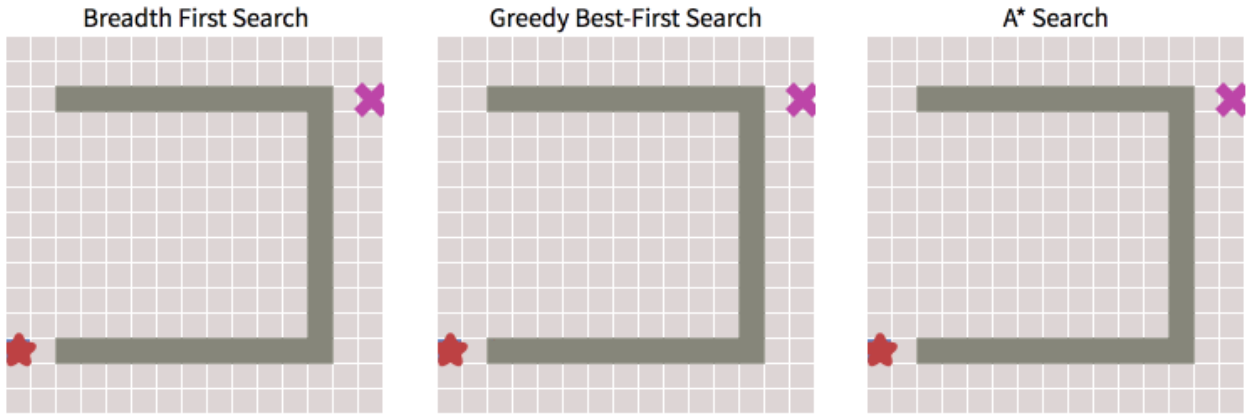


A* Algorithm : Example



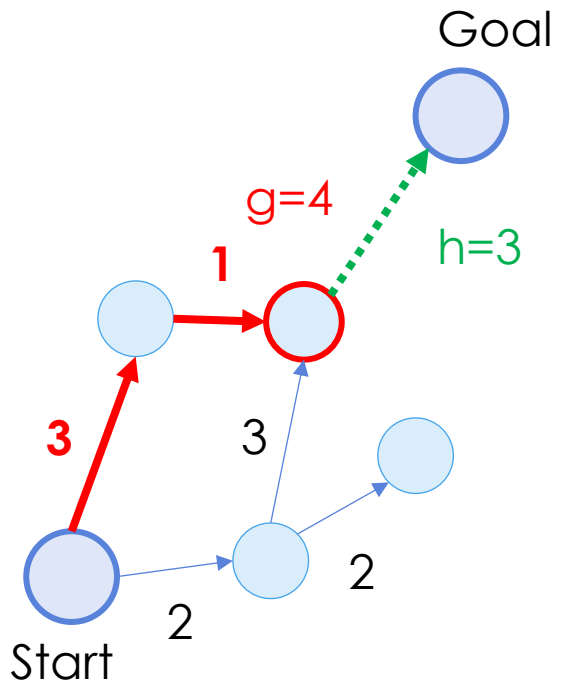
A* Algorithm : Analysis

- In easy cases, A* can find a great path as fast as Best First Search. In hard cases, A* can find a great path as good as Dijkstra
- Special condition of A*
  - h(v) = 0 : degrade to Dijkstra's Algorithm
  - g(v)=0 or h(v) >> Edge weight: degrade to Best-First Search
- In summary, a good choice of heuristic function is needed



Heuristic Function

- For path Planning, we can utilize the distance between the node and the goal as the heuristic Function



Path Planning in Grid Space

- For mobile robot, we can discrete the 2D plane into grid space.
- The definition of distance can be in the following forms

2	1	2
1	0	1
2	1	2

$L_1\ Distance$:
$|1| + |1| = 2$

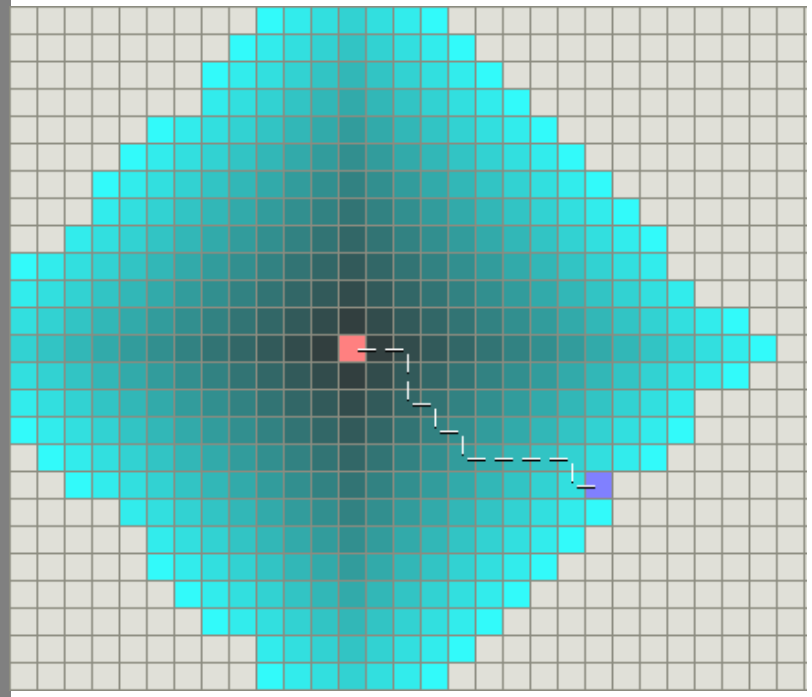
$\sqrt{2}$	1	$\sqrt{2}$
1	0	1
$\sqrt{2}$	1	$\sqrt{2}$

$L_2\ Distance$:
$\sqrt{1^2 + 1^2} = \sqrt{2}$

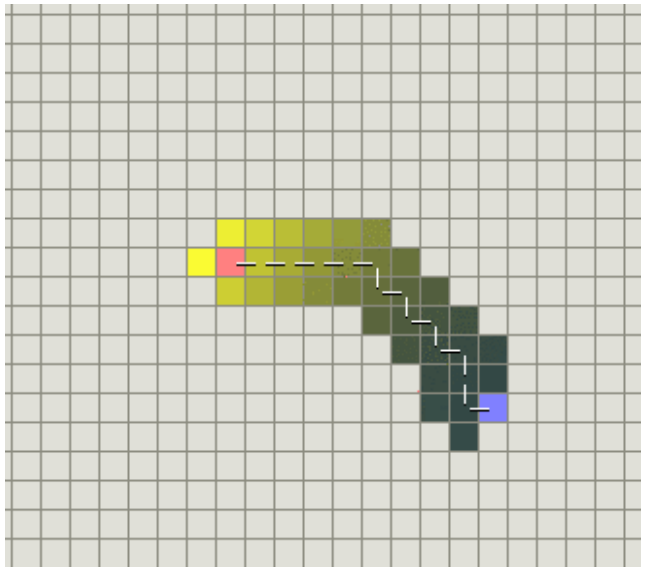
1	1	1
1	0	1
1	1	1

$L_\infty\ Distance$:
$\sqrt[\infty]{1^\infty + 1^\infty} = \max(1,1)$
$= 1$

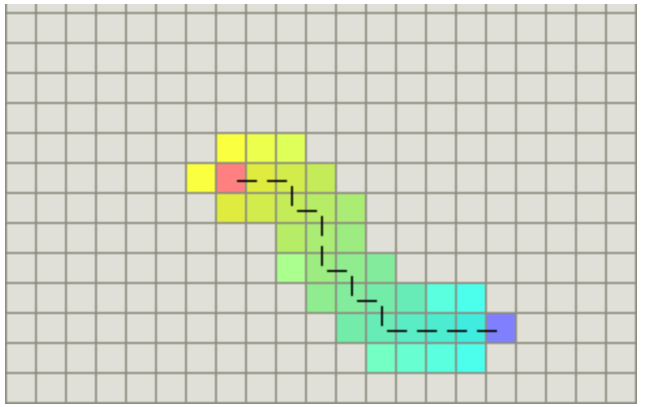
Comparison : Easy Case



Dijkstra
Great Path
Slow

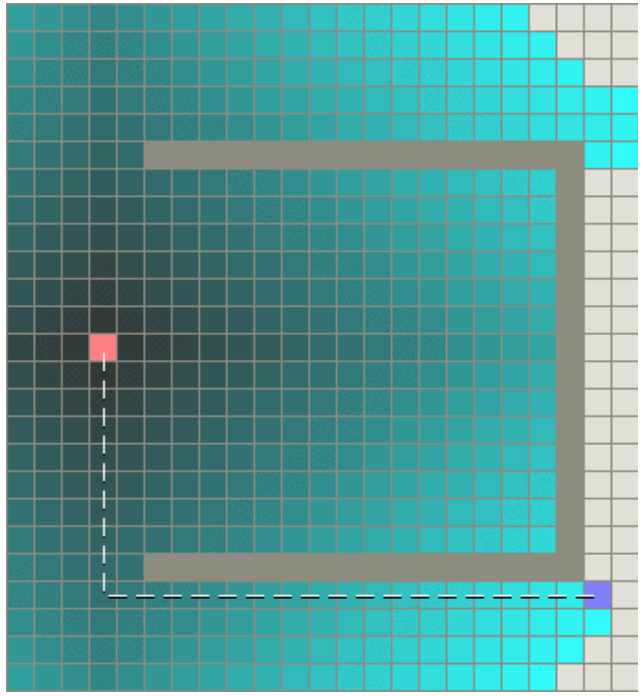


Best-First Search
Great Path
Fast

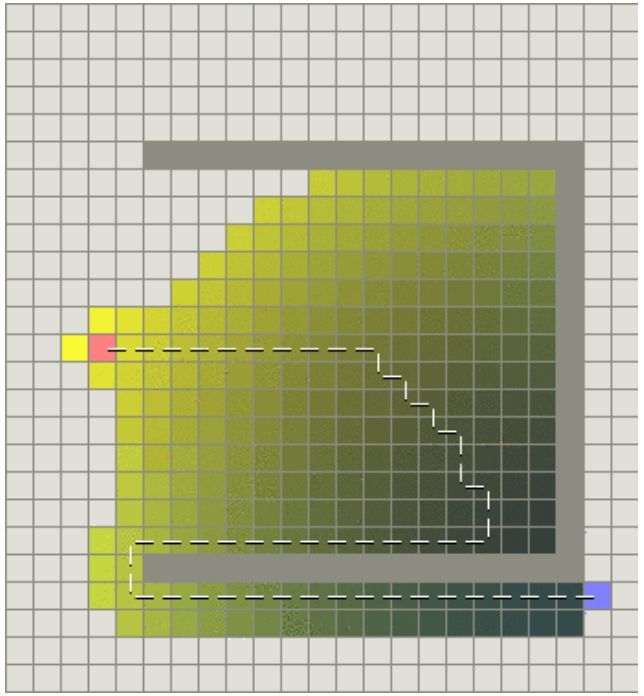


A* Search
Great Path
Fast

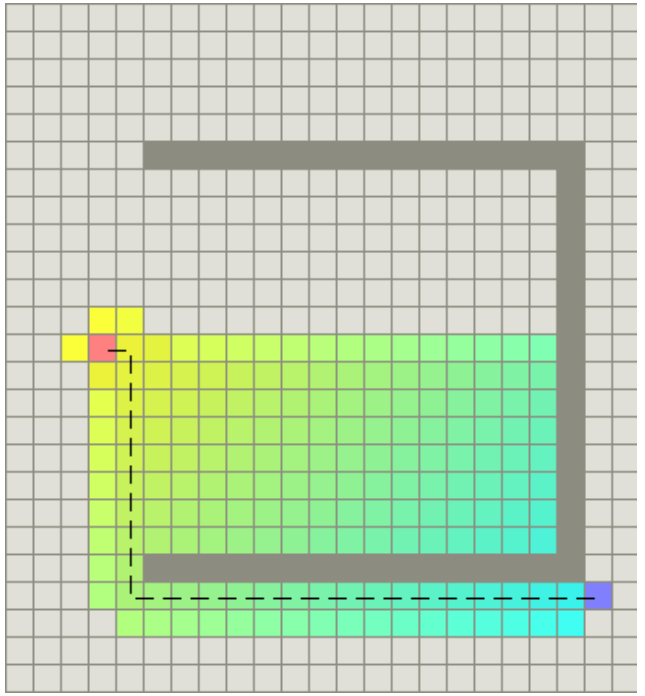
Comparison : Hard Case



Dijkstra
Great Path
Slow



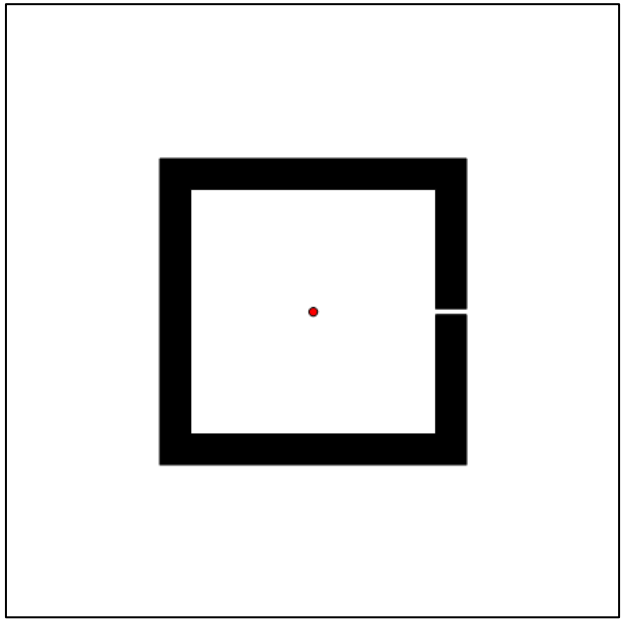
Best-First Search
Bad Path
Fast

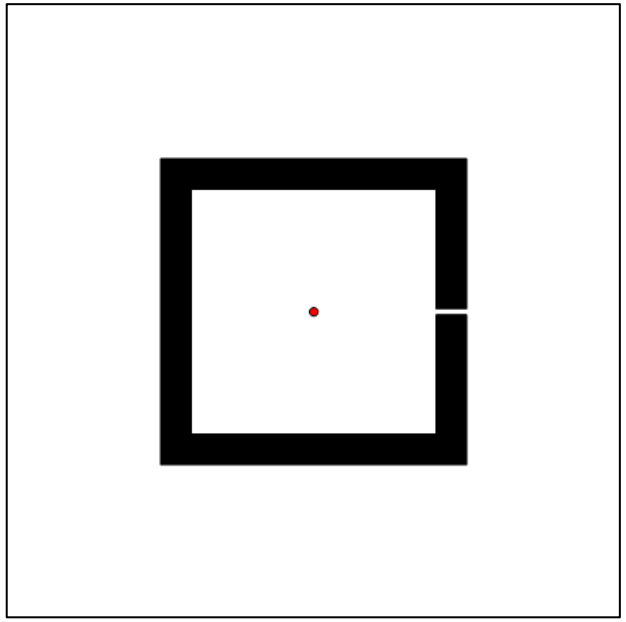


A* Search
Great Path
Fast

Sampling Based Planning Methods

- A* is resolution complete, while it still takes much time to search each grid in 2D space.
- Sampling based methods decrease the search state in 2D space, while the algorithms only output sub-optimal solution (probabilistic completeness).





Probabilistic Road-Map (PRM)

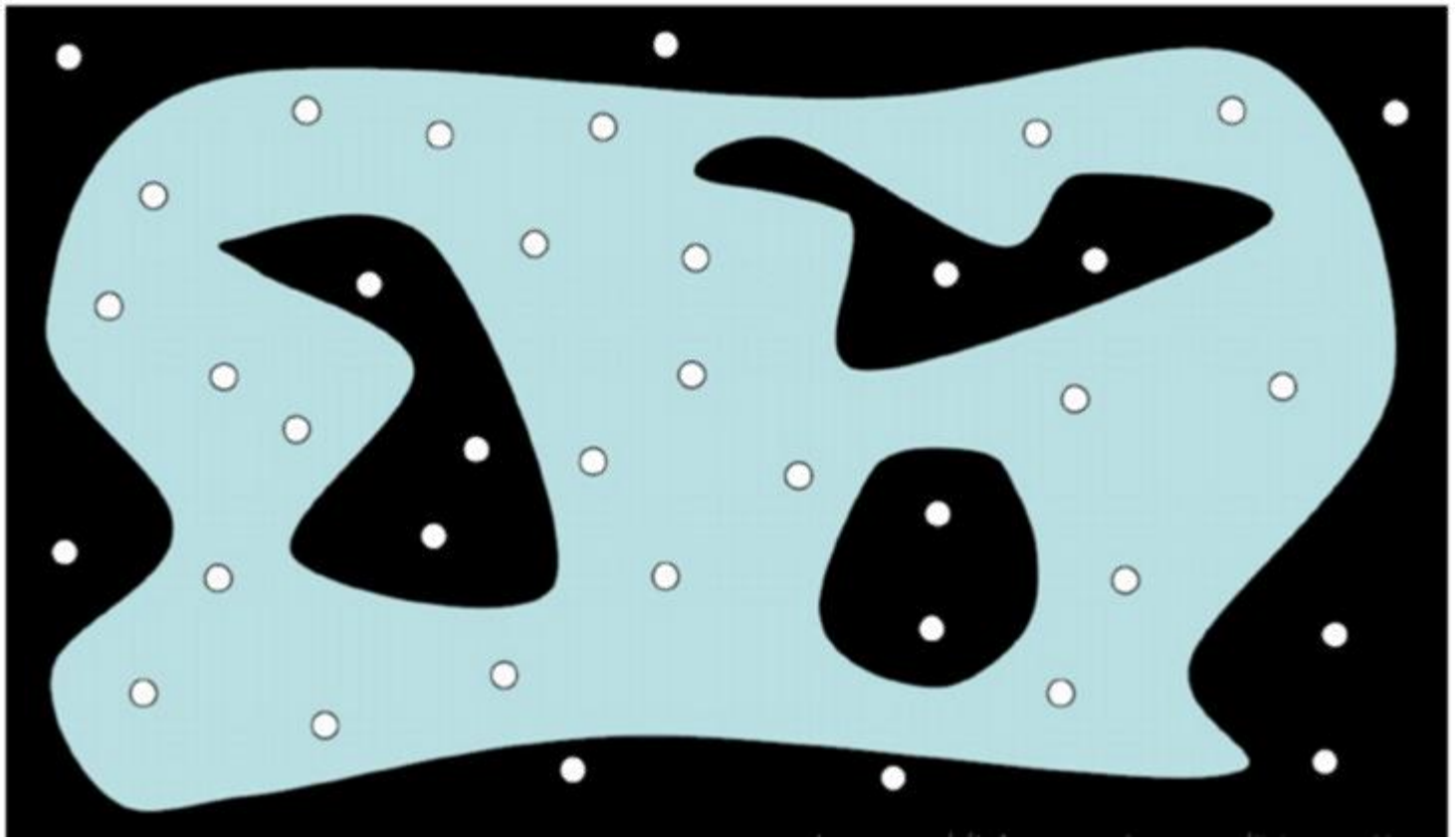
- To reduce search cost, PRM algorithm constructs a graph of the 2D space by randomly sampling positions.

- PRM Algorithm

1. Randomly generate points in free area.
2. Connect k-nearest neighbor points.
3. Delete the edges which cross the occupied area.
4. Connect connected components.
5. Plan the path in the generated graph.

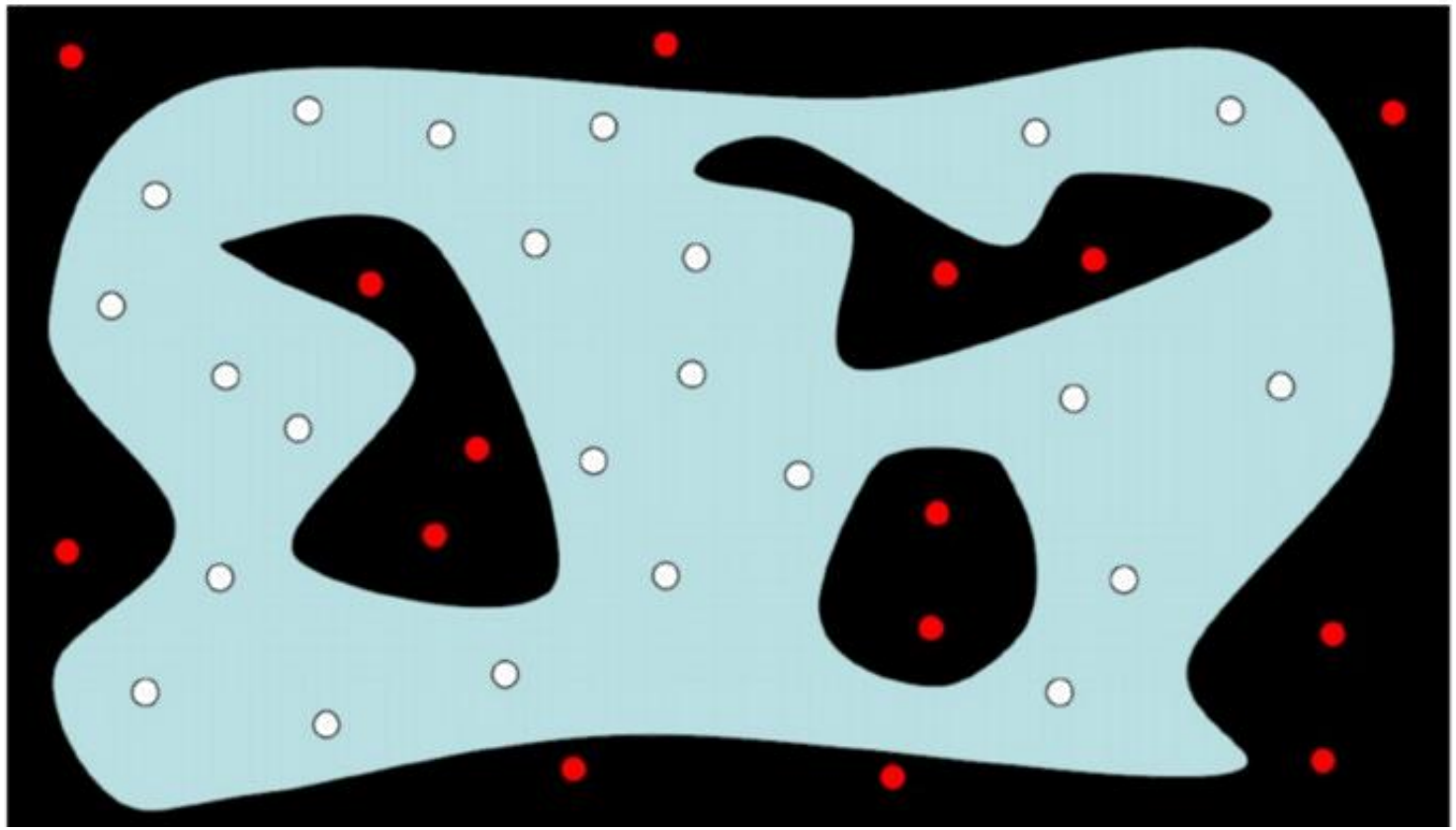
Probabilistic Road-Map (PRM)

- Randomly sample points in 2D space.



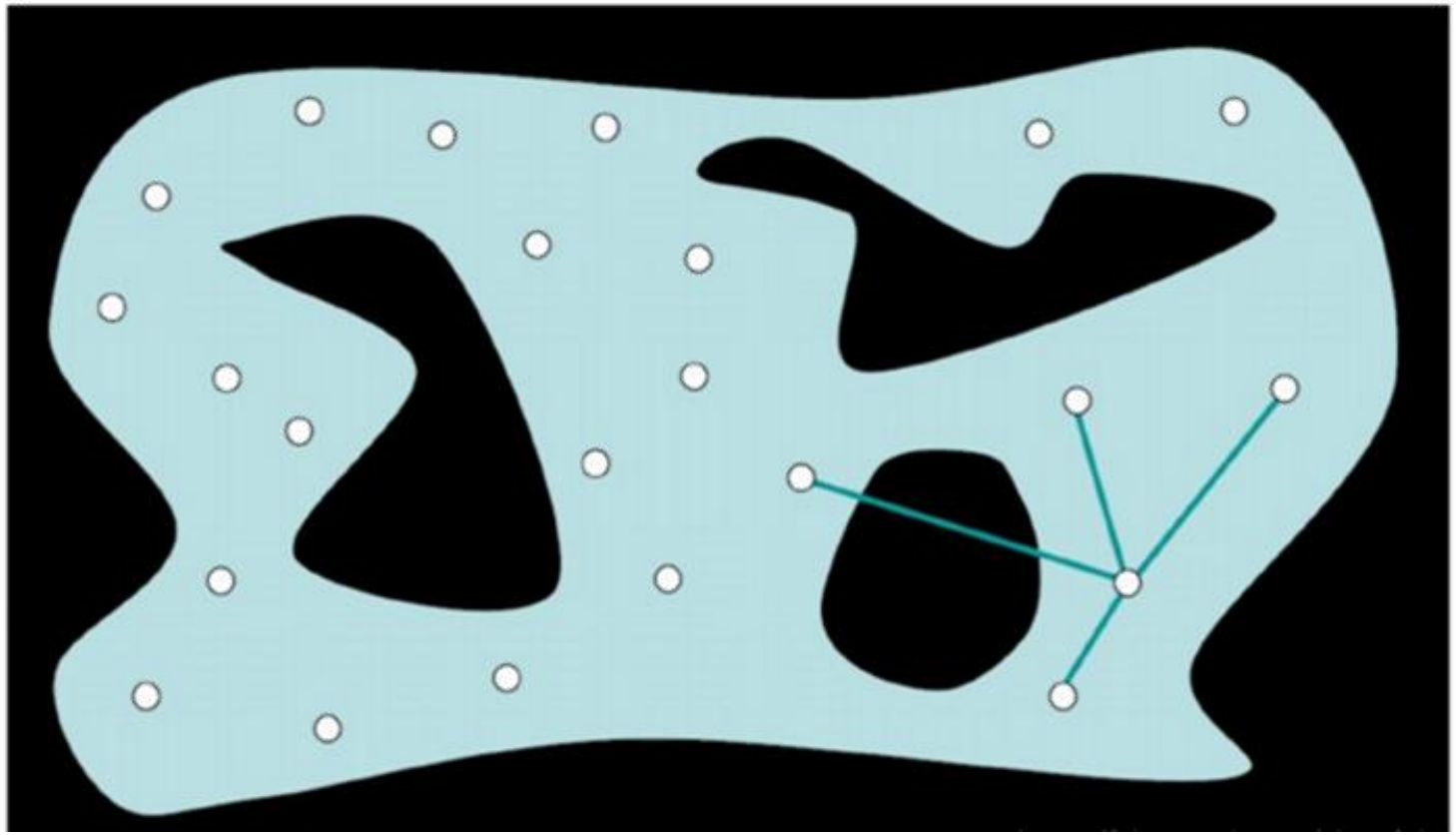
Probabilistic Road-Map (PRM)

- Delete the points which are at the occupied area.



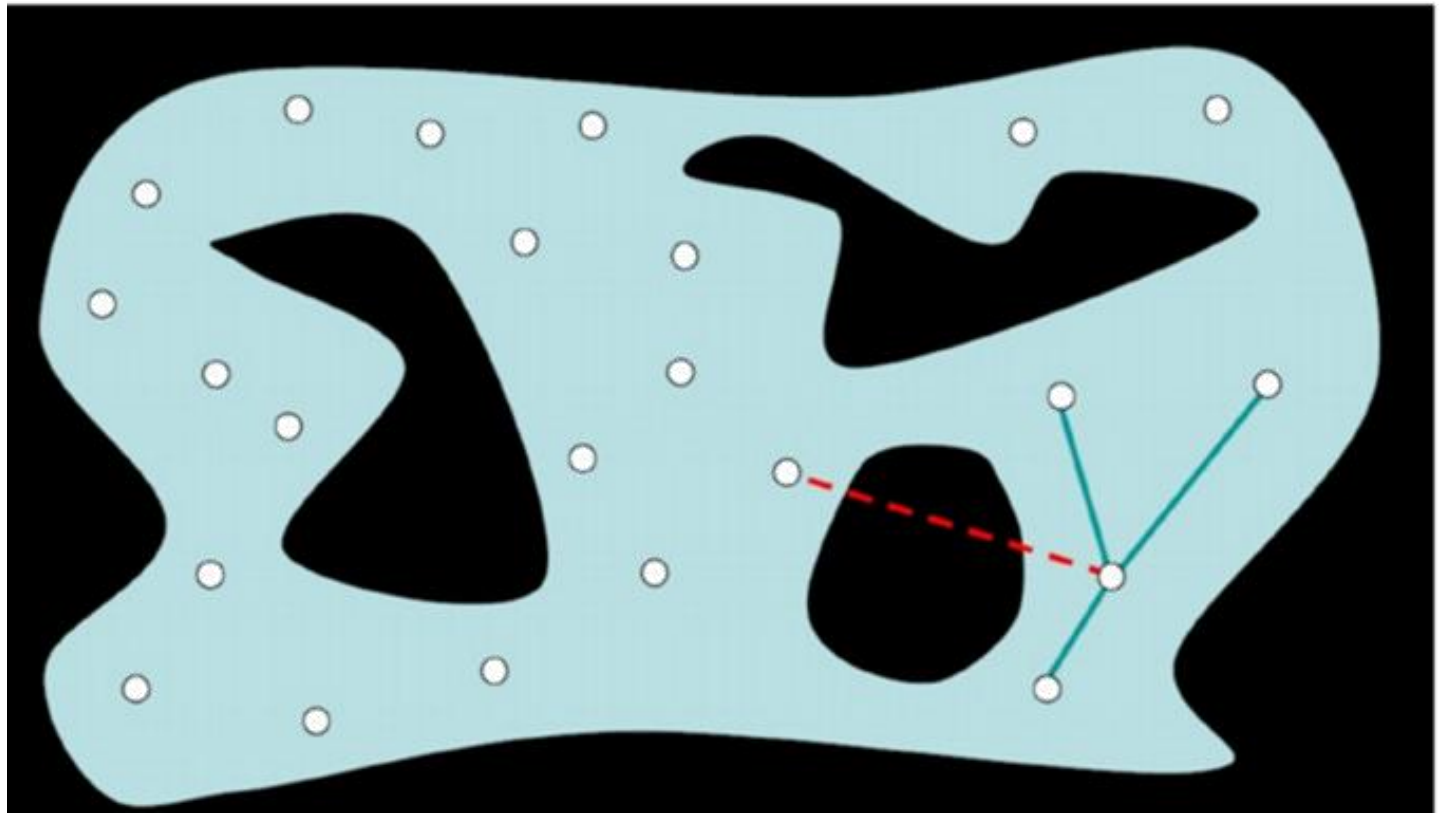
Probabilistic Road-Map (PRM)

- Connect the k-nearest neighbors.



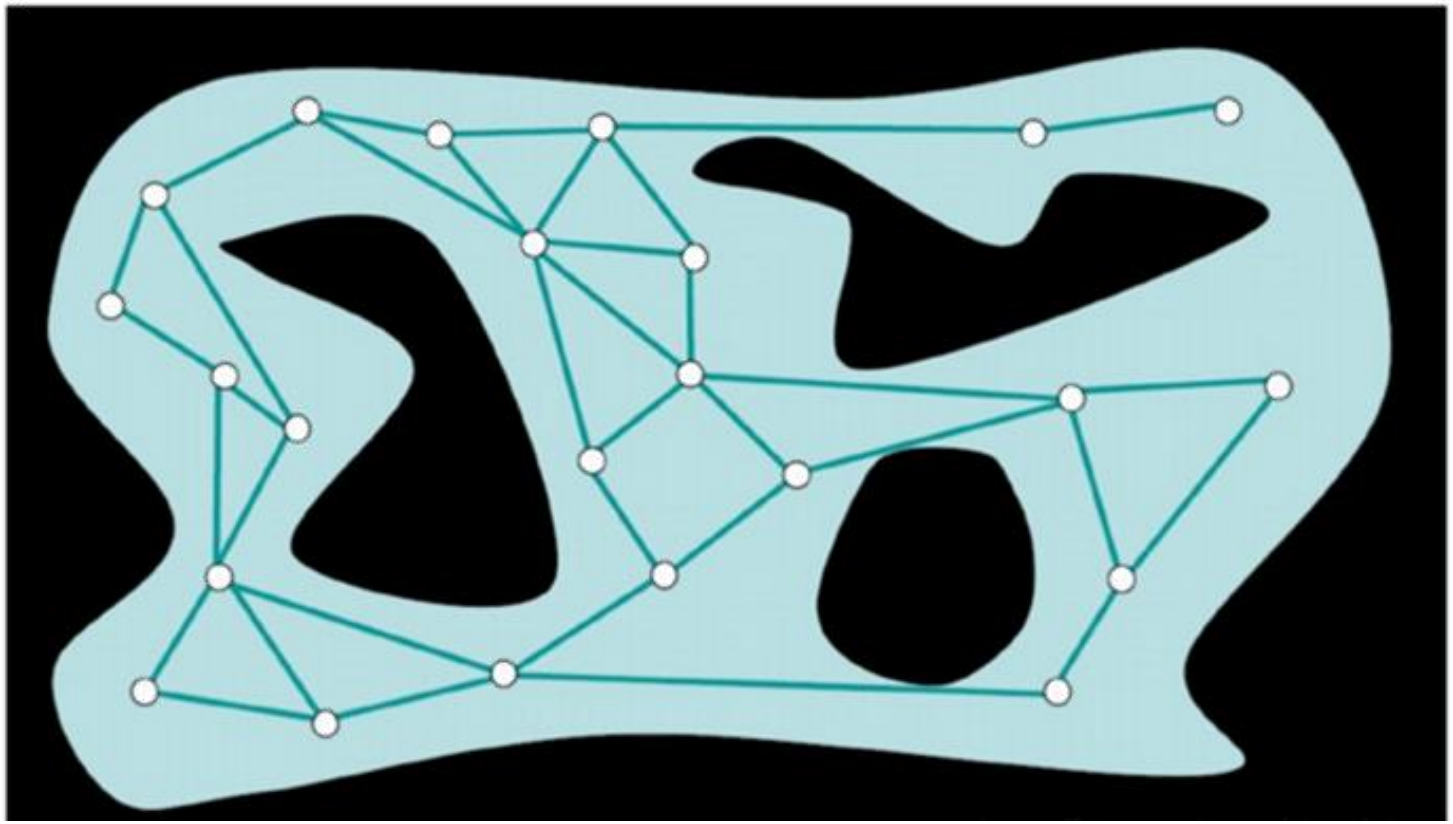
Probabilistic Road-Map (PRM)

- Delete the edges that cross through the occupied area.



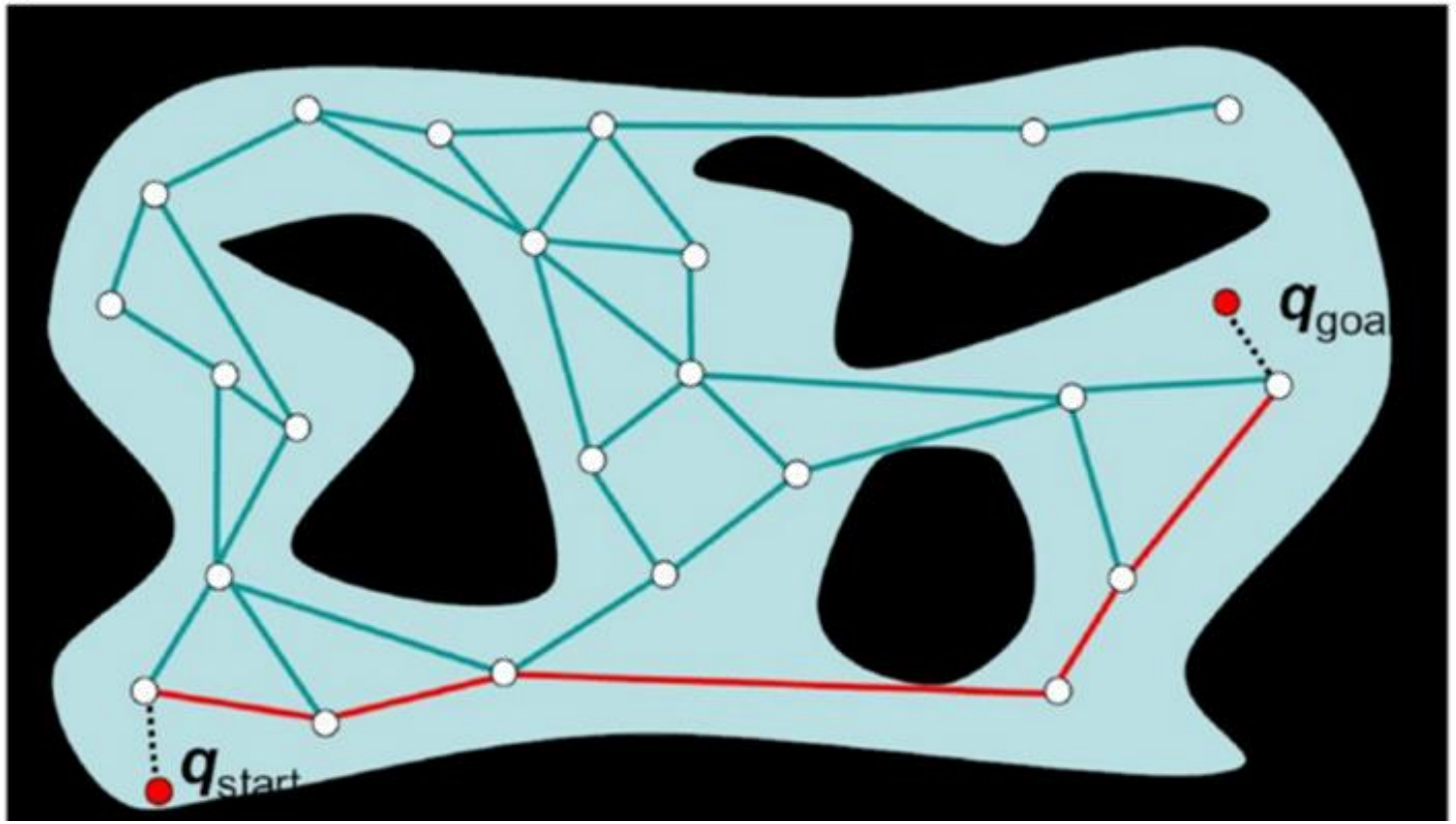
Probabilistic Road-Map (PRM)

- Connect each connected component to get the probabilistic road-map.



Probabilistic Road-Map (PRM)

- Connect the source and destination points to the constructed graph, and plan the path by utilizing graph search methods.



Rapidly Exploring Random Tree (RRT)

- Probabilistic Road-Map (PRM) spends lots of time to build the graph. To solve this problem, Rapidly exploring Random Tree (RRT) dynamically generates the tree branch and checks collision, which is more efficient.
- The improved version of RRT is called RRT*, which is the most commonly used path planning algorithm for mobile robot.
- RRT Algorithm:

G.initialize(x_{start}) # Initialize Graph
for i=1 **to** max_iter **do**
  $x_{rand} \leftarrow$ sample() # Sample the points in 2d space.
  $x_{nearest} \leftarrow$ nearest(x_{rand} , G) # Find the nearest point in graph.
  $x_{new} \leftarrow$ steer(x_{rand} , $x_{nearest}$, step_size) # Collision detection.
  G.add_node(x_{new})
  G.add_edge(x_{new} , x_{parent})
  if $||x_{new} - x_{goal}|| <$ threshold **then**
    return G

Rapidly Exploring Random Tree (RRT) (Cont.)

- Initialize graph

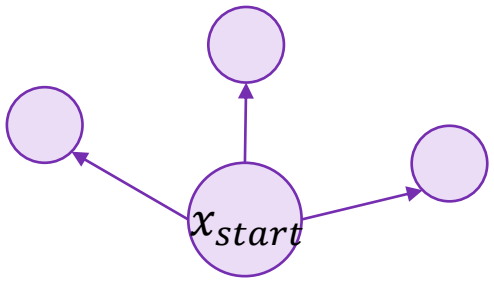
x_{goal}

x_{start}

Rapidly Exploring Random Tree (RRT) (Cont.)

- Sample points on 2d space

Probability (p) for sampling random point,
and probability (1-p) for choosing the goal point.

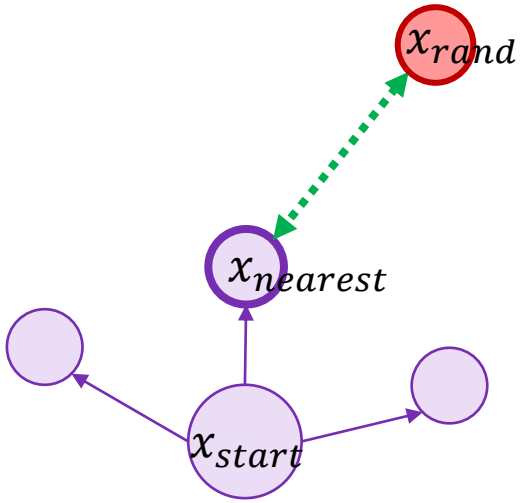


x_{rand}

x_{goal}

Rapidly Exploring Random Tree (RRT) (Cont.)

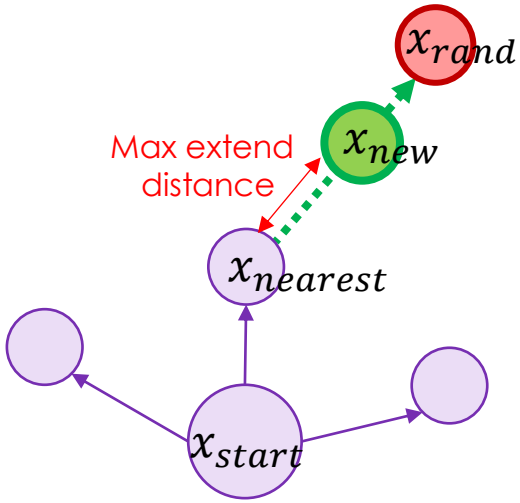
- Find the nearest point in graph.



x_{goal}

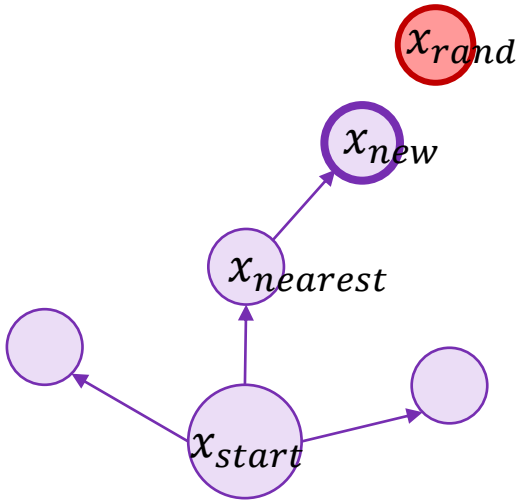
Rapidly Exploring Random Tree (RRT) (Cont.)

- Extend the branch of the nearest node and check the collision.



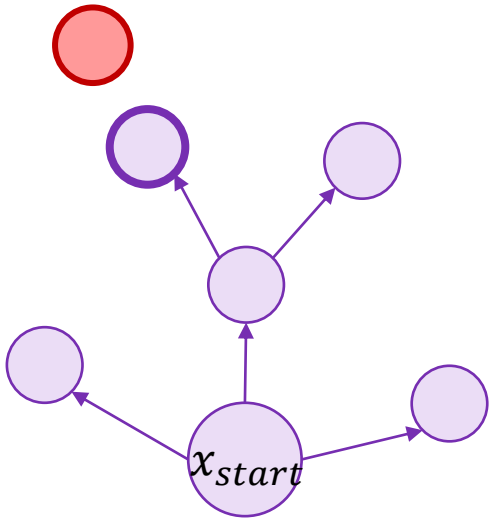
Rapidly Exploring Random Tree (RRT) (Cont.)

- Add the new node and edge into the graph

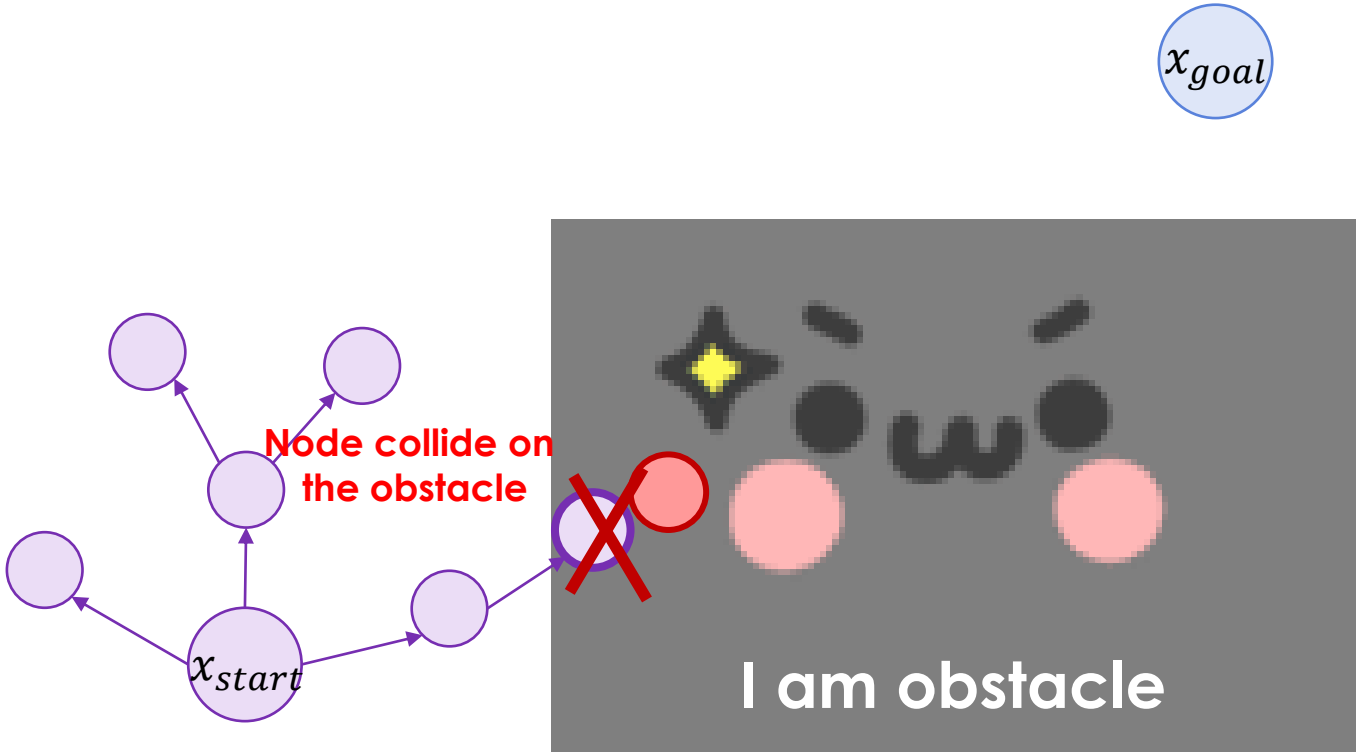


x_{goal}

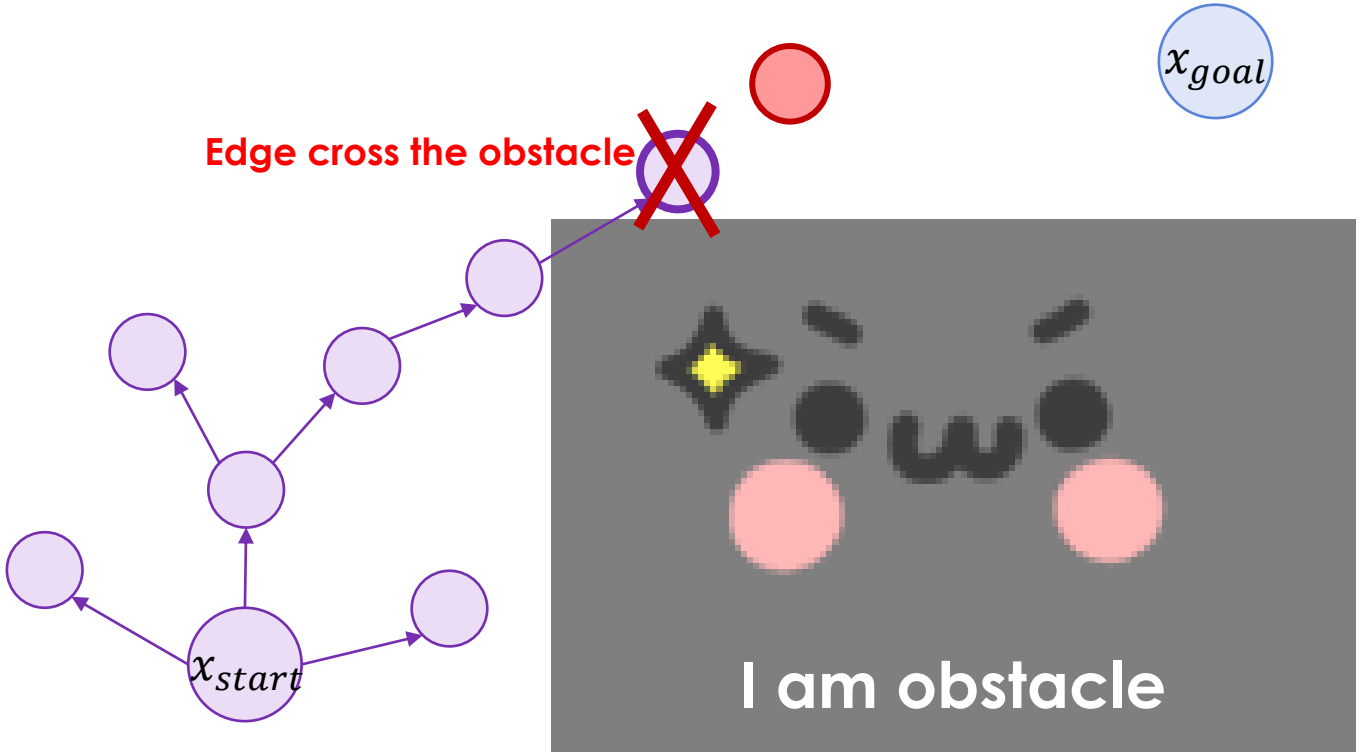
Rapidly Exploring Random Tree (RRT) (Cont.)



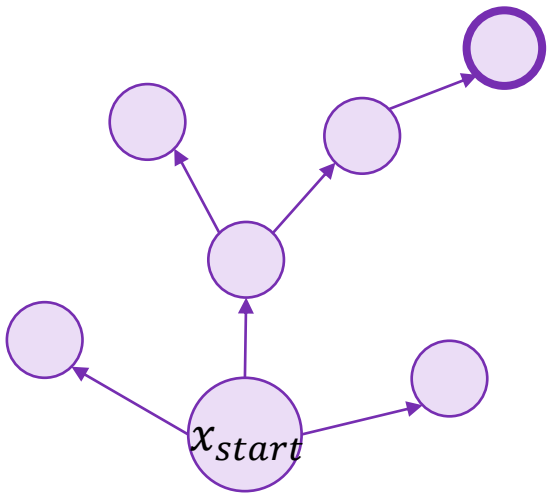
Rapidly Exploring Random Tree (RRT) (Cont.)



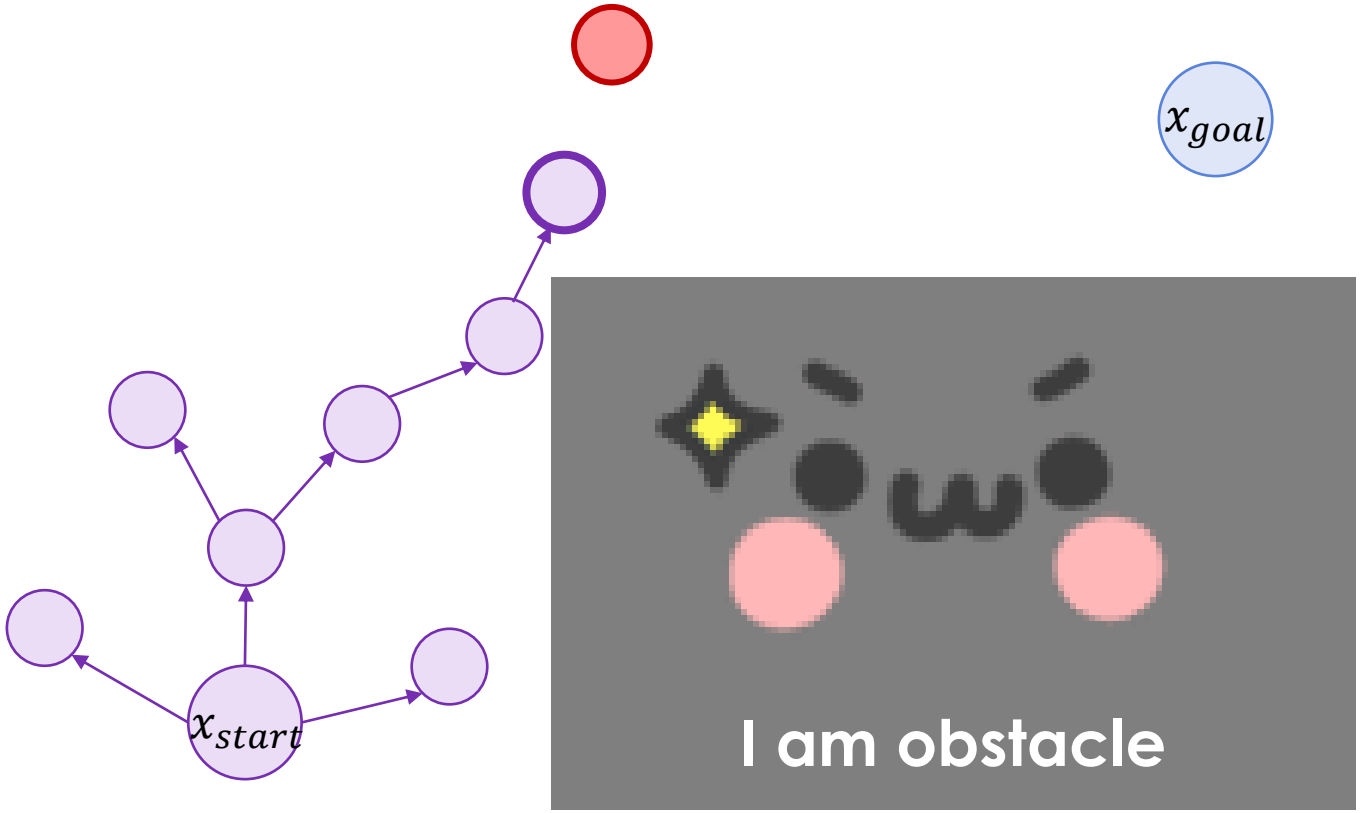
Rapidly Exploring Random Tree (RRT) (Cont.)



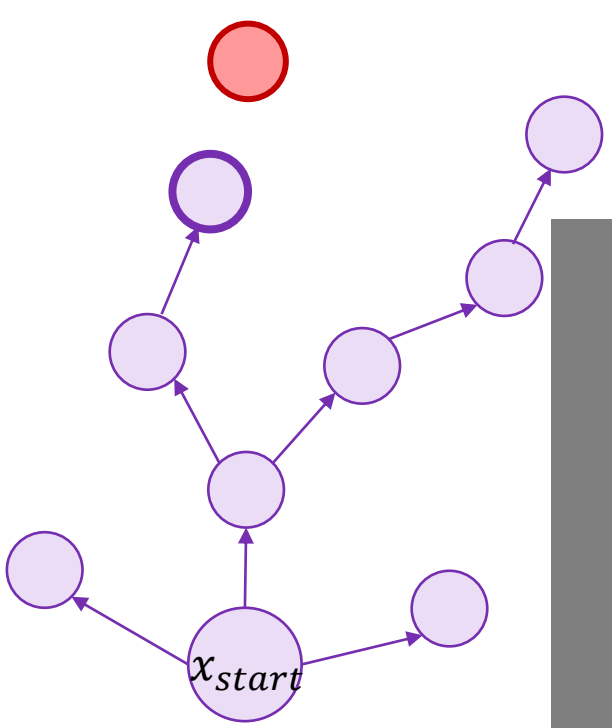
Rapidly Exploring Random Tree (RRT) (Cont.)



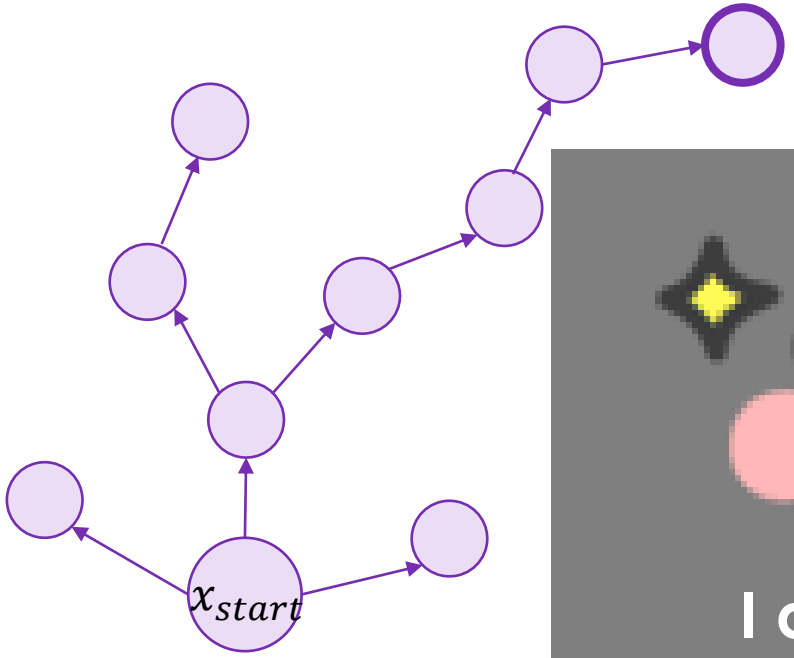
Rapidly Exploring Random Tree (RRT) (Cont.)

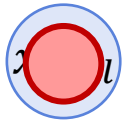


Rapidly Exploring Random Tree (RRT) (Cont.)

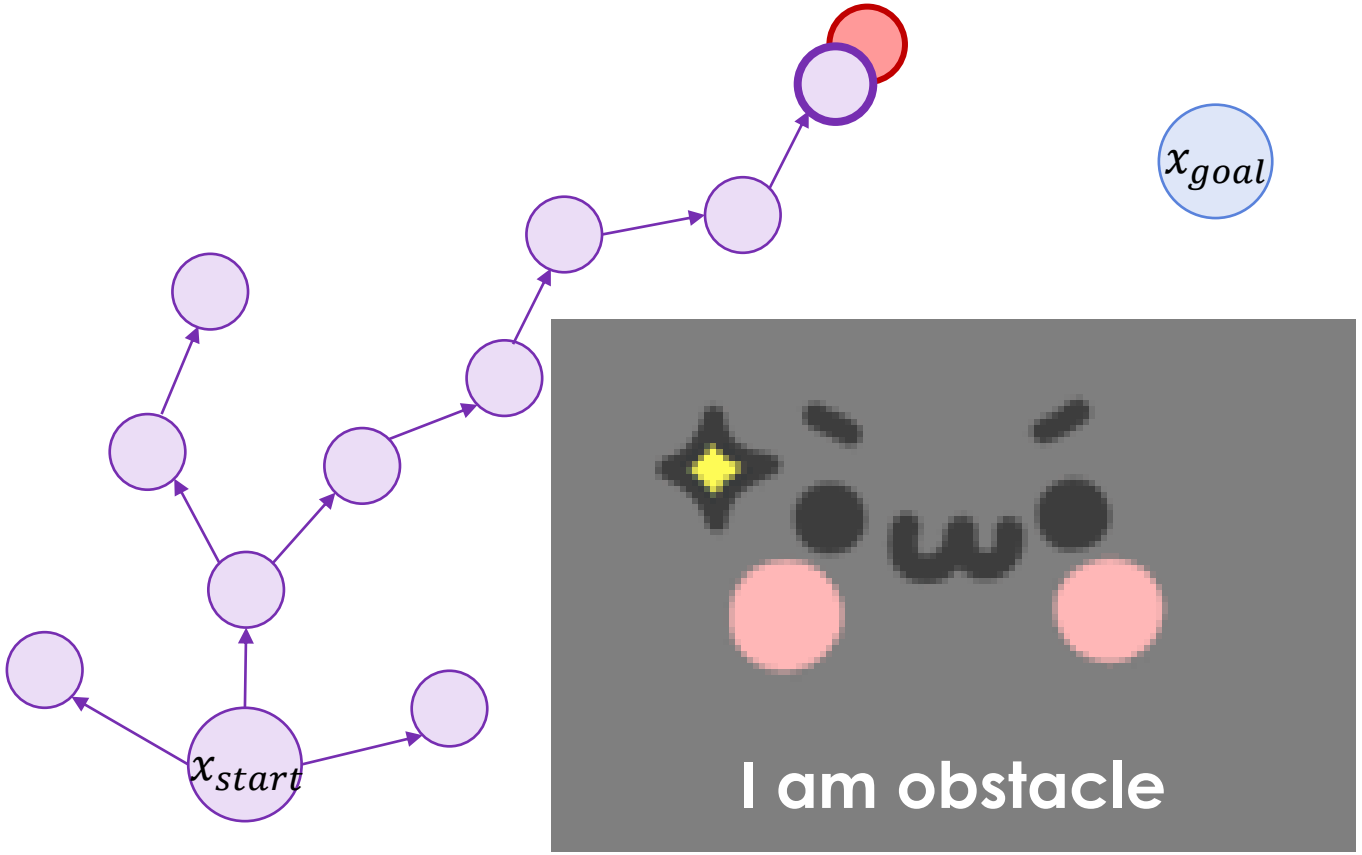


Rapidly Exploring Random Tree (RRT) (Cont.)

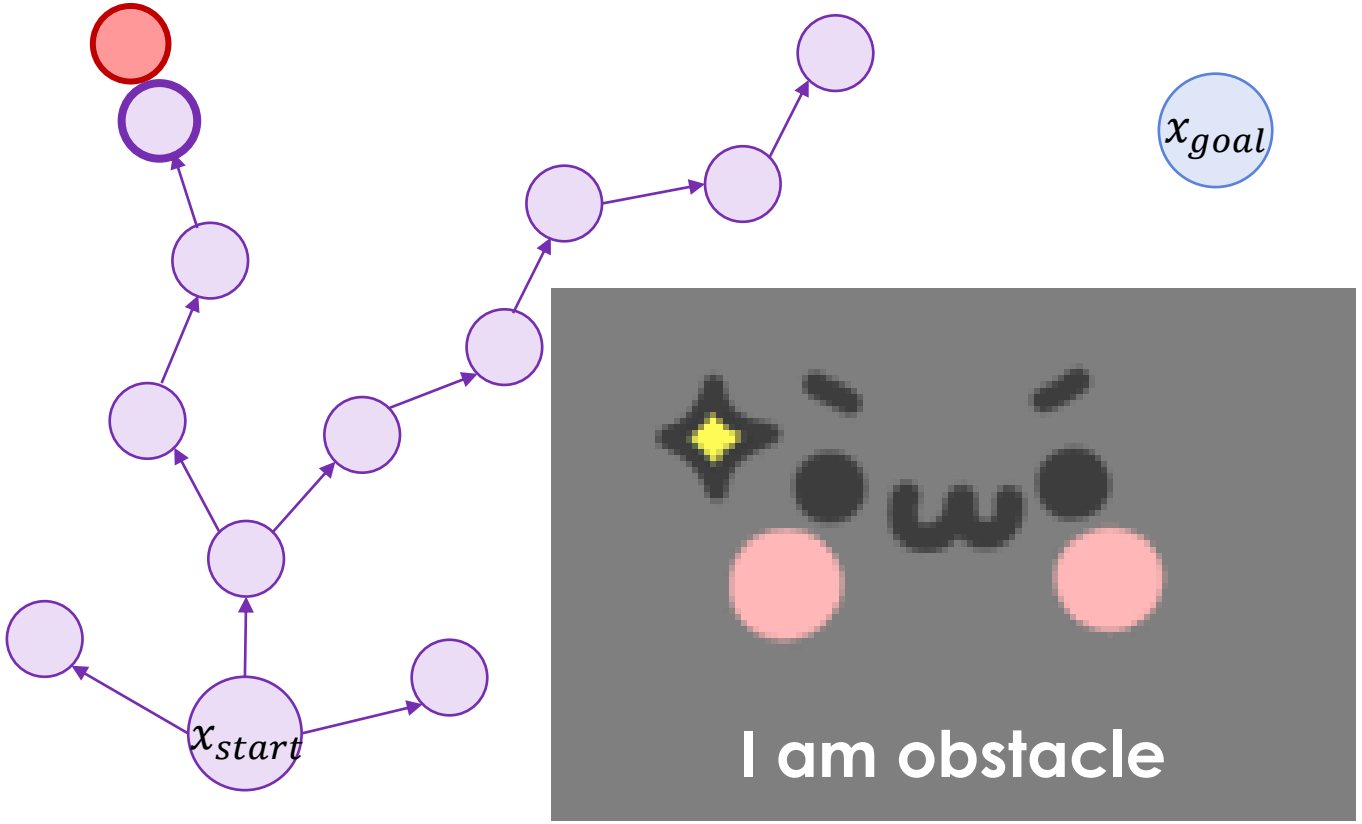




Rapidly Exploring Random Tree (RRT) (Cont.)

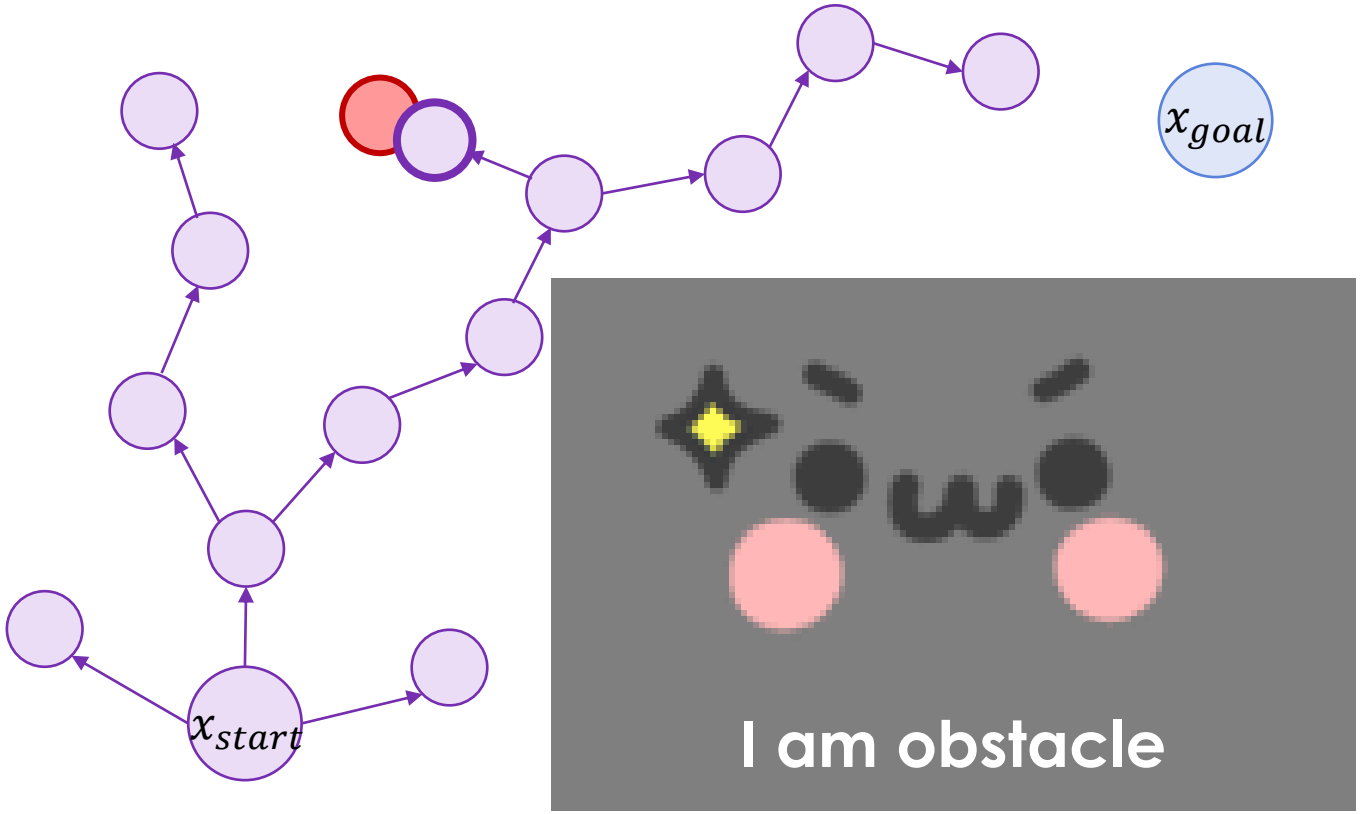


Rapidly Exploring Random Tree (RRT) (Cont.)



Rapidly Exploring Random Tree (RRT) (Cont.)

Rapidly Exploring Random Tree (RRT) (Cont.)



Rapidly Exploring Random Tree (RRT) (Cont.)

Rapidly Exploring Random Tree (RRT) (Cont.)

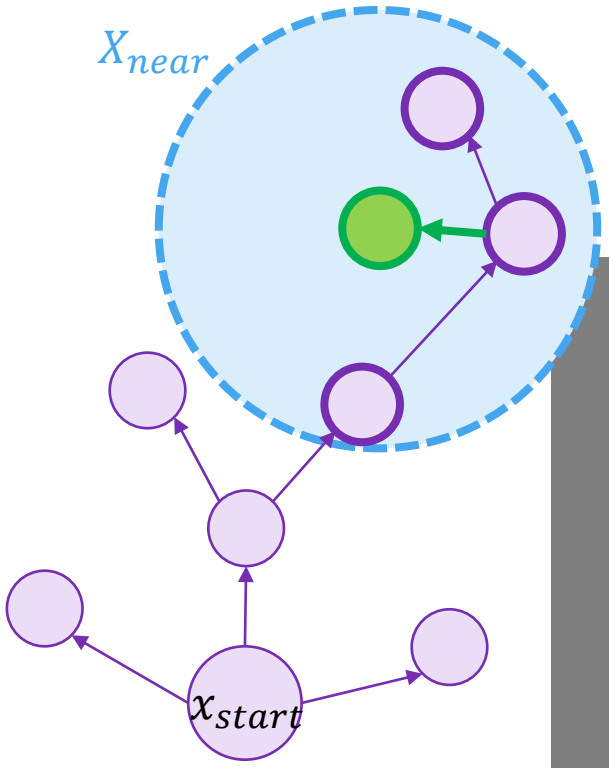
RRT*

- The path planned by RRT is winding and jerky, RRT star utilizes the re-parent and the re-wire process to improve the smoothness.
- RRT* Algorithm

G.initialize(x_{start}) # Initialize Graph
for i=1 **to** max_iter **do**
  $x_{rand} \leftarrow$ sample() # Sample the points in 2d space.
  $x_{nearest} \leftarrow$ nearest(x_{rand} , G) # Find the nearest point in graph.
  $x_{new} \leftarrow$ steer(x_{rand} , $x_{nearest}$, step_size) # Collision detection.
  $X_{near} \leftarrow$ near_node(x_{new} , G)
  $x_{parent} \leftarrow$ best_parent(x_{parent} , X_{near}) # Re-Parent
  G.add_node(x_{new})
  G.add_edge(x_{new} , x_{parent})
  G.rewire(x_{new} , X_{near}) # Rewire
  if $||x_{new} - x_{goal}|| <$ threshold **then**
    return G

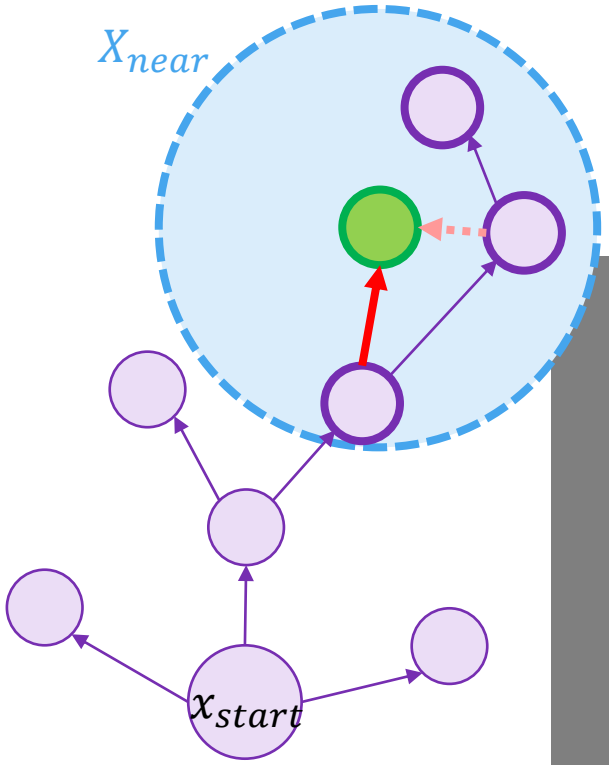
RRT* - Near Nodes

- Find the near nodes set of the new node in a distance range.



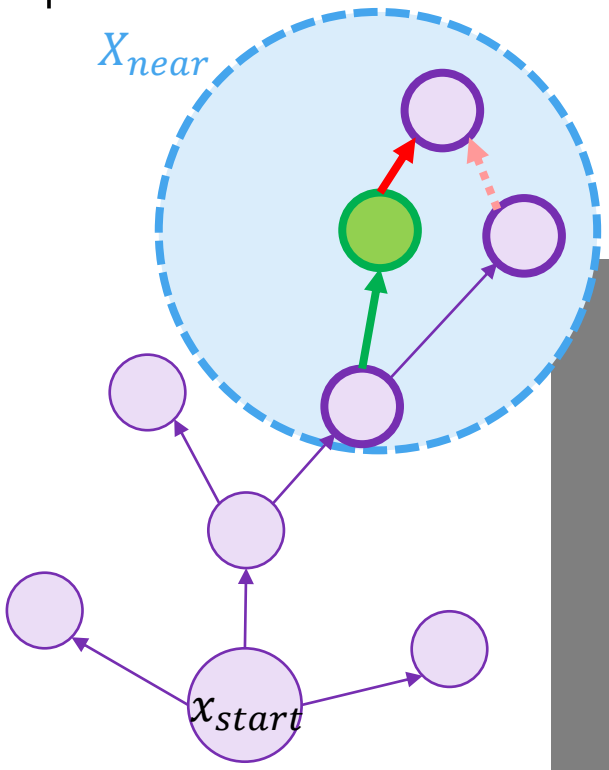
RRT* - Re-Parents Process

- Check if the cost of the path is smaller when selecting a different parent for the new node in the near nodes sets.



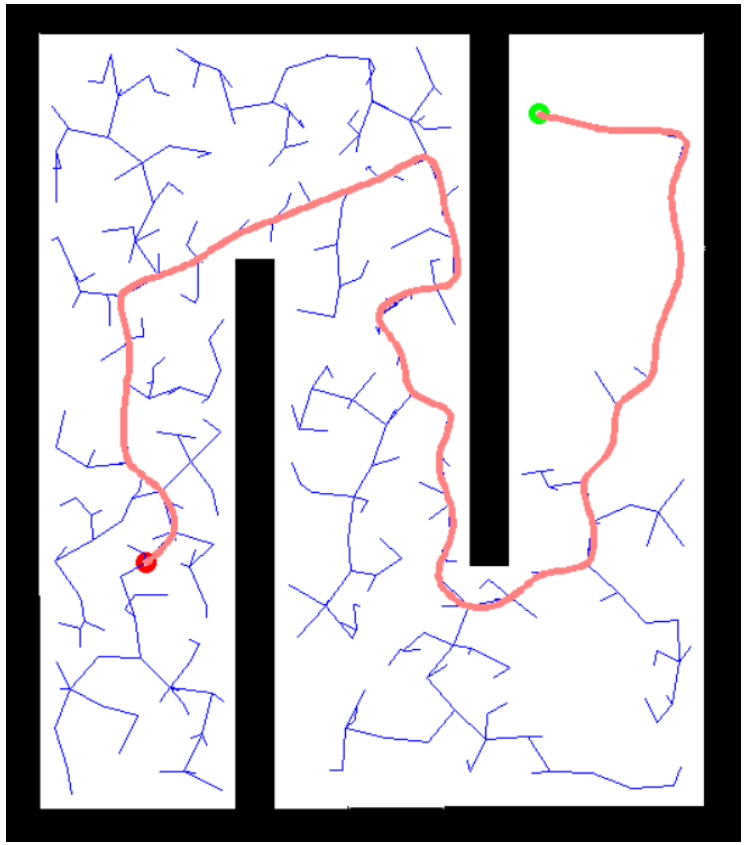
RRT* - Re-Wire Process

- Check if the cost of path is smaller when selecting the new node as the parent of the node in near nodes set.

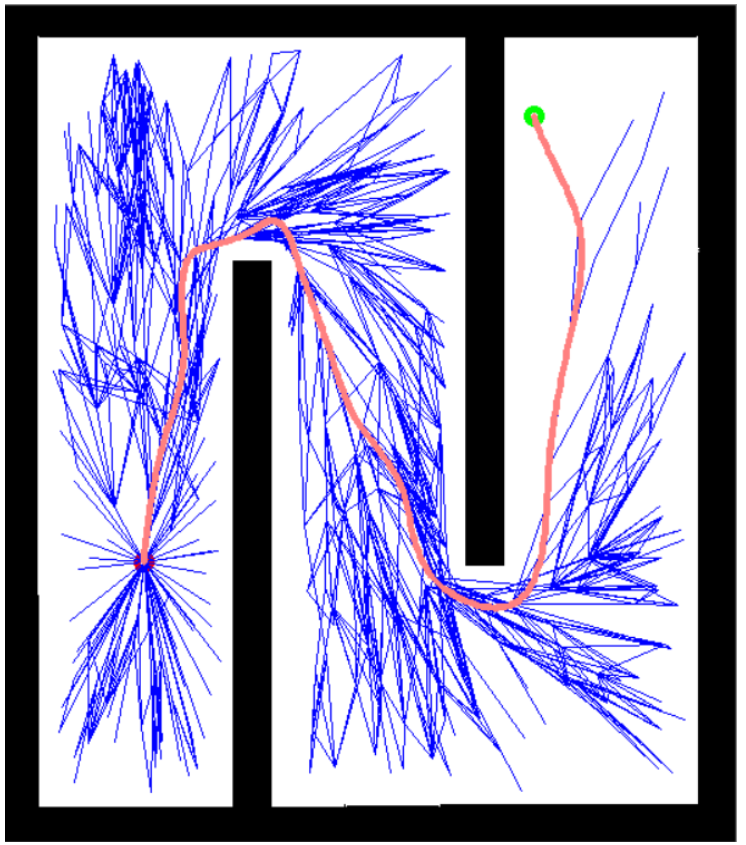


Comparison of RRT and RRT*

RRT

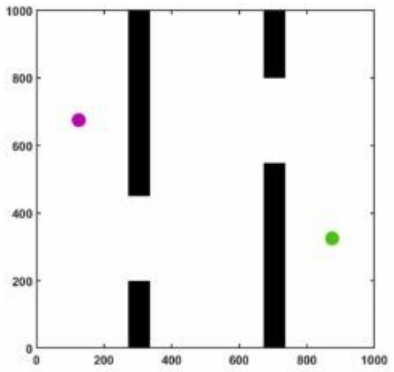


RRT*

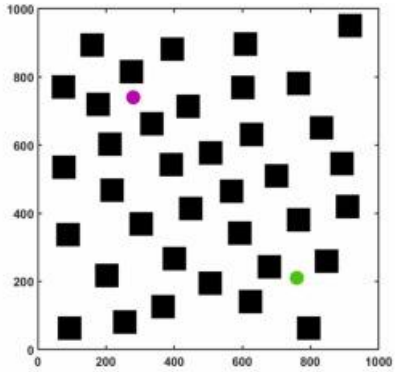


RRT* Algorithm Improvement

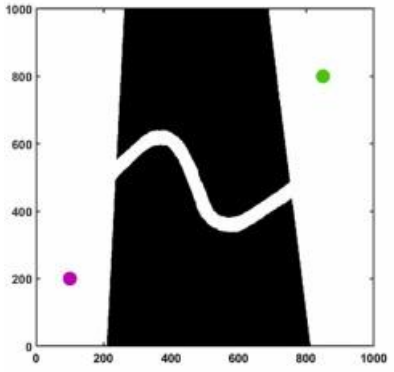
- To improve the convergence time for finding the path solution in narrow and maze environments:
  - Greedy heuristic search strategy
  - Path-expanding methods for reducing the sampling area



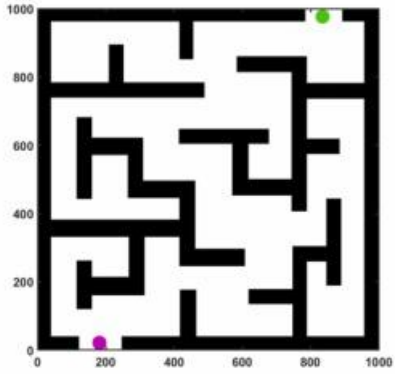
(a) General environment



(b) Cluttered environment



(c) Narrow corridor environment

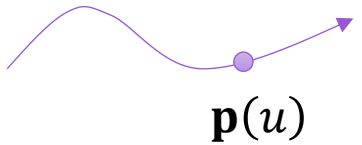


(d) Maze environment

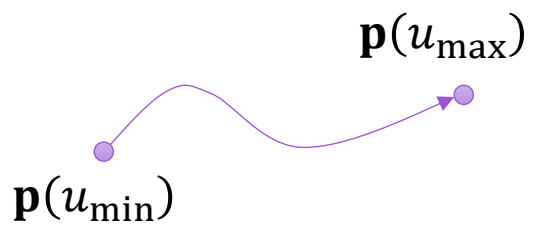
Curve Interpolation

Representation of Curves

- Explicit Representation:
  - $y = f(x)$ or $x = g(y)$
  - No guarantee that either form exists for a given curve (e.g., vertical line and circle)
- Implicit Representation:
  - $f(x, y) = 0$
    - Line: $ax + by + c = 0$
    - Circle: $x^2 + y^2 - r^2 = 0$
  - Does represent all lines and circles, but difficult to obtain all points on the curve
- Parametric Representation:
  - Curve: $\mathbf{p}(u) = [x(u) \quad y(u)]^T$
    - $\frac{d\mathbf{p}(u)}{du} = \left[\frac{dx(u)}{du} \quad \frac{dy(u)}{du}\right]^T$
      - the velocity with which the curve is traced out (tangent direction at point **p**)
  - Most flexible and robust form, but not unique



Parametric Cubic Polynomial Curves



- We can use polynomial functions to form curves:
  - $\mathbf{p}(u) = \boldsymbol{c_0} + \boldsymbol{c_1}u + \boldsymbol{c_2}u^2 + \cdots \boldsymbol{c_n}u^n$
  - degree of freedom: n+1
  - cubic polynomial curve: $\mathbf{p}(u) = \boldsymbol{c_0} + \boldsymbol{c_1}u + \boldsymbol{c_2}u^2 + \boldsymbol{c_3}u^3 = \mathbf{u}^T\mathbf{c}$
    - Low freedom, but sufficient to produce the desired shape in a small region
  - $0 \le u \le 1$
- The problem is how to efficiently find out the coefficient $\boldsymbol{c_i}$
- Least square curve fitting:

$$x(u) = c_{x0} + c_{x1}\,u + c_{x2}u^2 + c_{x3}u^3$$
$$y(u) = c_{y0} + c_{y1}\,u + c_{y2}u^2 + c_{y3}u^3$$

➡ Need 4 points to solve 8 unknowns

$$\mathbf{p}(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3$$

Least Square Curve Fitting

$$\mathbf{c}_k = \begin{bmatrix} c_{kx} \\ c_{ky} \end{bmatrix}$$

- Given 4 control points $\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}$, and $\mathbf{p_3}$
  - Assume the 4 points are with equally spaced values *u*:

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \end{bmatrix}$$

$$\left.\begin{aligned} \mathbf{p}_0 &= \mathbf{p}(0) = \mathbf{c}_0 \\ \mathbf{p}_1 &= \mathbf{p}\left(\frac{\mathbf{1}}{\mathbf{3}}\right) = \mathbf{c}_0 + \frac{1}{3}\mathbf{c}_1 + \left(\frac{\mathbf{1}}{\mathbf{3}}\right)^2 \mathbf{c}_2 + \left(\frac{\mathbf{1}}{\mathbf{3}}\right)^3 \mathbf{c}_3 \\ \mathbf{p}_2 &= \mathbf{p}\left(\frac{\mathbf{2}}{\mathbf{3}}\right) = \mathbf{c}_0 + \frac{2}{3}\mathbf{c}_1 + \left(\frac{\mathbf{2}}{\mathbf{3}}\right)^2 \mathbf{c}_2 + \left(\frac{\mathbf{2}}{\mathbf{3}}\right)^3 \mathbf{c}_3 \\ \mathbf{p}_3 &= \mathbf{p}(\mathbf{1}) = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3 \end{aligned}\right\} \Rightarrow \mathbf{P} = \mathbf{A}\mathbf{c}$$

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & \left(\frac{\mathbf{1}}{\mathbf{3}}\right)^2 & \left(\frac{\mathbf{1}}{\mathbf{3}}\right)^3 \\ 1 & \frac{2}{3} & \left(\frac{\mathbf{2}}{\mathbf{3}}\right)^2 & \left(\frac{\mathbf{2}}{\mathbf{3}}\right)^3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

nonsingular

$$\mathbf{M_I} = \mathbf{A}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix} \quad \Rightarrow \mathbf{c} = \mathbf{M_I}\mathbf{P}$$

$$\mathbf{p}(u) = \mathbf{u}^T\mathbf{c} = \mathbf{u}^T\mathbf{M_I}\mathbf{P}$$

Interpolating Geometry Matrix

Cubic Interpolating Curves

- Rather than deriving a single interpolating curve of degree m for all the points, derive a set of cubic interpolating curves.
- If each segment is derived by letting u varying equally over the interval [0,1], then the matrix $\mathbf{M_I}$ is the same for each segment.
- Derivatives at the joint points will not be continuous.

Hermite Curves

$$\mathbf{p}(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3$$
$$\mathbf{p}'(u) = \mathbf{c}_1 + 2u\mathbf{c}_2 + 3u^2\mathbf{c}_3$$

- A Hermite curve is a curve for which the user provides:
  - The endpoints of the curve:

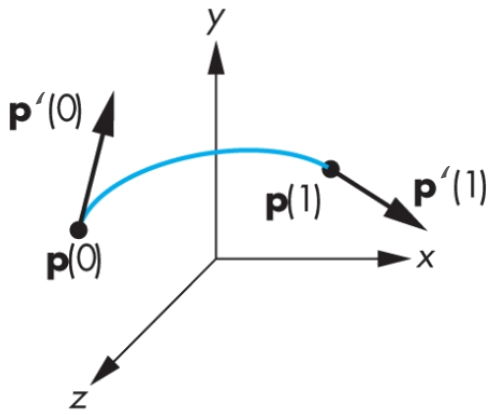
    $\mathbf{p}_0 = \mathbf{p}(0) = \mathbf{c}_0$

    $\mathbf{p}_3 = \mathbf{p}(1) = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3$

  - The derivatives of the curve at the endpoints:

    $\mathbf{p}'_0 = \mathbf{p}'(0) = \mathbf{c}_1$

    $\mathbf{p}'_3 = \mathbf{p}'(1) = \mathbf{c}_1 + 2\mathbf{c}_2 + 3\mathbf{c}_3$



$$\mathbf{Q} = \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_3 \\ \mathbf{p}'_0 \\ \mathbf{p}'_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \mathbf{c} \qquad \mathbf{M_H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & 2 & 1 & 1 \end{bmatrix}$$

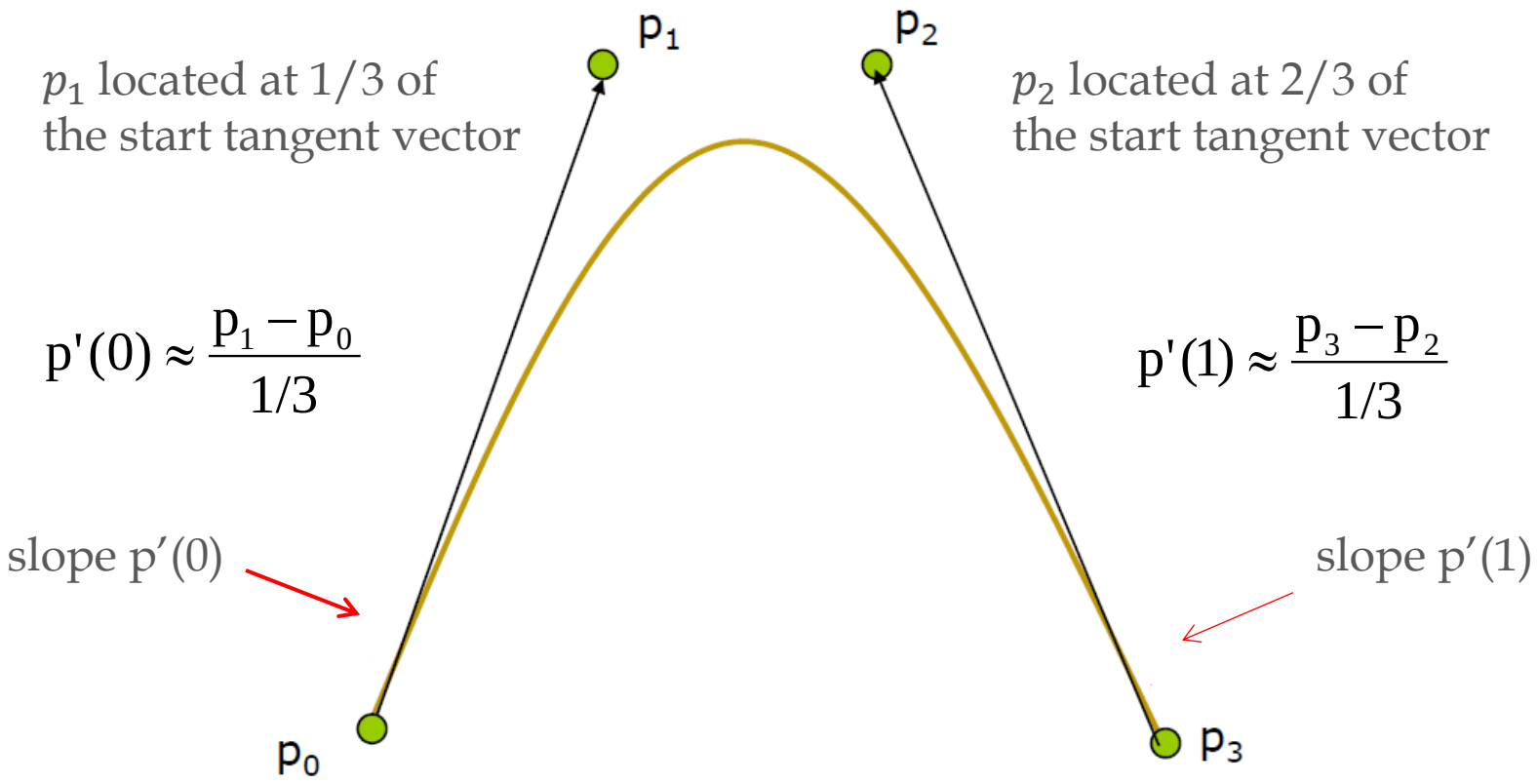
Hemite Geometry Matrix

$$\mathbf{c} = \mathbf{M_H Q}$$

$$\mathbf{p}(u) = \mathbf{u}^T \mathbf{c} = \mathbf{u}^T \mathbf{M}_H \mathbf{Q}$$

Bezier Curves

- Two control points define endpoints, and two points control the tangents.



$$\mathbf{p}(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3$$

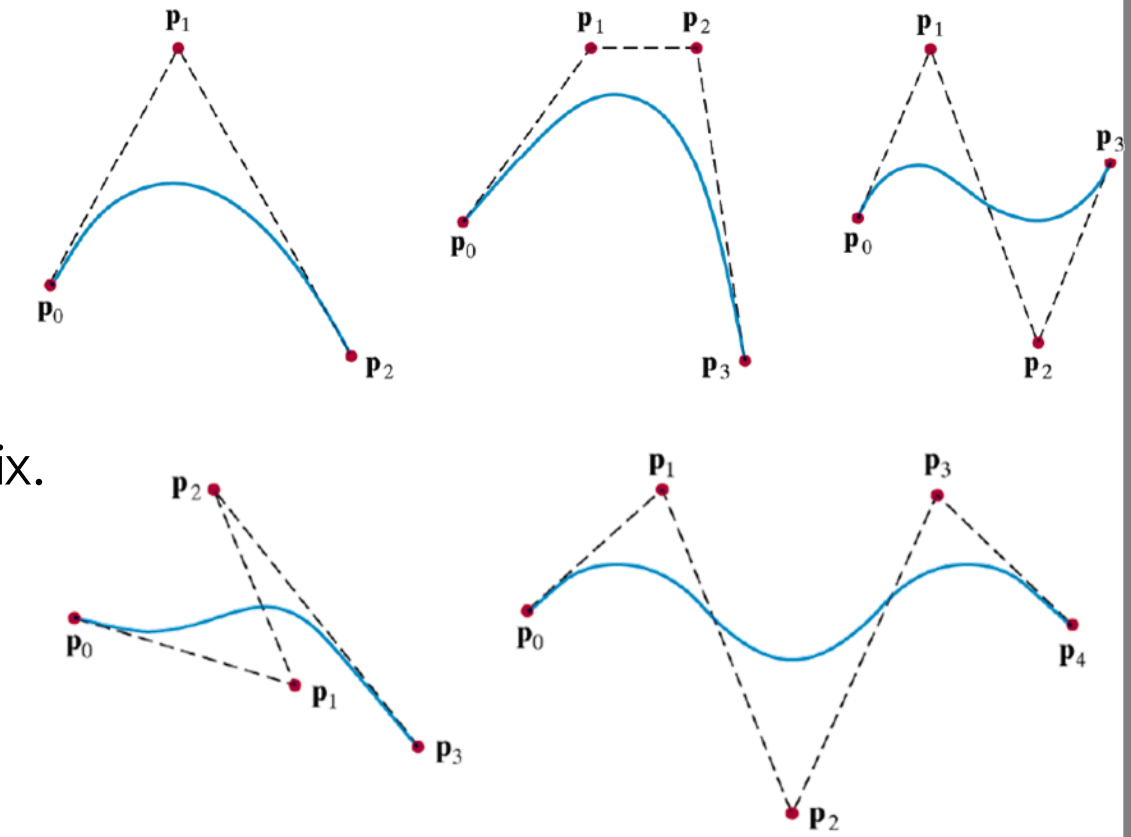
Bezier Curves (Cont.)

- The endsite conditions are the same
  - $P(0) = P_0 = c_0$
  - $P(1) = P_3 = c_0 + c_1 + c_2 + c_3$
- Approximating derivative conditions
  - $P'(0) = 3(P_1 - P_0) = c_1$
  - $P'(1) = 3(P_3 - P_2) = c_1 + 2c_2 + 3c_3$
- Replacing the original Hermite matrix.

$$\mathbf{c} = \mathbf{M}_B\mathbf{P} \qquad \mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

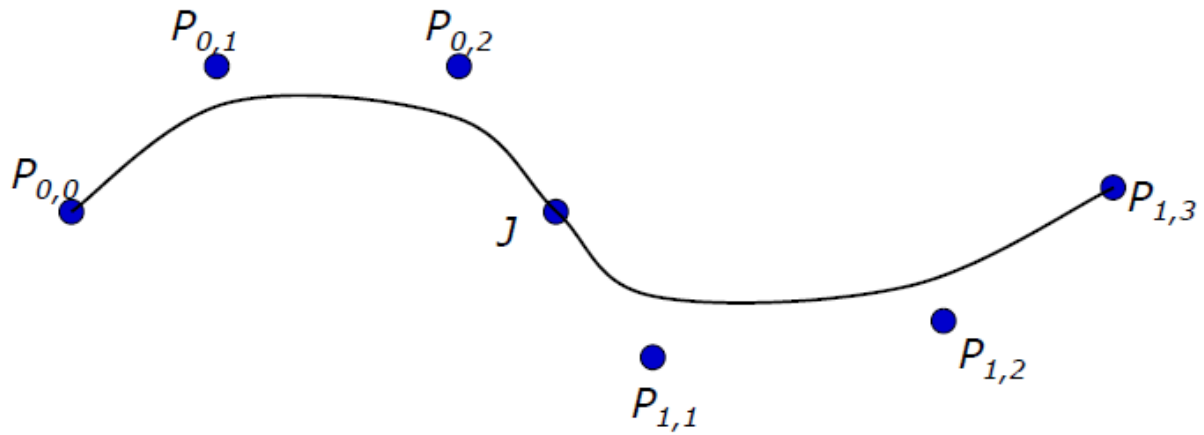
Bezier Geometry Matrix

$$\mathbf{p}(u) = \mathbf{u}^T\mathbf{c} = \mathbf{u}^T\mathbf{M}_B\mathbf{P}$$



Bezier Continuity

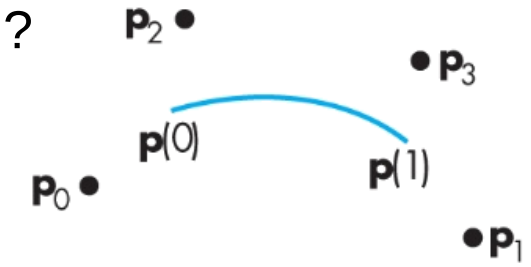
- We can make a long curve by concatenating multiple short Bezier curves.

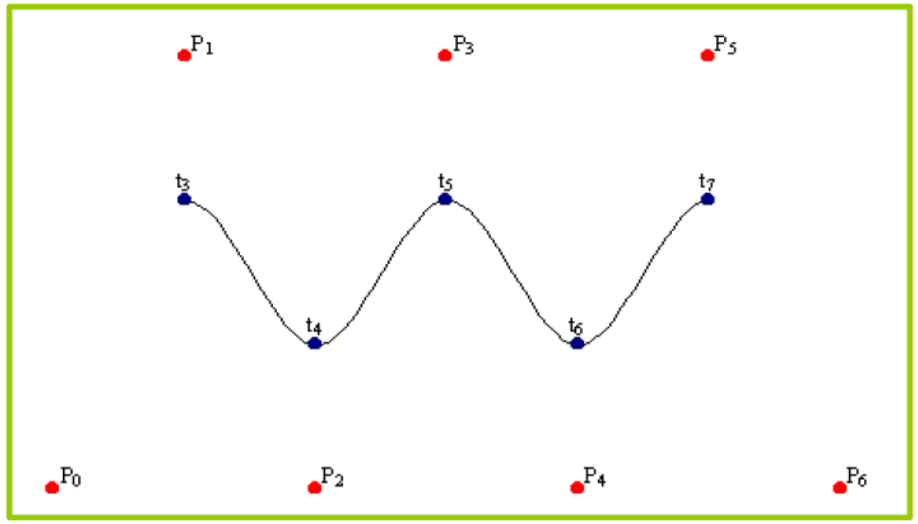


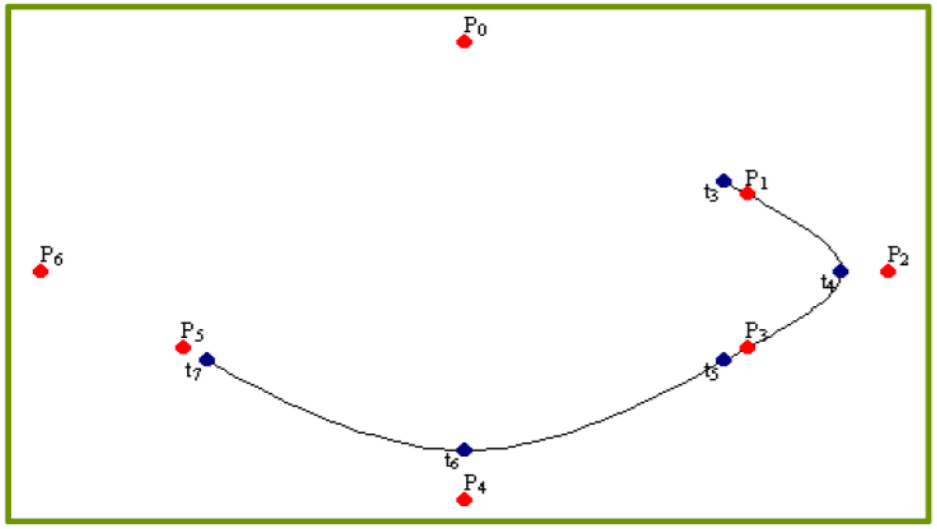
- How to keep the continuity?

Cubic B-spline Curves

- How to reach both C^2 continuity and local controllability ?
  - Slightly loose the endpoint constraints.
  - B-splines do not interpolate any of control points.







Figures from CG lecture note, U. Virginia

B-spline Curves (Cont.)

$$\mathbf{p}(0) = \mathbf{q}(1) = \frac{1}{6}(\mathbf{p}_{i-2} + 4\mathbf{p}_{i-1} + \mathbf{p}_i)$$

$$\mathbf{p}'(0) = \mathbf{q}'(1) = \frac{1}{2}(\mathbf{p}_i - \mathbf{p}_{i-2})$$

(for symmetric)

Since $\mathbf{p}(u) = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3 = u^T \mathbf{c}$

$$\mathbf{p}(0) = \mathbf{c}_0 = \frac{1}{6}(\mathbf{p}_{i-2} + 4\mathbf{p}_{i-1} + \mathbf{p}_i)$$

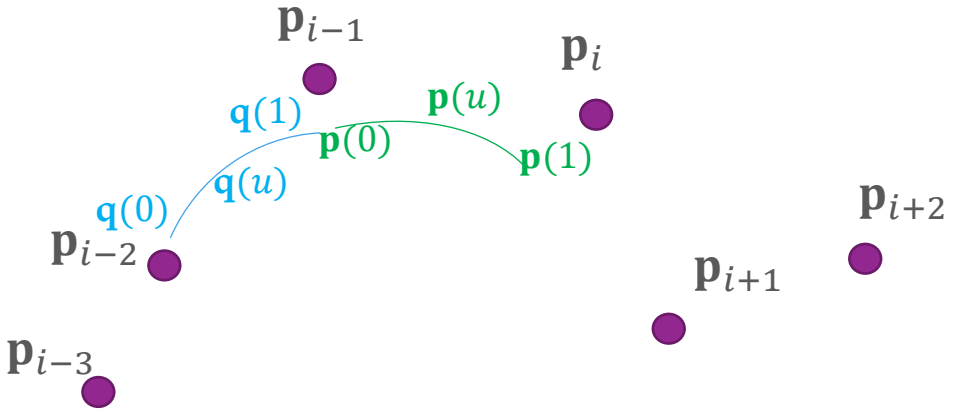
$$\mathbf{p}'(0) = \mathbf{c}_1 = \frac{1}{2}(\mathbf{p}_i - \mathbf{p}_{i-2})$$

$$\mathbf{p}(1) = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3 = \frac{1}{6}(\mathbf{p}_{i-1} + 4\mathbf{p}_i + \mathbf{p}_{i+1})$$

$$\mathbf{p}'(1) = \mathbf{c}_1 + 2\mathbf{c}_2 + 3\mathbf{c}_3 = \frac{1}{2}(\mathbf{p}_{i+1} - \mathbf{p}_{i-1})$$

$$\Rightarrow \mathbf{P} = \mathbf{A}\mathbf{c}$$

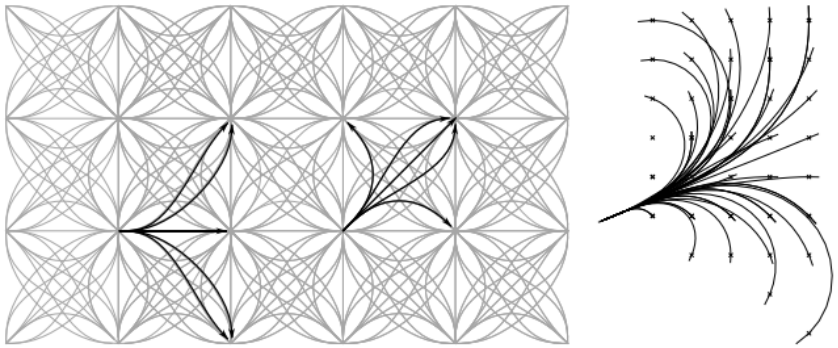
$$\mathbf{M}_S = \mathbf{A}^{-1} = \frac{1}{6}\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \quad \Rightarrow \mathbf{c} = \mathbf{M}_S\mathbf{P}$$



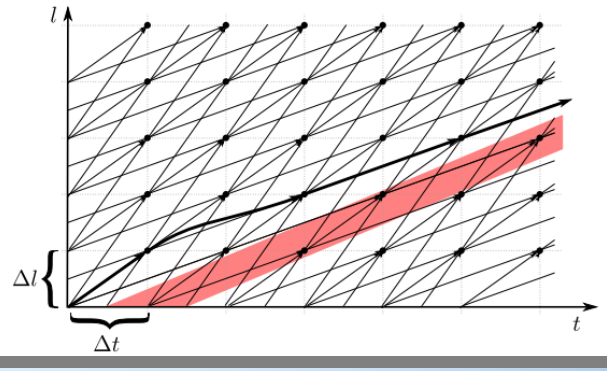
Trajectory Planning

State Lattices Planning

- Originally, the concept of state lattice is based on the configuration space. The following image illustrates the nontemporal state lattice on 2D workspace which considers the curvature.



- Later, the space-time manifold is constructed by combining the configuration space and time, which can be used in the dynamic environment.



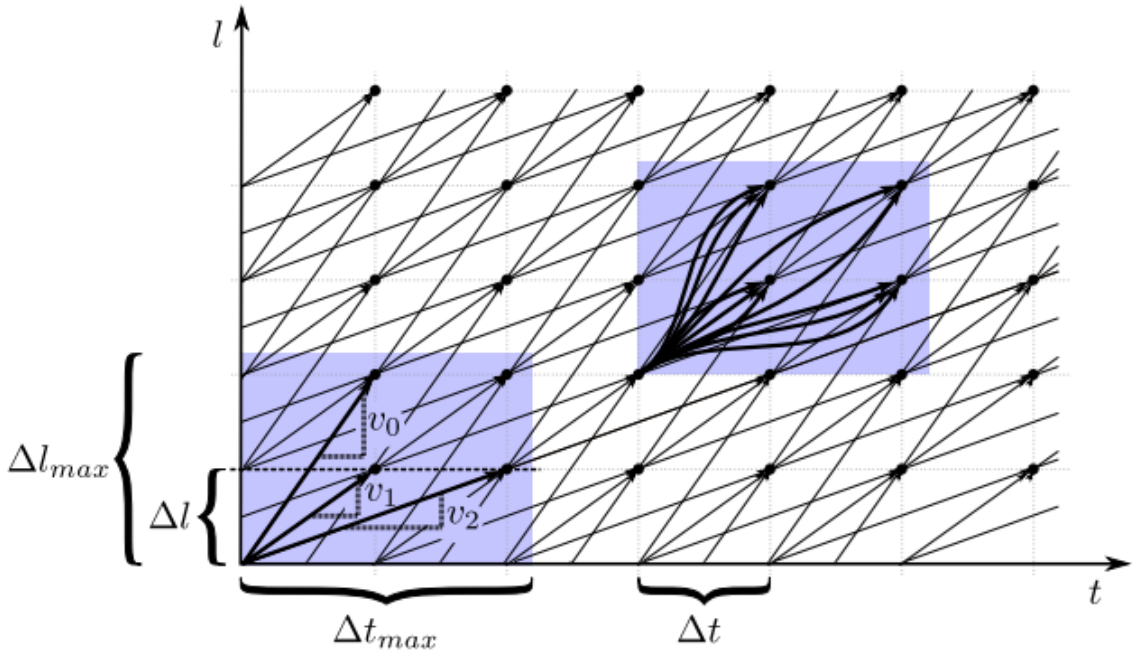
State Lattices Planning

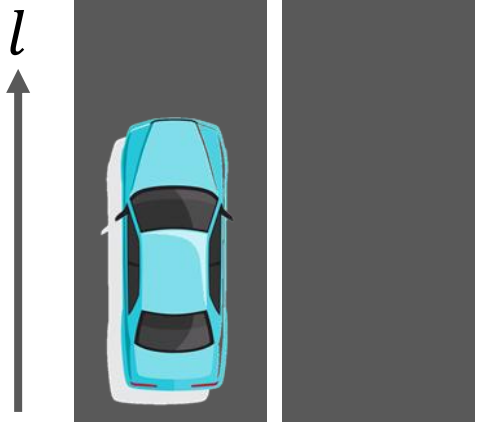
- State Lattice Planning Workflow

1. Sample the candidate trajectories.
2. Compute the cost by pre-defined cost function.
3. Check the collision and Motion constraints and select the low-cost trajectory as output.

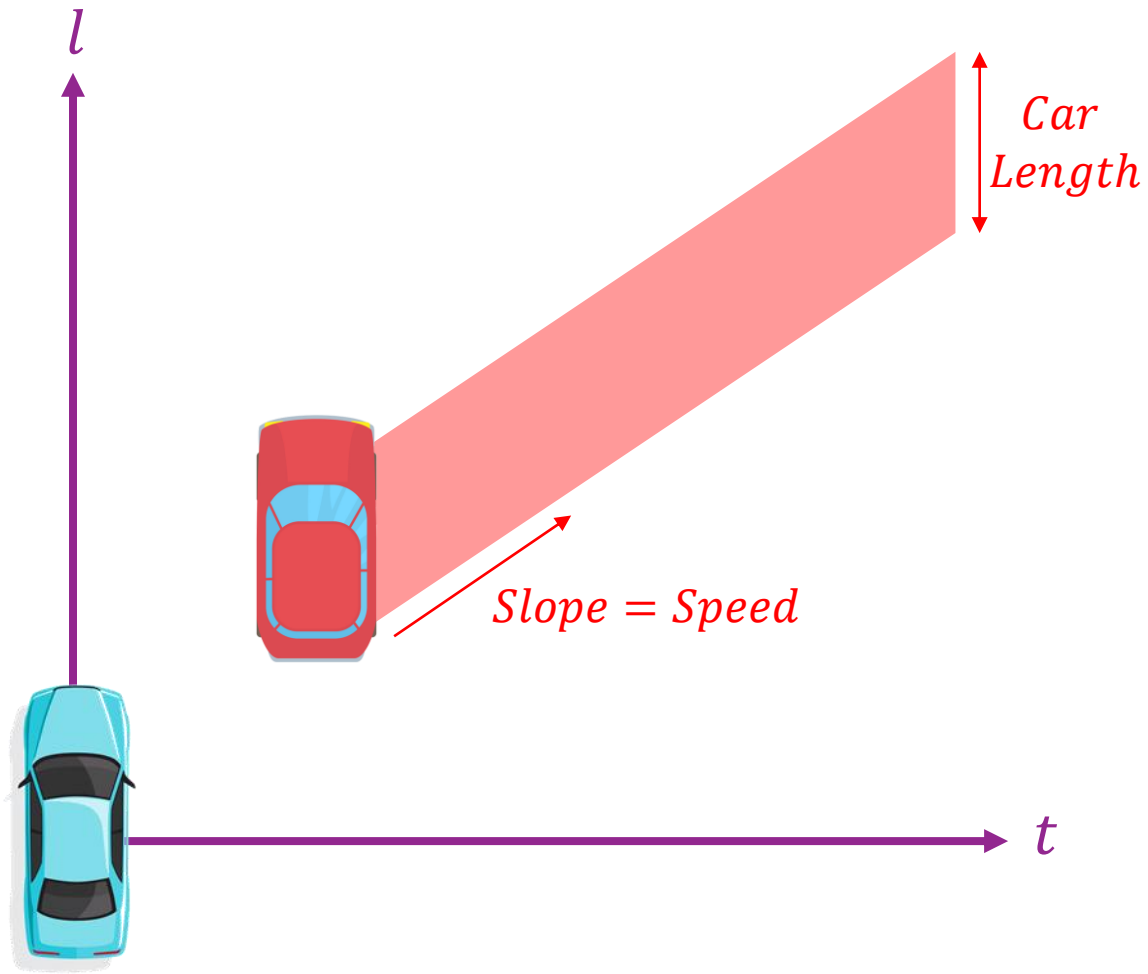
Spatiotemporal State Lattices

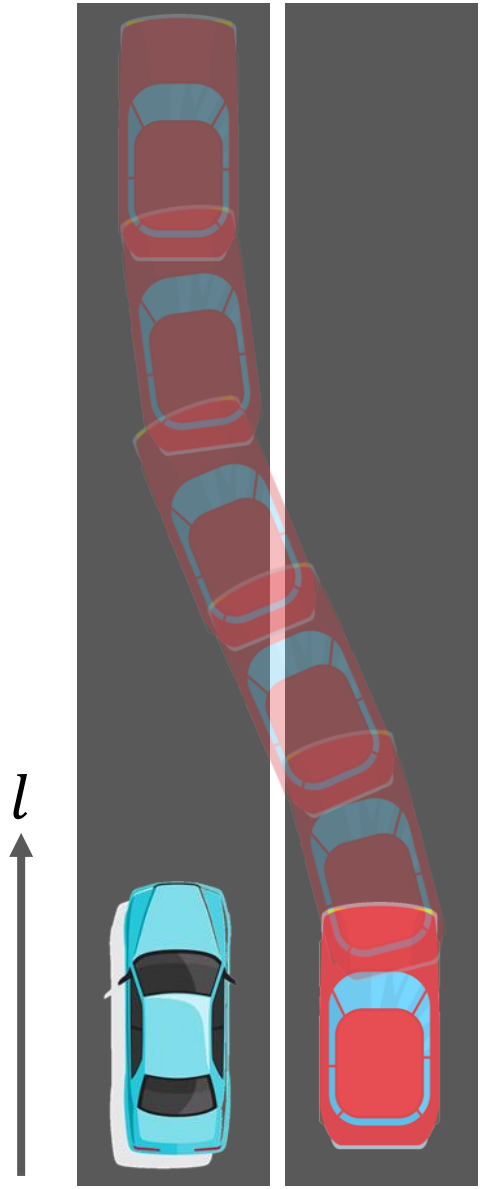
- Consider a spatiotemporal state lattice over a one dimensional workspace.
- $\Delta l, \Delta t$: spatial and temporal resolution.
- $\Delta l_{max}, \Delta t_{max}$: spatial and temporal constraints.



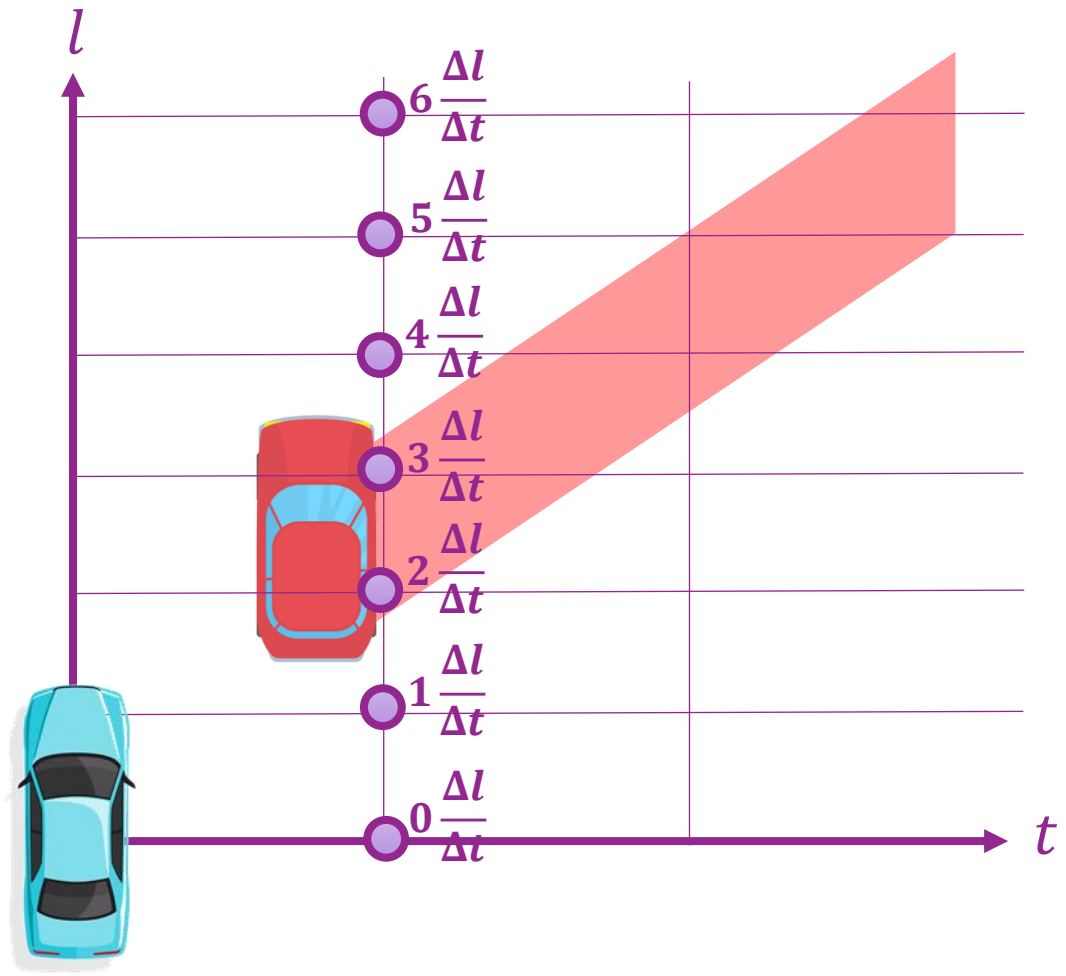


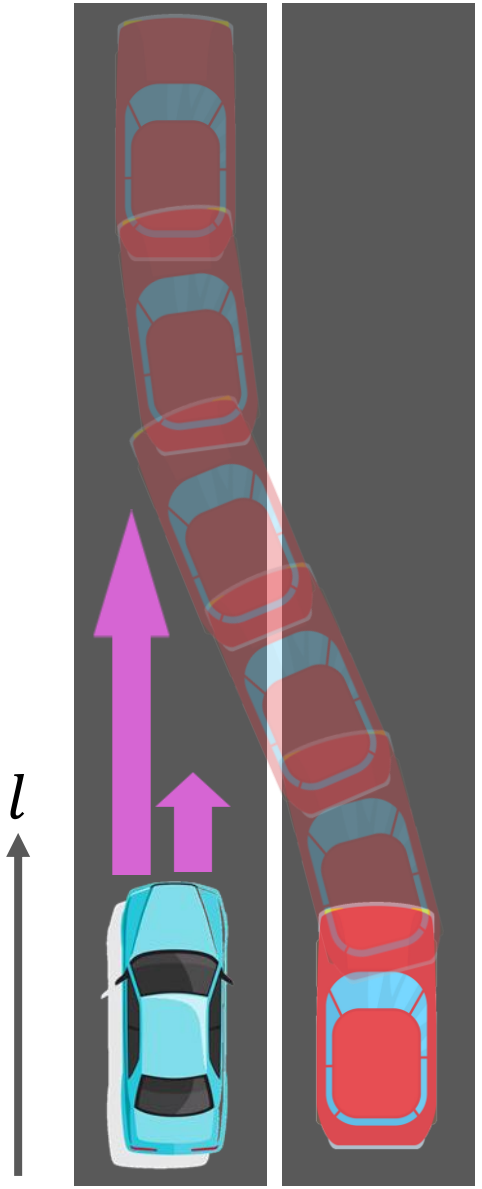
Spatiotemporal State Lattices



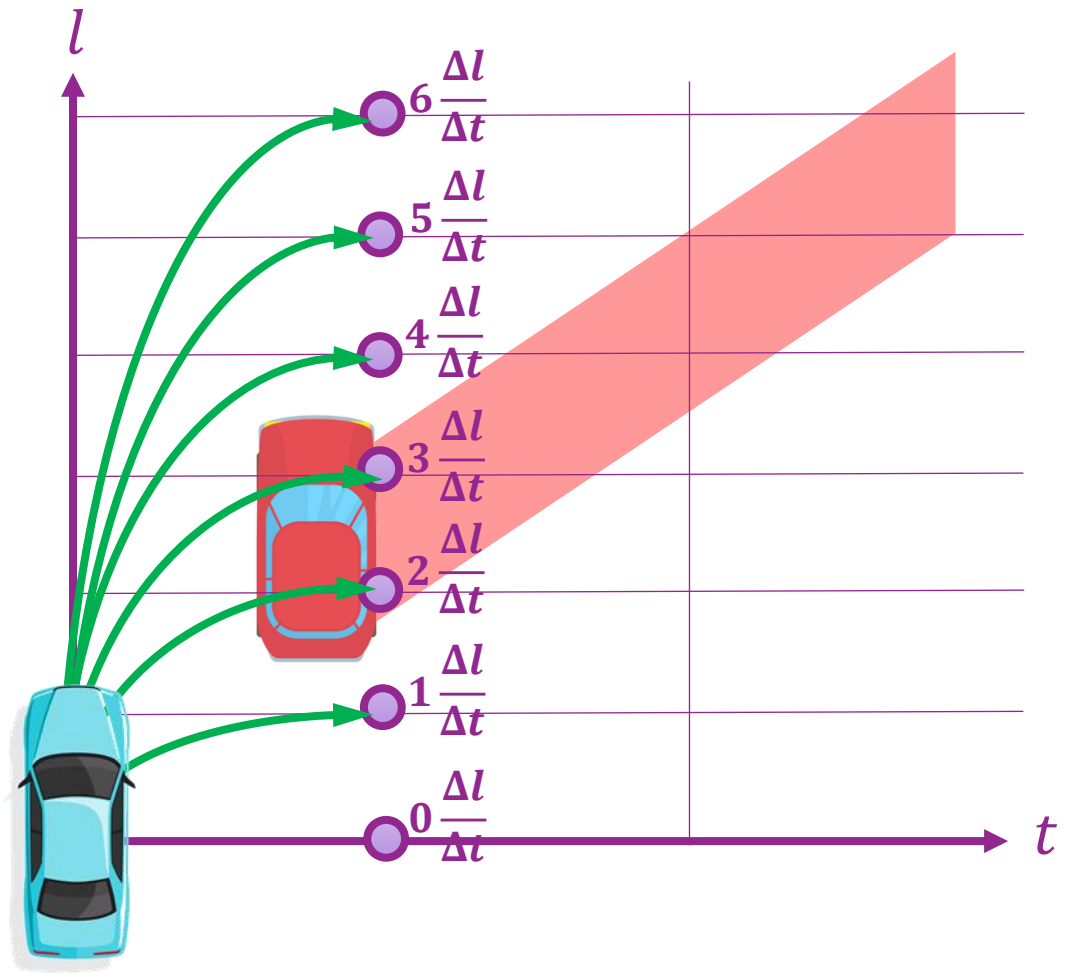


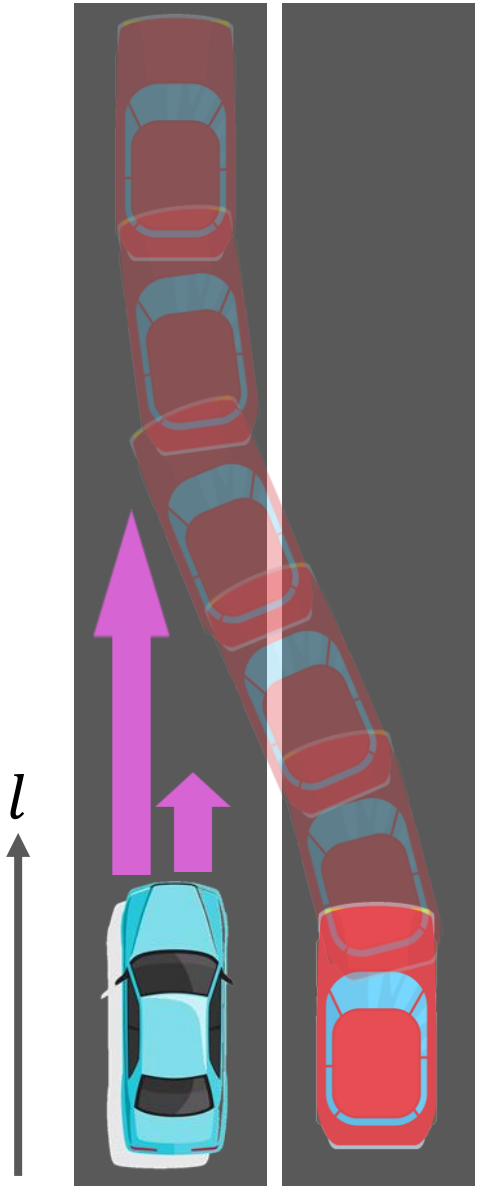
Spatiotemporal State Lattices



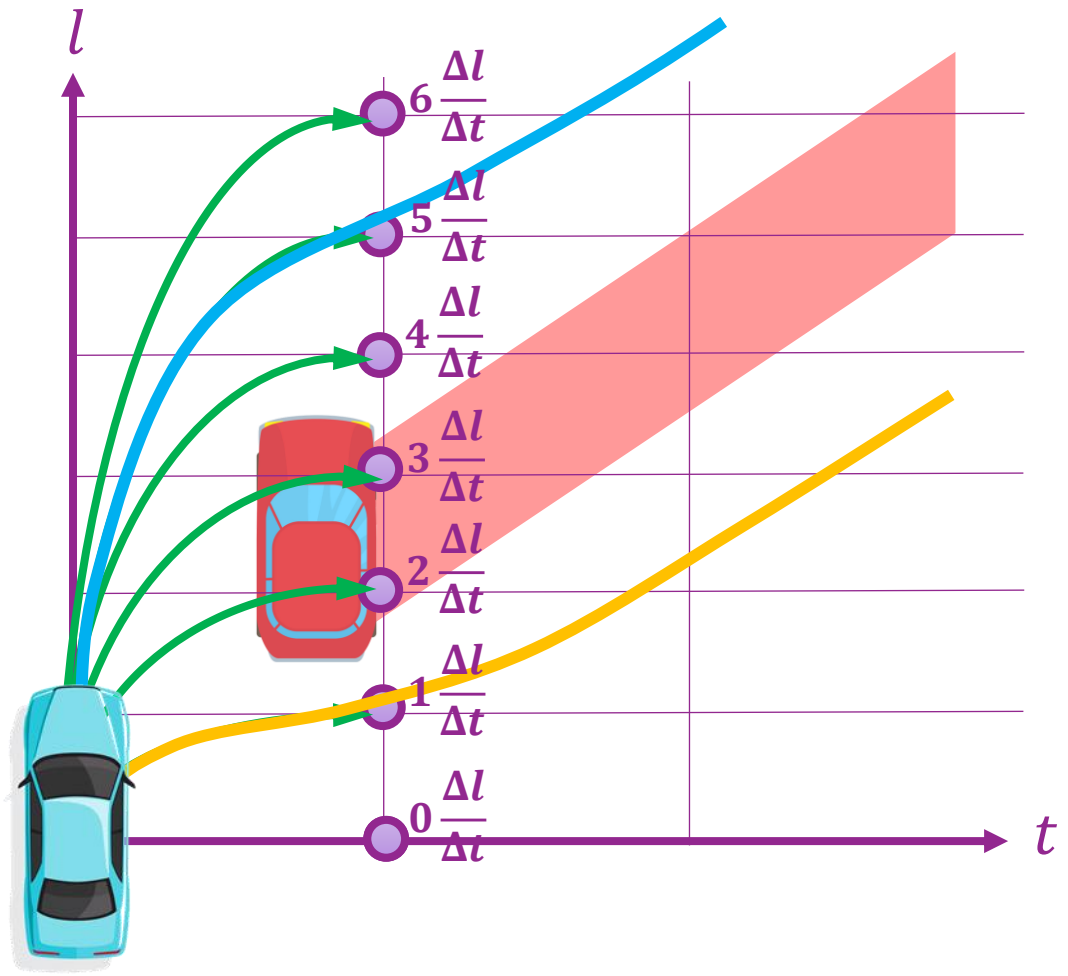


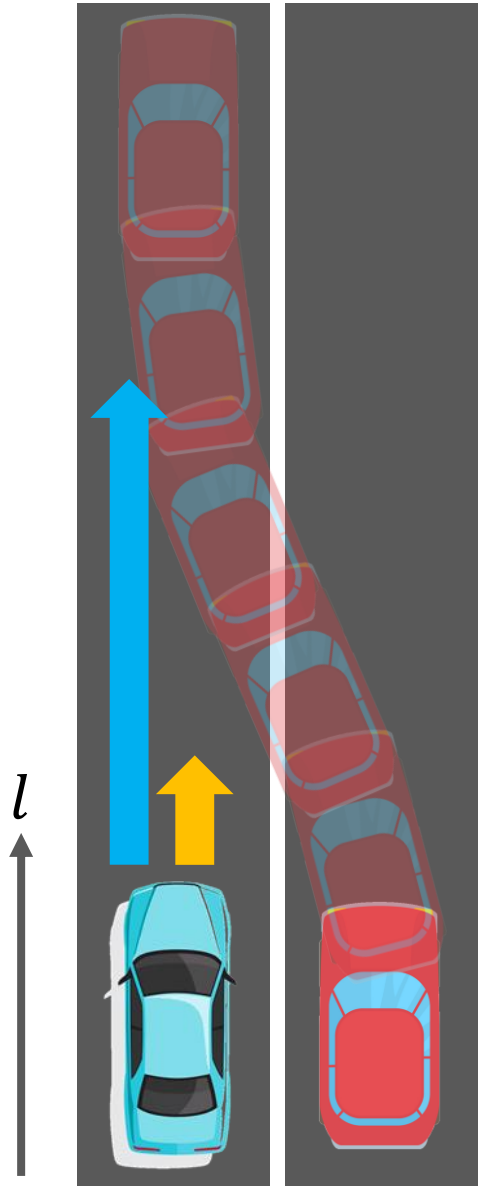
Spatiotemporal State Lattices



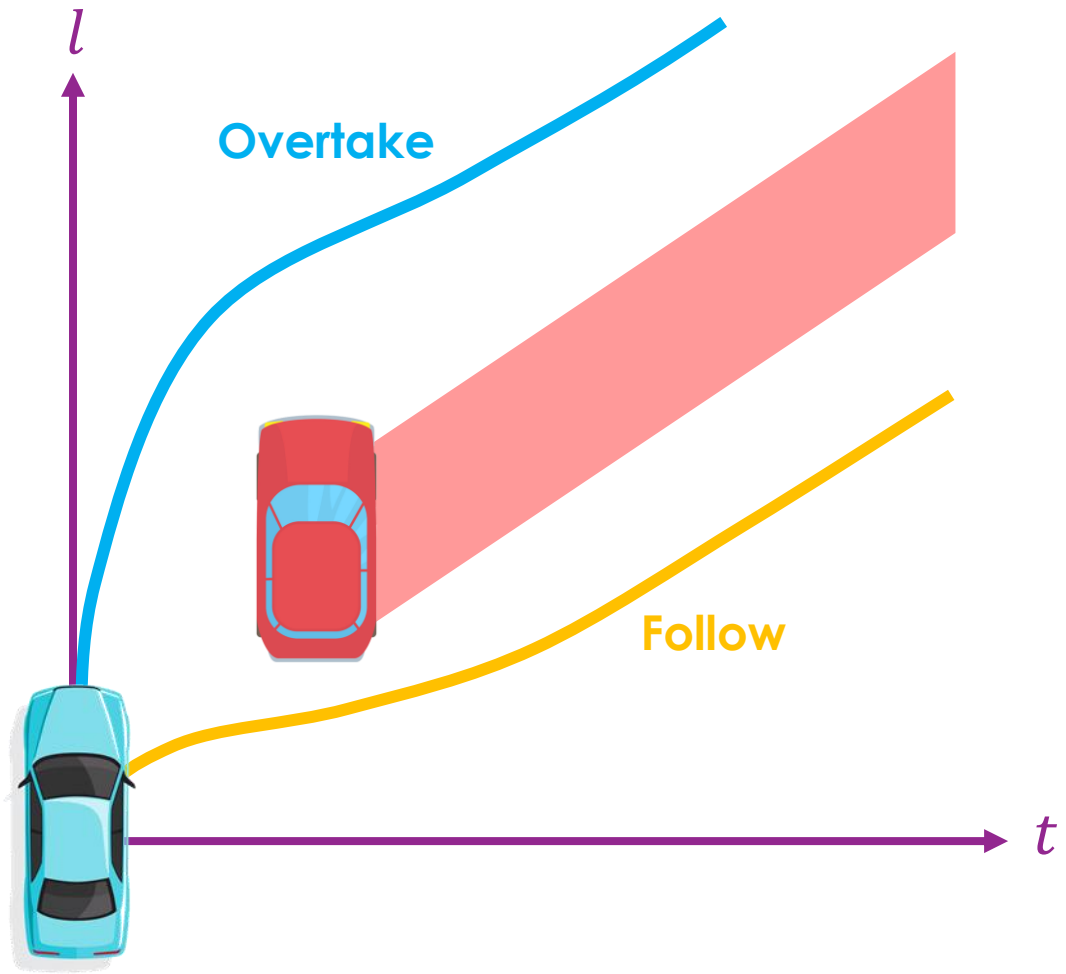


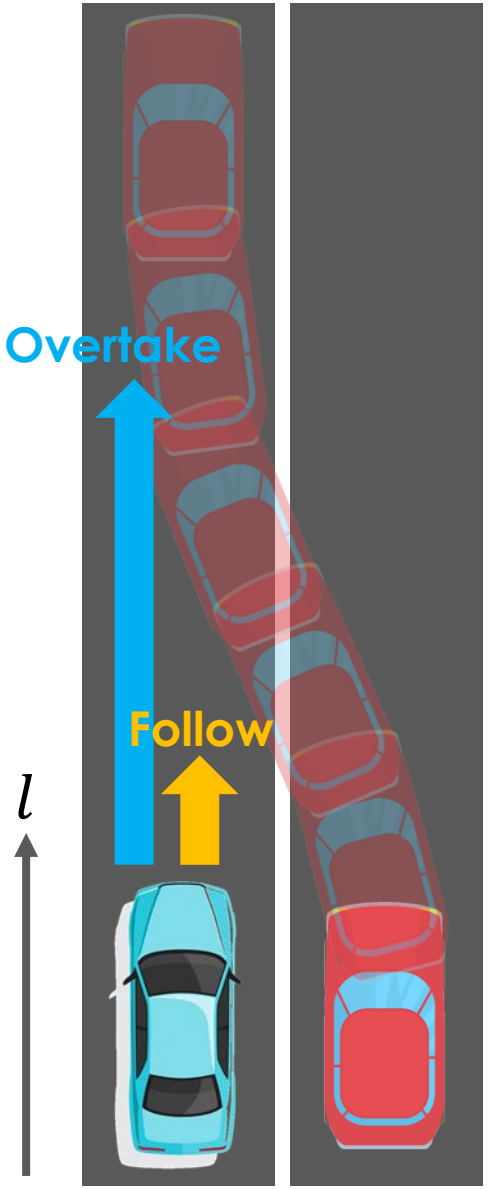
Spatiotemporal State Lattices





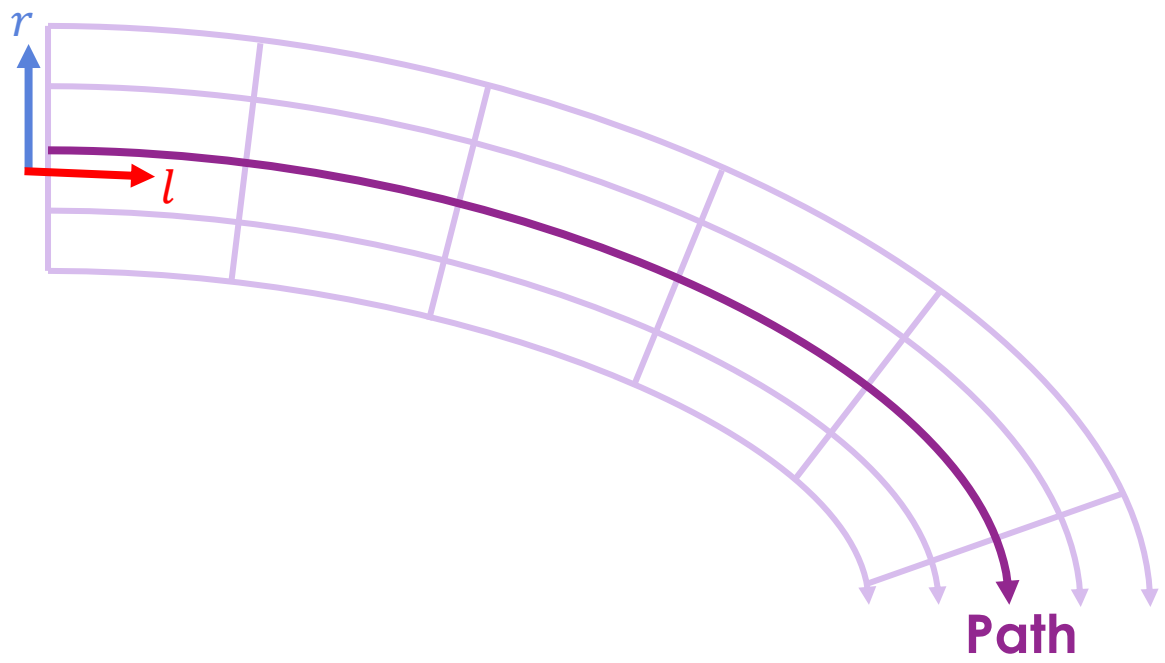
Spatiotemporal State Lattices





Frenet Coordinate

- Frenet coordinate projects the vehicle position onto the reference path.
- l represents the distance along the path and r represents the bias distance from the path.

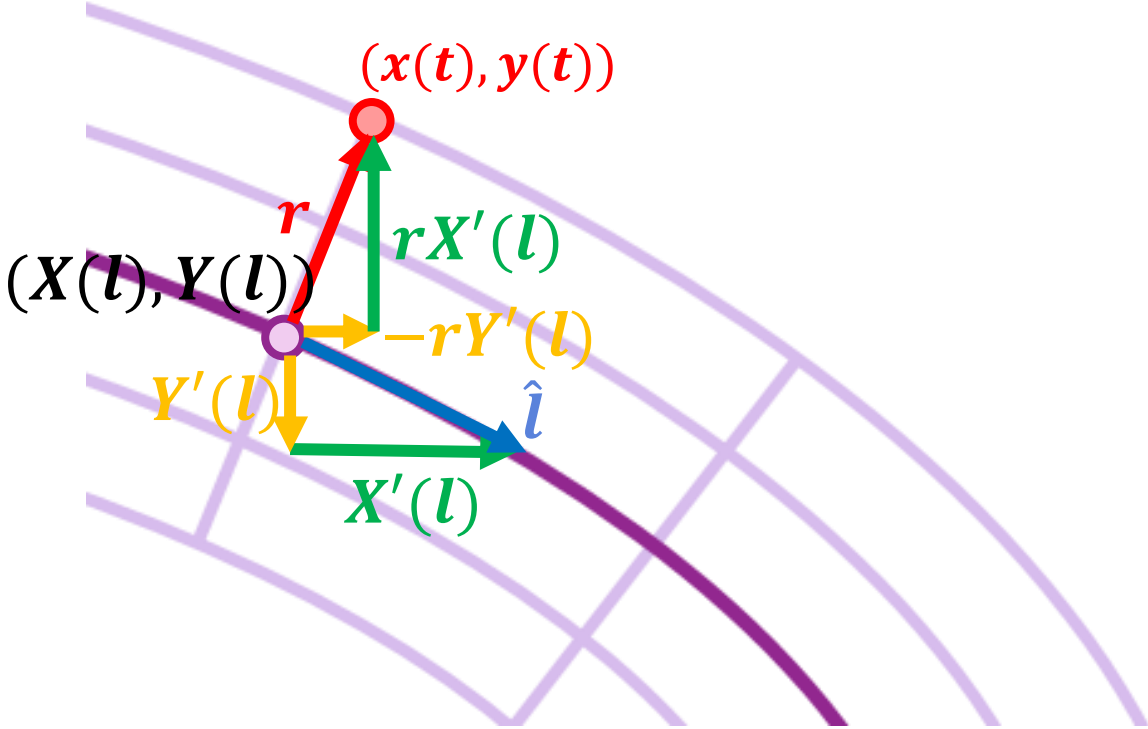


Frenet Coordinate (Cont.)

- (l, r) : reparameterization parameters along the path.
- $(X(l), Y(l))$: point on the path parameterized by l.
- $(x(t), y(t))$: Standard Cartesian coordinate.

- Transform Function

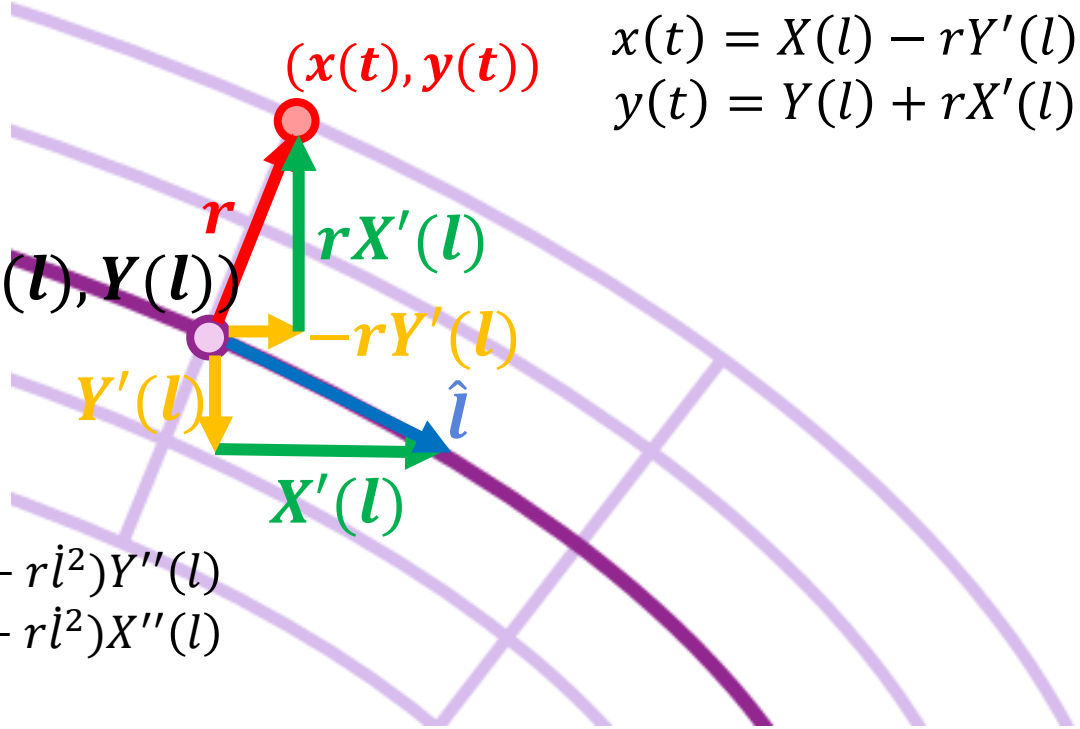
$$x(t) = X(l) - rY'(l)$$
$$y(t) = Y(l) + rX'(l)$$



Frenet Coordinate (Cont.)

- (l, r) : reparameterization parameters along the path.
- $(X(l), Y(l))$: point on the path parameterized by l.
- $(x(t), y(t))$: Standard Cartesian coordinate.

$$x(t) = X(l) - rY'(l)$$
$$y(t) = Y(l) + rX'(l)$$



- Differential Transform Function

$$\dot{x}(t) = \dot{l}X'(l) - \dot{r}Y'(l) - r\dot{l}Y'(l)$$
$$\dot{y}(t) = \dot{l}Y'(l) + \dot{r}X'(l) + r\dot{l}X'(l)$$

and

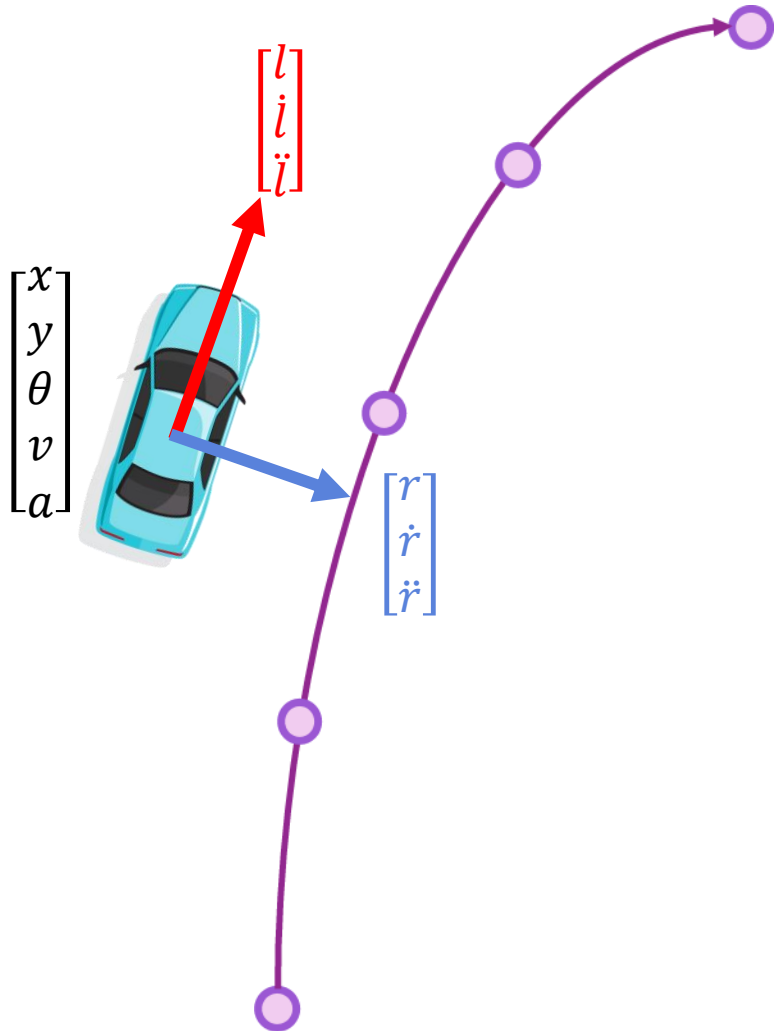
$$\ddot{x}(t) = \ddot{l}X'(l) + \dot{l}^2X''(l) - (\ddot{r} + \dot{r}\dot{l} + r\ddot{l})Y'(l) - (\dot{r}\dot{l} + r\dot{l}^2)Y''(l)$$
$$\ddot{y}(t) = \ddot{l}Y'(l) + \dot{l}^2Y''(l) - (\ddot{r} + \dot{r}\dot{l} + r\ddot{l})X'(l) - (\dot{r}\dot{l} + r\dot{l}^2)X''(l)$$

Trajectory Generation

- Vehicle States $[x, y, \theta, v, a]$

- Longitude States
  - l : Longitude distance
  - $\dot{l} = \frac{dl}{dt}$: Longitude speed
  - $\ddot{l} = \frac{d^2 l}{dt^2}$: Longitude acceleration

- Lateral States
  - r : Lateral offset
  - $\dot{r} = \frac{dr}{dt}$: Lateral speed
  - $\ddot{r} = \frac{d^2 r}{dt^2}$: Lateral acceleration



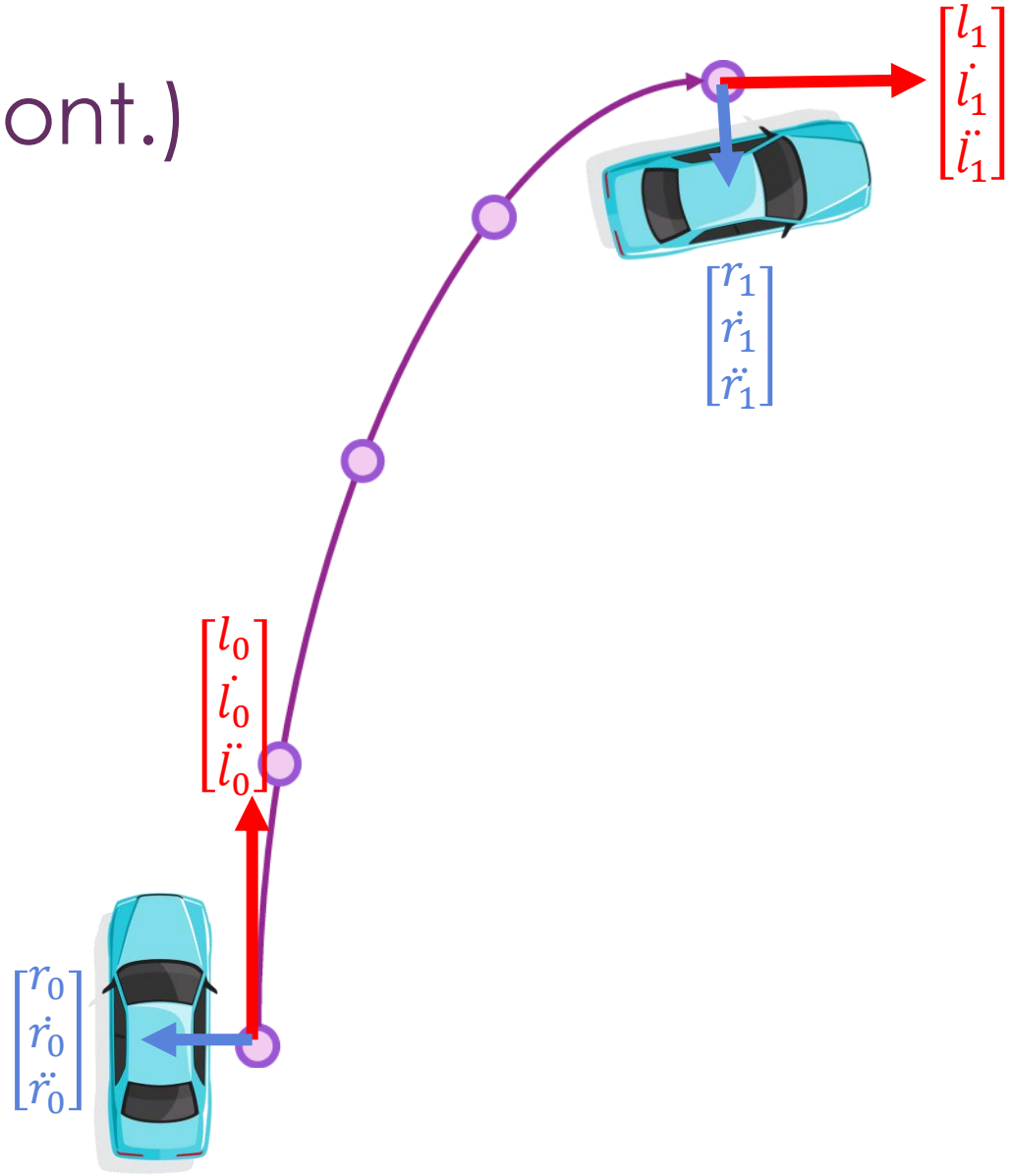
Trajectory Generation (Cont.)

- Given initial state at time t_0

  $\{(r_0, \dot{r}_0, \ddot{r}_0), (l_0, \dot{l}_0, \ddot{l}_0)\}$

- Given initial state at time t_1

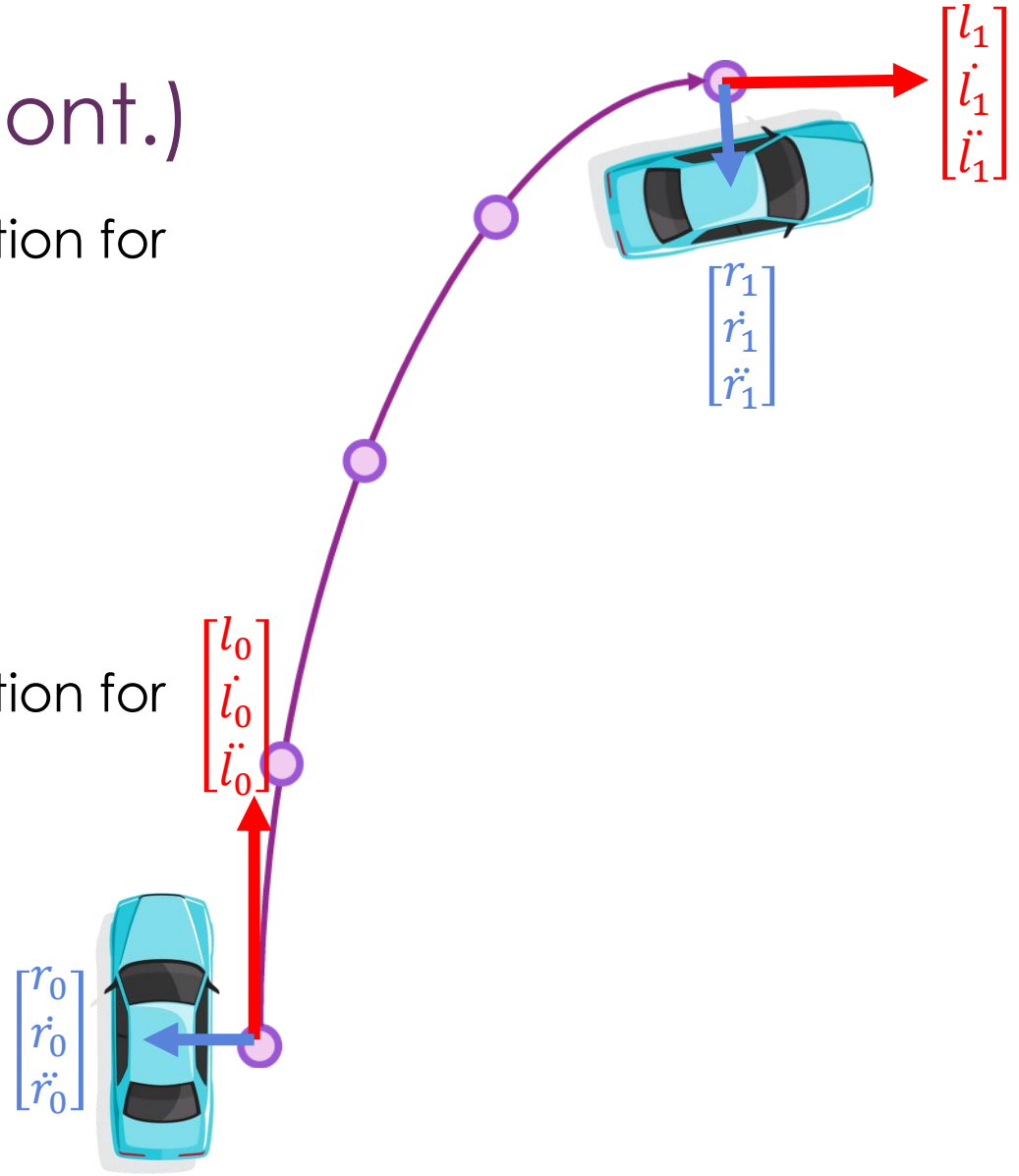
  $\{(r_1, \dot{r}_1, \ddot{r}_1), (l_1, \dot{l}_1, \ddot{l}_1)\}$



Trajectory Generation (Cont.)

- Set boundary condition of curve function for fitting the longitude trajectory $\boldsymbol{l(t)}$.
  - $l(t_0) = l_0\,, l(t_1) = l_1$
  - $\dot{l}(t_0) = \dot{l_0}\,, \dot{l}(t_1) = \dot{l_1}$
  - $\ddot{l}(t_0) = \ddot{l_0}\,, \ddot{l}(t_1) = \ddot{l_1}$

- Set boundary condition of curve function for fitting the lateral trajectory **r(t)**.
  - $r(t_0) = r_0\,, r(t_1) = r_1$
  - $\dot{r}(t_0) = \dot{r_0}\,, \dot{r}(t_1) = \dot{r_1}$
  - $\ddot{r}(t_0) = \ddot{r_0}\,, \ddot{r}(t_1) = \ddot{r_1}$



Trajectory Generation (Cont.)

- For time point $\boldsymbol{t}$
  - ✓ Get the longitude and lateral value from curve function

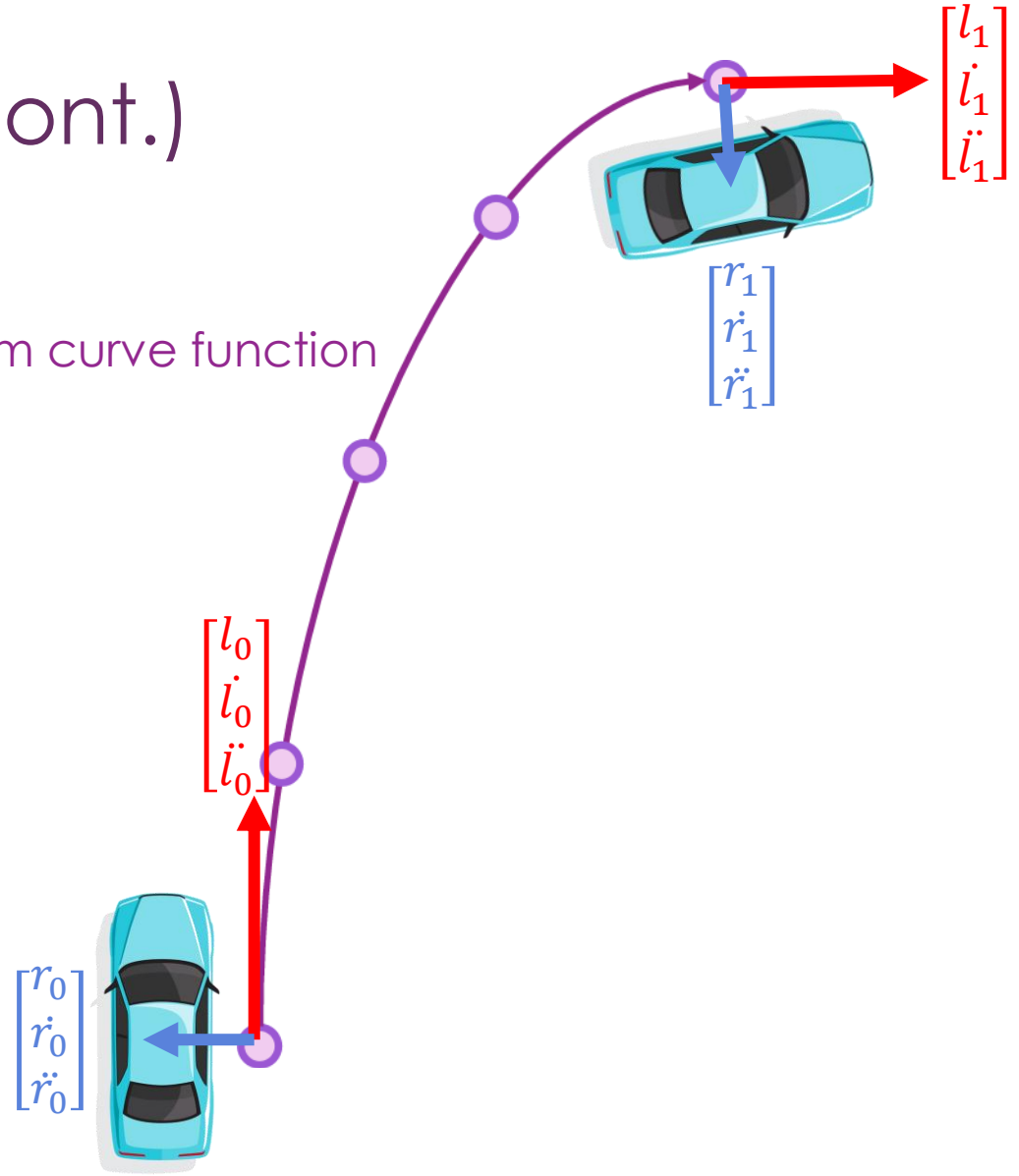
$$t \Rightarrow l = l(t), r = r(t)$$

  - ✓ Transform to standard coordinate

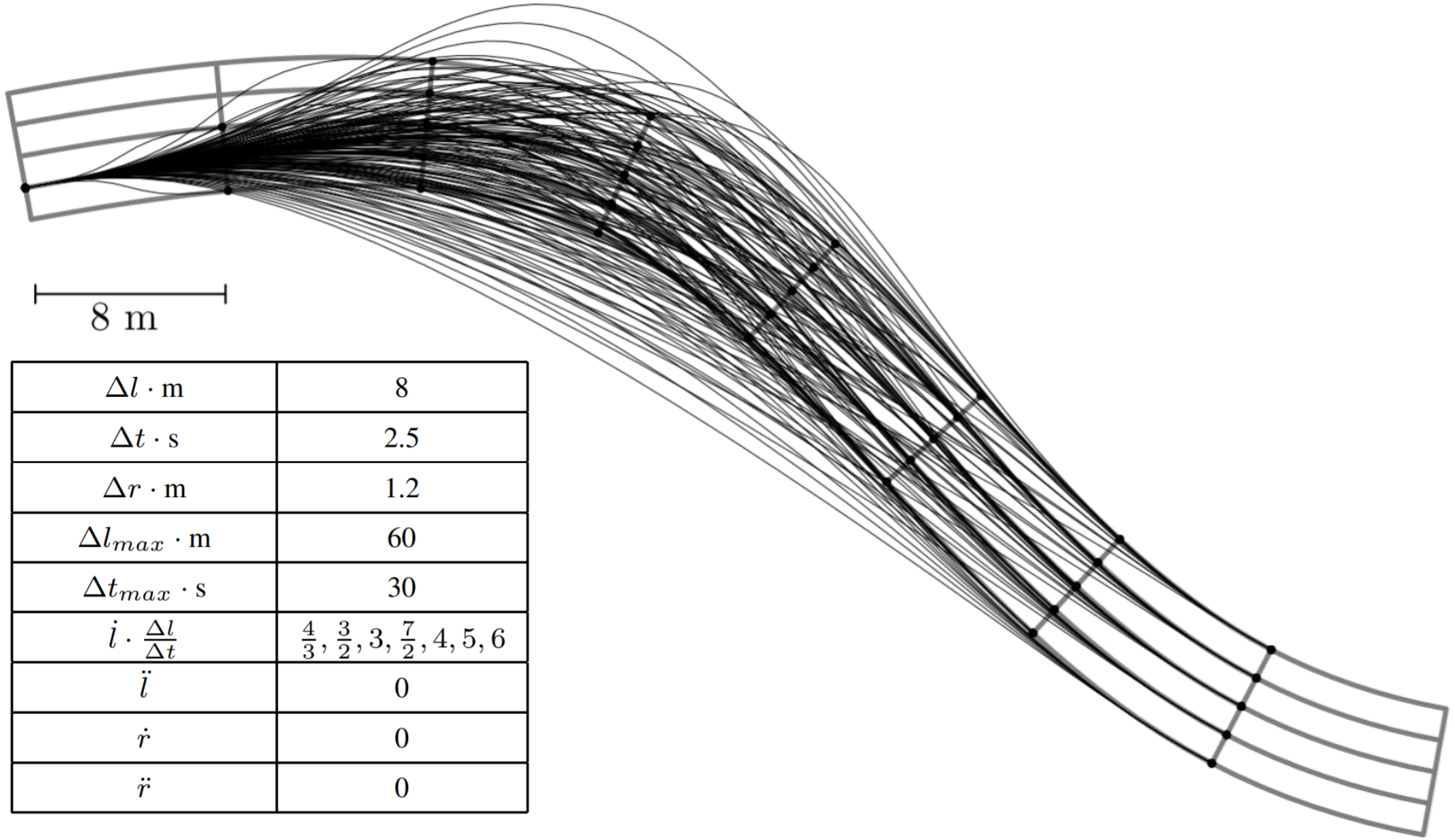
$$\begin{bmatrix} l \\ r \\ t \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ t \end{bmatrix}$$

$$x(t) = X(l) - rY'(l)$$
$$y(t) = Y(l) + rX'(l)$$

$$\dot{x}(t) = \dot{l}X'(l) - \dot{r}Y'(l) - r\dot{l}Y'(l)$$
$$\dot{y}(t) = \dot{l}Y'(l) + \dot{r}X'(l) + r\dot{l}X'(l)$$



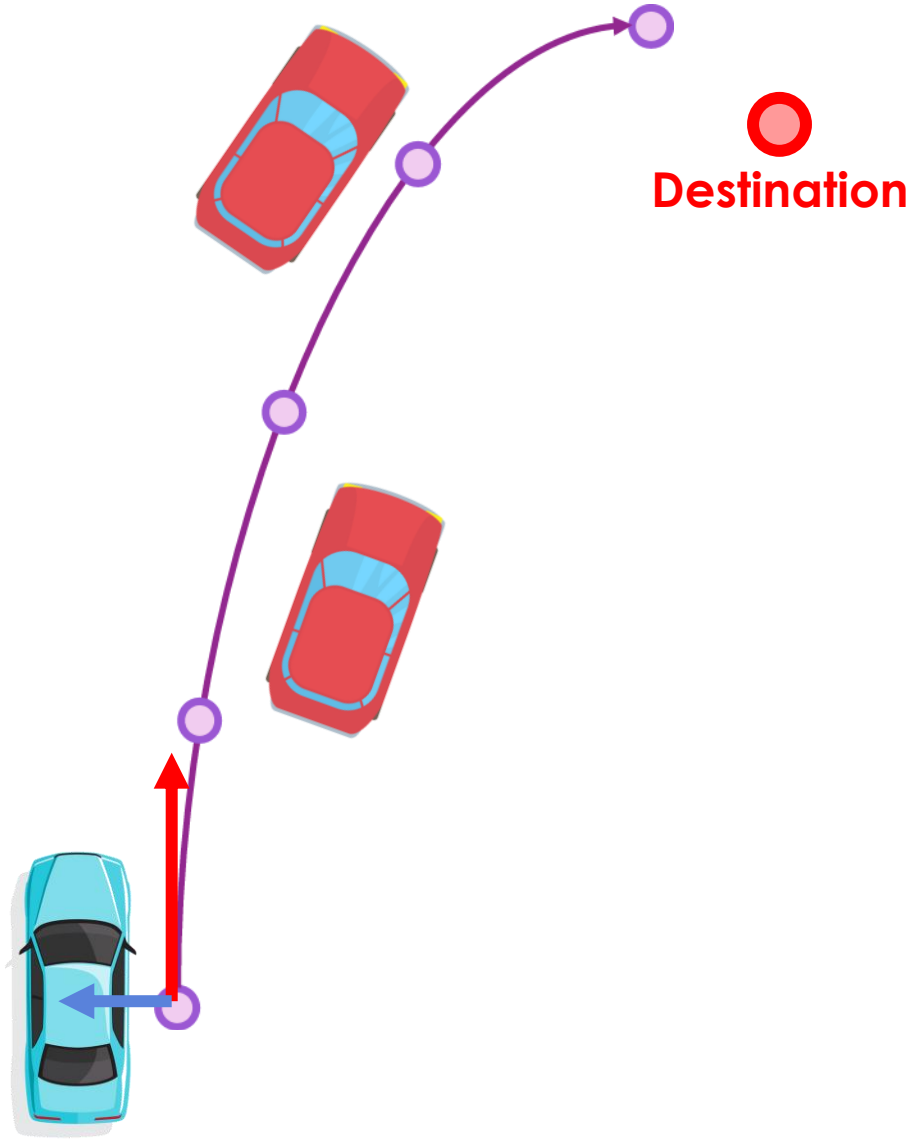
Planning on 2D Road



$\Delta l \cdot \mathrm{m}$	8
$\Delta t \cdot \mathrm{s}$	2.5
$\Delta r \cdot \mathrm{m}$	1.2
$\Delta l_{max} \cdot \mathrm{m}$	60
$\Delta t_{max} \cdot \mathrm{s}$	30
$\dot{l} \cdot \frac{\Delta l}{\Delta t}$	$\frac{4}{3}, \frac{3}{2}, 3, \frac{7}{2}, 4, 5, 6$
$\ddot{l}$	0
$\dot{r}$	0
$\ddot{r}$	0

Cost Function Design

- Achieve Destination
  - Objective achievement cost
- Follow the path
  - Lateral offset cost
- Avoid Collision
  - Collision cost
- Comfortable
  - Longitude jerk cost
  - Lateral acceleration cost



Q&A