Self-Driving Cars

Lecture 12 – Decision Making and Planning

Prof. Dr.-Ing. Andreas Geiger

Autonomous Vision Group

University of Tübingen / MPI-IS

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

ellis
European Laboratory for Learning and Intelligent Systems

Agenda

12.1 Introduction

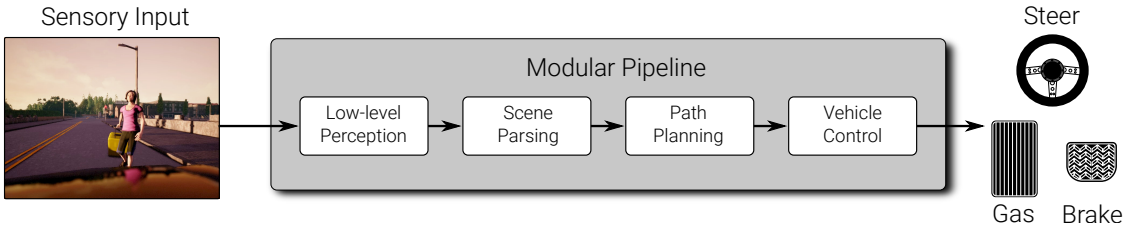
12.2 Route Planning

12.3 Behavior Planning

12.4 Motion Planning

12.1
Introduction

Modular Pipeline



Course Content:

- Lectures 5+6: Vehicle Dynamics and Control
- Lectures 7-11: Low-level Perception and Scene Parsing
- Lecture 12: Decision Making and Planning

Planning and Decision Making

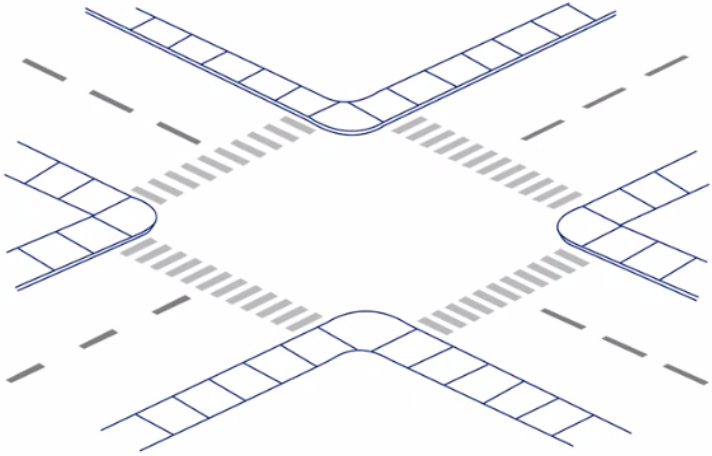
Problem Definition:

- Goal: find and follow path from current location to destination
- Take static infrastructure and dynamic objects into account
- Input: vehicle and environment state (via perception stack)
- Output: path or trajectory as input to vehicle controller

Challenges:

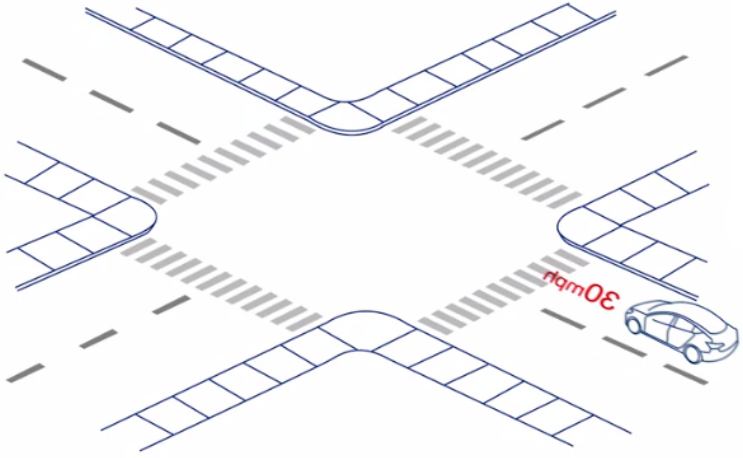
- Driving situations and behaviors are very complex
- Thus difficult to model as a single optimization problem
- Let's have a look at a few examples ...

Planning and Decision Making



Slide Credit: Steven Waslander.

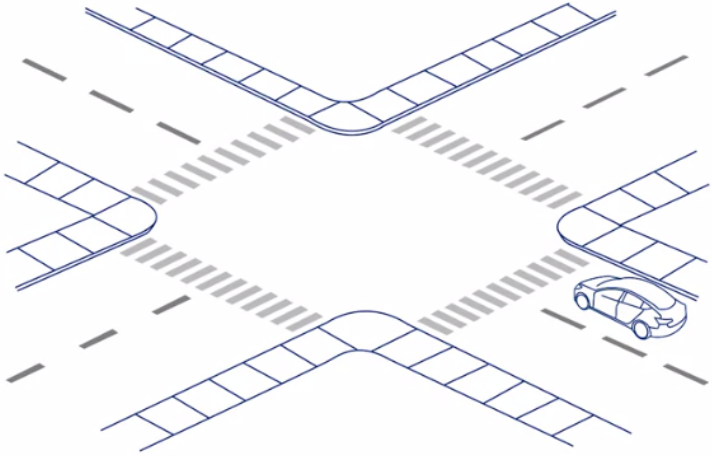
Planning and Decision Making



Speed tracking (30 mph)

Slide Credit: Steven Waslander.

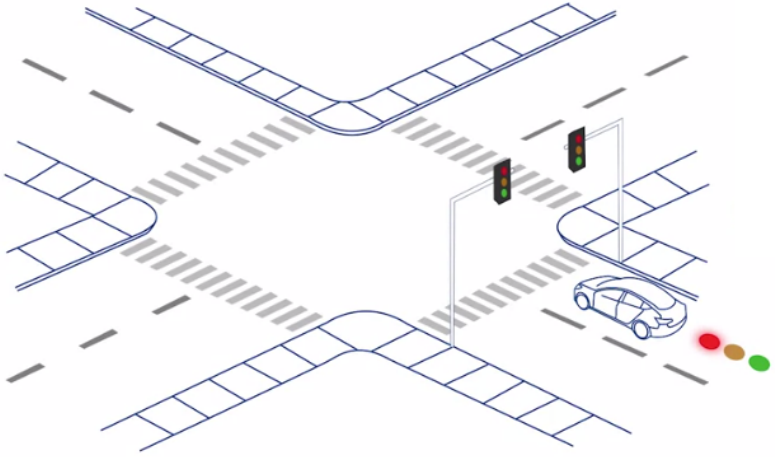
Planning and Decision Making



Decelerate to stop

Slide Credit: Steven Waslander.

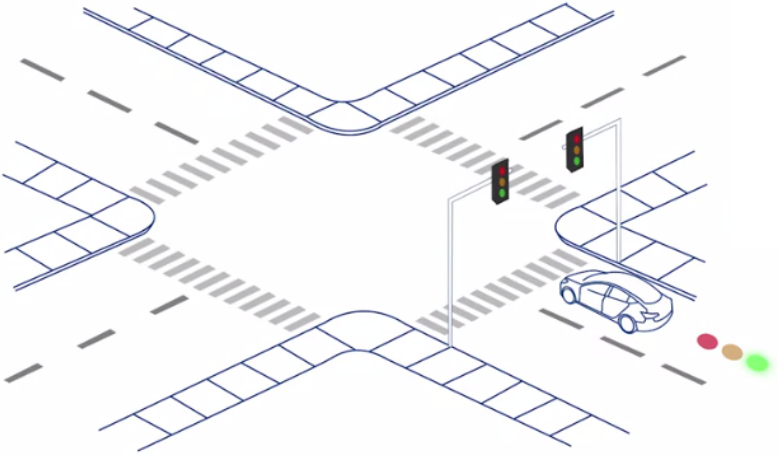
Planning and Decision Making



Stay stopped

Slide Credit: Steven Waslander.

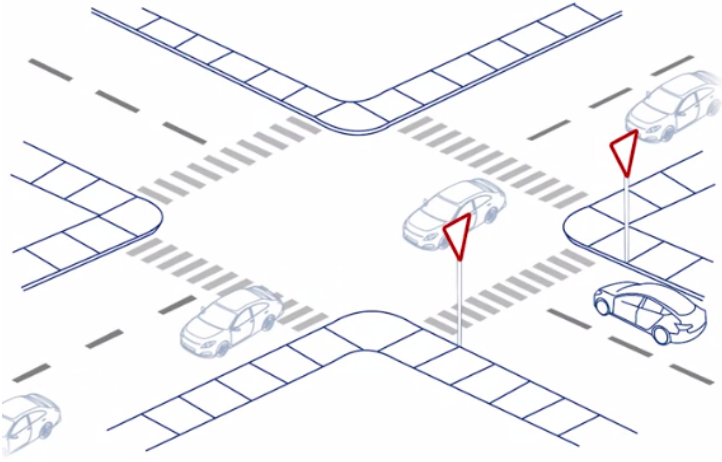
Planning and Decision Making



Start driving

Slide Credit: Steven Waslander.

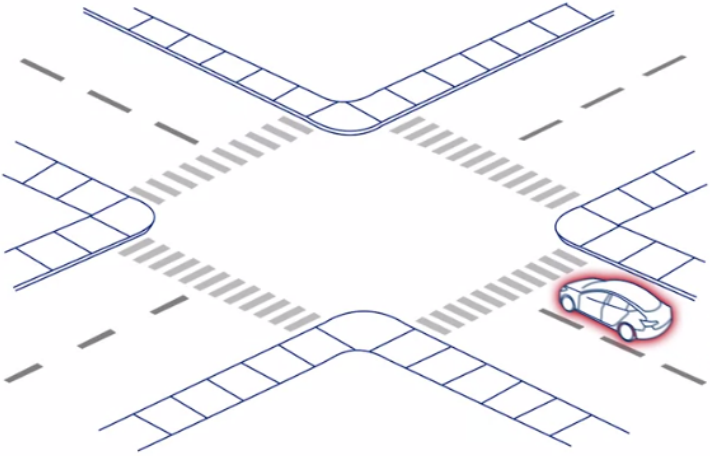
Planning and Decision Making



Yield to traffic

Slide Credit: Steven Waslander.

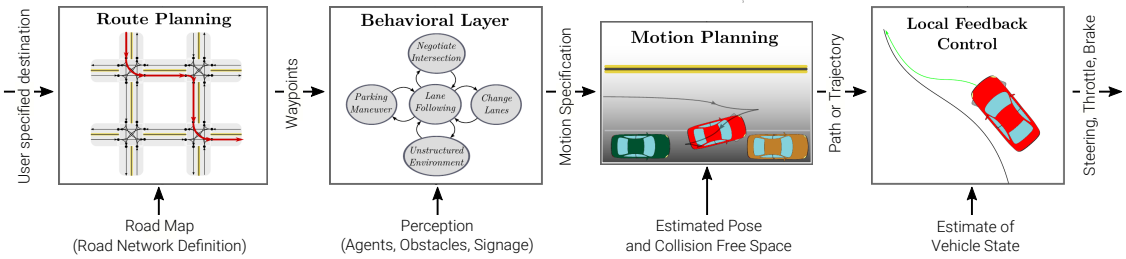
Planning and Decision Making



Emergency stop (and many more ...)

Slide Credit: Steven Waslander.

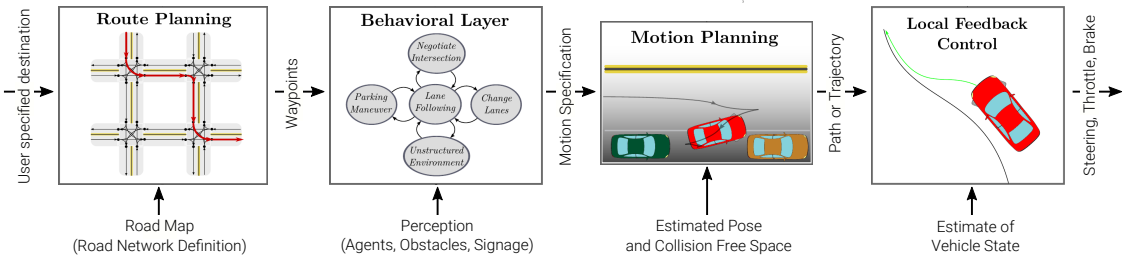
Planning and Decision Making



- Idea: Break planning problem into a **hierarchy of simpler problems**
- Each problem tailored to its scope and level of abstraction
- Earlier in this hierarchy means higher level of abstraction
- Each optimization problem will have constraints and objective functions

Paden et al.: A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. TIV, 2016.

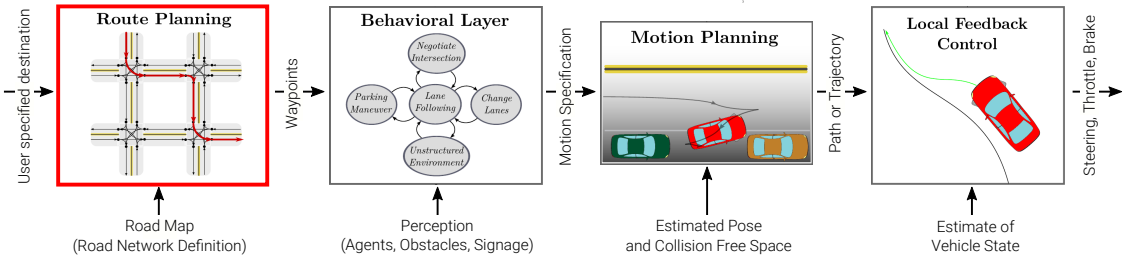
Planning and Decision Making



Hierarchy: A destination is passed to a route planner that generates a route through the road network. A behavioral layer reasons about the environment and generates a motion specification to progress along the selected route. A motion planner then solves for a feasible motion accomplishing the specification. A feedback control adjusts actuation variables to correct errors in executing the reference path.

Paden et al.: A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. TIV, 2016.

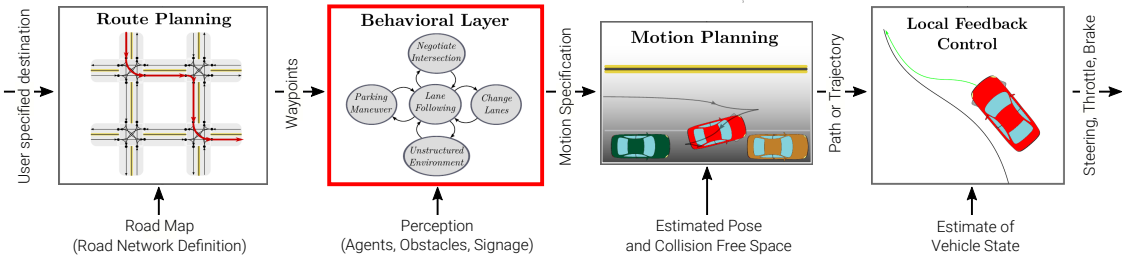
Step 1: Route Planning



- Represent **road network** as **directed graph**
- Edge weights correspond to road segment length or travel time
- Problem translates into a minimum-cost graph network problem
- Inference algorithms: Dijkstra, A*, …

Paden et al.: A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. TIV, 2016.

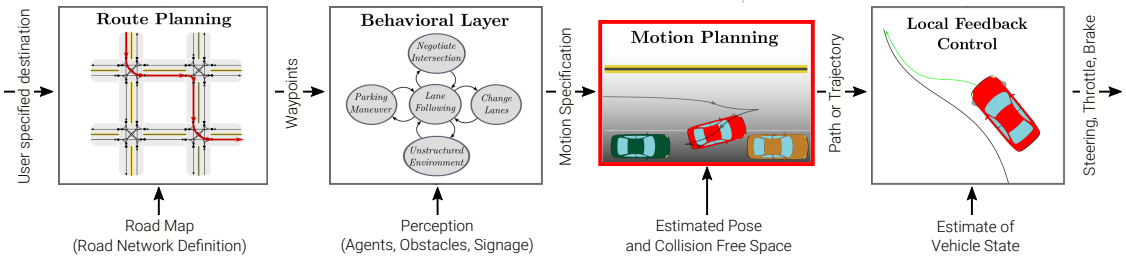
Step 2: Behavior Planning



- Select **driving behavior** based on current vehicle/environment state
- E.g. at stop line: stop, observe other traffic participants, traverse
- Often modeled via **finite state machines** (transitions governed by perception)
- Can be modeled probabilistically, e.g., using Markov Decision Processes (MDPs)

Paden et al.: A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. TIV, 2016.

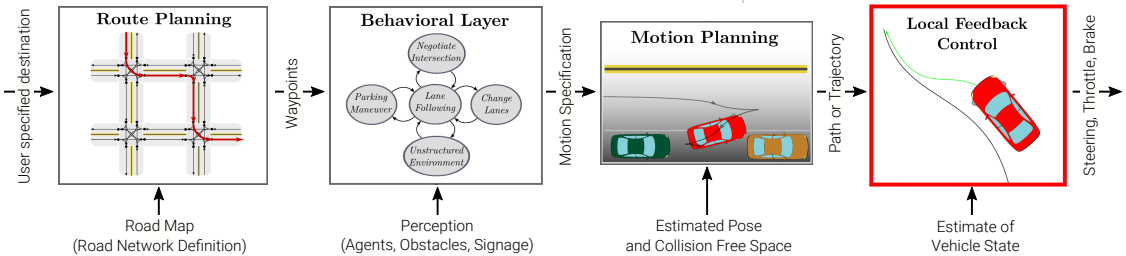
Step 3: Motion Planning



- Find feasible, comfortable, safe and fast **vehicle path/trajectory**
- Exact solutions in most cases computationally intractable
- Thus often **numerical approximations** are used
- Approaches: variational methods, graph search, incremental tree-based

Paden et al.: A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. TIV, 2016.

Step 4: Local Feedback Control

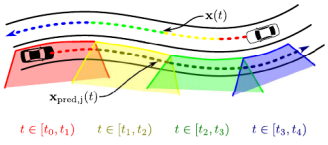


- **Feedback controller** executes the path/trajectory from the motion planner
- Corrects errors due to inaccuracies of the vehicle model
- Emphasis on **robustness, stability and comfort**
- Vehicle dynamics and control has been discussed in lecture 5+6

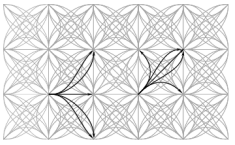
Paden et al.: A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. TIV, 2016.

Planning Algorithms

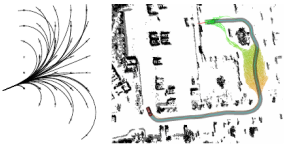
(a) Dijkstra [29]



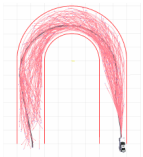
(b) FunctionOptimization [38]



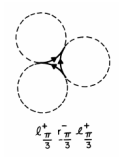
(c) Lattices [39]



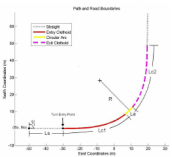
(d) A* [36]



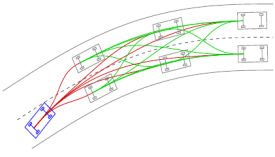
(e) RRT [40]



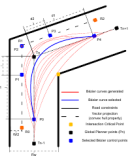
(f) Line&Circle [41]



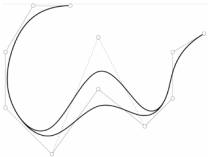
(g) Clothoid [42]



(h) Polynomial [43]



(i) Bezier [44]



(j) Spline [45]

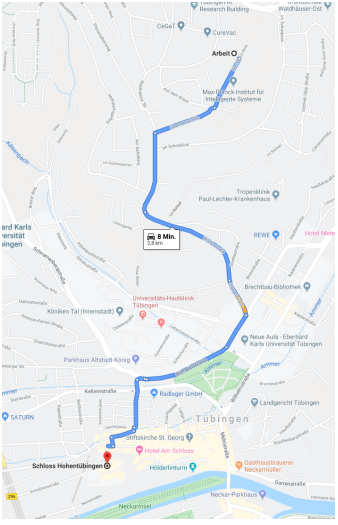
- Planning algorithms used in the **autonomous driving literature**
- There are many of them – we will focus only on a few today

12.2
Route Planning

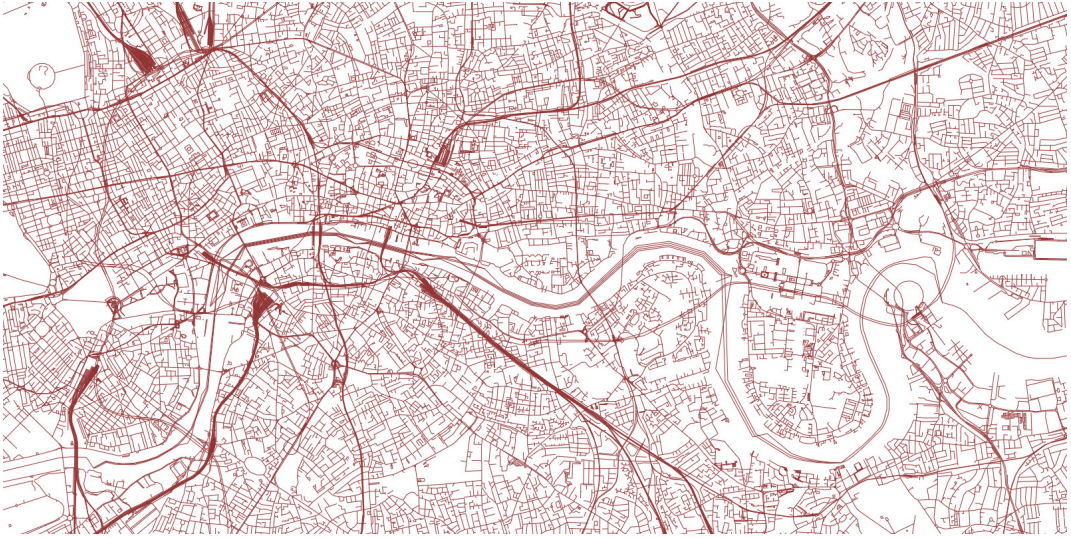
Route Planning

Goal:

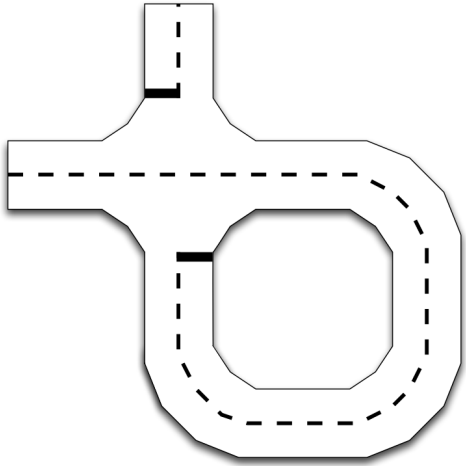
- Navigate from point A to point B
  on a road map using the **most efficient path**
  (in terms of time or distance)
- Route planning is **high-level planning**
- Low-level details are abstracted away
- Route planning is sometimes
  also referred to as “mission planning”

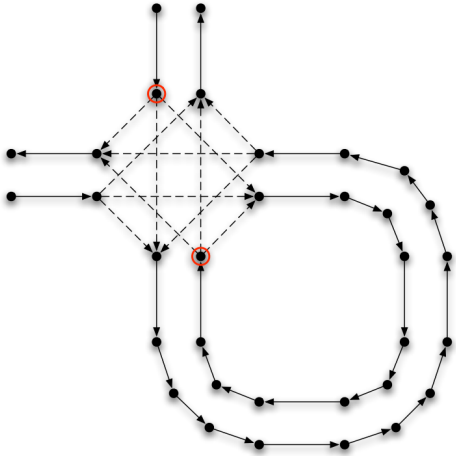


Road Networks as Graphs



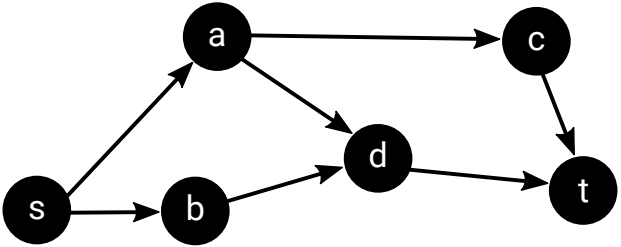
Road Networks as Graphs





- **Road networks** can be represented as **directed graphs**

Road Networks as Graphs

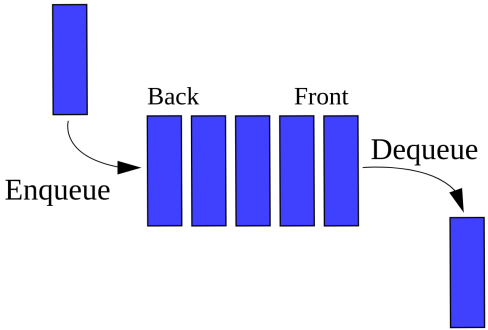


- Road networks can be represented as **directed graphs** $G = (V, E)$ with vertices $v \in V$ and directed edges $(u, v) \in E$
- Edges correspond to road or lane segments
- s denotes the source and t denotes the sink
- We are interested in finding the **shortest path** from s to t (=least edges)

Breadth First Search

Queue

- Collection of **sequential entities**
- First-in-first-out (FIFO) data structure
- **Enqueue** (=addition) at one end (back)
- **Dequeue** (=removal) from other end (front)
- Often implemented via linked lists
- Enqueue and dequeue in $O(1)$



Breadth First Search

Algorithm 1 BFS(G,s,t)

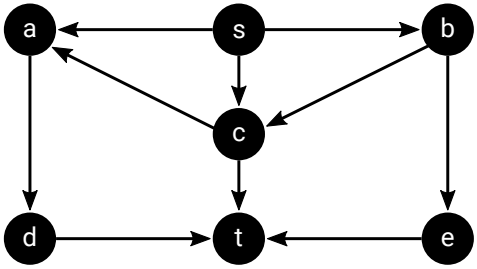
```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```

- Algorithm for unweighted graphs. Returns optimal solution in $O(|V| + |E|)$.

Breadth First Search: Example

Algorithm 1 BFS(G,s,t)

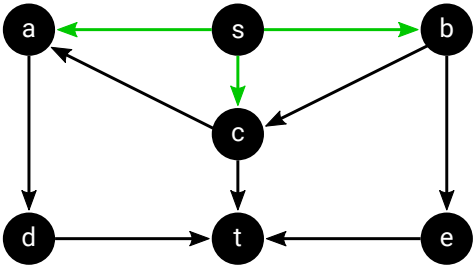
```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```



Breadth First Search: Example

Algorithm 1 BFS(G,s,t)

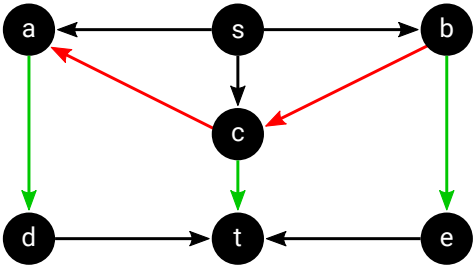
```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```



Breadth First Search: Example

Algorithm 1 BFS(G,s,t)

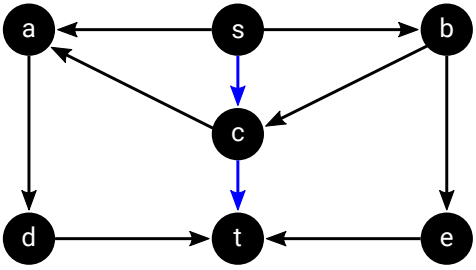
```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```



Breadth First Search: Example

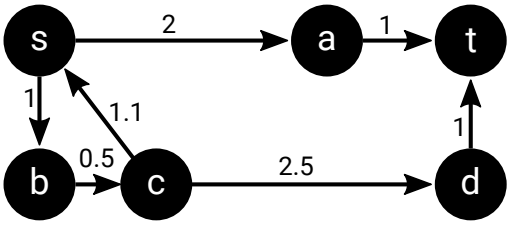
Algorithm 1 BFS(G,s,t)

```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```



Road Segment Length

- Problem? Unweighted graphs cannot represent **road segments of variable length**
- How can we solve this? Chunk road network into small segments of equal length
- Problem? Increase in computation
- Better solution? Use **weighted graph:**



- Each edge carries a weight corresponding to road segment length or travel time

Dijkstra’s Shortest Path Algorithm

Dijkstra's Shortest Path Algorithm

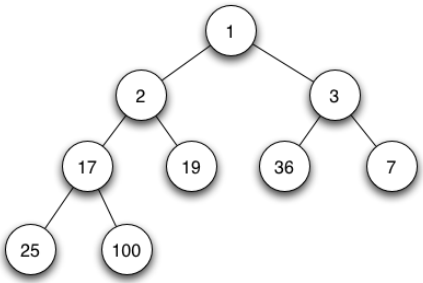
"What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame."

Edsger Dijkstra, in an interview with Philip L. Frana, Communications of the ACM, 2001

Dijkstra: A note on two problems in connexion with graphs. Numerische Mathematik, 1959.

Min Heap

- Specialized tree-based data structure
- **Min heap property:** The value of any node in the tree is smaller than all its children
- Efficient implementation of **priority queue**
- **Smallest element** stored at **root node**
- Partially ordered, but not sorted
- Common implementation: binary tree
- Insertion and removal in $O(\log n)$



Dijkstra: A note on two problems in connexion with graphs. Numerische Mathematik, 1959.

Dijkstra's Shortest Path Algorithm

Algorithm 1 BFS(G,s,t)

```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```

Algorithm 2 Dijkstra(G,s,t)

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

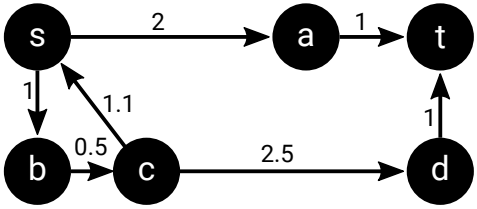
- Returns optimal solution in $O(|V| \log |V| + |E|)$

Dijkstra: A note on two problems in connexion with graphs. Numerische Mathematik, 1959.

Dijkstra's Shortest Path Algorithm: Example

Algorithm 2 Dijkstra(G,s,t)

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



Open Min Heap:

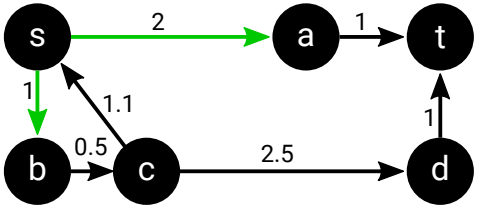
Node	s
Cost to Vertex	0

Closed Set:

Dijkstra's Shortest Path Algorithm: Example

Algorithm 2 Dijkstra(G,s,t)

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



Open Min Heap:

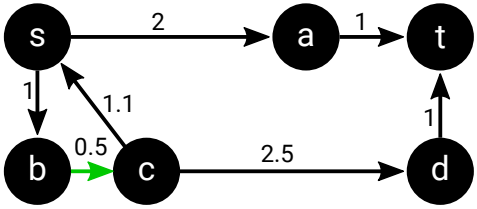
Node	b	a
Cost to Vertex	1	2

Closed Set: s

Dijkstra's Shortest Path Algorithm: Example

Algorithm 2 Dijkstra(G,s,t)

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



Open Min Heap:

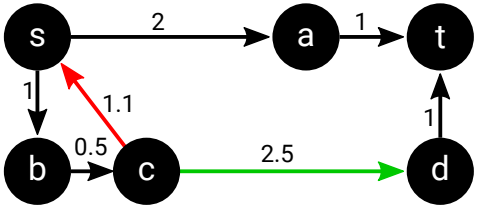
Node	c	a
Cost to Vertex	1.5	2

Closed Set: s b

Dijkstra's Shortest Path Algorithm: Example

Algorithm 2 Dijkstra(G,s,t)

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



Open Min Heap:

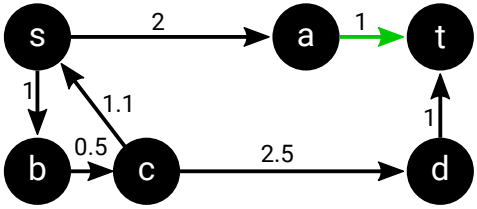
Node	a	d
Cost to Vertex	2	4

Closed Set: s b c

Dijkstra's Shortest Path Algorithm: Example

Algorithm 2 Dijkstra(G,s,t)

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



Open Min Heap:

Node	t	d
Cost to Vertex	3	4

Closed Set:

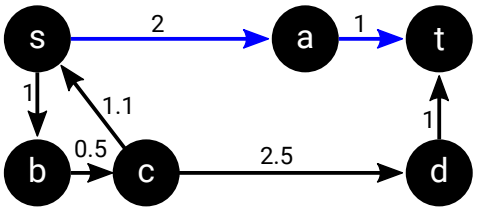
s	b	c	a

Dijkstra: A note on two problems in connexion with graphs. Numerische Mathematik, 1959.

Dijkstra's Shortest Path Algorithm: Example

Algorithm 2 Dijkstra(G,s,t)

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



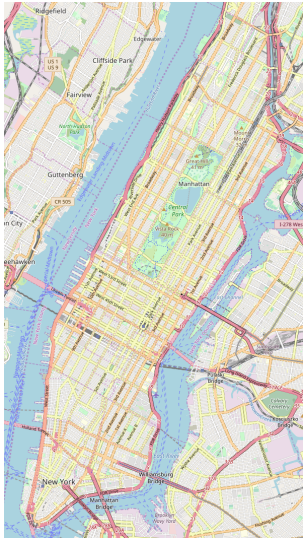
Final Path: s a t

Closed Set: s b c a

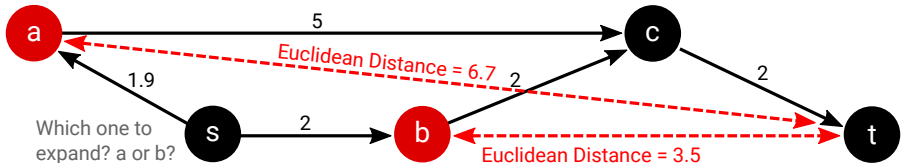
Dijkstra's Shortest Path Algorithm: Scalability

What is the problem with Dijkstra?

- Method does not scale to large graphs
- Computationally very expensive / slow
- Example: Map of New York City
  - 54,837 vertices
  - 140,497 edges
- Solution? Use **planning heuristics**
- Example: Use **distance to target** as heuristic to select "promising" vertices to expand ..



Euclidean Planning Heuristics



- Euclidean planning heuristics **exploits structure of planar graphs**
- Straight line between vertices is a useful estimate of the distance along the path
- In this example, Dijkstra's algorithm would expand from s to a
- However, b is more likely to be on the shortest path as it is closer to the target t
- The A* algorithm uses this heuristics to plan **more efficiently**
- It always returns the **optimal solution**, independent of the chosen heuristic

A* Algorithm

A* Algorithm

Algorithm 2 Dijkstra(G,s,t)

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

Algorithm 3 A*(G,s,t)

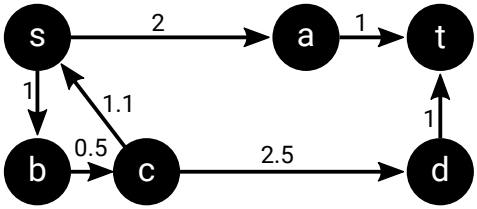
```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0,h(s))
3:  while !open.isEmpty() do
4:      u, uCost, uHeuristic ← open.pop()   [based on cost+heuristic]
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvCost ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uCost + uvCost + h(v) < fullCost(v) then
12:                     open[v] ← uCost + uvCost, h(v)
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uCost + uvCost, h(v))
16:                 predecessors[v] ← u
17:     closed.add(u)
```

Hart et al.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. TSSC, 1968.

A* Algorithm: Example

Algorithm 3 A*(G,s,t)

```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0,h(s))
3:  while !open.isEmpty() do
4:      u, uCost, uHeuristic ← open.pop()    [based on cost+heuristic]
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvCost ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uCost + uvCost + h(v) < fullCost(v) then
12:                     open[v] ← uCost + uvCost, h(v)
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uCost + uvCost, h(v))
16:                 predecessors[v] ← u
17:     closed.add(u)
```



Open Min Heap:

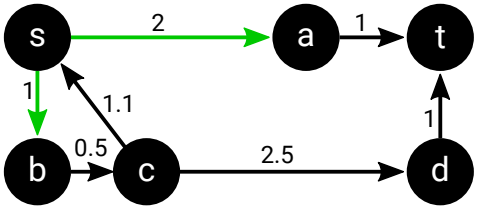
Node	s
Cost + Heuristic	3

Closed Set:

A* Algorithm: Example

Algorithm 3 A*(G,s,t)

```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0,h(s))
3:  while !open.isEmpty() do
4:      u, uCost, uHeuristic ← open.pop()    [based on cost+heuristic]
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvCost ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uCost + uvCost + h(v) < fullCost(v) then
12:                     open[v] ← uCost + uvCost, h(v)
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uCost + uvCost, h(v))
16:                 predecessors[v] ← u
17:     closed.add(u)
```



Open Min Heap:

Node	a	b
Cost + Heuristic	3	4.2

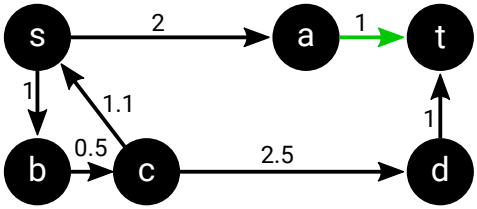
Closed Set: s

Hart et al.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. TSSC, 1968.

A* Algorithm: Example

Algorithm 3 A*(G,s,t)

```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0,h(s))
3:  while !open.isEmpty() do
4:      u, uCost, uHeuristic ← open.pop()   [based on cost+heuristic]
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvCost ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uCost + uvCost + h(v) < fullCost(v) then
12:                     open[v] ← uCost + uvCost, h(v)
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uCost + uvCost, h(v))
16:                 predecessors[v] ← u
17:     closed.add(u)
```



Open Min Heap:

Node	t	b
Cost + Heuristic	3	4.2

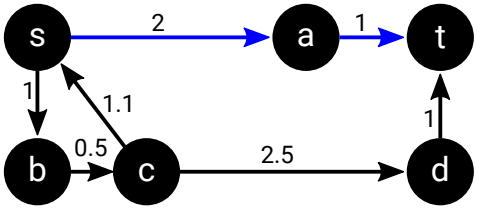
Closed Set: s a

Hart et al.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. TSSC, 1968.

A* Algorithm: Example

Algorithm 3 A*(G,s,t)

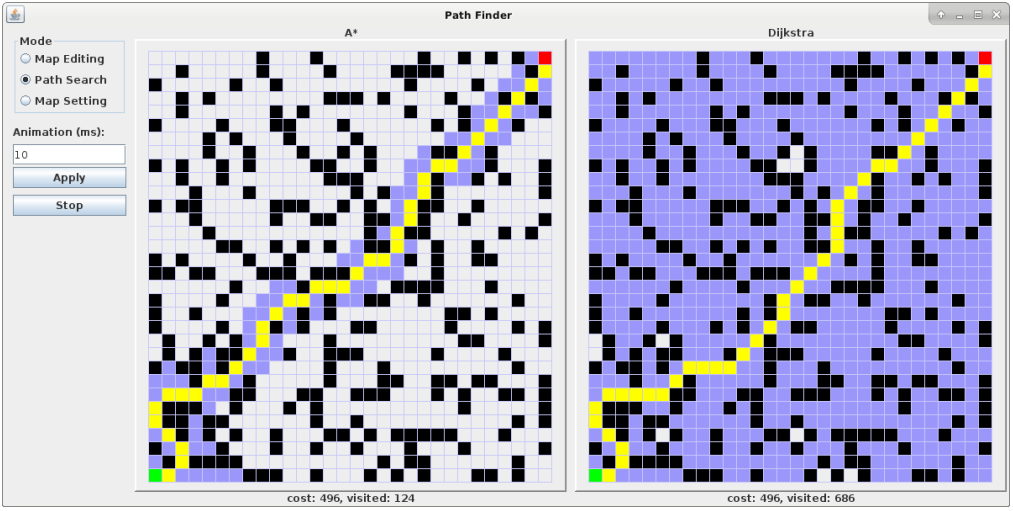
```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0,h(s))
3:  while !open.isEmpty() do
4:      u, uCost, uHeuristic ← open.pop()    [based on cost+heuristic]
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvCost ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uCost + uvCost + h(v) < fullCost(v) then
12:                     open[v] ← uCost + uvCost, h(v)
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uCost + uvCost, h(v))
16:                 predecessors[v] ← u
17:     closed.add(u)
```



Final Path: s a t

Closed Set: s a

PathFinder: A* and Dijkstra Visualization



https://github.com/kevinwang1975/PathFinder

12.3
Behavior Planning

Behavior Planning

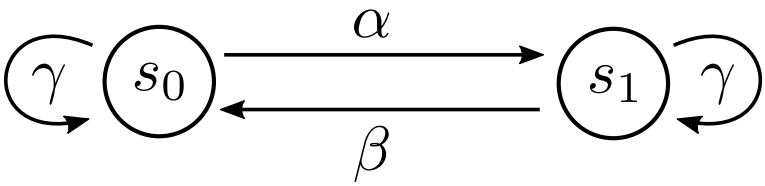
- To follow a planned route, the vehicle must conduct **various maneuvers**
- Examples include: speed tracking, car following, stopping, merging, etc.
- It is difficult to design a motion planner for all maneuvers jointly
- The **behavior planning** stage thus **discretizes the behaviors** into simpler (atomic) maneuvers, each of which can be addressed with a dedicated motion planner
- The behavior layer must take into account traffic rules, static and dynamic objects
- **Input:** High-level route plan and output of perception stack
- **Output:** Motion planner constraints: corridor, objects, speed limits, target, ...
- Frequently used models:
  - Deterministic: Finite State Machines (FSMs) and variants
  - Probabilistic: Markov Decision Processes (MDPs, POMDPs, see RL lecture)

Finite State Machine

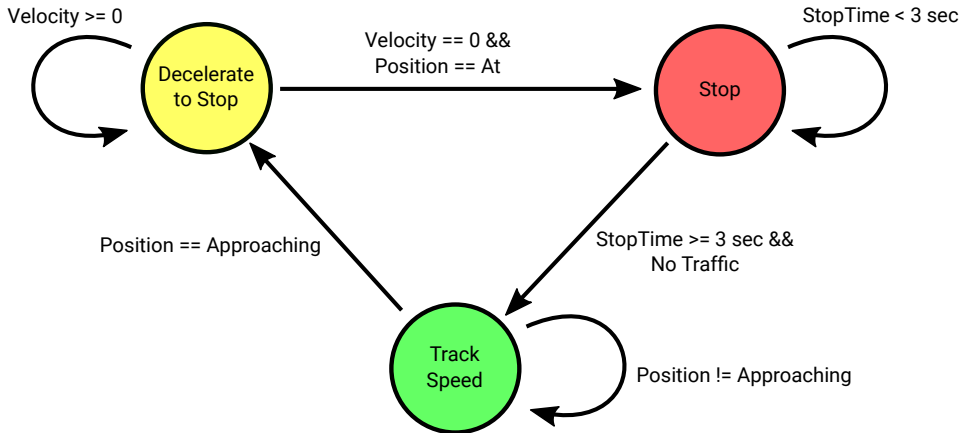
A Finite State Machine (FSM) is defined by quintuple $(\Sigma, \mathcal{S}, \mathcal{F}, s_0, \delta)$

- Σ is the input alphabet
- $\mathcal{S}$ is a non-empty set of states
- $\mathcal{F} \subseteq \mathcal{S}$ is the (possibly empty) set of final states
- $s_0 \in \mathcal{S}$ is the initial state
- $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$ is the state transition function

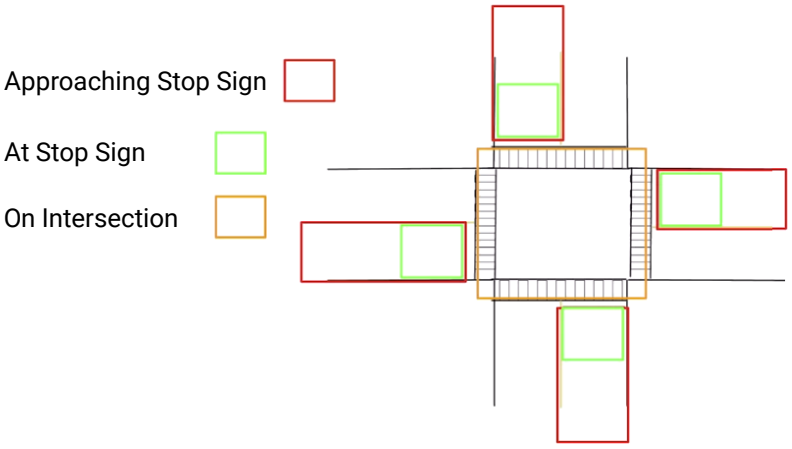
Example:



FSM for a Simple Vehicle Behavior

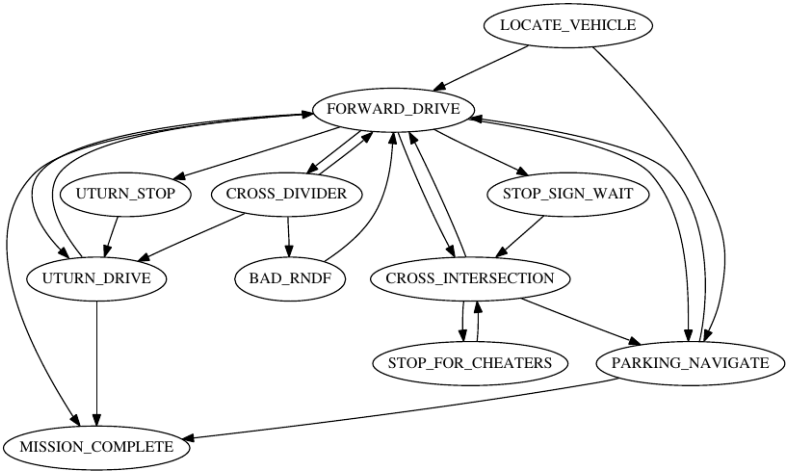


Example 4-Way Stop



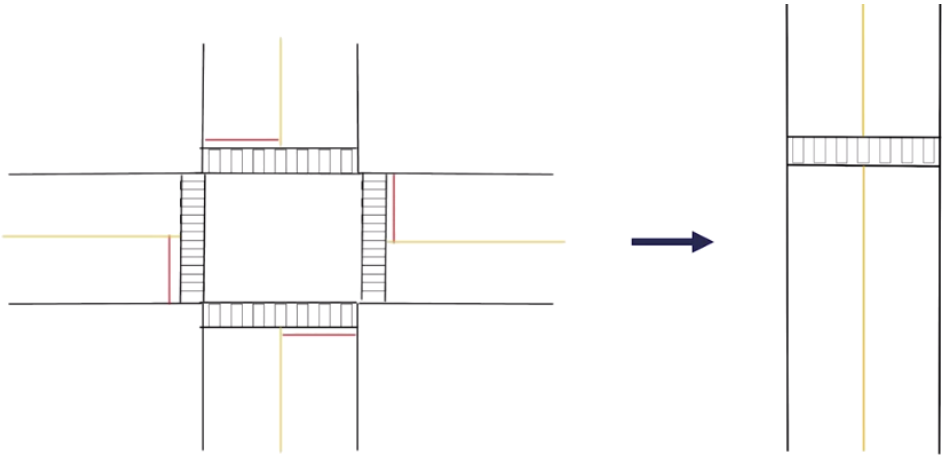
[Slide Credit: Steven Waslander]

Example from DARPA Challenge: Stanford Junior's FSM



Montemerlo et al.: Junior: The Stanford Entry in the Urban Challenge. JFR, 2008.

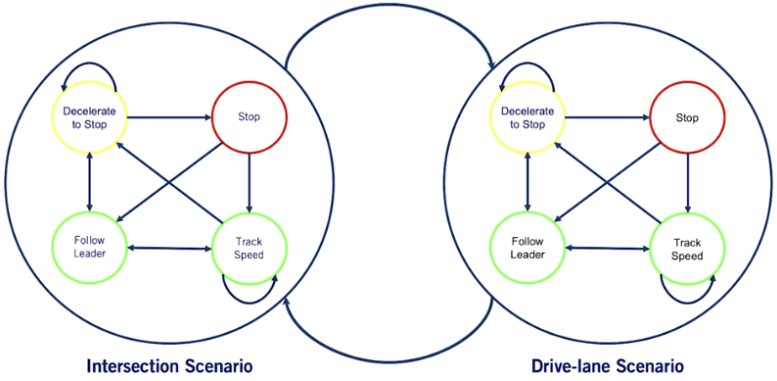
Handling Multiple Scenarios



[Slide Credit: Steven Waslander]

- How to handle multiple scenarios? Extending single FSM leads to rule explosion!

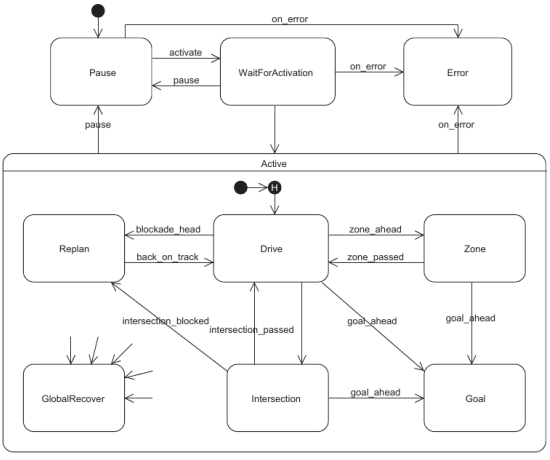
Handling Multiple Scenarios



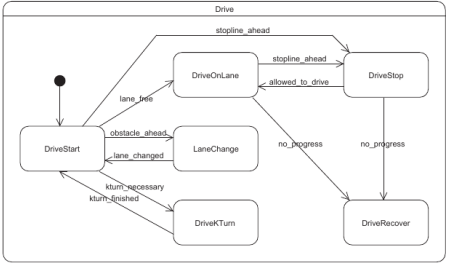
[Slide Credit: Steven Waslander]

- Hierarchical State Machine (HSM)
- Advantages: Simpler, more efficient – Disadvantages: Rule duplication

Example from DARPA Challenge: Karlsruhe AnnieWAY's Planner



(a) State chart of the main level of the FSM.



(b) Substate of the state *Drive*.

Gindele et al.: Design of the planner of team AnnieWAY's autonomous vehicle used in the DARPA Urban Challenge 2007. IV, 2008.

Summary Finate State Machines

- Elegant way to break complex behaviors into simpler maneuvers
- Interpretable and easy to design
- Rule explosion when dealing with complex scenarios
- Cannot handle noise / uncertainty $\Rightarrow$ MDPs
- Expert-designed hyperparameters $\Rightarrow$ Reinforcement Learning

Gindele et al.: Design of the planner of team AnnieWAY's autonomous vehicle used in the DARPA Urban Challenge 2007. IV, 2008.

12.4
Motion Planning

Motion Planning

Goal:

- Compute safe, comfortable and feasible trajectory from the vehicle's current configuration to the goal based on the output of the behavioral layer
- Local goal: center of lane a few meters ahead, stop line, parking spot
- Takes as input static and dynamic obstacles around vehicle and generates collision-free trajectory

Two types of output representations:

- Path: $\sigma(l) : [0, 1] \rightarrow \mathcal{X}$ (does not specify velocity)
- Trajectory: $\pi(t) : [0, T] \rightarrow \mathcal{X}$ (explicitly considers time)
- ... with $\mathcal{X}$ the configuration space of the vehicle and T the planning horizon

Motion Planning

Main Formulations:

- Variational Methods
- Graph Search Methods
- Incremental Search Techniques

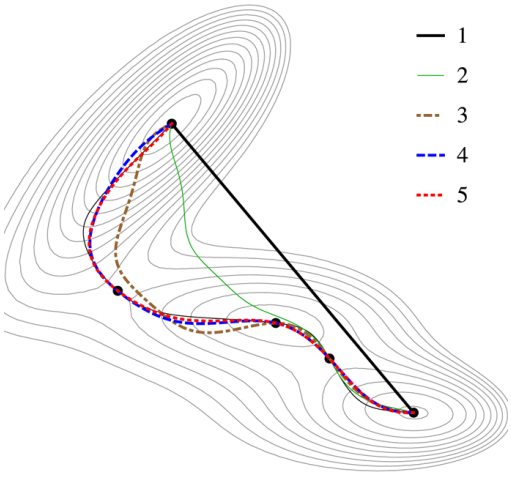
Variational Methods

Variational methods **minimize a functional**:

$$\begin{aligned} \underset{\pi}{\operatorname{argmin}} \quad & J(\pi) = \int_0^T f(\pi)\, dt \\ \text{s.t.} \quad & \pi(0) = \mathbf{x}_{init} \;\; \wedge \;\; \pi(T) \in \mathbf{X}_{goal} \end{aligned}$$

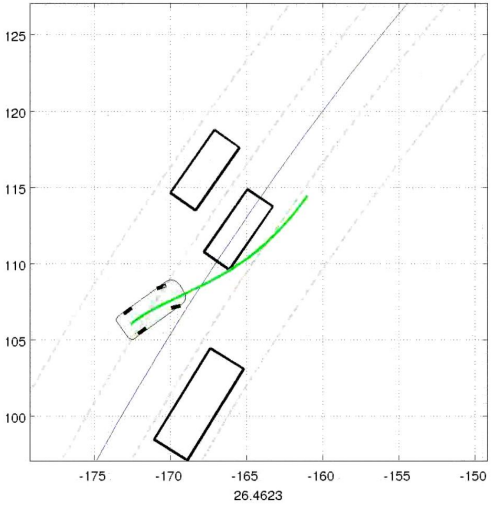
- The functional $f(\cdot)$ integrates soft constraints (spatial, velocity, jerk, etc.)
- Additional hard constraints can be formulated (minimum turn radius, etc.)
- Solved using numerical optimization
- Often non-linear problem $\Rightarrow$ converges to local minimum

Variational Optimization Example

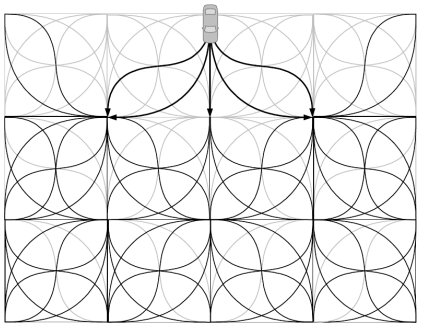


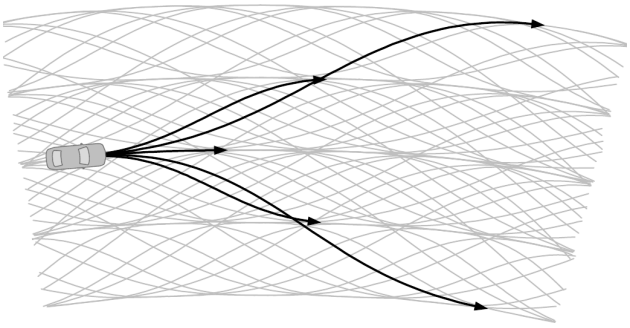
- Illustration of variational function optimization without hard constraints

Variational Optimization Example



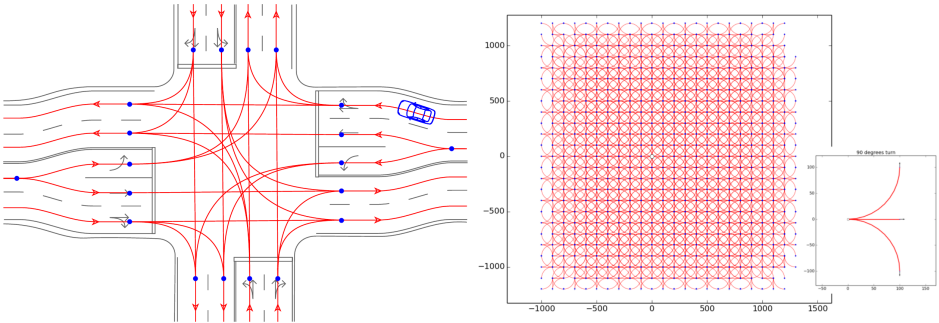
Graph Search Methods





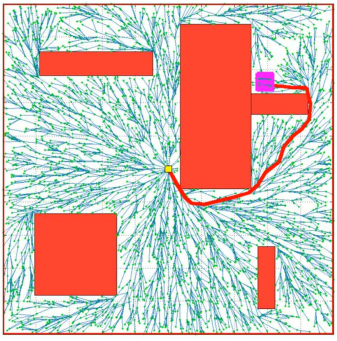
- Idea: Discretize configuration space $\mathcal{X}$ into graph $G = (V, E)$
- Various algorithms for constructing graphs
- Search strategies: Dijkstra, A^*, …

Graph Search Methods



- Idea: Discretize configuration space $\mathcal{X}$ into graph $G = (V, E)$
- Various algorithms for constructing graphs
- Search strategies: Dijkstra, A^*, …

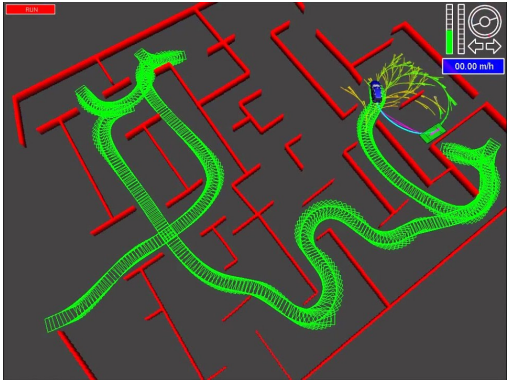
Incremental Search Techniques



- Idea: Incrementally build increasingly finer discretization of configuration space $\mathcal{X}$
- Guaranteed to provide feasible path given enough computation time
- But: computation time can be unbounded
- Prominent example: Rapidly exploring random trees (RRTs)

LaValle: Rapidly-exploring random trees: A new tool for path planning. Techical Report, 1998.

Hybrid A* Path Planning



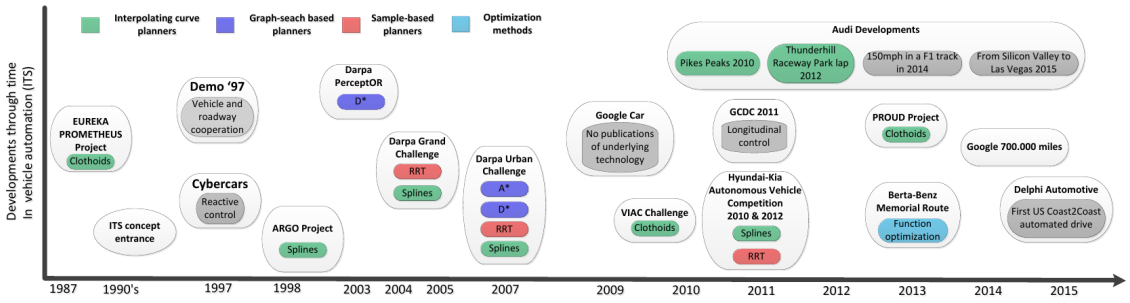
- Hybrid A* is an A* variant that guarantees kinematic feasibility of the path
- Planning is re-applied continuously as the car explores the environment

Dolgov et al.: Practical Search Techniques in Path Planning for Autonomous Driving. STAIR, 2008.

Summary

- Driving situations and behaviors are very complex
- Thus, we break the problem into a hierarchy of simpler problems:
- Route planning, behavior planning and motion planning
- Each problem is tailored to its scope and level of abstraction
- Road networks can be represented as weighted directed graphs
- Dijkstra's algorithm finds the shortest path in such a graph
- A* exploits planning heuristics to improve efficiency
- Behavior planning can be implemented using finite state machines
- For motion planning, variational and graph search methods are often used

Summary



- Various planning algorithms have been used in self-driving demonstrations
- There is no consensus on which algorithm works best, this is problem dependent

Thank You!

All the best for the exam