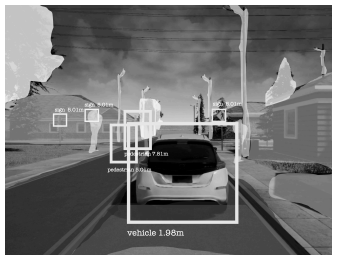
Search and Planning: Part 2

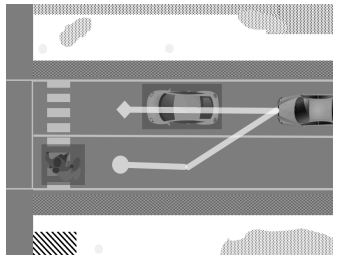
Lecture 11

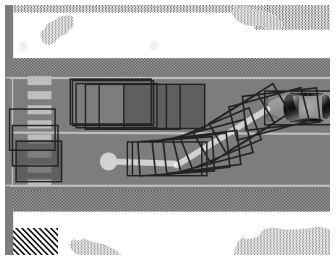
Sayan Mitra

GEM platform

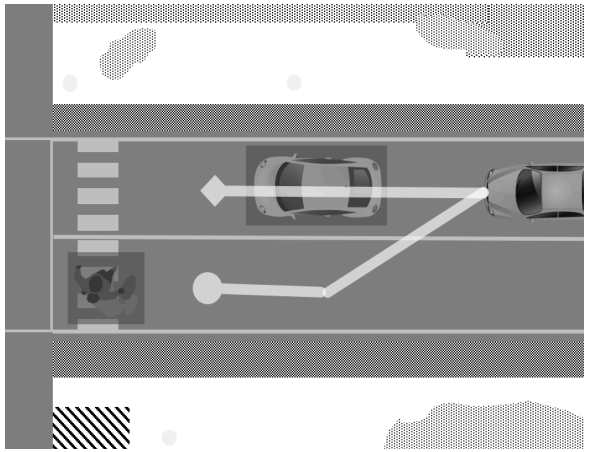
Autonomy pipeline







Sensing	Perception	Decisions and planning	Control
Physics-based models of camera, LIDAR, RADAR, GPS, etc.	Programs for object detection, lane tracking, scene understanding, etc.	Programs and multi-agent models of pedestrians, cars, etc.	Dynamical models of engine, powertrain, steering, tires, etc.

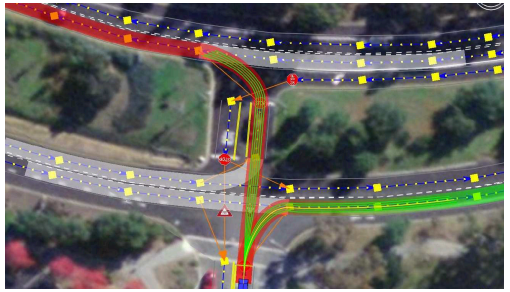


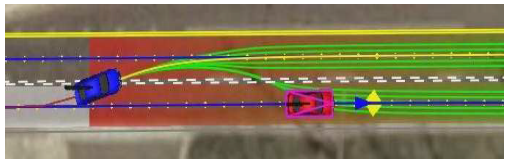
Decisions and planning

Programs and multi-agent models of pedestrians, cars, etc.

Search in different levels of navigation, planning, decision making, and control

- Global path planner --- invoked at each new *checkpoint*
  - finds paths from every point in the map to next checkpoint
  - dynamic programming
- Road navigation
  - For each path, the planner rolls out several discrete trajectories that are parallel to the smoothed center of the lane
- Freeform navigation (parking lots)
  - Generate arbitrary trajectories (irrespective of road structure) using modified A*





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

Outline

- Last lecture
  - Problem statement: path finding in a graph
  - Uninformed search
  - Informed search: Cost to go
  - Optimal search: A, A*
- Admissible Heuristics
  - Dominated heuristics
  - Consistent heuristics
- Hybrid A*
- Clustering

Problem statement: find shortest path

- Input: $\langle V, E, w, start, goal \rangle$
  - V: (finite) set of vertices
  - $E \subseteq V \times V$: (finite) set of edges
  - $w : E \to \mathbb{R}_{>0}$: a function that associates to each edge e to a strictly positive weight $w(e)$ (cost, length, time, fuel, prob. of detection)
  - $start, goal \in V$: respectively, start and end vertices.
- Output: $\langle P \rangle$
  - P is a path (starting in start and ending in goal, such that its weight $w(P)$ is minimal among all such paths
  - The weight of a path is the sum of the weights of its edges
  - The graph may be unknown, partially known, or known

Uniform cost search (Uninformed search)

```
Q ← ⟨start⟩                                                    //  maintains paths
                                                               // initialize queue with start
while Q ≠ ∅:
    pick (and remove) the path P with lowest cost g = w(P) from Q
    if head(P) =  goal then return P ;                         // Reached the goal
    foreach vertex v such that (head(P), v) ∈  E, do           // for all neighbors
        add ⟨v, P⟩ to Q ;                                      // Add expanded paths
return FAILURE ;                                               // nothing left to consider
```

Note no visited list; Use no information obtained from the environment

Example of Uniform-Cost Search

Q:

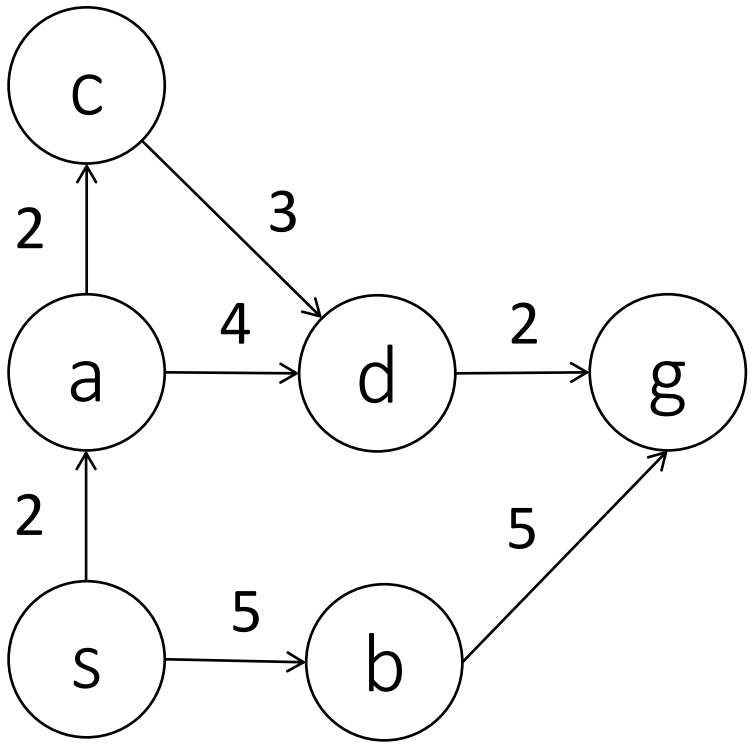
Path	Cost
<b, s>	5
<d, a, s>	6
<d, c, a, s>	7

<g, b, s> 10

<g, d, a, s> 8

<g, d, c, a, s> 9

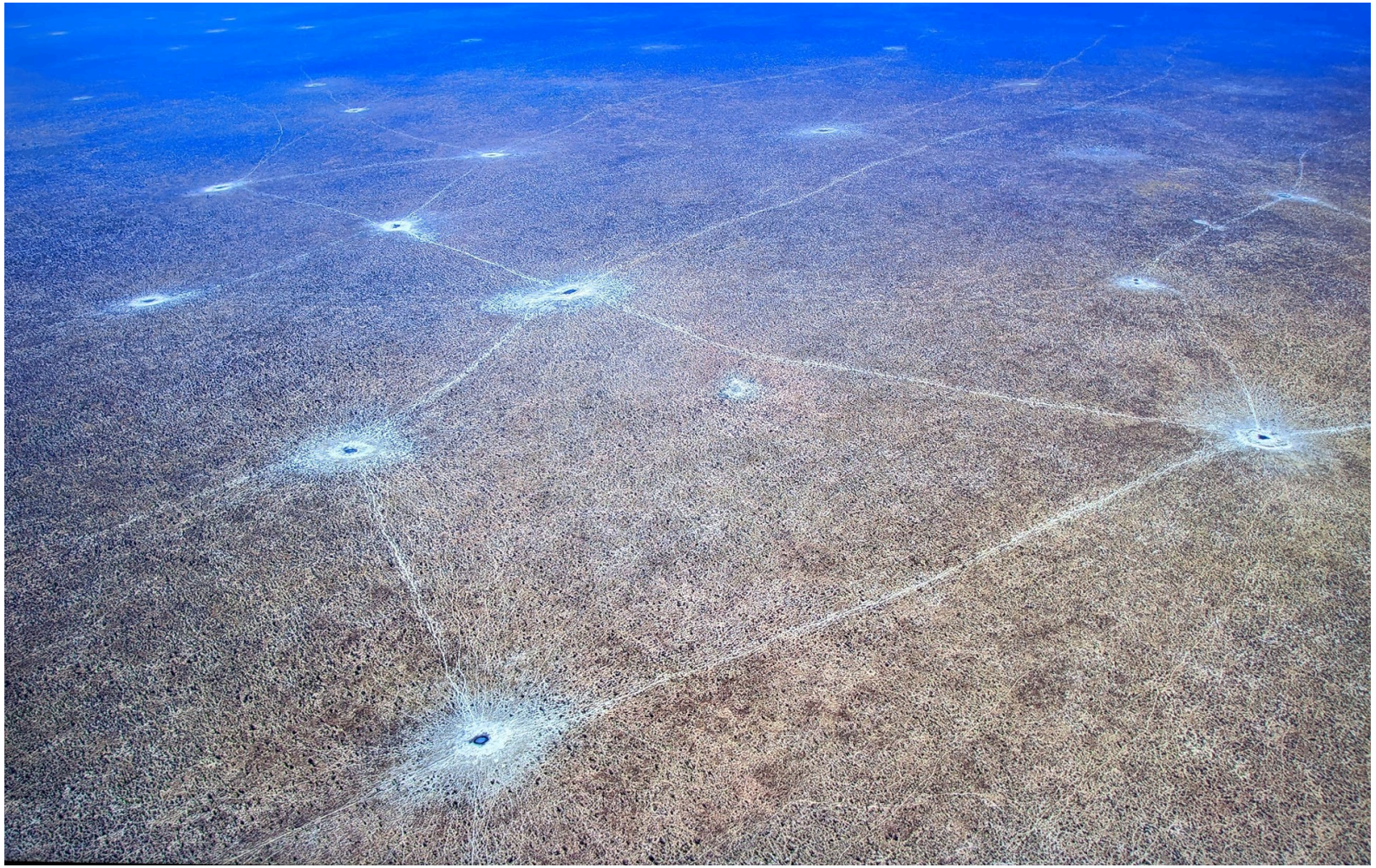
Stops when smallest cost path has 'g' as the head



Remarks on Uniform Cost Search

- UCS is an extension of BFS to the weighted-graph case (UCS = BFS if all edges have the same cost)
- UCS is *complete* and *optimal* (assuming costs bounded away from zero)
- UCS is guided by path cost rather than path depth, so it may get in trouble if some edge costs are very small
- Worst-case time and space complexity $O(b^{W^*/\epsilon})$, where W^* is the optimal cost, and ϵ is such that all edge weights are no smaller than

Slow search can be life or death



Elephants migrate in thousands from Okavango delta, Botswana, drawn by the need to find water

Greedy or Best-First Search

- UCS explores paths in all directions, with no bias towards the goal state
- What if we try to get "closer" to the goal?
- We need a measure of distance to the goal
  - It would be ideal to use the length of the shortest path...
  - but this is exactly what we are trying to compute!
- We can ***estimate*** the distance to the goal through a "heuristic function," $h : V \rightarrow \mathbb{R}_{\geq 0}$. E.g., the Euclidean distance to the goal (as the crow flies)
  - $h(v)$ is the estimate of the distance from v to goal
- A reasonable strategy is to always try to move in such a way to minimize the estimated distance to the goal: this is the basic idea of the greedy (best-first) search

Greedy/Best-first search

$Q \leftarrow \langle start \rangle$ // initialize queue with start

while $Q \neq \emptyset$:

  pick $(and\ remove)\ the\ path\ P\ with\ lowest\ heuristic\ cost\ h(head(P))\ from\ Q$

  if $head(P) = goal$ then return P // Reached the goal

  foreach $vertex\ v\ such\ that\ (head(P), v) \in E$, do // for all neighbors

    add $\langle v, P \rangle\ to\ Q$; // Add expanded paths

return $FAILURE$; // nothing left to consider

Example of Greedy search

Q:

Path	Cost	h
$\langle s \rangle$	0	10

→ $\langle a, s \rangle$ 2 2

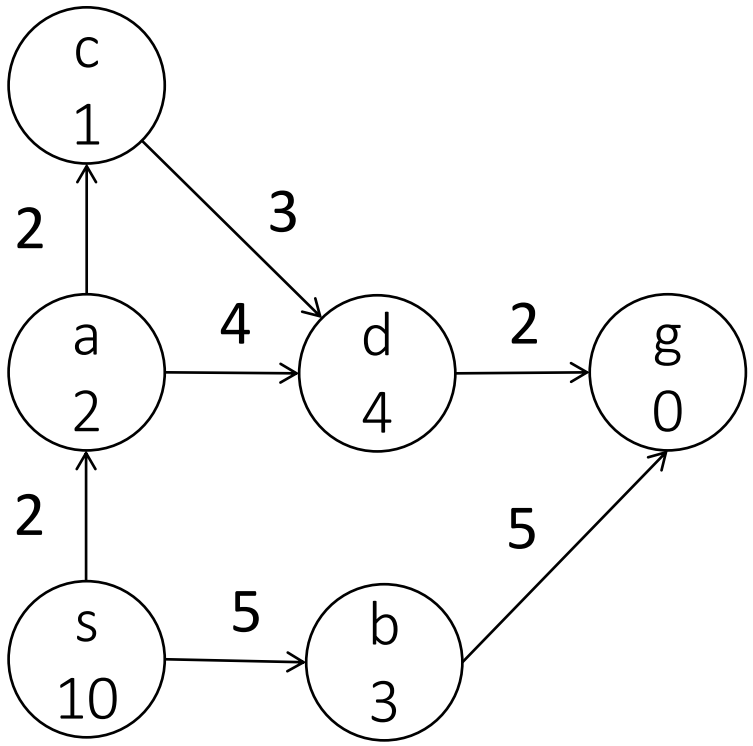
→ $\langle b, s \rangle$ 5 3

→ $\langle c, a, s \rangle$ 4 1

$\langle d, a, s \rangle$ 6 4

$\langle d, c, a, s \rangle$ 7 4

→ $\langle g, b, s \rangle$ 10 0 not optimal



Remarks on greedy/best-first search

- Greedy (Best-First) search is similar to Depth-First Search
  - keeps exploring until it has to back up due to a dead end
- **Not complete** (why?) and not optimal, but is often fast and efficient, depending on the heuristic function h
- Worst-case time and space complexity?

A search: informed search

- The problems
  - UCS is optimal, but may wander around a lot before finding the goal
  - Greedy is not optimal, but can be efficient, as it is heavily biased towards moving towards the goal. The non-optimality comes from neglecting "the past."

- The idea
  - Keep track both of the cost of the partial path to get to a vertex, say g(v), and of the heuristic function estimating the cost to reach the goal from a vertex, h(v)
  - In other words, choose as a "ranking" function the sum of the two costs:

$$f(v) = g(v) + h(v)$$

  - g(v) cost-to-come (from the start to v)
  - h(v): cost-to-go estimate (from v to the goal)
  - f (v): estimated cost of the path (from the start to v and then to the goal)

A search

open set and closed set

$Q \leftarrow \langle start \rangle$ // initialize queue with start

while $Q \neq \emptyset$:

  pick $(and\ remove)\ \ path\ P\ with\ lowest\ estimated\ cost\ f(P) = g(P) + h\big(head(P)\big)\ from\ Q$

  if $head(P) = \ goal$ then return P // Reached the goal

  foreach $vertex\ v\ such\ that\ (head(P), v) \in \ E$, do // for all neighbors

    add $\langle v, P \rangle\ to\ Q$; // Add expanded paths

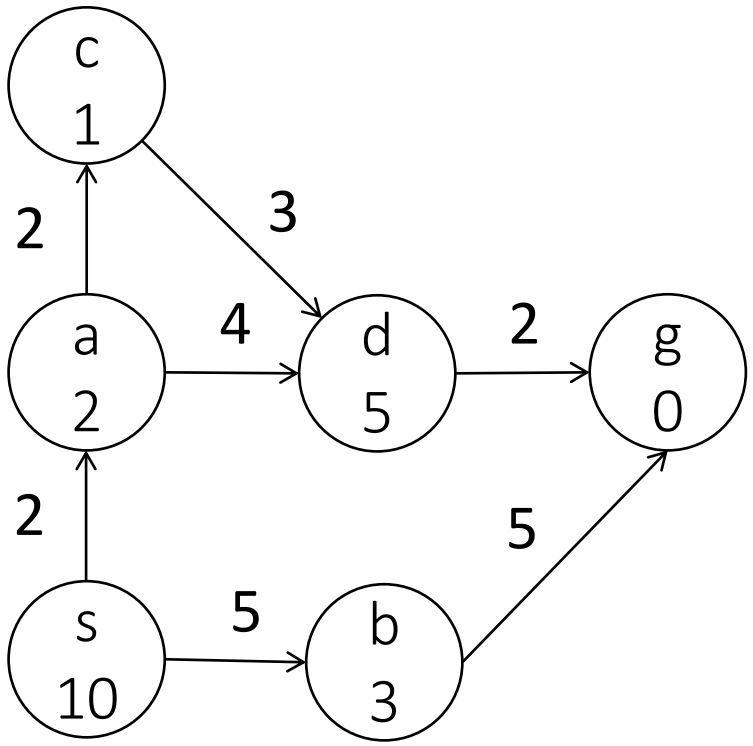
return $FAILURE$; // nothing left to consider

Example of A search

Q:

Path	g	h	f
~~$\langle a, s\rangle$~~	~~2~~	~~2~~	~~4~~
$\langle b, s\rangle$	5	3	8
$\langle c, a, s\rangle$	4	1	5
$\langle d, a, s\rangle$	6	5	11
$\langle d, c, a, s\rangle$	7	5	12
$\langle g, b, s\rangle$	10	0	10

not optimal



Remarks on A search

- A search is similar to UCS, with a bias induced by the heuristic h
- If h = 0, A = UCS.
- The A search is complete, but is *not optimal*
  - What is wrong? (Recall that if h = 0 then A = UCS, and hence optimal...)

A $*$ Search

- Choose an admissible heuristic, i.e., such $that\ h(v) \ \leq h^*(v)$
  - $h^*(v)$ is the "optimal" heuristic---perfect cost to go
  - To be admissible $h(v)$ should be at most $h^*(v)$
  - A search with an admissible heuristic is called A* --- guaranteed to find optimal path

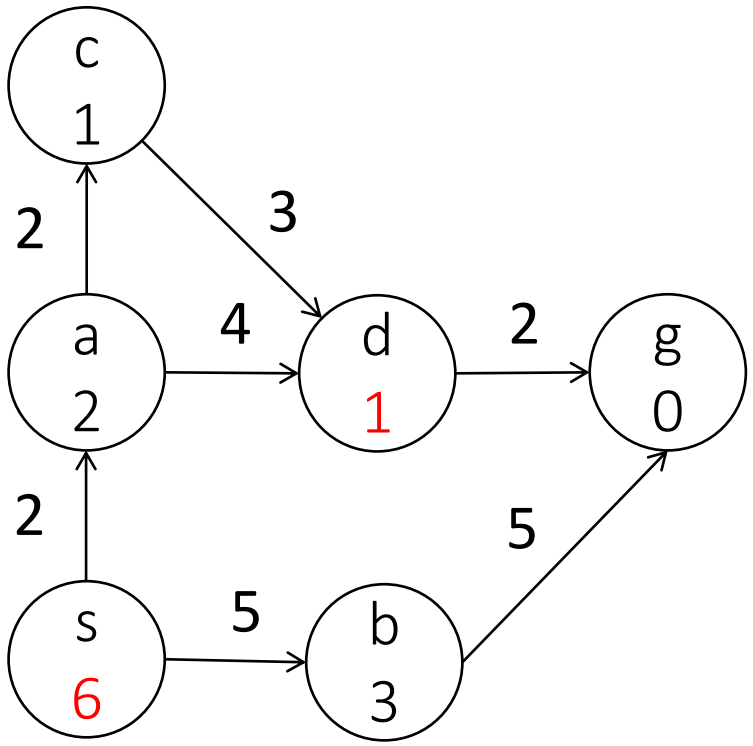
Example of A* search

Q:

Path	g	h	f
$\langle s \rangle$	0	6	6

changed h

finds ⟨g, d, a, s⟩
optimal path



Proof of optimality of A*

- Let w^* be the cost of the optimal path
- Suppose for the sake of contradiction, that A* returns P with $w(P) > w^*$
- Find the first unexpanded node on the optimal path P^*; call it n
- $f(n) > w(P)$, otherwise n would have been expanded
- $f(n) = g(n) + h(n)$

  $= g^*(n) + h(n)$ [since n is on the optimal path]

  $<= g^*(n) + h^*(n)$ [since h is admissible]

  $= f^*(n) = w^*$ [by def. of f, and since w^* is the cost of the optimal path]
- Hence $w^* >= f(n) = w(P)$, which is a contradiction

Outline

- Last lecture
  - Problem statement: path finding in a graph
  - Uninformed search
  - Informed search: Cost to go
  - Optimal search: A, A*
- Admissible Heuristics
  - Dominated heuristics
  - Consistent heuristics
- Hybrid A*
- Clustering

Admissible heuristics

- How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go.
- Examples of admissible heuristics
  - $h(v) = 0$: this always works! However, it is not very useful, A$*$ = UCS
  - $h(v) = distance(v, g)$ when the vertices of the graphs are physical locations
  - $h(v) = \left|\left|v - g\right|\right|_p$, when the vertices of the graph are points in a normed vector space
- A general method
  - Choose h as the optimal cost-to-go function for a relaxed problem, that is easy to compute
  - Relaxed problem: ignore some of the constraints in the original problem

Admissible heuristics for the 8-puzzle

Initial state:

1		5
2	6	3
7	4	8

Goal state:

1	2	3
4	5	6
7	8	

Which of the following are admissible heuristics?

- h = 0 — YES, always good
- h = 1 — not valid in goal state
- h = number of tiles in the wrong position — YES, “teleport” each tile to the goal in one move
- h = sum of (Manhattan) distance between tiles and their goal position — YES, move each tile to the goal ignoring other tiles.

A **partial** order of admissible heuristic functions

- Some admissible heuristics are better than others
  - h = 0 is an admissible heuristic, but is not very useful
  - h = h^* is also an admissible heuristic, and it the "best" possible one (it give us the optimal path directly, no searches/backtracking)
- Partial order
  - We say that h_1 dominates h_2 if $h_1(v) \geq h_2(v)$ for all vertices v
  - h^* dominates all admissible heuristics, and 0 is dominated by all admissible heuristics
- Choosing the right heuristic
  - In general, we want a heuristic that is as close to $h*$ as possible
  - However, such a heuristic may be too complicated to compute
  - There is a tradeoff between complexity of computing h and the complexity of the search

Consistent heuristics

- An additional useful property for A∗ heuristics is called consistency
  - A heuristic $h : X \rightarrow \mathbb{R}_{\geq 0}$ is said consistent if $\forall (u, v) \in E$

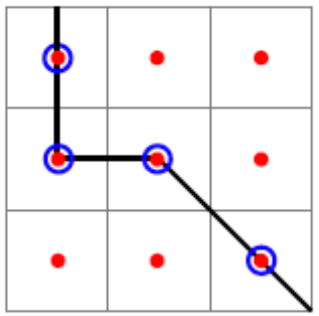
    $$h(u) \leq w\left(e = (u, v)\right) + h(v),$$

    In other words, a consistent heuristics satisfies a triangle inequality
- If h is a consistent heuristics, then $f = g + h$ is non-decreasing along paths: $f(v) = g(v) + h(v) = g(u) + w(u, v) + h(v) \geq f(u)$
- Hence, the values of f on the sequence of nodes expanded by A∗ is non-decreasing: the first path found to a node is also the optimal path ⇒ no need to compare costs!

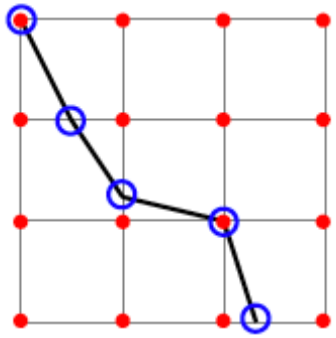
A* to Continuous State Spaces

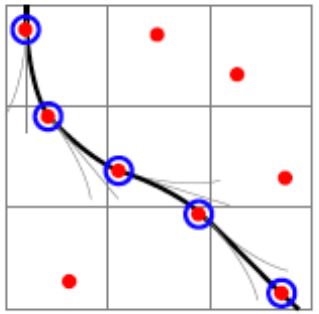
How to apply A* to continuous state space: Hybrid A*

(a)



- Represent vehicle state in a *uniform* discrete grid
  - 4D grid: $x, y, \theta\ (heading), dir$ (fwd,rev)
- A path (a) over this discrete grid is a start for a plan
- But, the discrete path (a) may not be executable by the vehicle dynamics
- *Hybrid A** solves this problem by shifting the points that represent the discrete cells

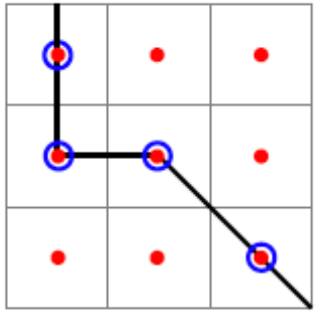




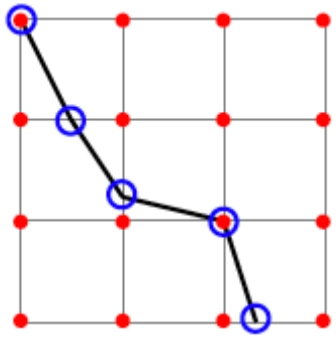
open cells: cells that are accessible from root and closed cells

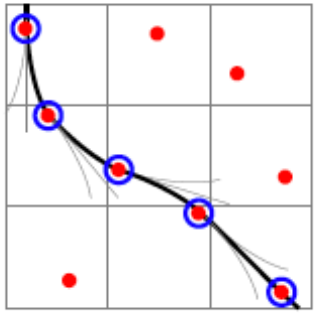
How to apply A* to continuous state spaces? Hybrid A*

(a)



- Represent vehicle state in a ~~*uniform*~~ discrete grid
  - 4D grid: $x, y, \theta\ (heading), dir$ (fwd,rev)
- If the current coordinate is $\langle x, y, \theta \rangle$ and those coordinates lie in cell c_i then the *representative continuous state* for cell c_i will be $x_i = x, y_i = y, \theta_i = \theta$
- After applying control input u to vehicle, suppose the predicted state is x', y', θ'
  - $x', y', \theta' = f(x, y, \theta, u); \dot{x} = \cdots$
  - representative for $c_j = x', y', \theta'$
  - This defines a transition from c_i to c_j





Junior: The Stanford Entry in the Urban Challenge

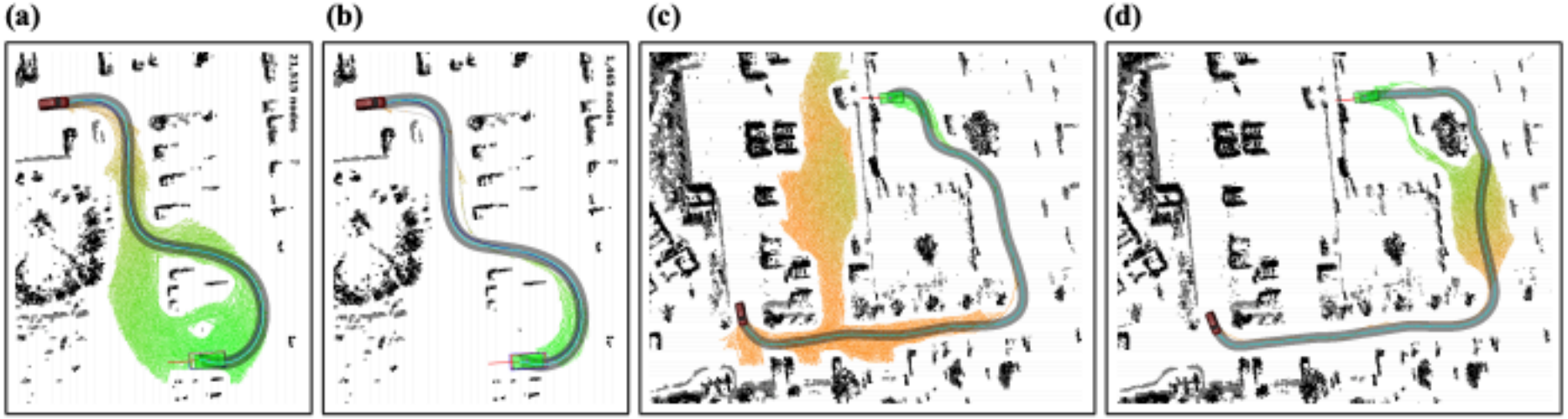
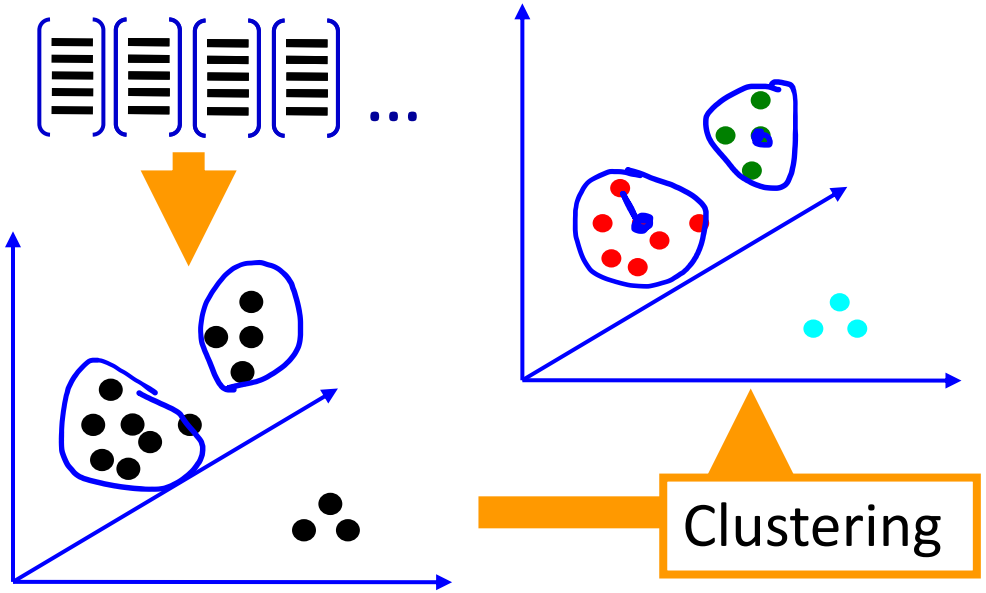


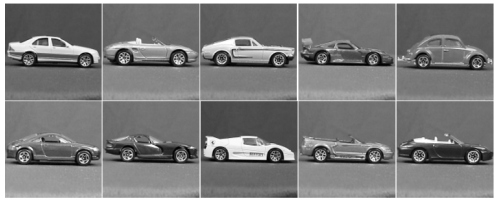
Figure 16: Hybrid-state A* heuristics. (a) Euclidean distance in 2-D expands 21,515 nodes. (b) The non-holonomic-without-obstacles heuristic is a significant improvement, as it expands 1,465 nodes, but as shown in (c), it can lead to wasteful exploration of dead-ends in more complex settings (68,730 nodes). (d) This is rectified by using the latter in conjunction with the holonomic-with-obstacles heuristic (10,588 nodes).

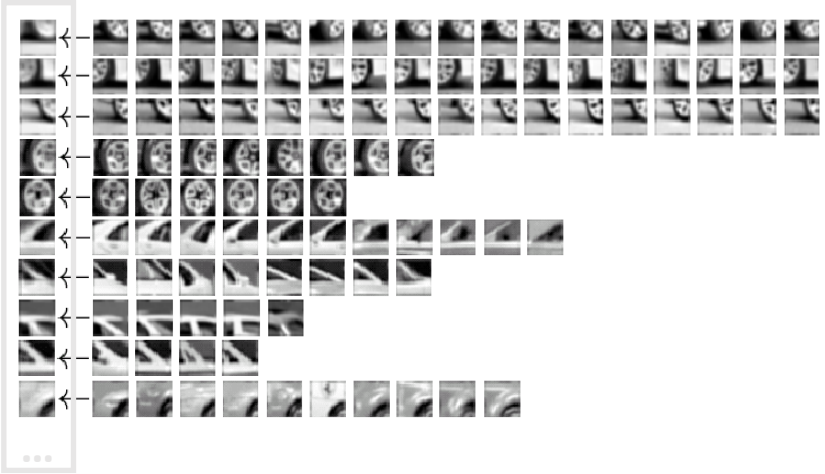
http://robots.stanford.edu/papers/junior08.pdf

Clustering

Recall we used clustering to learn visual vocabulary







Clustering

Problem statement. Given N vectors $x_1, \dots, x_N \in \mathbb{R}^n$, the goal is to partition them into k groups so that the vectors in the same group are close to one another

Examples: image compression (vectors are pixel values); patient clustering (patient attributes, tests);

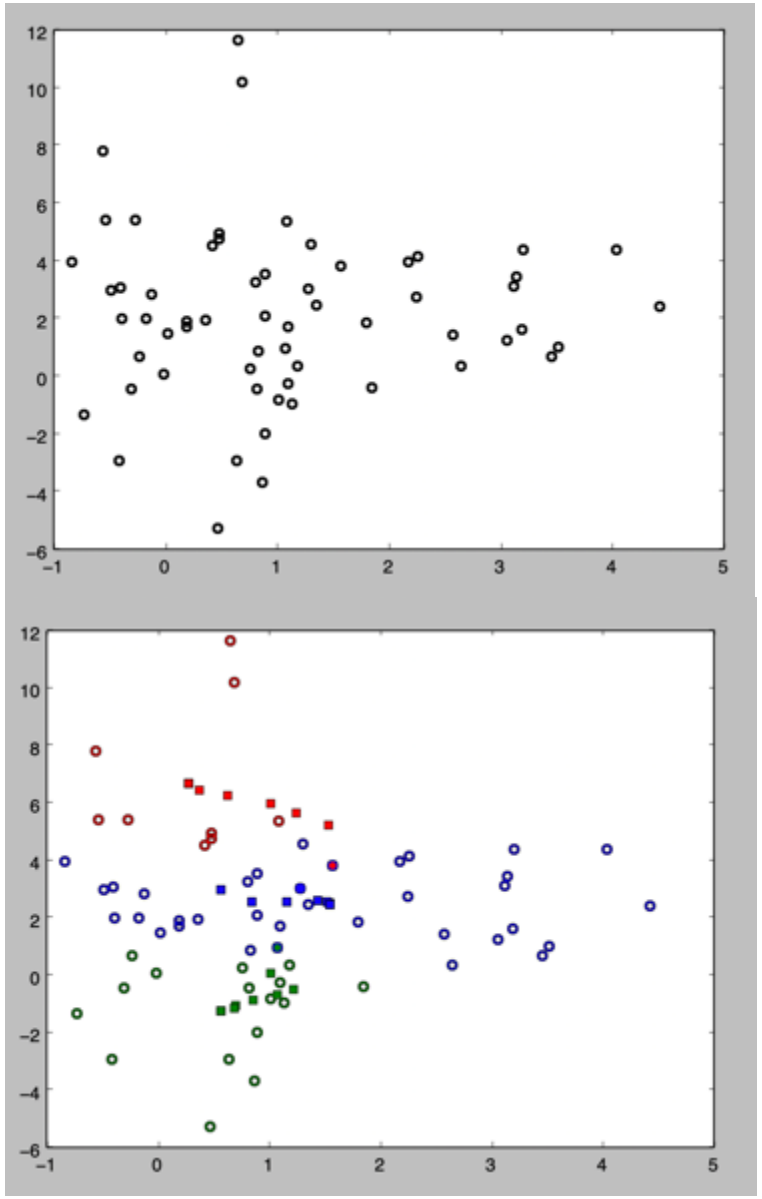
$c_i \in \{1, \dots, k\}$ is the group x_i belongs to

$G_{c_i} \subseteq \{x_1, \dots, x_n\}$ group

z_{c_i} group representative

k-means clustering objective minimize $\mathrm{J}_{clust} = \frac{1}{N}\sum_{i=1}^{N}\left|x_i - z_{c_i}\right|^2$ by choosing the groups $\{c_i\}$ and the representatives

Theorem. **2-means clustering** is NP hard.



from Boyd & Vandenberghe

K-means clustering algorithm: Step 1

- Suppose the representatives $z_1, \dots, z_k$ are given, how do we assign the vectors $x_1, \dots, x_N$ to the k groups?
- Recall the clustering cost is $\mathrm{J}_{clust} = \frac{1}{n}\sum_{i=1}^{N}\left|x_i - z_{c_i}\right|^2$
- $\min_j \frac{1}{N}\sum_{i=1}^{N}\left|x_i - z_j\right|^2 = \quad \frac{1}{N}\sum_{i=1}^{N}\min_j\left|x_i - z_j\right|^2$
- That is, assign x_i to the nearest representative z_j

Algorithm: Step 2

- Given the partition $G_1, \dots, G_k$, how to choose the representatives $z_1, \dots, z_k$?
- $\mathrm{J}_{clust} = J_1 + \cdots + J_k = \sum_{j=1}^{k} \frac{1}{|G_j|} \sum_{i \in G_j} \left| x_i - z_j \right|^2$
- Choose z_j to minimize J_j, that is $z_j = \frac{1}{|G_j|} \sum_{i \in G_j} x_i$ the mean (centroid)

Algorithm: Combined

- alternate between updating the partition, then the representatives
- well-known algorithm called *k-means clustering*
- objective J_{clust} decreases in each step

given $x_1, \ldots, x_N \in \mathbb{R}^n$ and $z_1, \ldots, z_k \in \mathbb{R}^n$
repeat
  update partition: assign i to G_j, $j = \mathrm{argmin}_j \left|x_i - z_j\right|_2^2$
  update centroids: $z_j = \frac{1}{|P_j|} \sum_{i \in P_j} x_i$
until $z_1, \ldots, z_k$ stop changing

Properties of k-means cluster and questions

The cost J_{clust} monotonically decreases in each iteration

Questions and problems

- How much time does each iteration of k-means algorithm take?
- Does K-means give optimal clustering?
  - Show an example where it does not.
- Code it and experiment with different k, n, and distance metrics

Summary

- A* algorithm combines cost-to-come g(v) and a heuristic function h(v) for cost-to-go to find shortest path
  - informed search
- Heuristic function must be *admissible* $h(v) \leq h^*(v)$
  - Never over-estimate the actual cost to go
  - Are all $h(v)$ values needed ?
  - What if h is not admissible
  - How to find heuristics
- K-means clustering