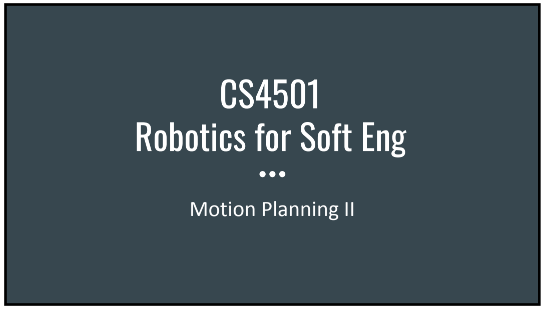
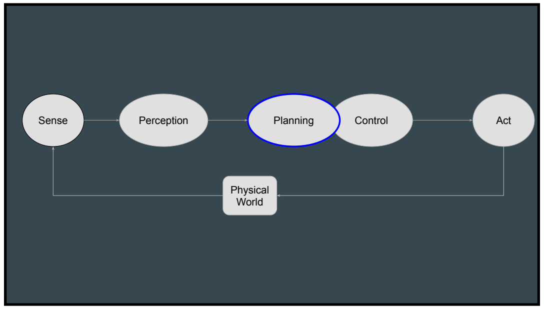


CS4501
Robotics for Soft Eng

•••

Motion Planning II



Sense

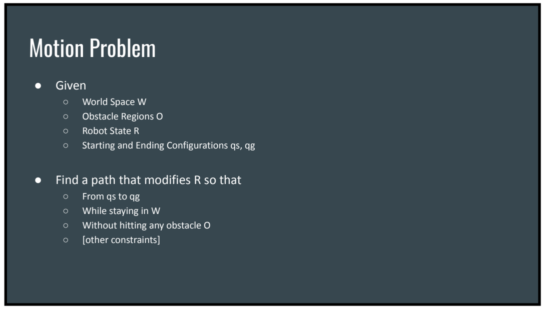
Perception

Planning

Control

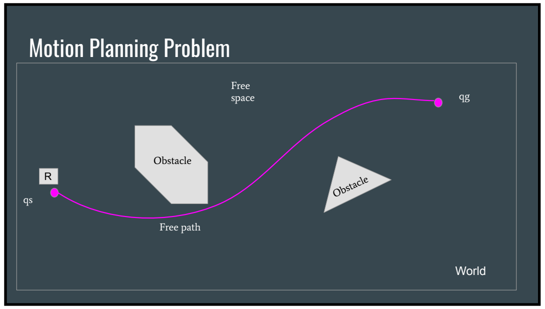
Act

Physical World



Motion Problem

- Given
  - World Space W
  - Obstacle Regions O
  - Robot State R
  - Starting and Ending Configurations qs, qg
- Find a path that modifies R so that
  - From qs to qg
  - While staying in W
  - Without hitting any obstacle O
  - [other constraints]



Motion Planning Problem

Free space

qg

Obstacle

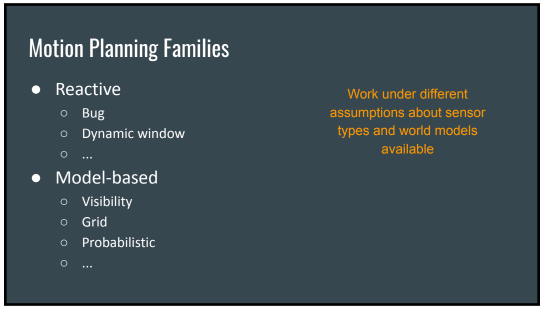
R

qs

Obstacle

Free path

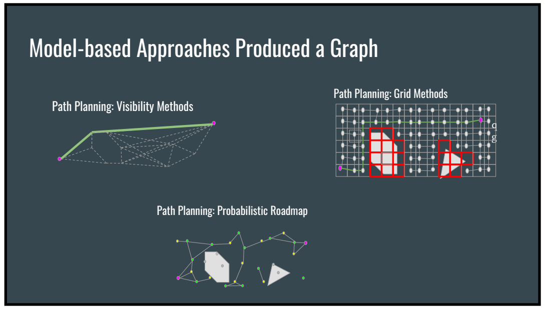
World



Motion Planning Families

- Reactive
  - Bug
  - Dynamic window
  - ...
- Model-based
  - Visibility
  - Grid
  - Probabilistic
  - ...

Work under different assumptions about sensor types and world models available

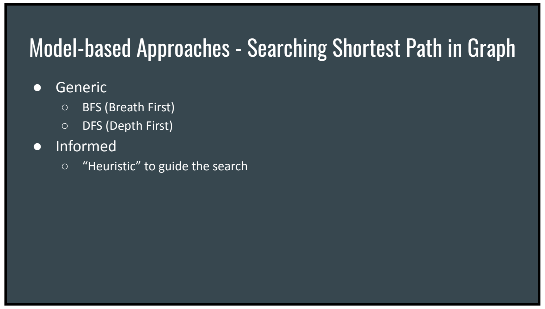


Model-based Approaches Produced a Graph

Path Planning: Visibility Methods

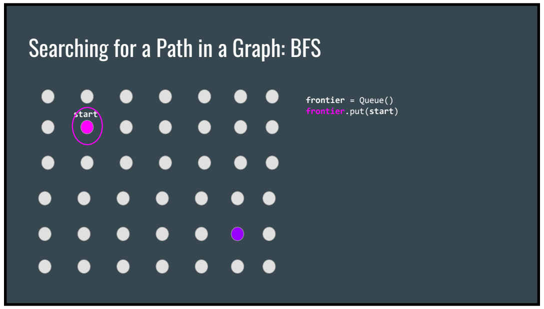
Path Planning: Grid Methods

Path Planning: Probabilistic Roadmap



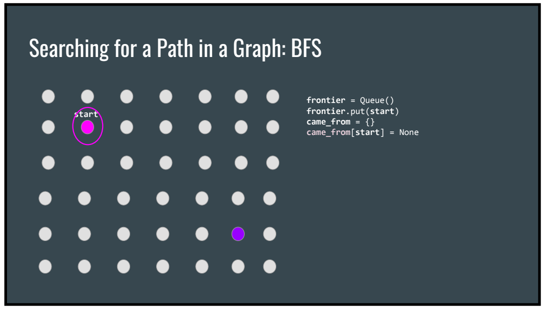
Model-based Approaches - Searching Shortest Path in Graph

- Generic
  - BFS (Breath First)
  - DFS (Depth First)
- Informed
  - “Heuristic” to guide the search



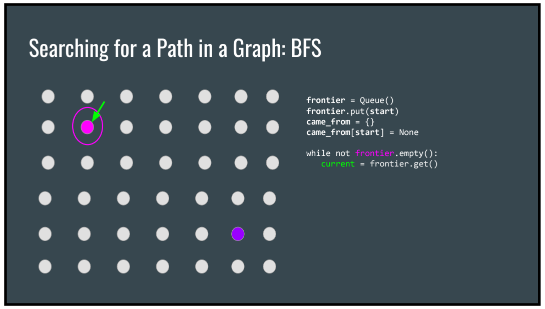
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
```



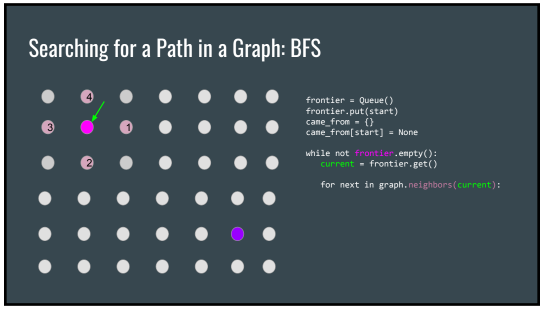
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None
```



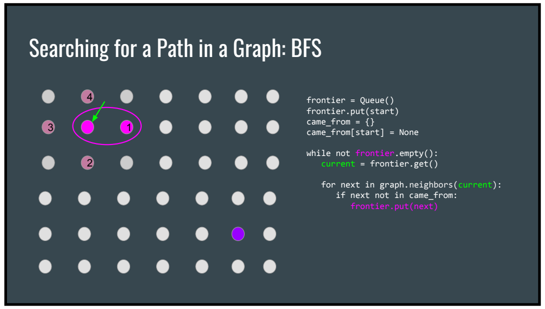
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
```



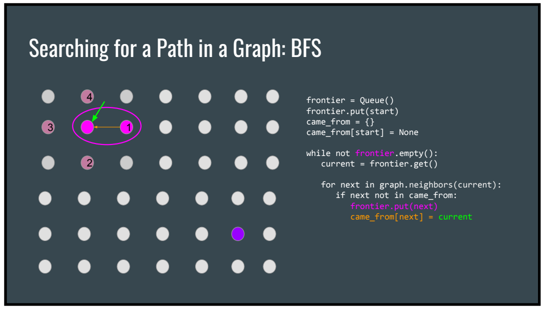
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
```



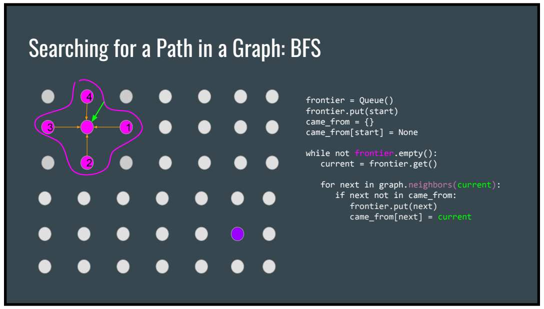
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
```



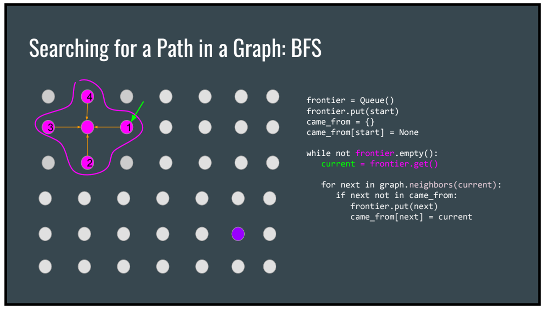
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



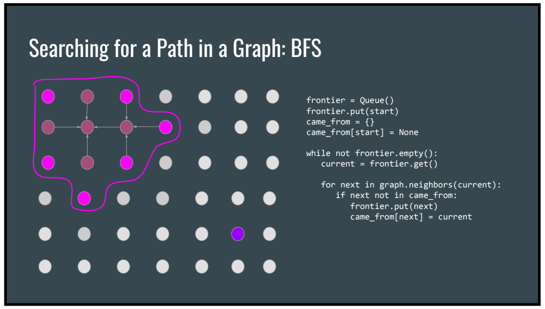
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



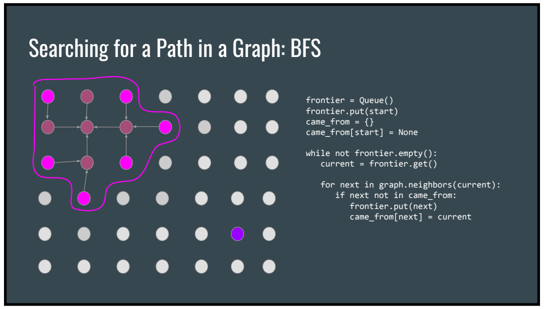
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



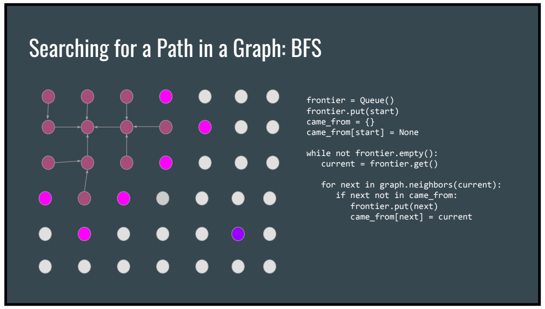
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



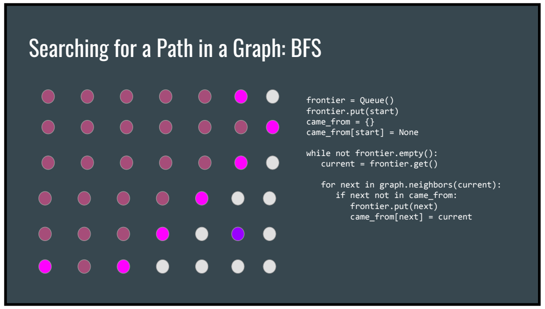
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



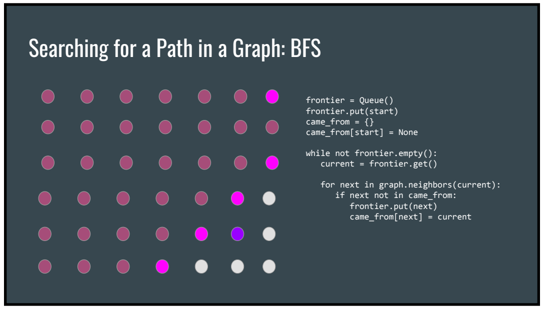
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



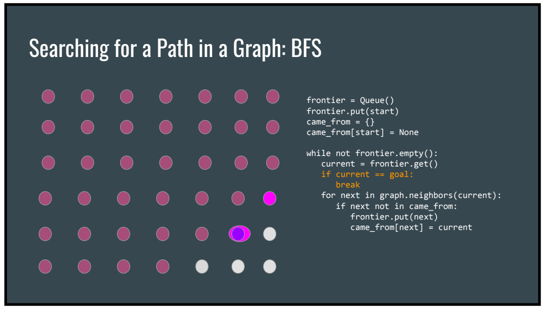
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



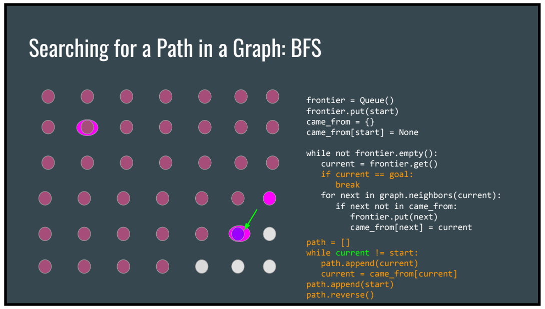
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



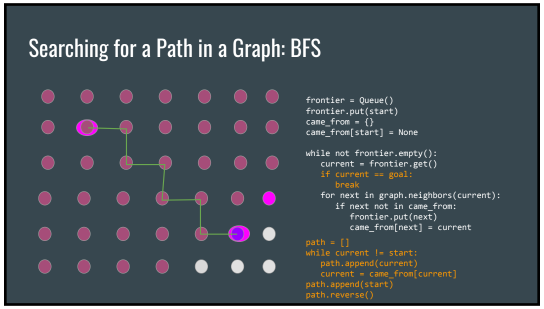
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```



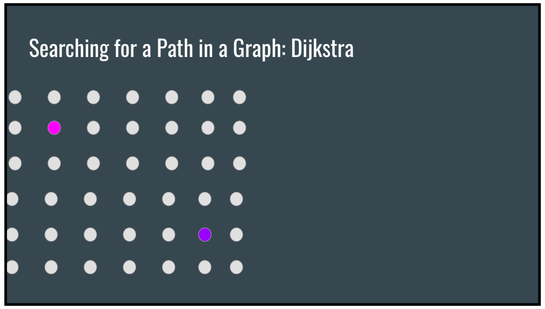
Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
path = []
while current != start:
   path.append(current)
   current = came_from[current]
path.append(start)
path.reverse()
```

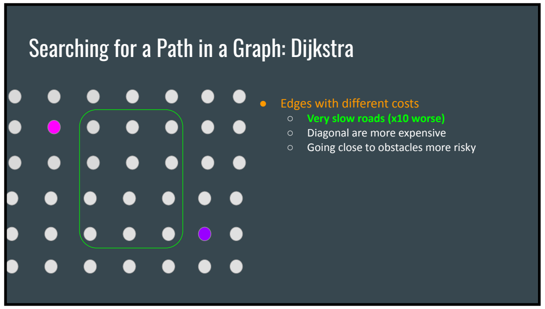


Searching for a Path in a Graph: BFS

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
path = []
while current != start:
   path.append(current)
   current = came_from[current]
path.append(start)
path.reverse()
```

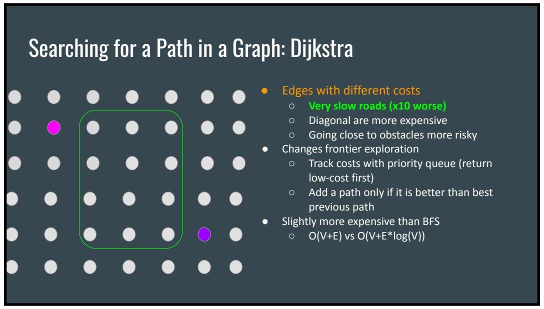


Searching for a Path in a Graph: Dijkstra



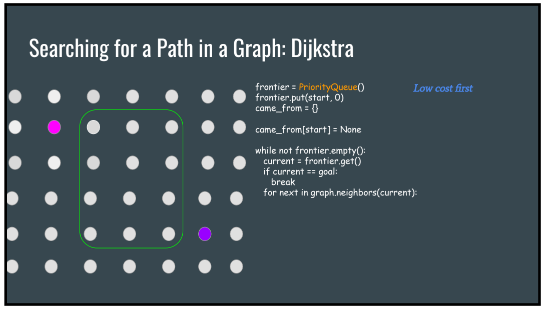
Searching for a Path in a Graph: Dijkstra

- Edges with different costs
  - **Very slow roads (x10 worse)**
  - Diagonal are more expensive
  - Going close to obstacles more risky



Searching for a Path in a Graph: Dijkstra

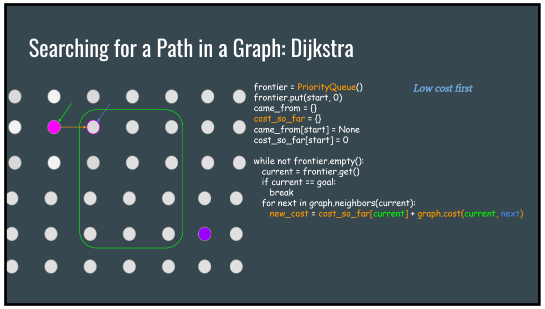
- Edges with different costs
  - **Very slow roads (x10 worse)**
  - Diagonal are more expensive
  - Going close to obstacles more risky
- Changes frontier exploration
  - Track costs with priority queue (return low-cost first)
  - Add a path only if it is better than best previous path
- Slightly more expensive than BFS
  - O(V+E) vs O(V+E*log(V))



Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}

came_from[start] = None

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
```

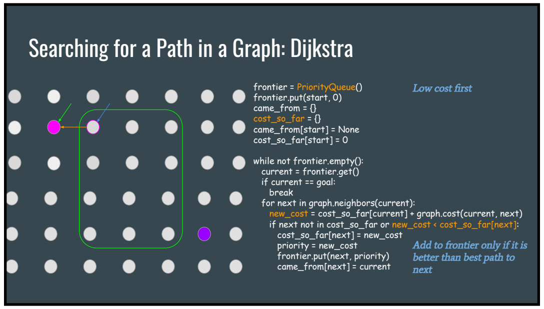
Low cost first



Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
```

Low cost first

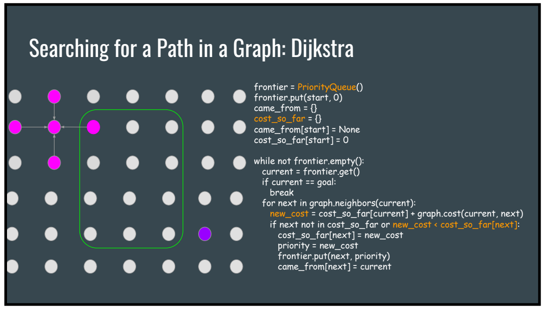


Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```

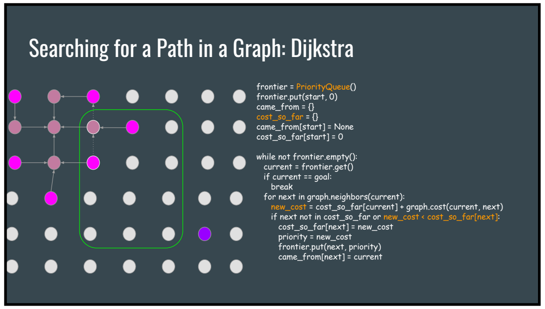
Low cost first

Add to frontier only if it is better than best path to next



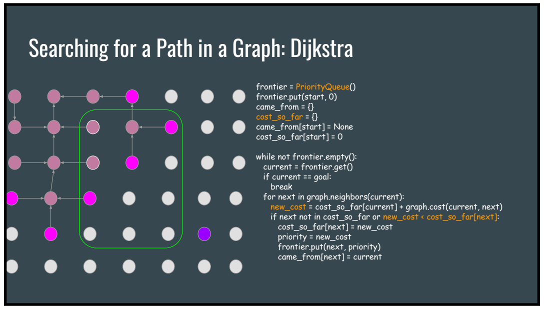
Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```



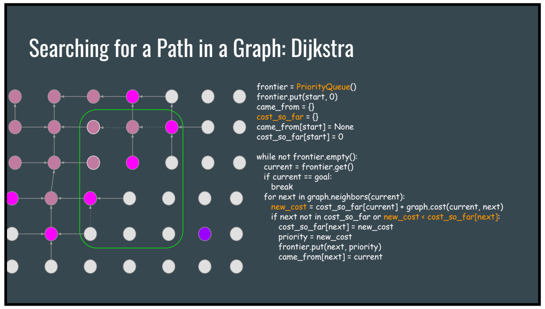
Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```



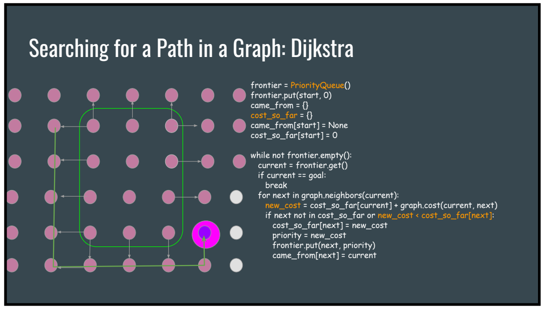
Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```



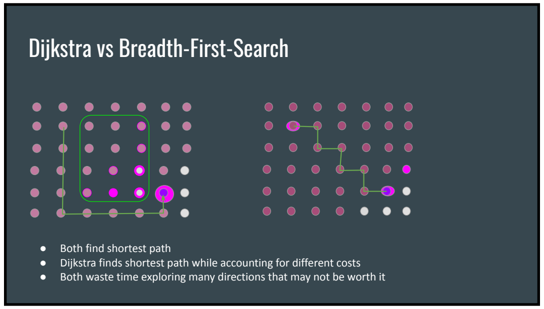
Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```



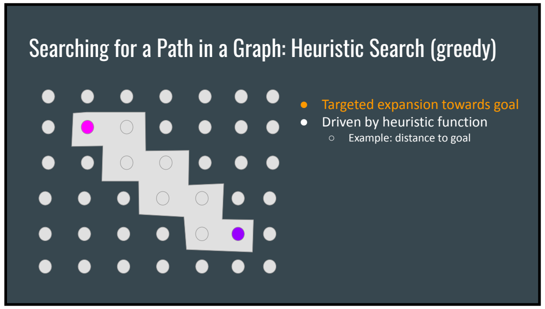
Searching for a Path in a Graph: Dijkstra

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()
   if current == goal:
      break
   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```



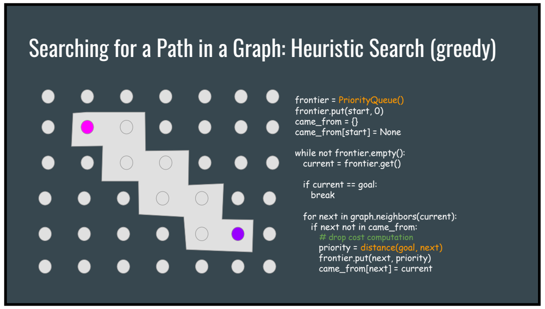
Dijkstra vs Breadth-First-Search

- Both find shortest path
- Dijkstra finds shortest path while accounting for different costs
- Both waste time exploring many directions that may not be worth it



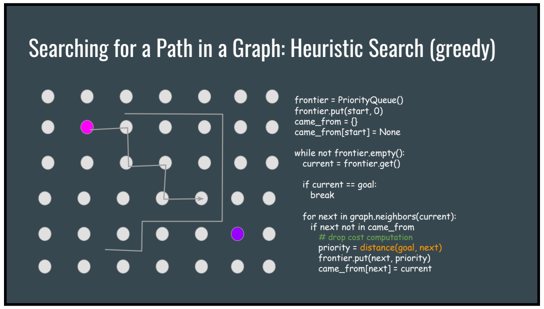
Searching for a Path in a Graph: Heuristic Search (greedy)

- Targeted expansion towards goal
- Driven by heuristic function
  - Example: distance to goal



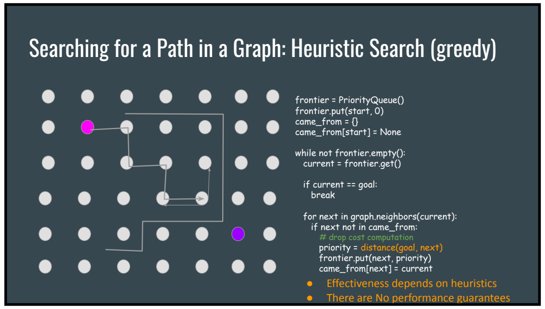
Searching for a Path in a Graph: Heuristic Search (greedy)

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         # drop cost computation
         priority = distance(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```



Searching for a Path in a Graph: Heuristic Search (greedy)

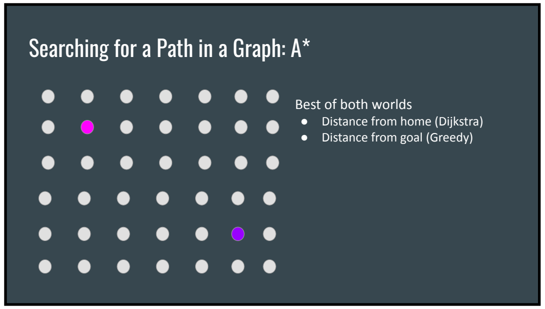
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from
         # drop cost computation
         priority = distance(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```



Searching for a Path in a Graph: Heuristic Search (greedy)

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         # drop cost computation
         priority = distance(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

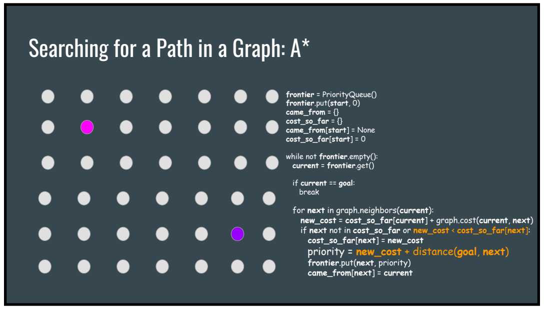
- Effectiveness depends on heuristics
- There are No performance guarantees



Searching for a Path in a Graph: A*

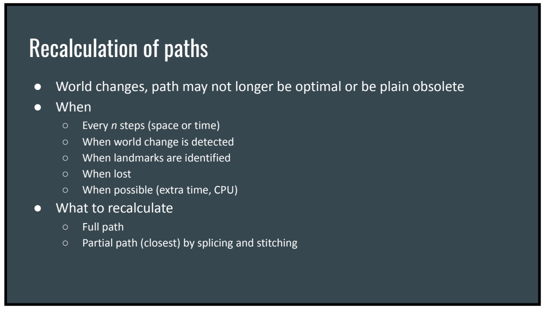
Best of both worlds

- Distance from home (Dijkstra)
- Distance from goal (Greedy)



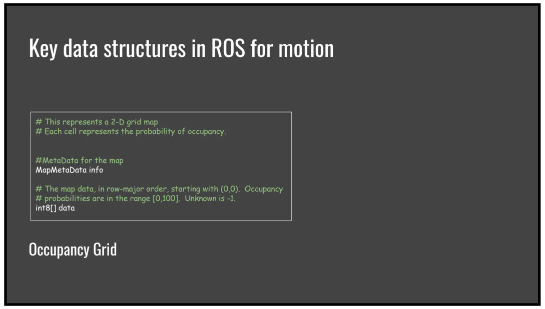
Searching for a Path in a Graph: A*

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + distance(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```



Recalculation of paths

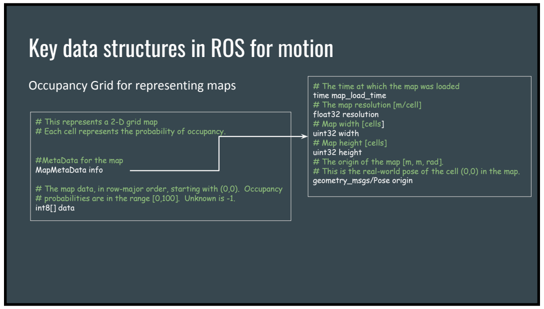
- World changes, path may not longer be optimal or be plain obsolete
- When
  - Every *n* steps (space or time)
  - When world change is detected
  - When landmarks are identified
  - When lost
  - When possible (extra time, CPU)
- What to recalculate
  - Full path
  - Partial path (closest) by splicing and stitching



Key data structures in ROS for motion

```
# This represents a 2-D grid map
# Each cell represents the probability of occupancy.


#MetaData for the map
MapMetaData info

# The map data, in row-major order, starting with (0,0).  Occupancy
# probabilities are in the range [0,100].  Unknown is -1.
int8[] data
```

Occupancy Grid

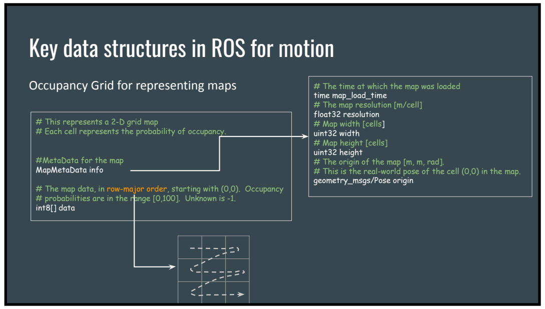


Key data structures in ROS for motion

Occupancy Grid for representing maps

```
# This represents a 2-D grid map
# Each cell represents the probability of occupancy.


#MetaData for the map
MapMetaData info

# The map data, in row-major order, starting with (0,0).  Occupancy
# probabilities are in the range [0,100].  Unknown is -1.
int8[] data
```

```
# The time at which the map was loaded
time map_load_time
# The map resolution [m/cell]
float32 resolution
# Map width [cells]
uint32 width
# Map height [cells]
uint32 height
# The origin of the map [m, m, rad].
# This is the real-world pose of the cell (0,0) in the map.
geometry_msgs/Pose origin
```

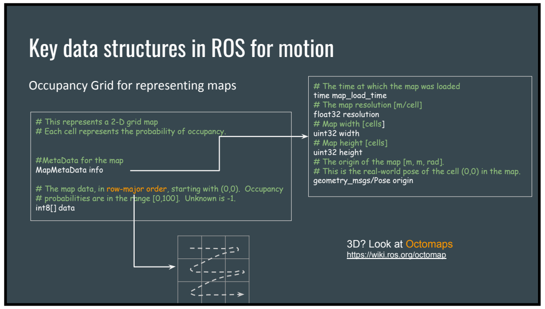


Key data structures in ROS for motion

Occupancy Grid for representing maps

```
# This represents a 2-D grid map
# Each cell represents the probability of occupancy.


#MetaData for the map
MapMetaData info

# The map data, in row-major order, starting with (0,0).  Occupancy
# probabilities are in the range [0,100].  Unknown is -1.
int8[] data
```

```
# The time at which the map was loaded
time map_load_time
# The map resolution [m/cell]
float32 resolution
# Map width [cells]
uint32 width
# Map height [cells]
uint32 height
# The origin of the map [m, m, rad].
# This is the real-world pose of the cell (0,0) in the map.
geometry_msgs/Pose origin
```



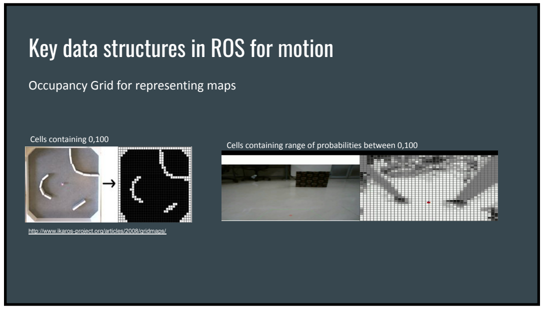
Key data structures in ROS for motion

Occupancy Grid for representing maps

```
# This represents a 2-D grid map
# Each cell represents the probability of occupancy.


#MetaData for the map
MapMetaData info

# The map data, in row-major order, starting with (0,0).  Occupancy
# probabilities are in the range [0,100].  Unknown is -1.
int8[] data
```

```
# The time at which the map was loaded
time map_load_time
# The map resolution [m/cell]
float32 resolution
# Map width [cells]
uint32 width
# Map height [cells]
uint32 height
# The origin of the map [m, m, rad].
# This is the real-world pose of the cell (0,0) in the map.
geometry_msgs/Pose origin
```

3D? Look at Octomaps
https://wiki.ros.org/octomap



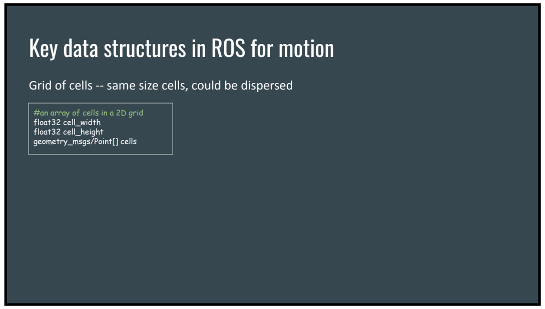
Key data structures in ROS for motion

Occupancy Grid for representing maps

Cells containing 0,100

http://www.ikaros-project.org/articles/2008/gridmaps/

Cells containing range of probabilities between 0,100



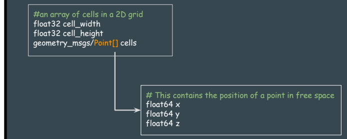
Key data structures in ROS for motion

Grid of cells -- same size cells, could be dispersed

```
#an array of cells in a 2D grid
float32 cell_width
float32 cell_height
geometry_msgs/Point[] cells
```

Key data structures in ROS for motion

Grid of cells -- same size cells, could be dispersed

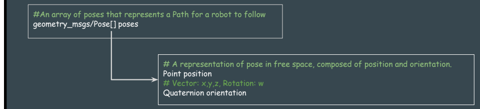


```
#an array of cells in a 2D grid
float32 cell_width
float32 cell_height
geometry_msgs/Point[] cells
```

```
# This contains the position of a point in free space
float64 x
float64 y
float64 z
```

Key data structures in ROS for motion

Path as a sequence of poses (waypoints + orientation)



```
#An array of poses that represents a Path for a robot to follow
geometry_msgs/Pose[] poses
```

```
# A representation of pose in free space, composed of position and orientation.
Point position
# Vector: x,y,z, Rotation: w
Quaternion orientation
```

Take Away

- Families of approaches to employ in tandem
  - Reactive
    - Local area and fast response
  - Model-based
    - Big picture and long paths
    - Build and searching graphs
  - ROS Support