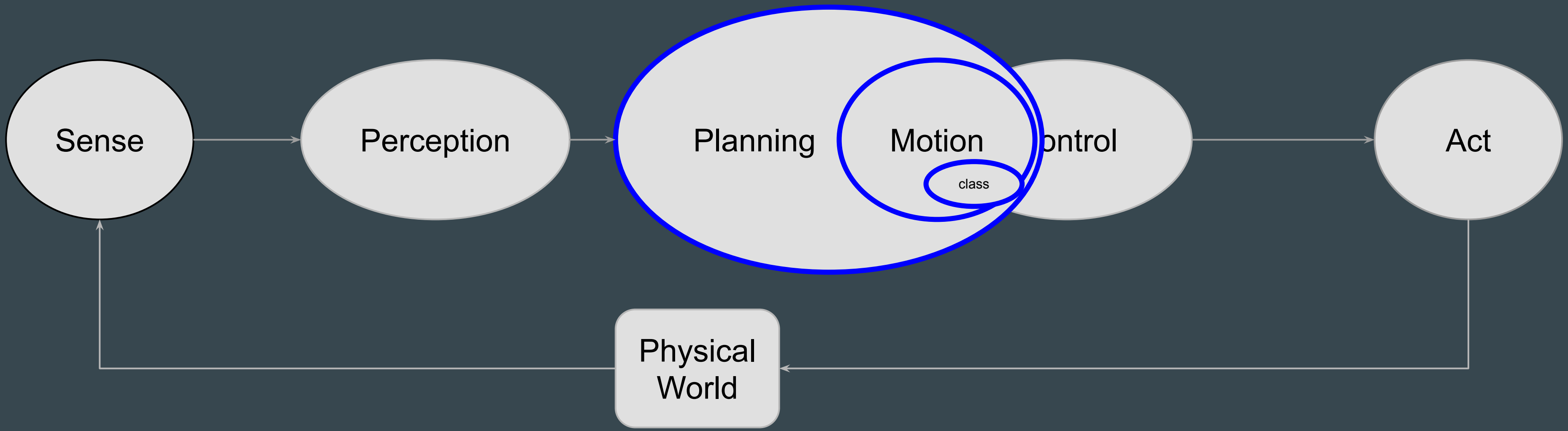
CS4501
Robotics for Soft Eng

•••

Motion Planning

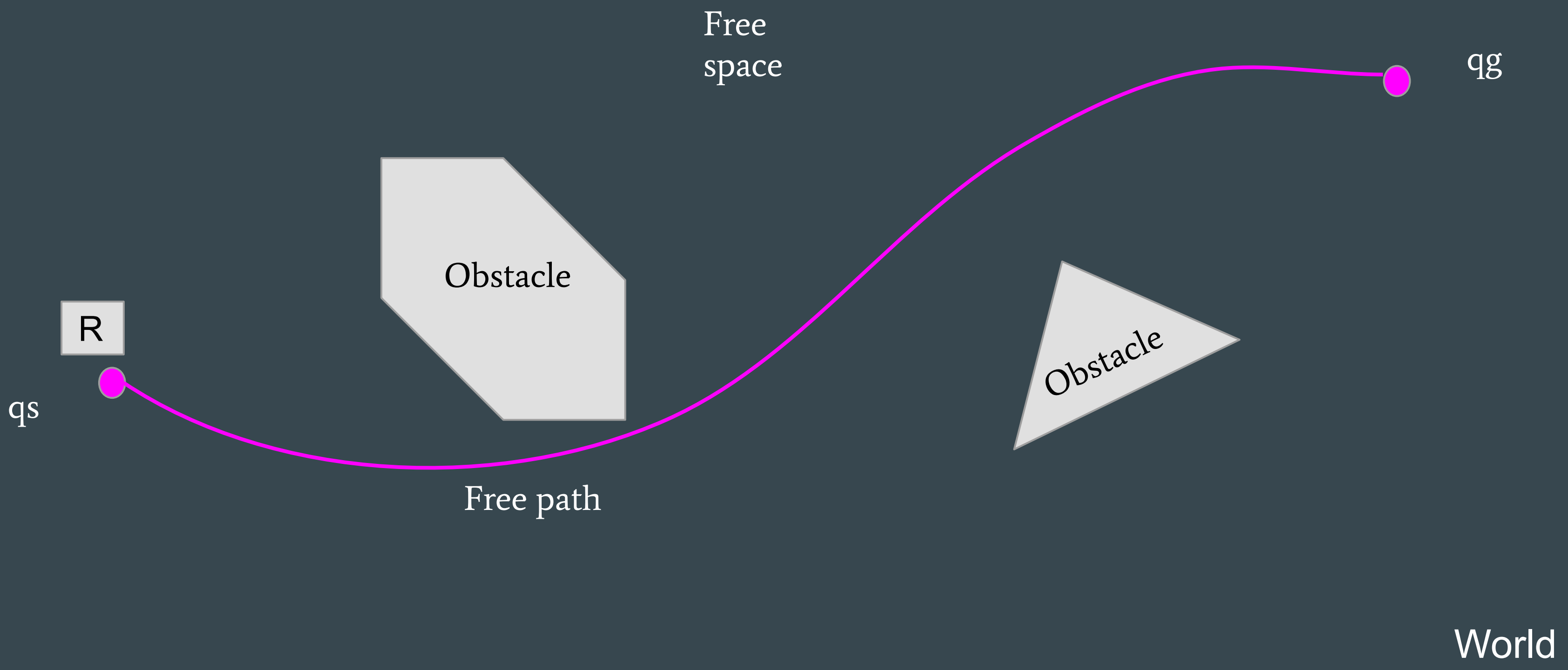

Sense
Perception
Planning
Motion
class
ontrol
Act
Physical
World

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



Motion Planning Families

- Reactive
- Model-based

Work under different assumptions about sensor types and world models available

Motion Planning Families

- Reactive
  - Online
  - Fast, non-optimal

Bug Algorithms

Robot

- Is modeled as a bounded point
  - Under-approximation of robot constraints induced by physical structure
  - Over-approximation of robot capabilities in terms of directionality

Bug Algorithms

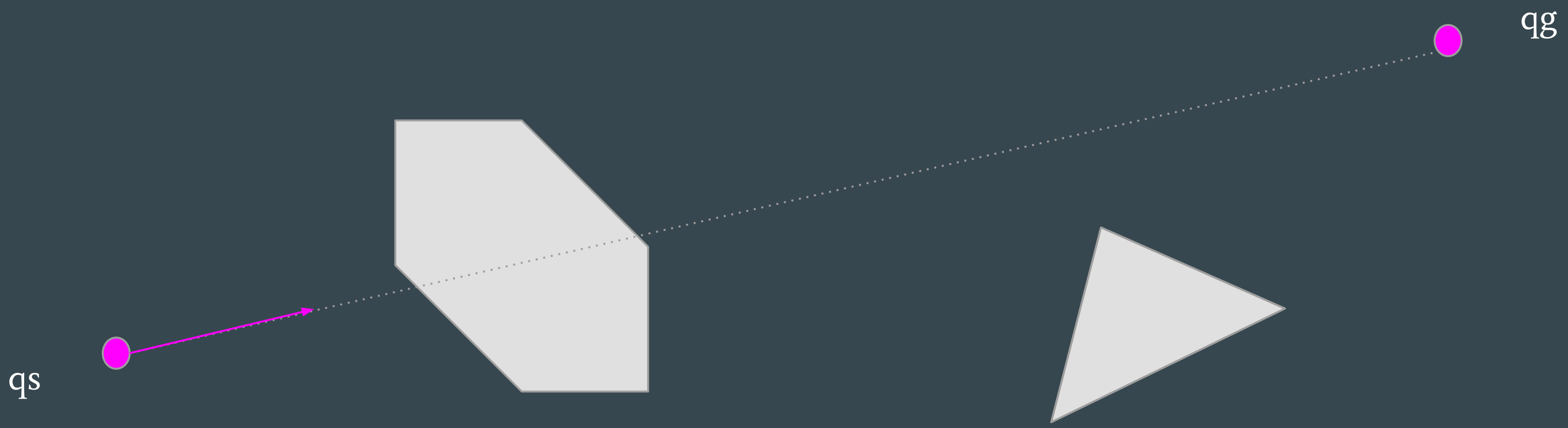
Robot

- Is modeled as a bounded point
- Can sense its location precisely
- Can sense contact with obstacles
- Can compute direction towards goal and distance between two points
- Does not know location of obstacles in advanced

Bug Algorithm 1

Repeat until Robot-pose = Goal

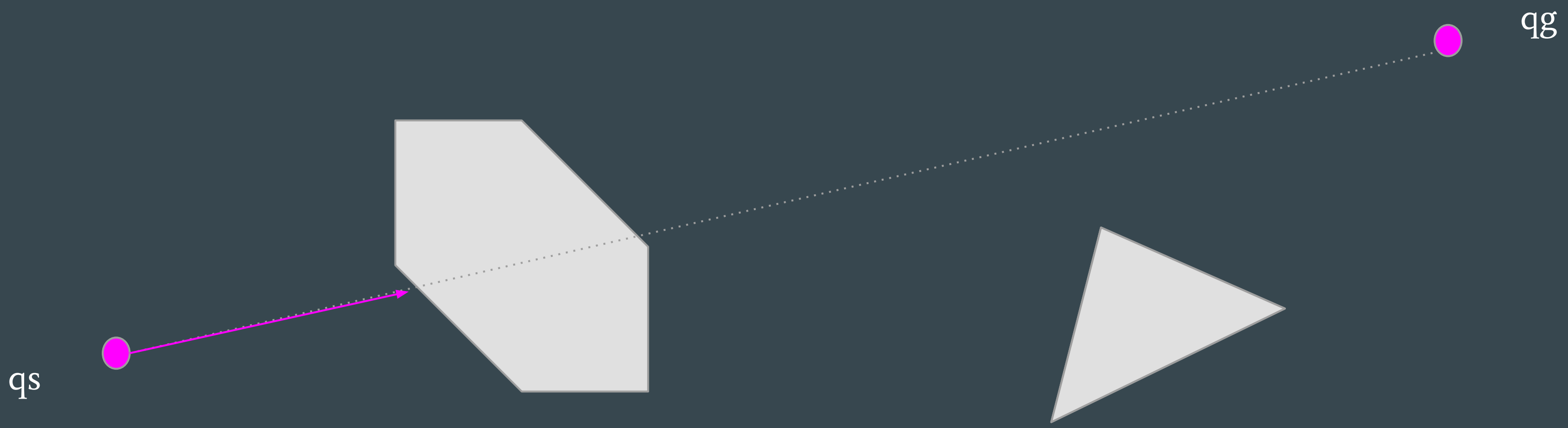
Bug Algorithm 1



Repeat until Robot-pose = Goal

Head towards goal

Path Planning Simplified: Bug Algorithm 1



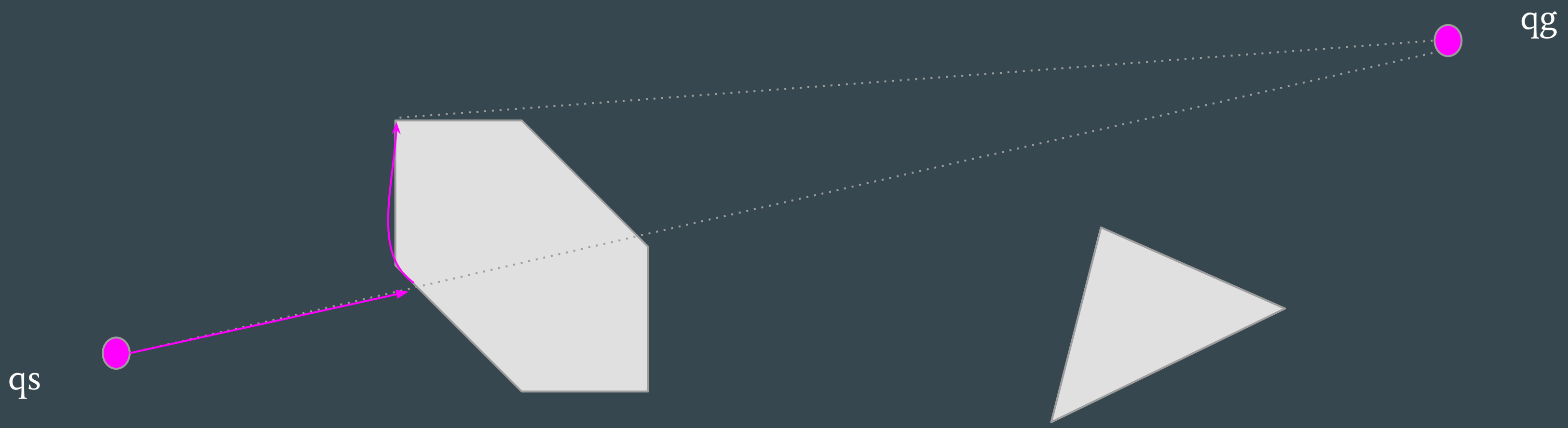
Repeat until Robot-pose = Goal

Head towards goal

If obstacle detected then

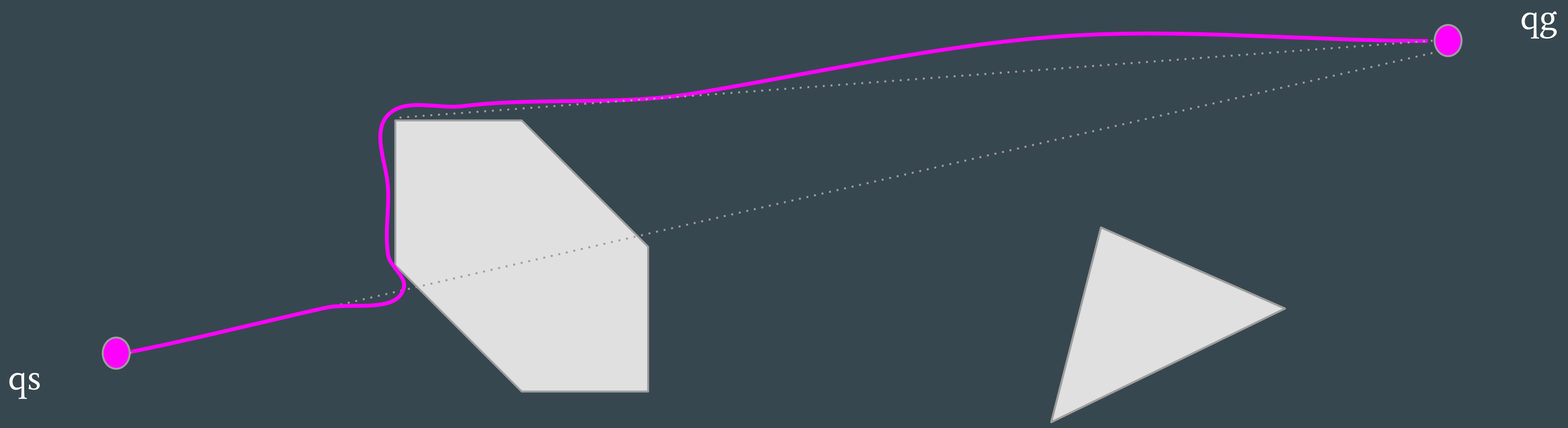
Navigate next to wall to the left until heading towards goal is possible

Bug Algorithm 1



Repeat until Robot-pose = Goal
- Head towards goal
- If obstacle detected then
  - Navigate next to wall to the left until heading towards goal is possible

Bug Algorithm 1



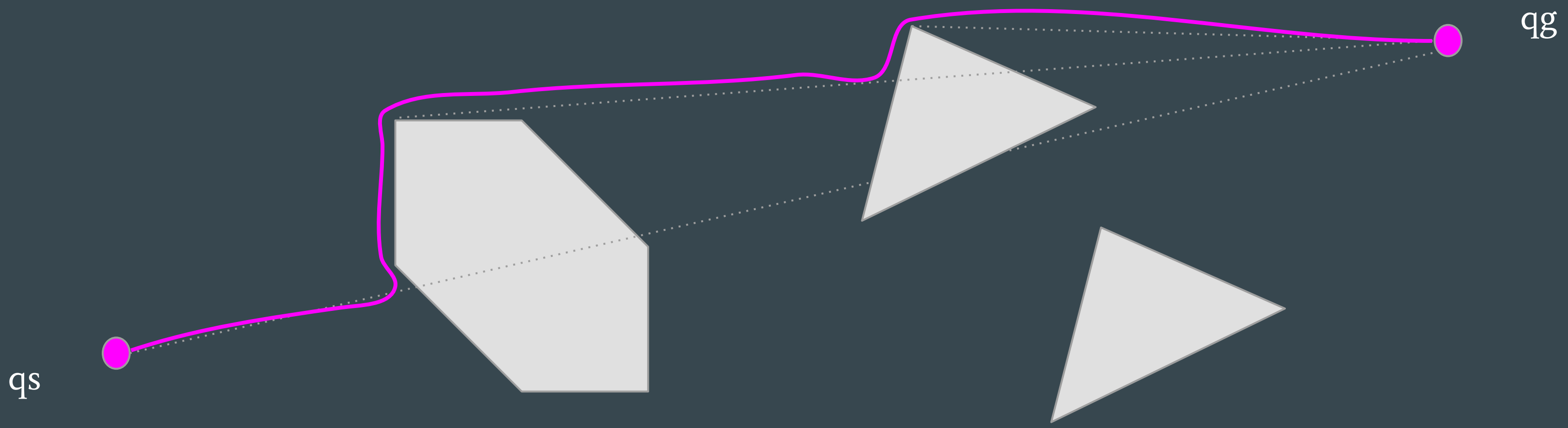
Repeat until Robot-pose = Goal

Head towards goal

If obstacle detected then

Navigate next to wall to the left until heading towards goal is possible

Bug Algorithm 1



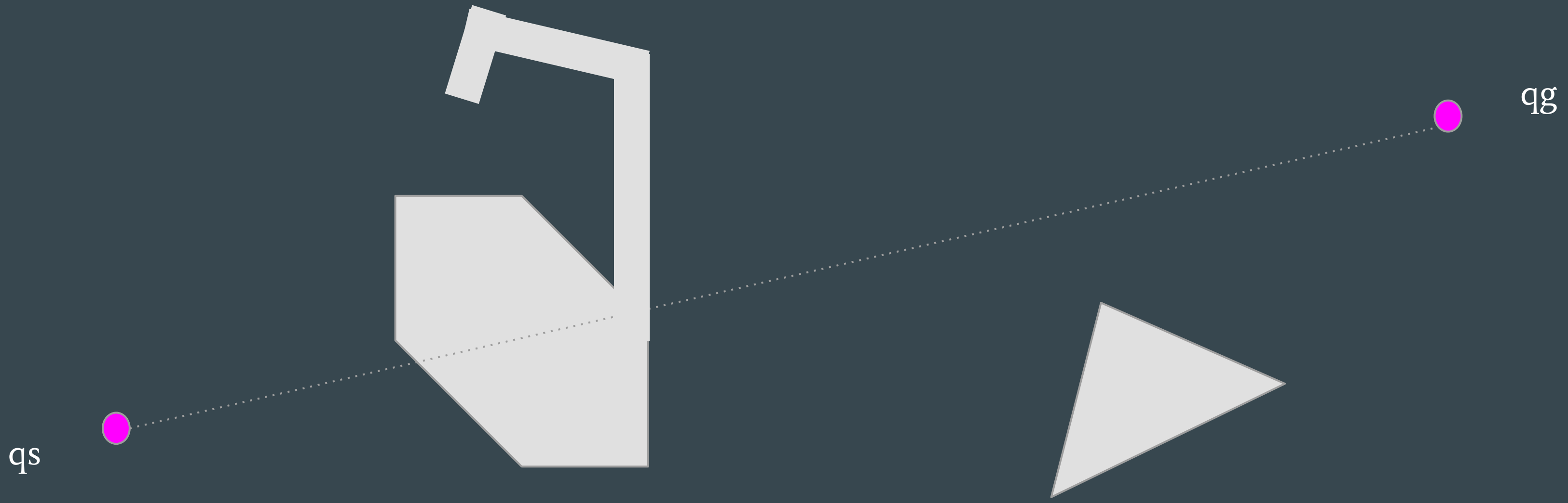
Repeat until Robot-pose = Goal

  Head towards goal

  If obstacle detected then

    Navigate next to wall to the left until heading towards goal is possible

Bug Algorithm 1



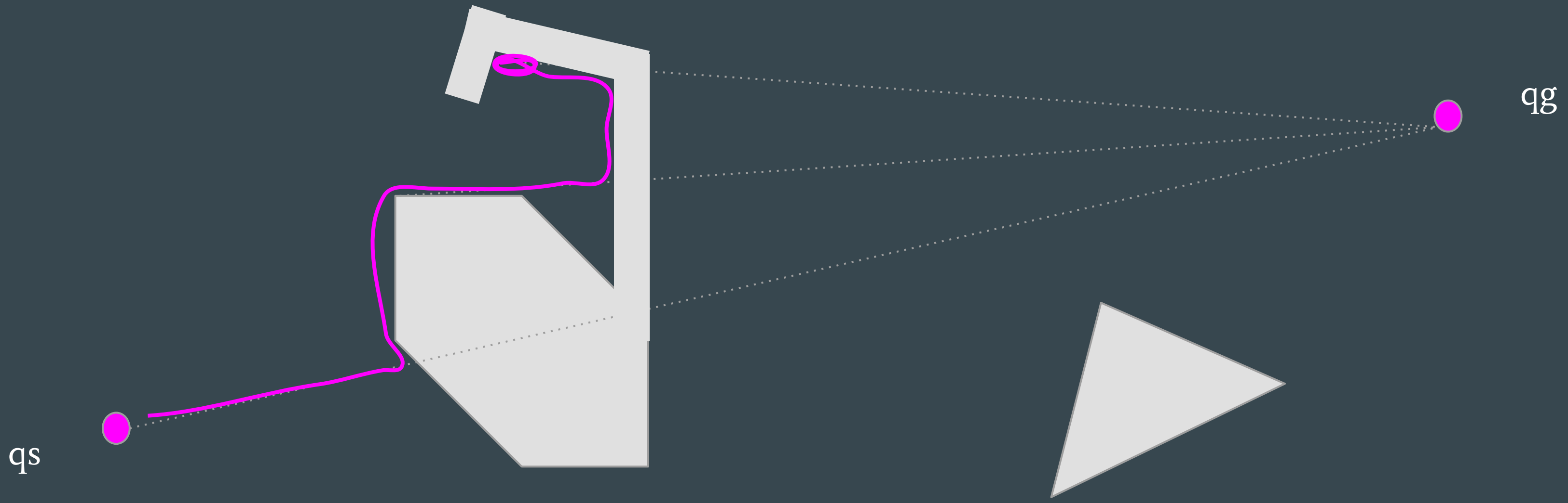
Repeat until Robot-pose = Goal

Head towards goal

If obstacle detected then

Navigate next to wall to the left until heading towards goal is possible

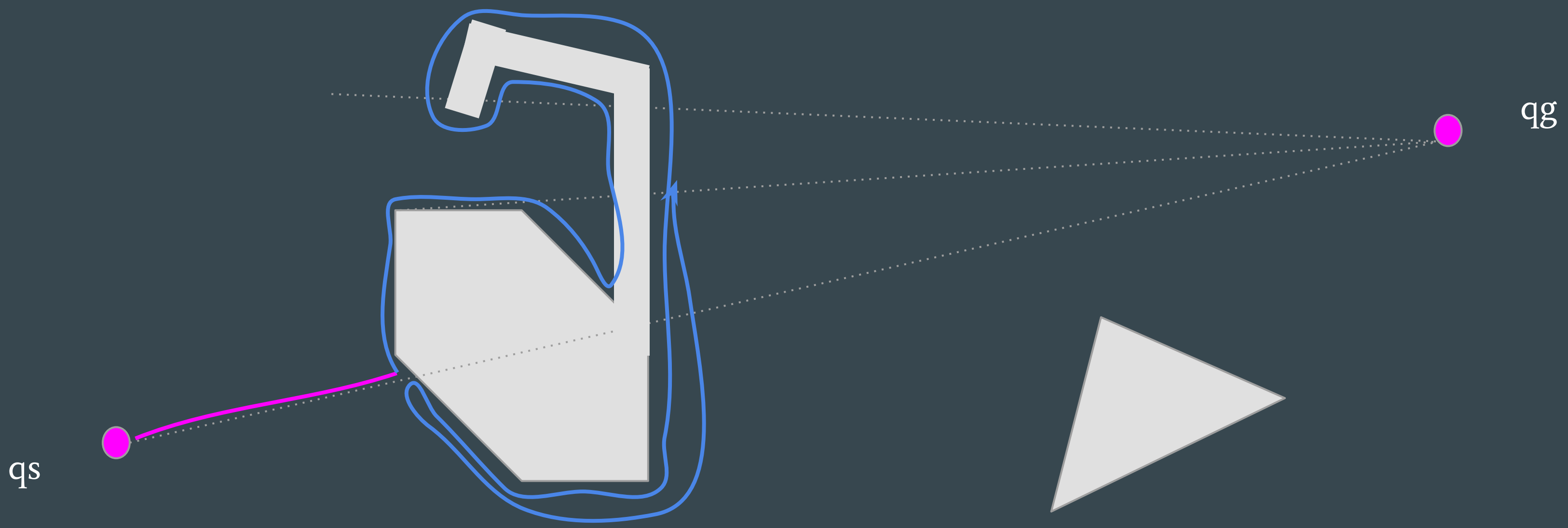
Bug Algorithm 1



Repeat until Robot-pose = Goal

- Head towards goal
- If obstacle detected then
  - Navigate next to wall to the left until heading towards goal is possible

Bug Algorithm 1+



Repeat until Robot-pose = Goal

Head towards goal

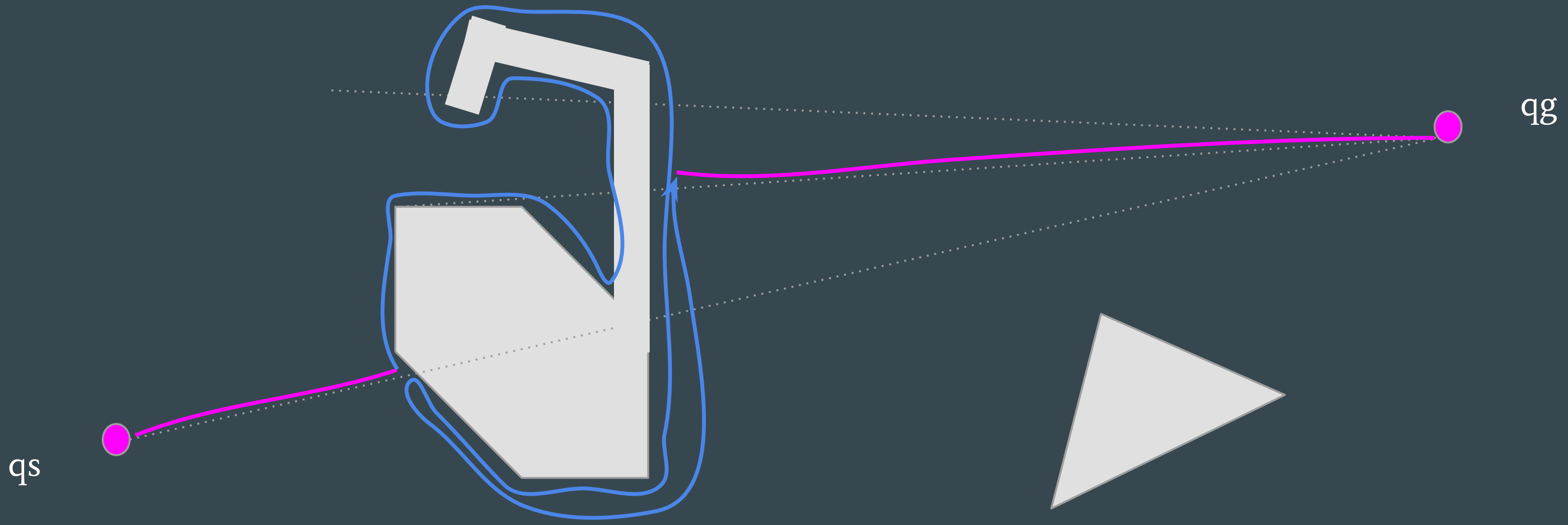
If obstacle detected then

Navigate next to wall completely

Identify closest boundary point to Goal

Return to this point by shortest path along obstacle boundary

Bug Algorithm 1+



Repeat until Robot-pose = Goal

Head towards goal

If obstacle detected then

Navigate next to wall completely

Identify closest boundary point to Goal

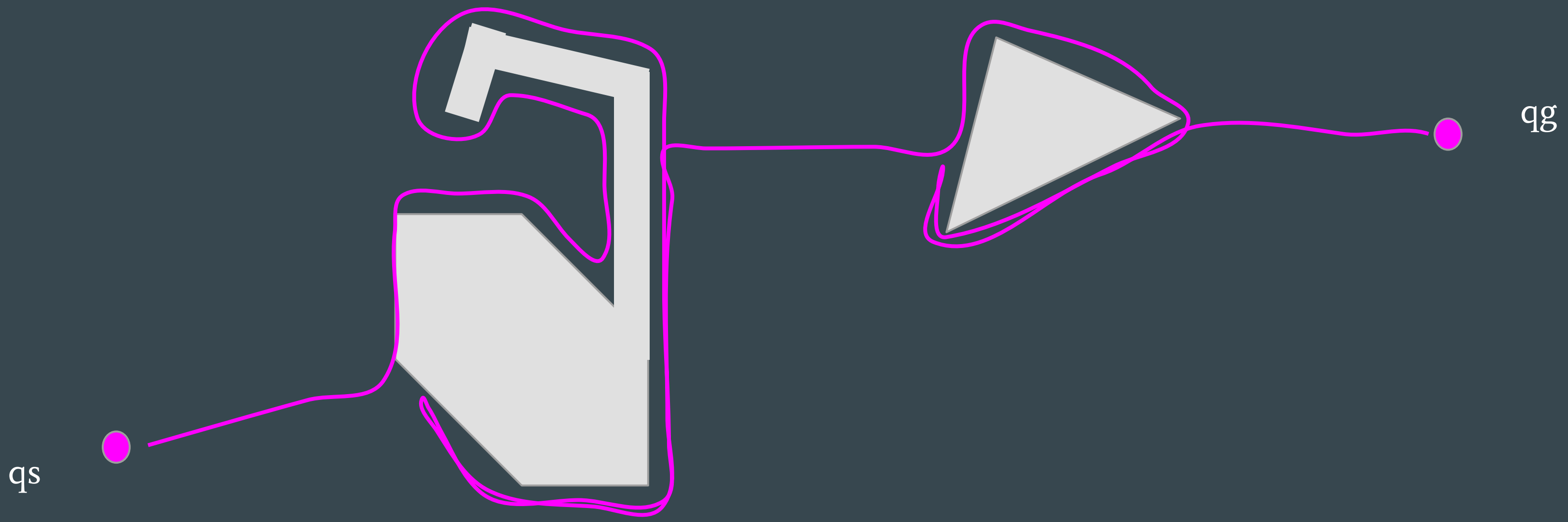
Return to this point by shortest path along obstacle boundary

Bug Algorithm 1+ Exercise

Repeat until Robot-pose = Goal

- Head towards goal
- If obstacle detected then
  - Navigate next to wall completely
  - Identify closest boundary point to Goal
  - Return to this point by shortest path along obstacle boundary

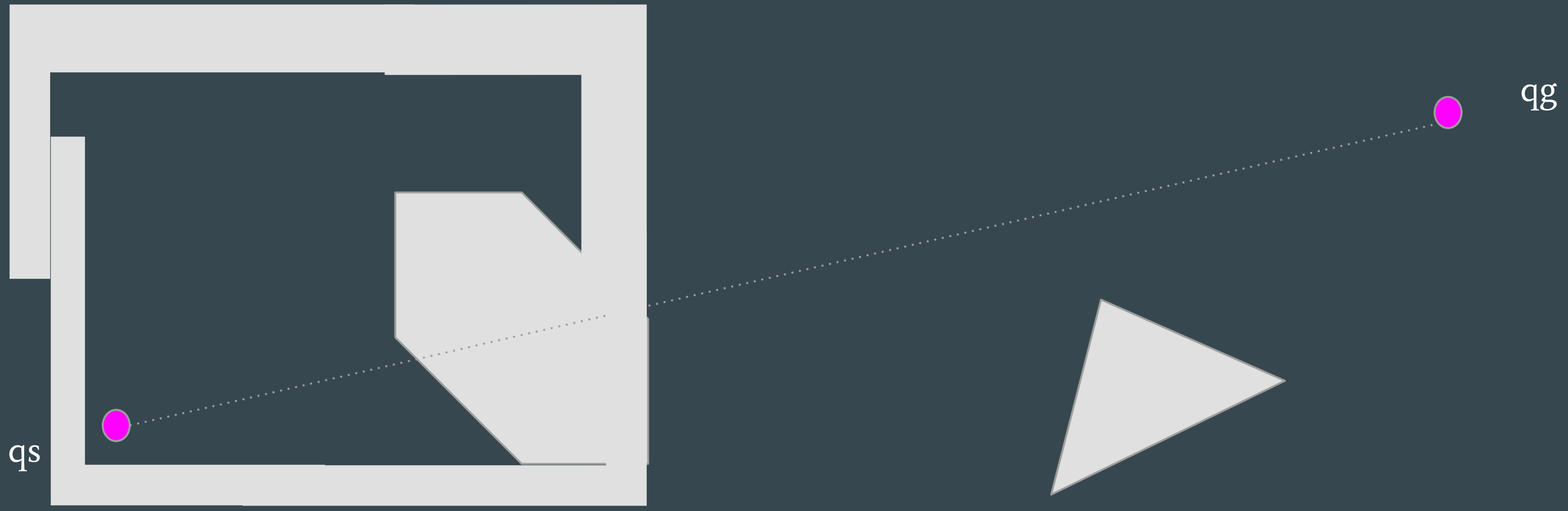
Bug Algorithm 1+ Exercise



Repeat until Robot-pose = Goal

- Head towards goal
- If obstacle detected then
  - Navigate next to wall completely
  - Identify closest boundary point to Goal
  - Return to this point by shortest path along obstacle boundary

Bug Algorithms 1++



Repeat until Robot-pose = Goal

Head towards goal

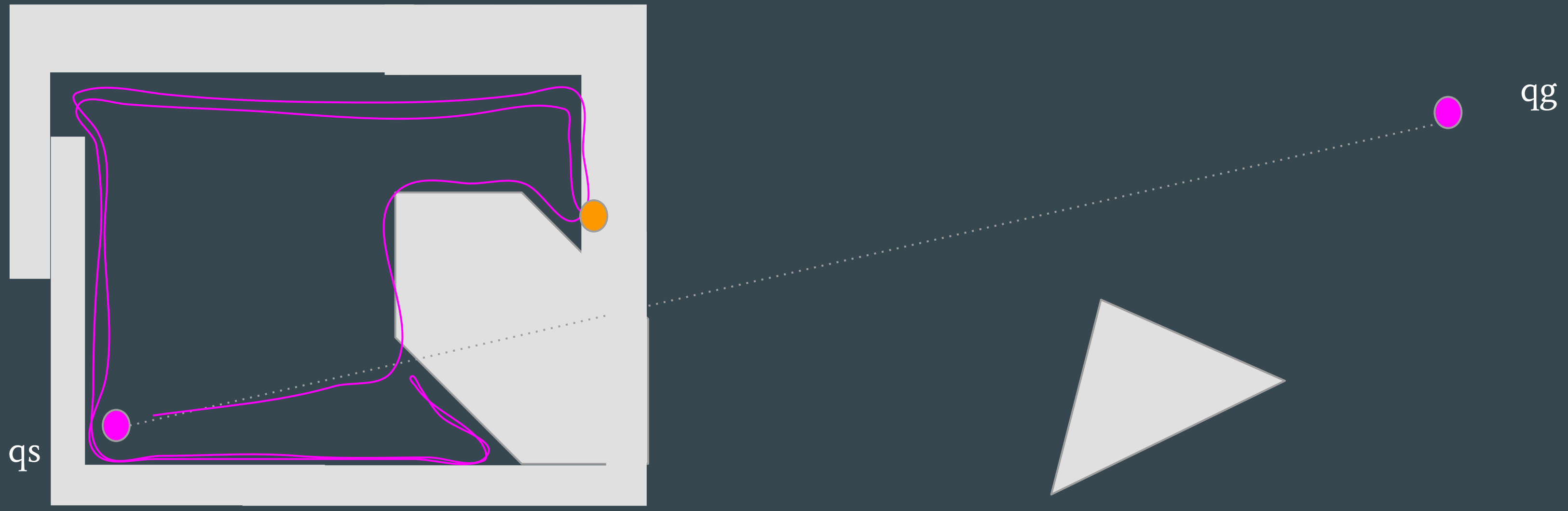
If obstacle detected then

Navigate next to wall completely

Identify closest boundary point to Goal, if direction towards the Goal hits obstacle break

Return to this point by shortest path along obstacle boundary

Bug Algorithms 1++



Repeat until Robot-pose = Goal

Head towards goal

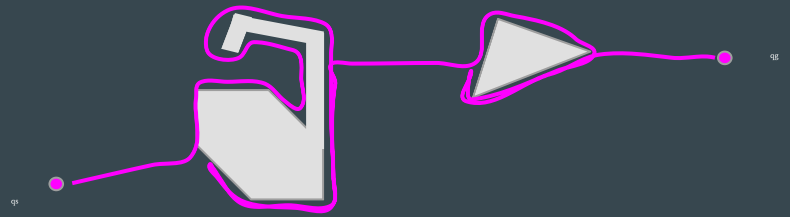
If obstacle detected then

Navigate next to wall completely

Identify closest boundary point to Goal, if direction towards the Goal hits obstacle break

Return to this point by shortest path along obstacle boundary

Bug Algorithm 1++



Repeat until Robot-pose = Goal

Head towards goal

If obstacle detected then

Navigate next to wall completely

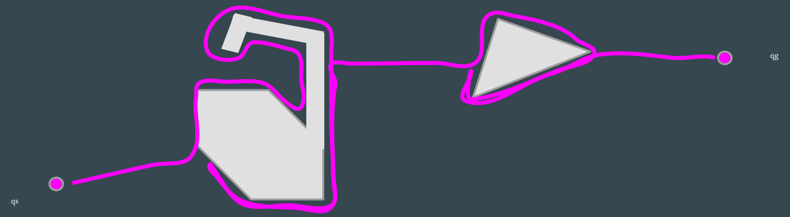
Identify closest boundary point to Goal, if direction towards the Goal hits obstacle break

Return to this point by shortest path along obstacle boundary

Distance **T** traveled by Bug-1 (based on D distance between qs and qg)

- Lower bound:
- Upper bound:
- Average:

Bug Algorithm 1++



Repeat until Robot-pose = Goal

Head towards goal

If obstacle detected then

Navigate next to wall completely

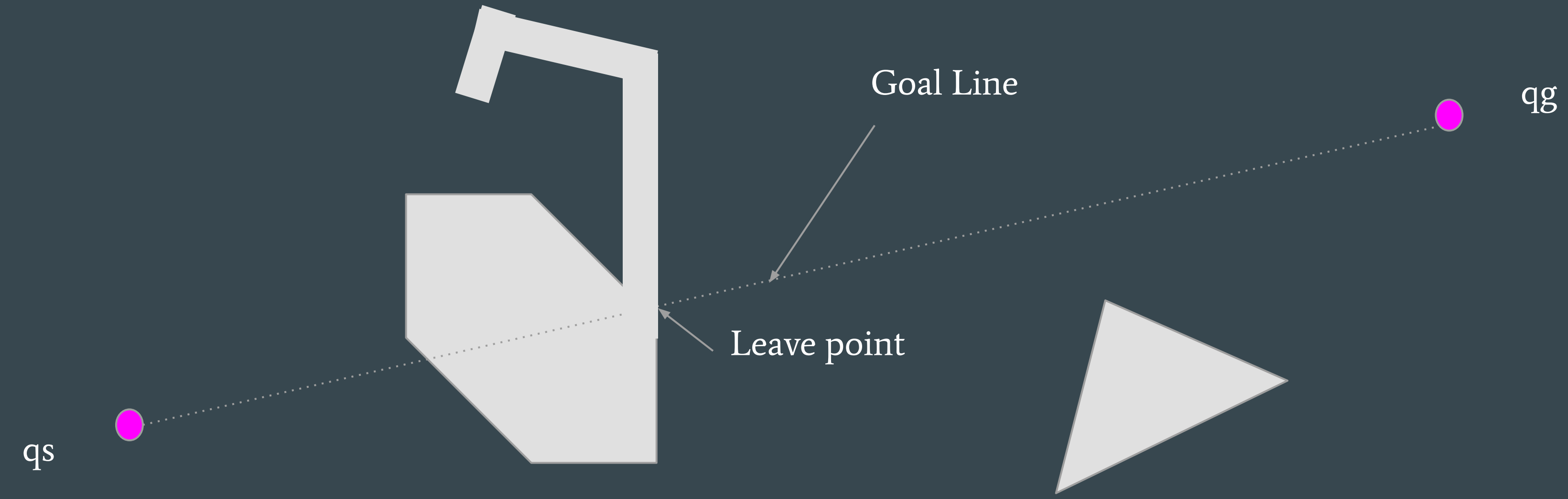
Identify closest boundary point to Goal, if direction towards the Goal hits obstacle break

Return to this point by shortest path along obstacle boundary

Distance T traveled by Bug-1 (based on D distance between qs and qg)

- Lower bound: $T >= D$
- Upper Bound: \inf
- Average: $T <= D + 1.5 \sum(\text{perimeter polygons})$

Bug Algorithm 2



Repeat until Robot-pose = Goal

Head towards goal

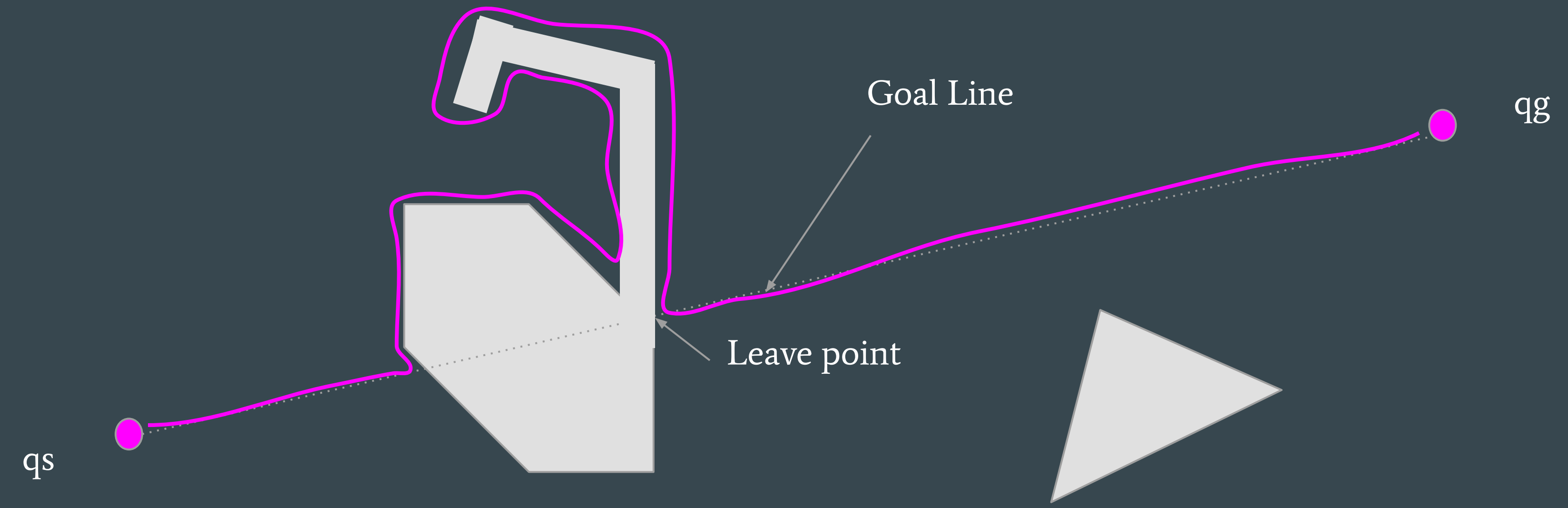
If obstacle detected then

Repeat

Navigate next to wall (start left)

Until Goal Line crossed at Leave point closer to goal on the same side than before

Bug Algorithm 2



Repeat until Robot-pose = Goal

Head towards goal

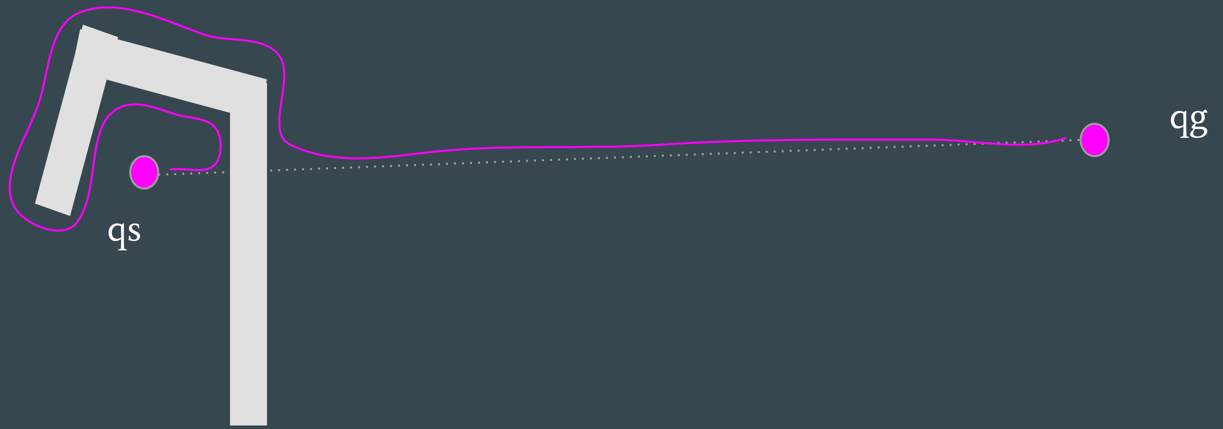
If obstacle detected then

Repeat

Navigate next to wall (start left)

Until Goal Line crossed at Leave point closer to goal on the same side than before

Bug Algorithm 2



Repeat until Robot-pose = Goal

  Head towards goal

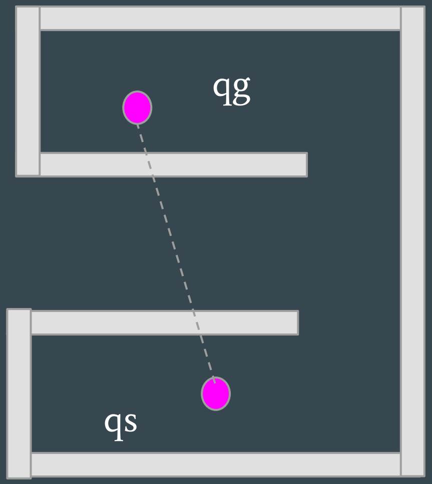
  If obstacle detected then

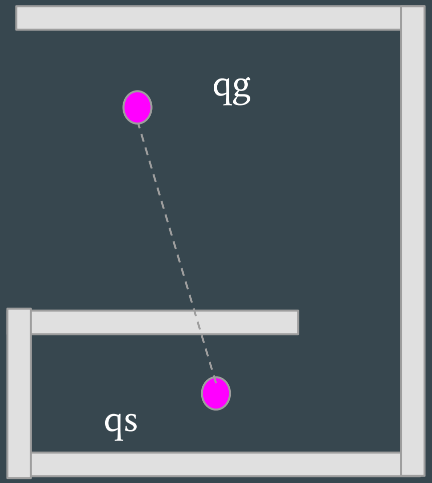
    Repeat

      Navigate next to wall (start left)

    Until Goal Line crossed at Leave point closer to goal on the same side than before

Bug Algorithm 2 Exercise





Repeat until Robot-pose = Goal

Head towards goal

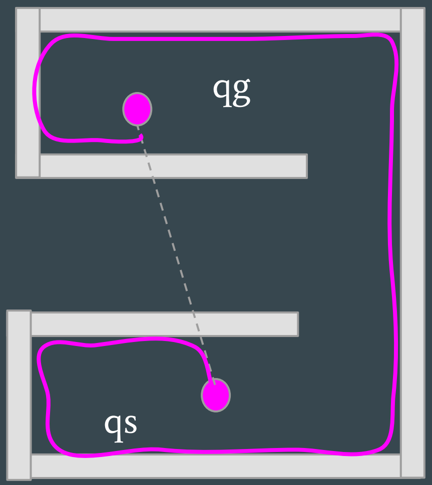
If obstacle detected then

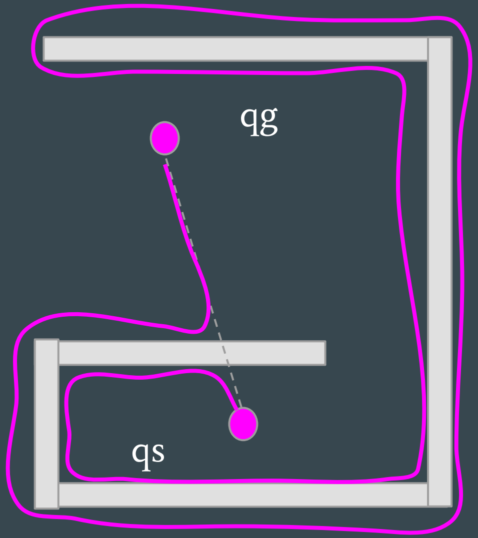
Repeat

Navigate next to wall (start left)

Until Goal Line crossed at Leave point closer to goal on the same side than before

Path Planning Simplified: Bug Algorithm 2 Exercise





Repeat until Robot-pose = Goal

Head towards goal

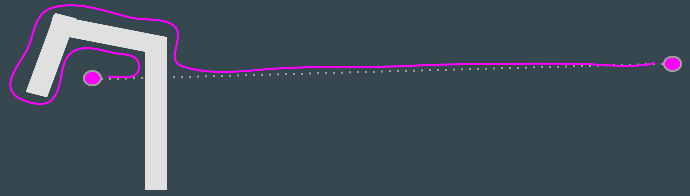
If obstacle detected then

Repeat

Navigate next to wall (start left)

Until Goal Line crossed at Leave point closer to goal on the same side than before

Bug Algorithm 2



Repeat until Robot-pose = Goal

    Head towards goal

    If obstacle detected then

        Repeat

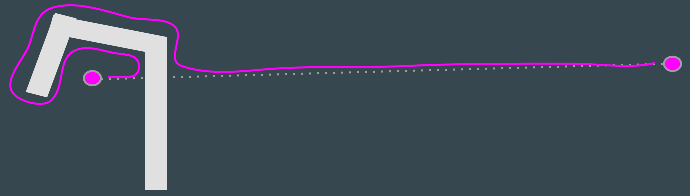
            Navigate next to wall (start left)

        Until Goal Line crossed at Leave point closer to goal on the same side than before

Distance T traveled by Bug-2 (based on D distance between qs and qg)

- Lower bound:
- Upper Bound:
- Average:

Bug Algorithm 2



Repeat until Robot-pose = Goal
    Head towards goal
    If obstacle detected then
        Repeat
            Navigate next to wall (start left)
        Until Goal Line crossed at Leave point closer to goal on the same side than before

Distance T traveled by Bug-2 (based on D distance between qs and qg)

- Lower bound: *T >= D*
- Upper bound: \inf
- Average: *T <= D + 0.5 ∑(Perimeters of obstacles intersected by goal line * number of times lines intersects each obstacle)*

Bug Algorithm 2 Exercise

Repeat until Robot-pose = Goal

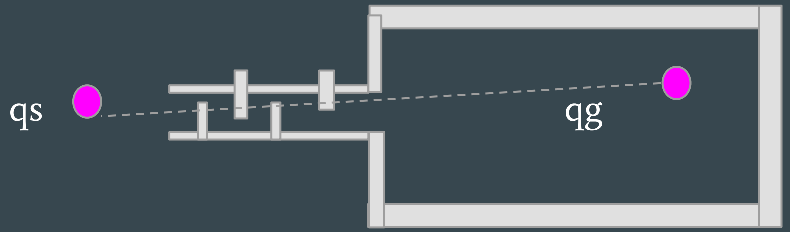
Head towards goal

If obstacle detected then

Repeat

Navigate next to wall (start left)

Until Goal Line crossed at Leave point closer to goal on the same side than before



Bug Algorithm 2 Exercise

Repeat until Robot-pose = Goal

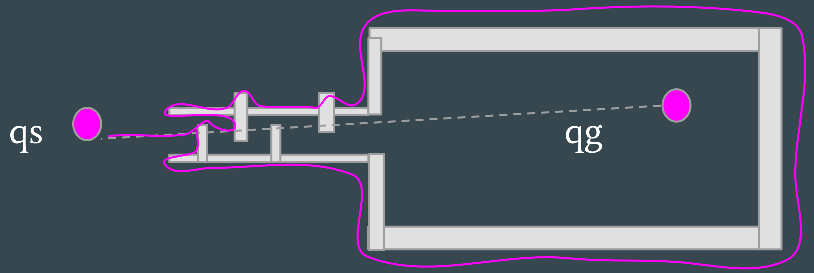
  Head towards goal

  If obstacle detected then

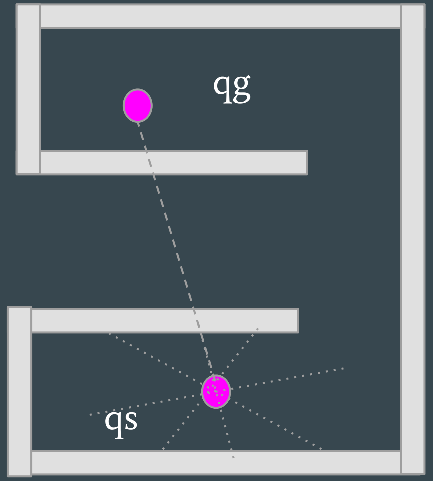
    Repeat

      Navigate next to wall (to the left)

    Until Goal Line crossed at Leave point closer to goal on the same side than before



Relaxing Bug Algorithm assumptions



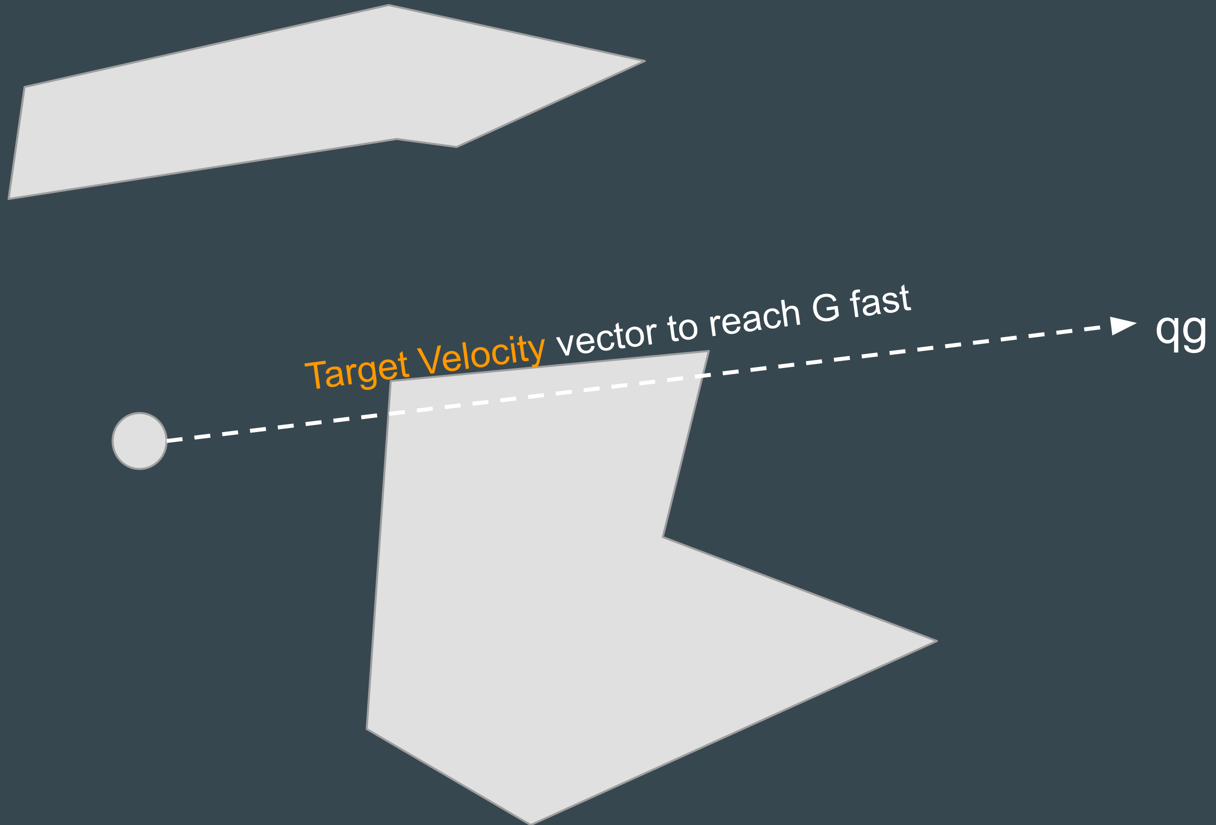
Robot

- Is modeled as a bounded point
- Can sense its location precisely
- Can sense contact with obstacles - can sense more…
- Can compute direction towards goal and distance between two points
- Does not know location of obstacles, has more memory

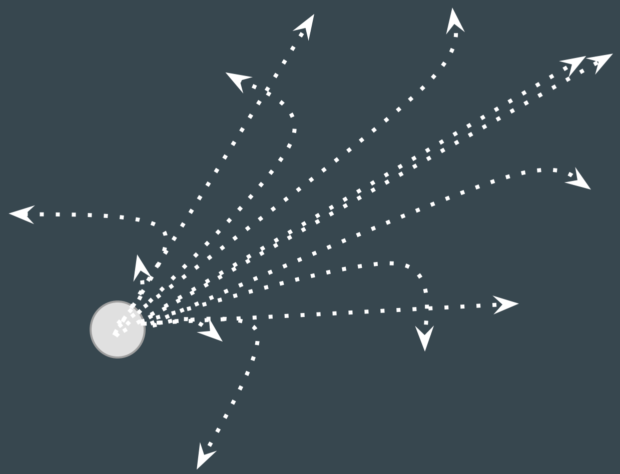
Motion Problem

- Reactive
  - Bug
  - Dynamic windows
- Model-based

Dynamic Windows



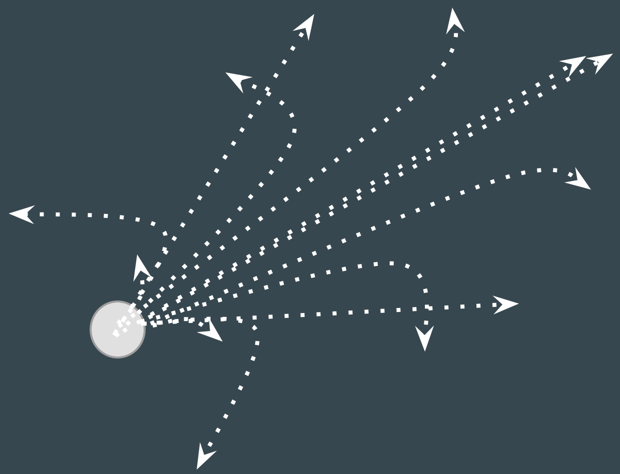
Dynamic Windows



For each time slice t

Enumerate allowed velocities in

Dynamic Windows



For each time slice *t*

Enumerate **allowed** velocities in

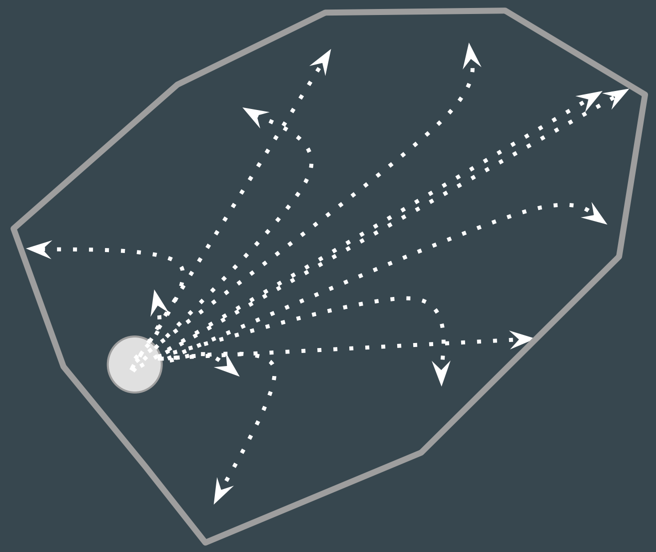
Robot
Physical Model

Racing drone

Values of the high power drone (Gemo-Copter):

- climb rate: over 40 m/s or 140 km/h
- acceleration from 0 to 100 km/h (vertical): far below 2 s
- duration to reach 100 meters above the ground from the hover: 2,8 s
- maximum acceleration: 3,6 g or 35 m/s2
- take off weight: 923 g
- maximum power: 2.250 Watt
- power to weight ratio: 2438 W/kg

Dynamic Windows



For each time slice ***t***

For each ***v*** in [curr.v - maxacc(t), curr.v + maxacc(t)]

If (v < maxV and v > minV)

validVelocities.add(***v***)

For each ***ω*** in [curr.ω - maxacc(t), curr.ω + maxacc.(t)]

If (ω < maxω and ω > minω)

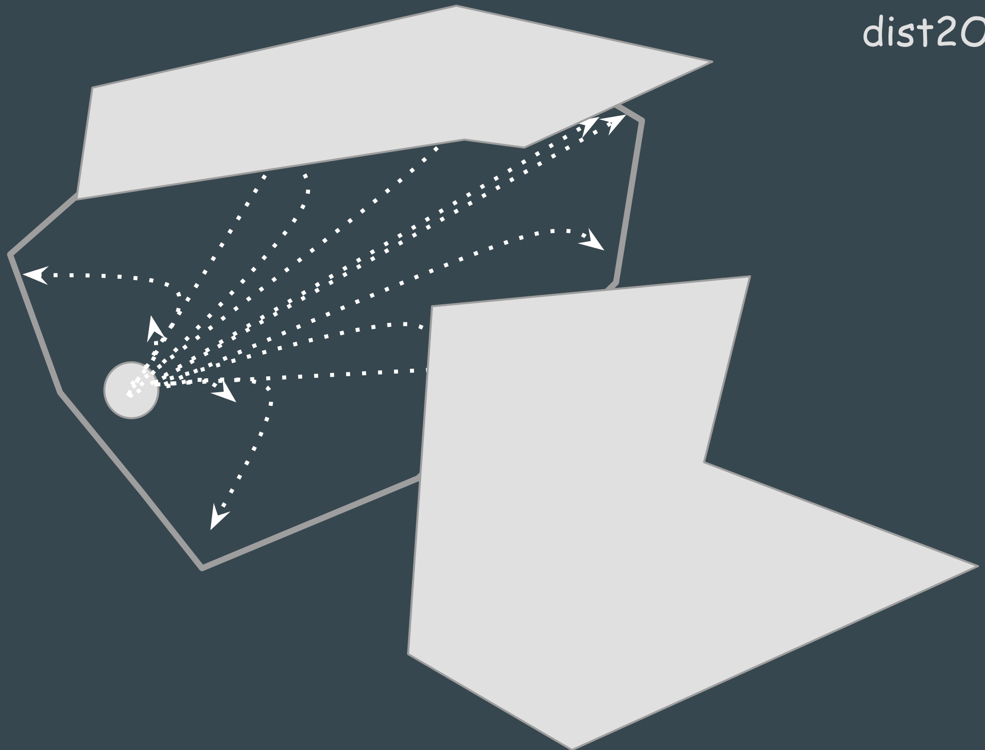
validAngVelocities.add(ω)

Dynamic Windows

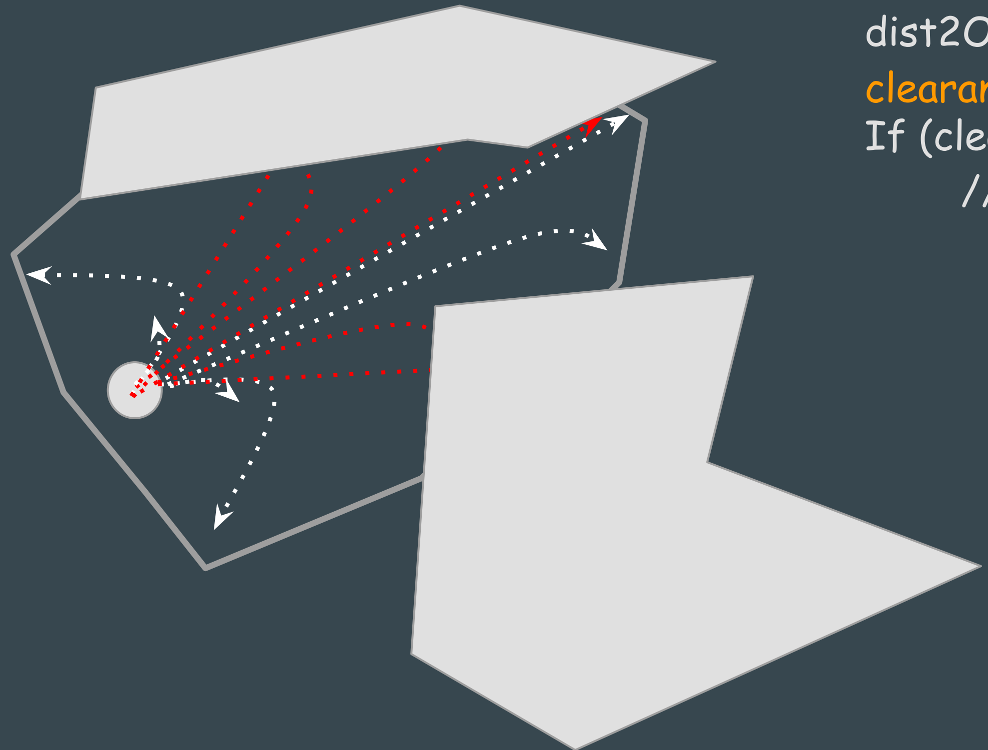
For each *v* in validVelocities

    For each *ω* in validAngVelocities

        dist2Obstacle = computeDist(v, ω, laserScan())



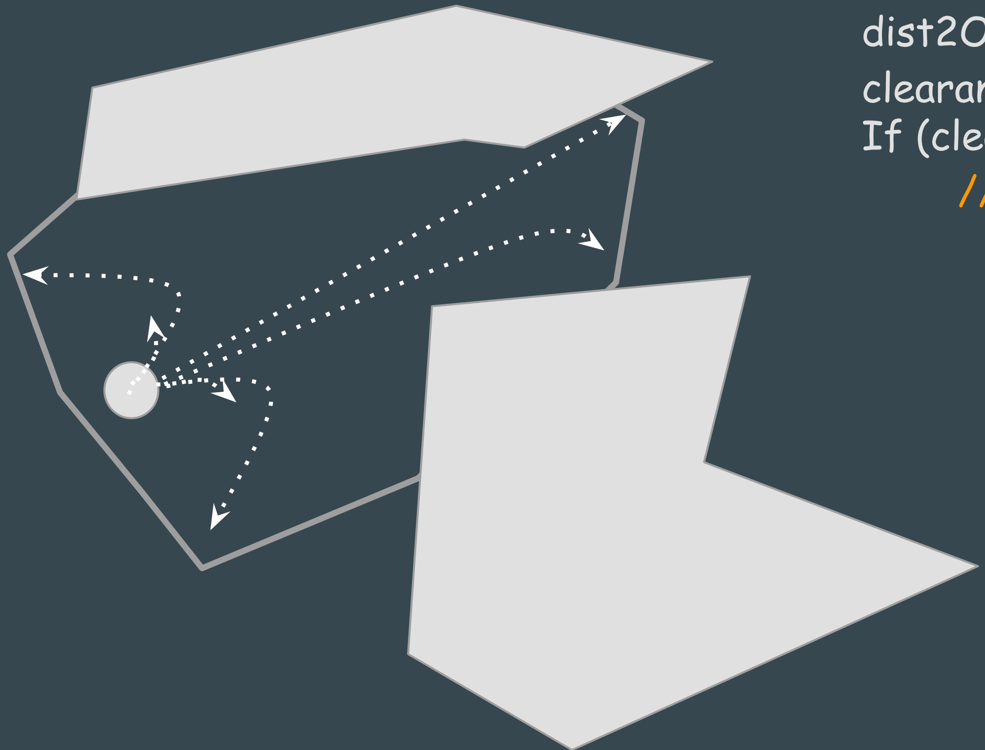
Dynamic Windows



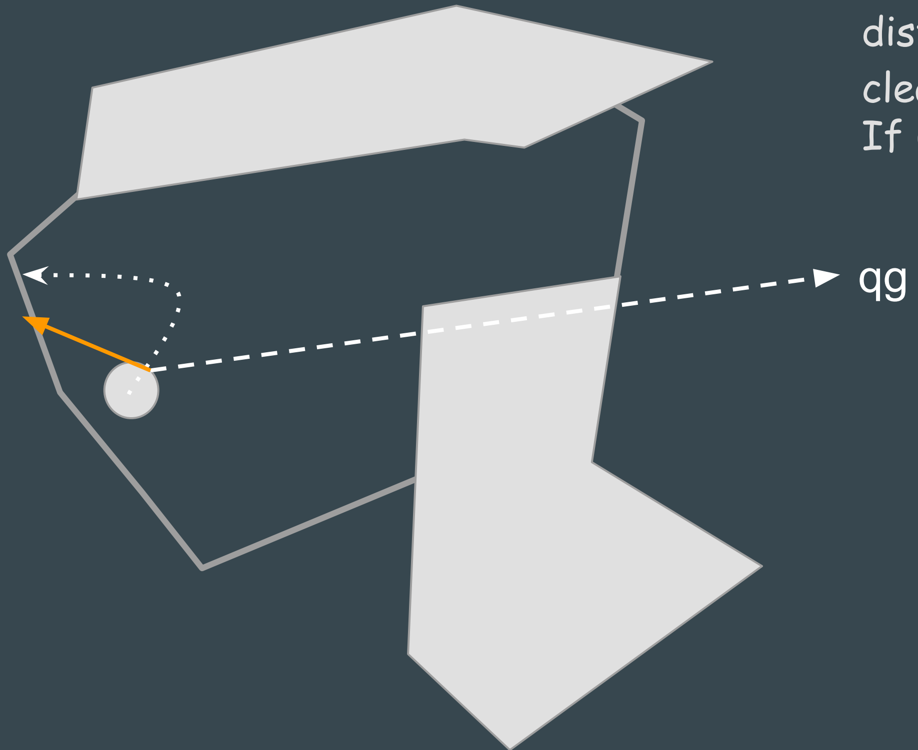
```
For each v in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v,ω)
        If (clearance > 0)
            // non-colliding velocities
```

Dynamic Windows

```
For each v in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v,ω)
        If (clearance > 0)
            // non-colliding velocities
```

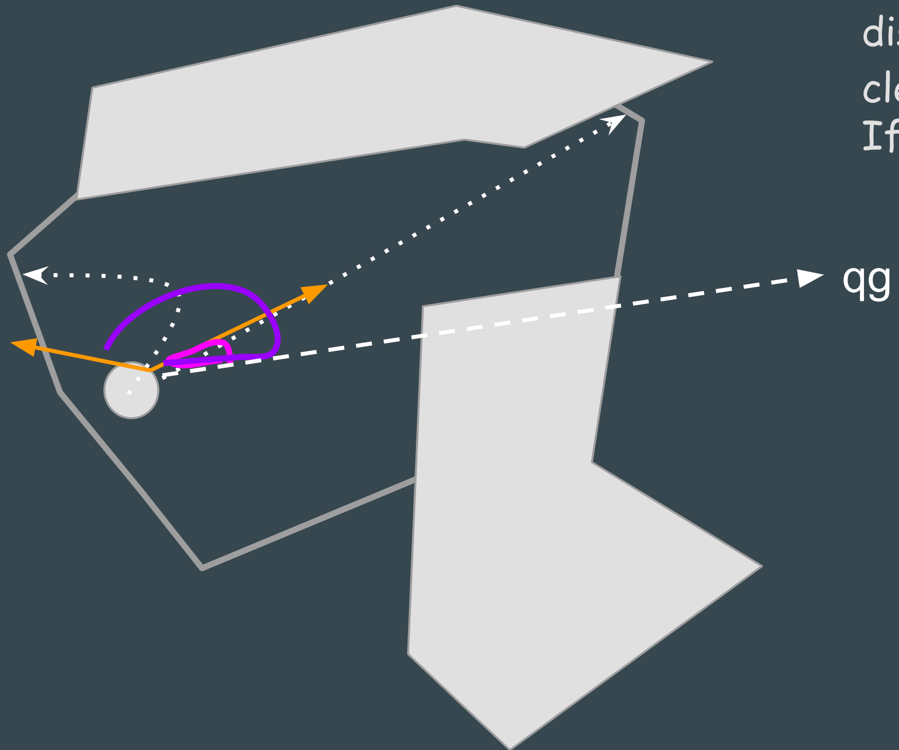


Dynamic Windows



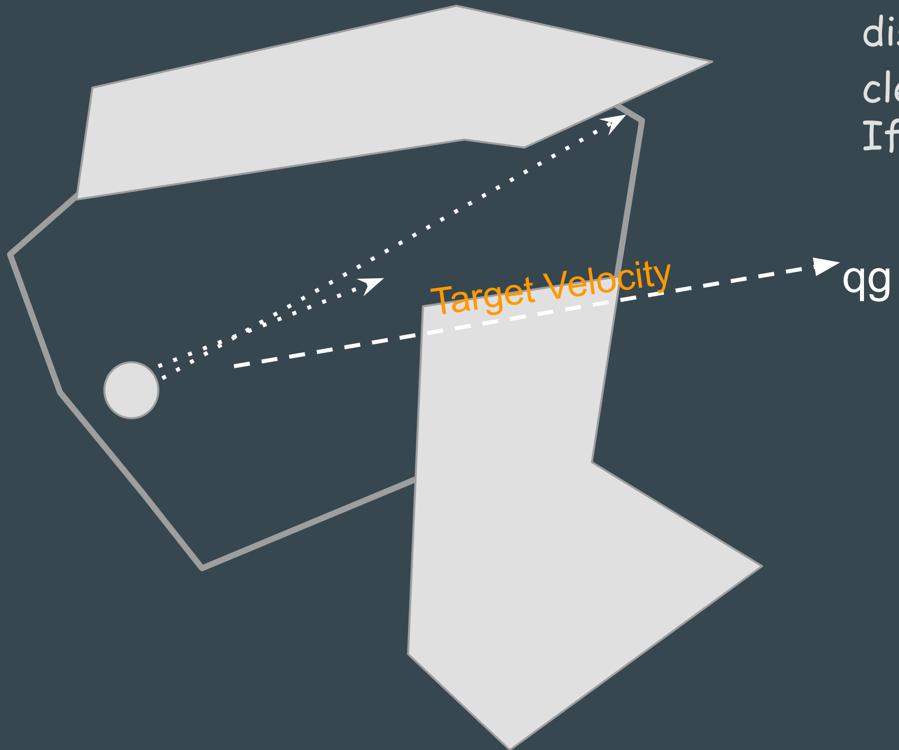
For each *v* in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v)
        If (clearance > 0)
            // non-colliding velocities
            offHeading = headingDiff(robot.pose, qg, v, **ω**)

Dynamic Windows




```
For each v in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v)
        If (clearance > 0)
            // non-colliding velocities
            offHeading = headingDiff(robot.pose, qg, v, ω)
```

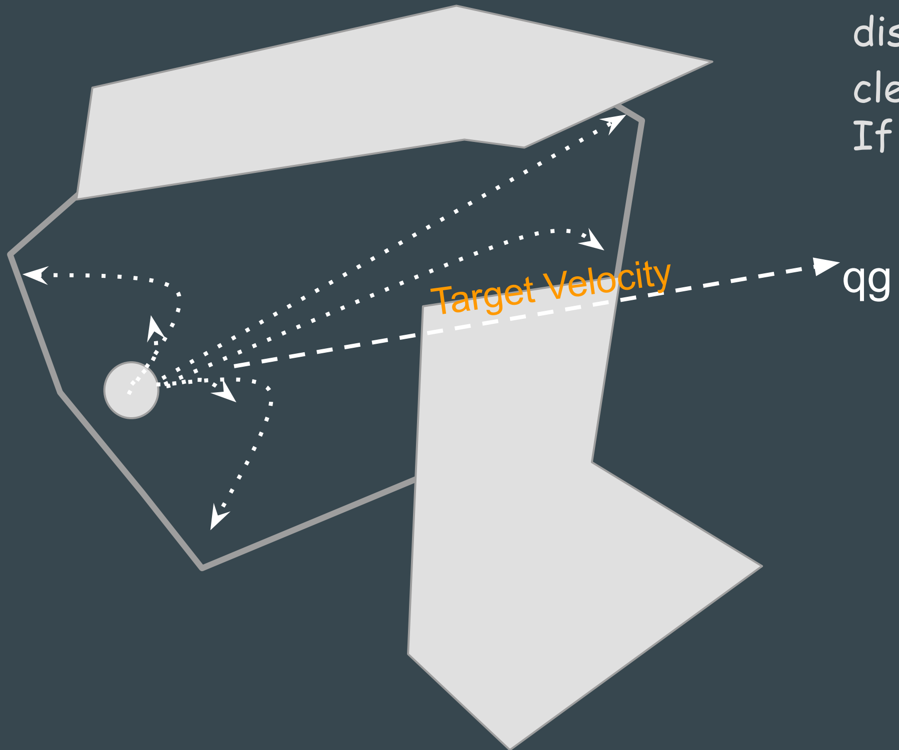
Dynamic Windows




```
For each v in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v)
        If (clearance > 0)
            // non-colliding velocities
            offHeading = headingDiff(robot.pose, qg, v, ω)

            offVel = abs(targetVelocity - v))
```

Dynamic Windows



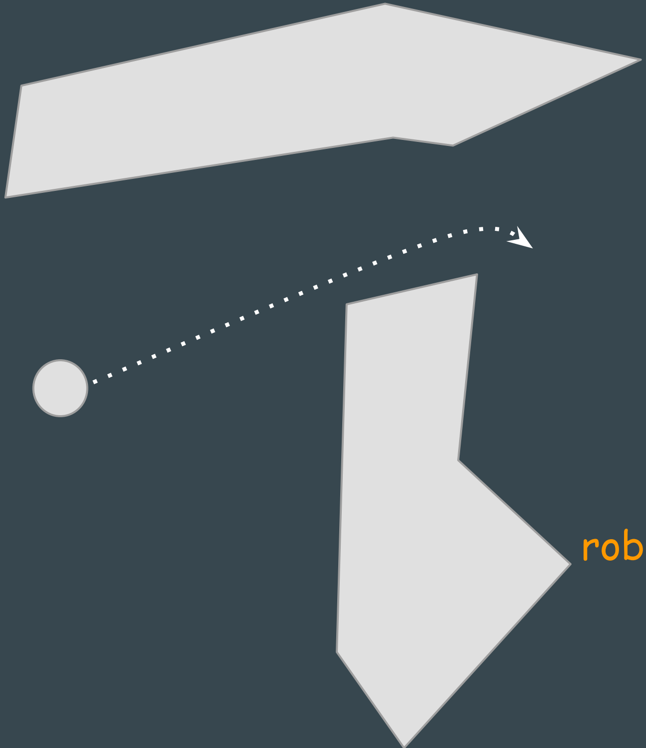
For each *v* in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v)
        If (clearance > 0)

            offHeading = headingDiff(robot.pose, qg, v, ω)

            offVel = abs(targetVelocity - v))

            output = ka*clearance + kb* offHeading + kc*offVel

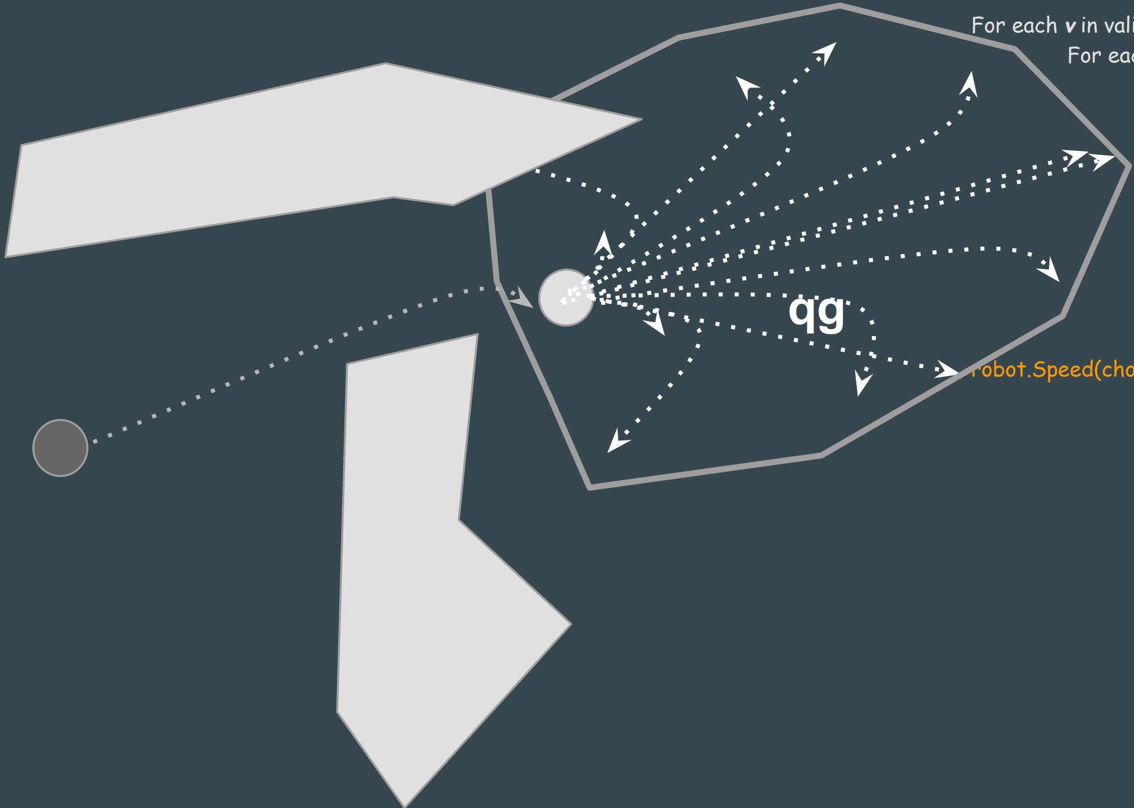
Dynamic Windows



```
For each v in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v)
        If (clearance > 0)
            offHeading = headingDiff(robot.pose, qg, v, ω)
            offVel = abs(targetVelocity - v))
            output = ka*clearance + kb* offHeading + kc*offVel
            if (output > chosen)
                chosenV = v
                chosenW = ω
                chosen = output
robot.Speed(chosenV, chosenW)
```

qg

Dynamic Windows




```
For each v in validVelocities
    For each ω in validAngVelocities
        dist2Obstacle = computeDist(v, ω, laserScan())
        clearance = dist2Obstacle - breakDist(v)
        If (clearance > 0)
            offHeading = headingDiff(robot.pose, qg, v, ω)
            offvel = abs(targetVelocity - v))
            output = ka*clearance + kb* offHeading + kc*offVel
            if (output > chosen)
                chosenV = v
                chosenW = ω
                chosen = output
robot.Speed(chosenV, chosenW)
```

Dynamic Windows

- Velocity planner (clearance, heading, velocity)
- Considers Robot's Dynamics for valid velocities

Motion Planning Families

- Reactive
- Model-based

Path Planning with Models

- Reactive
- Model-based
  - Predictive model of robot actions in **known** world
  - Build simplified representation
  - Search for solution in world representation

Path Planning: Visibility Methods

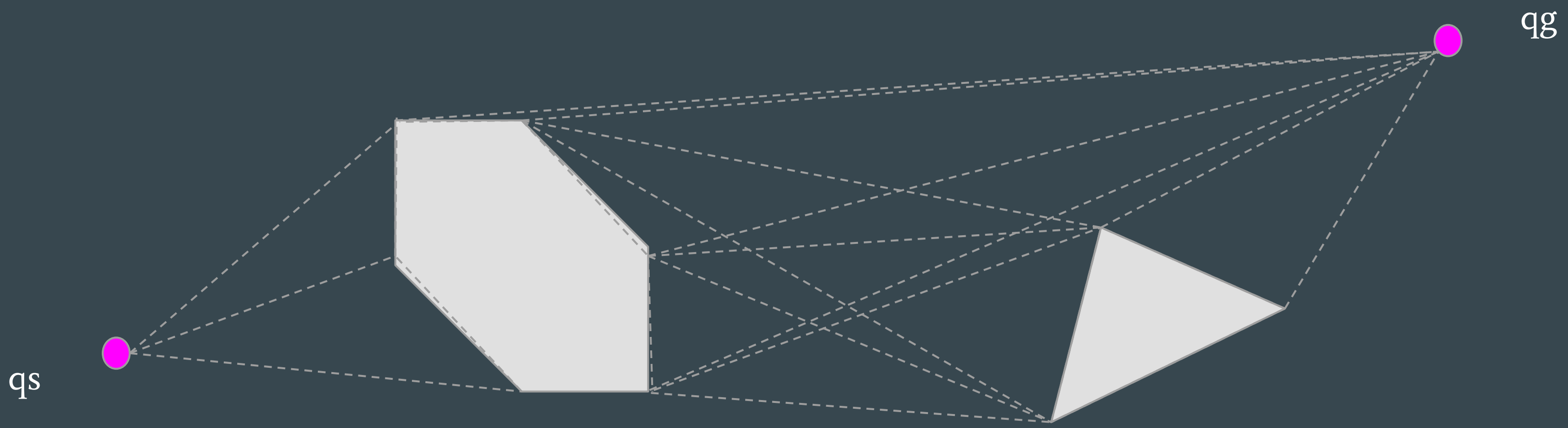
Assumptions

- Robot modeled as a bounded point
- Can sense its location precisely
- Can compute direction towards goal and distance between two points
- Knows location of obstacles in advanced - **polygonal obstacles**

Path Planning: Visibility Methods

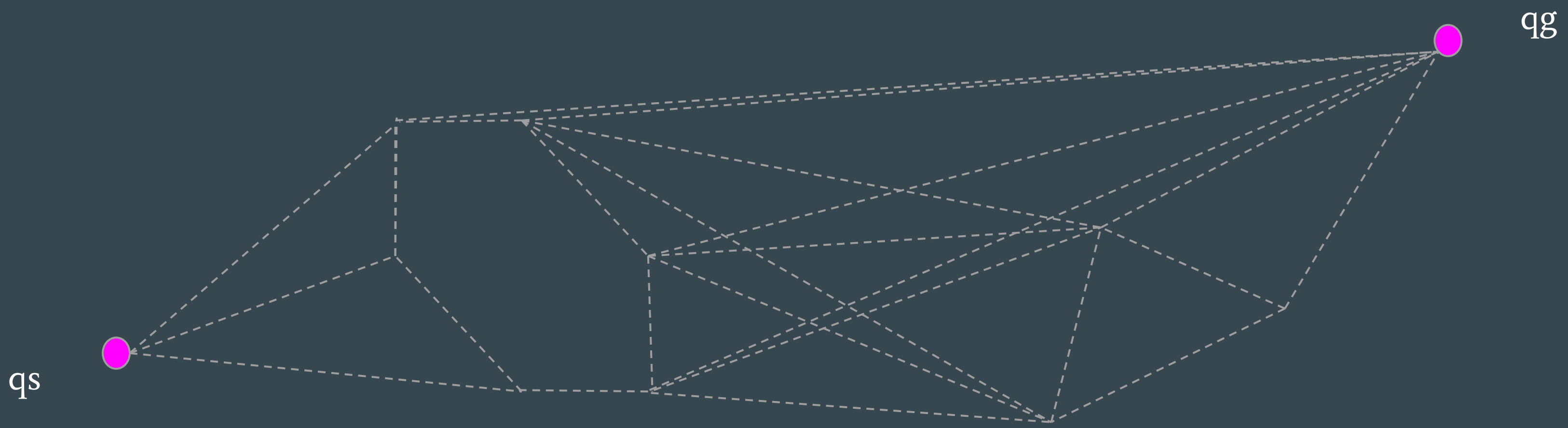
- Assumption: known polygonal obstacles

Path Planning: Visibility Methods



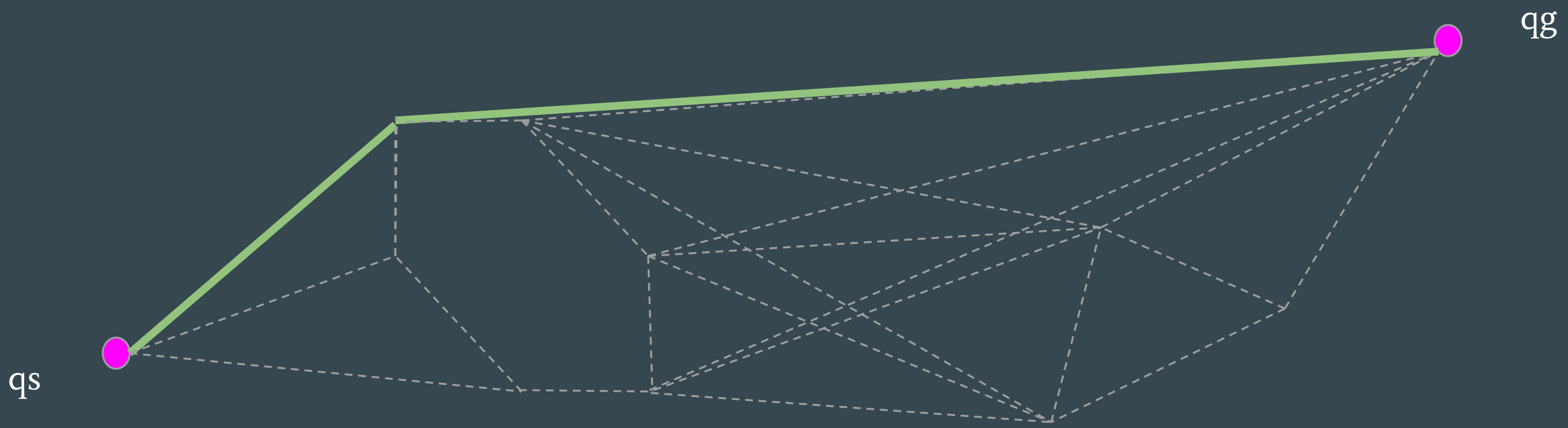
- Assumption: known polygonal obstacles
- Connect all vertices without obstacles in between

Path Planning: Visibility Methods



- Assumption: known polygonal obstacles
- Connect all vertices without obstacles in between
- Graph search!

Path Planning: Visibility Methods



- Assumption: known polygonal obstacles
- Connect all vertices without obstacles in between
- Graph search algorithm

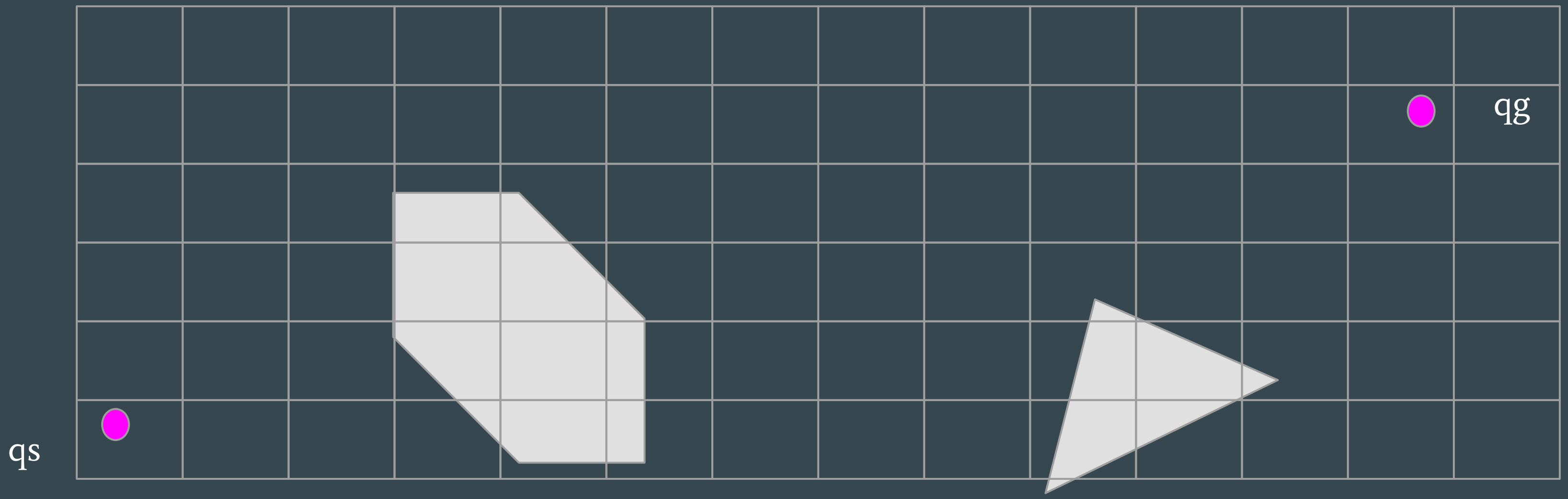
Path Planning: Visibility Methods

When does it struggle?

Path Planning with Models

- Reactive
- Model-based
  - Visibility
  - Grid

Path Planning: Grid Methods

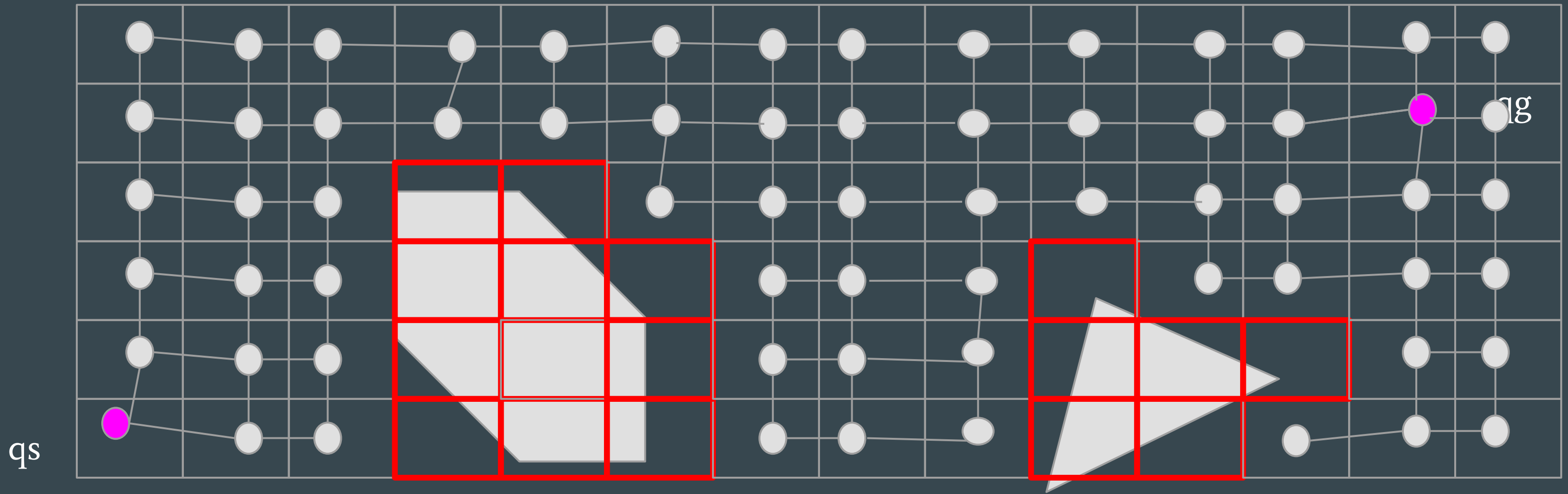


- Discretization of space - resolution

Path Planning: Grid Methods

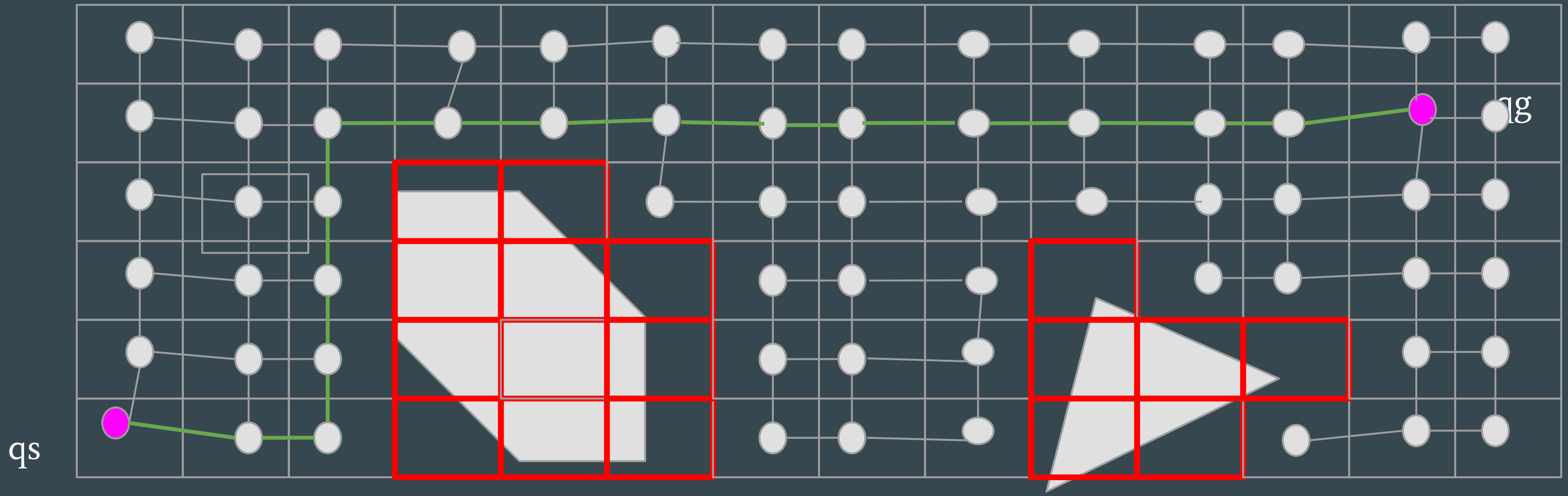
- Discretization of space
- Occupancy checker - probability

Path Planning: Grid Methods



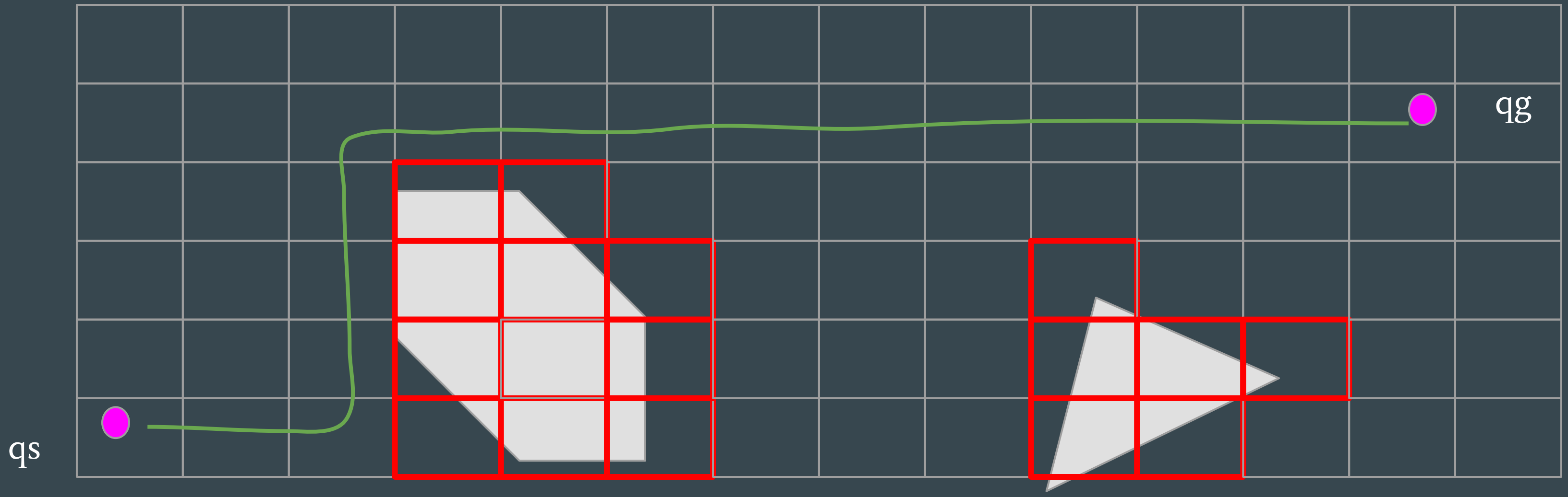
- Discretization of space
- Occupancy checker - probability

Path Planning: Grid Methods



- Discretization of space
- Occupancy checker
- Graph search algorithm on free cells

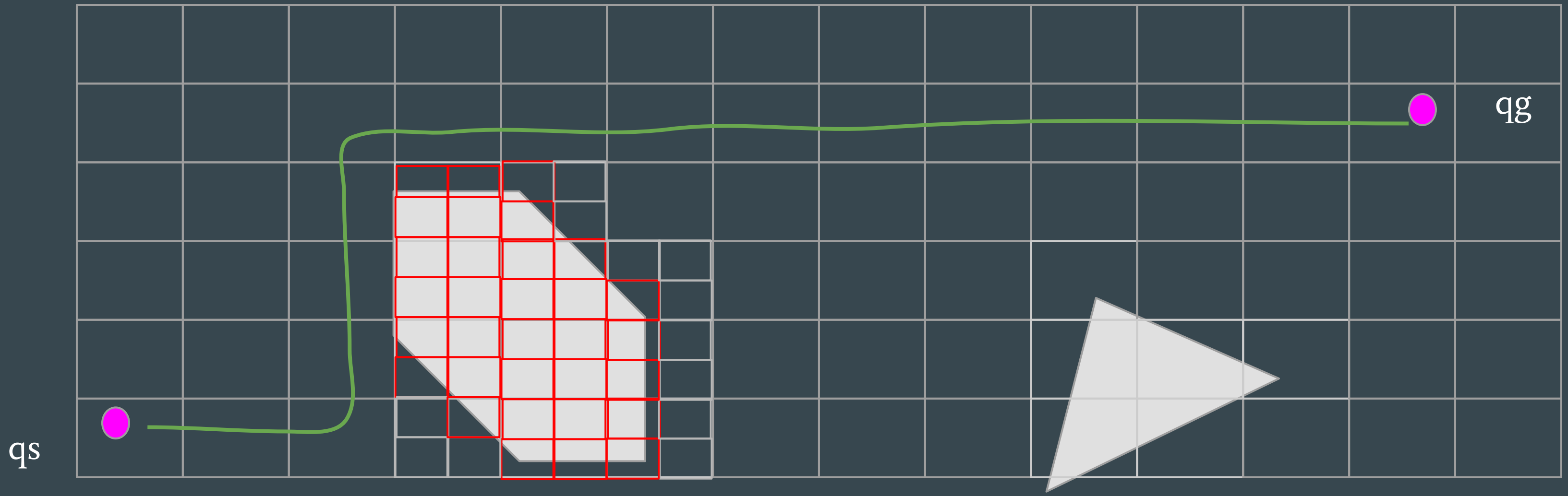
Path Planning: Grid Methods



- Discretization of space
- Occupancy checker
- Graph search algorithm on free cells

- Dependent on cell dimensions
- Subject to shape of objects

Path Planning: Grid Methods with Refinement



- Discretization of space
- Occupancy checker
- Graph search algorithm on free cells

- Dependent on cell dimensions
- Subject to shape of objects

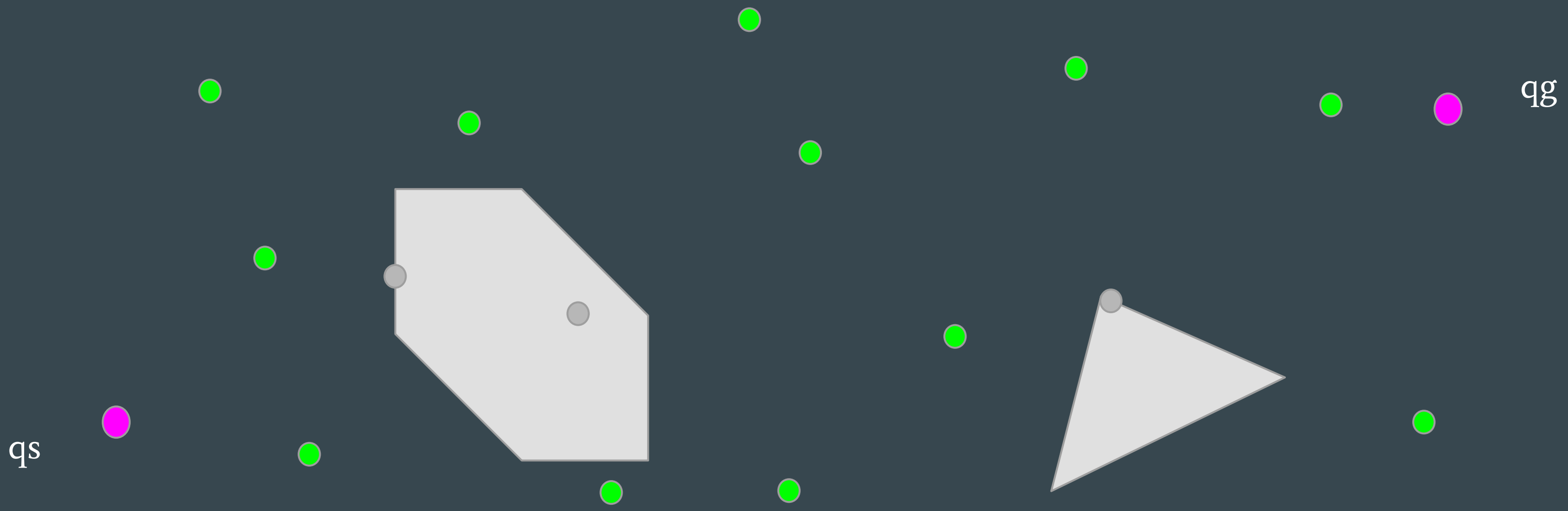
Path Planning with Models

- Reactive
- Model-based
    - Visibility
    - Grid
    - Probabilistic

Path Planning: Probabilistic Roadmap

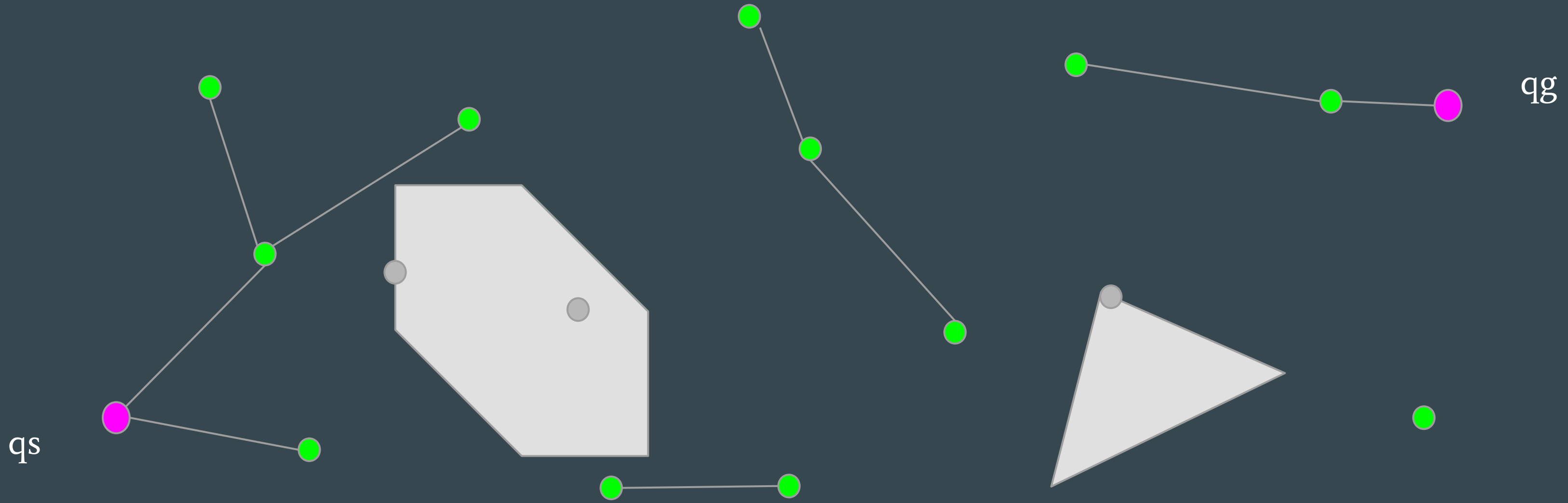
- Random sample of points in space

Path Planning: Probabilistic Roadmap



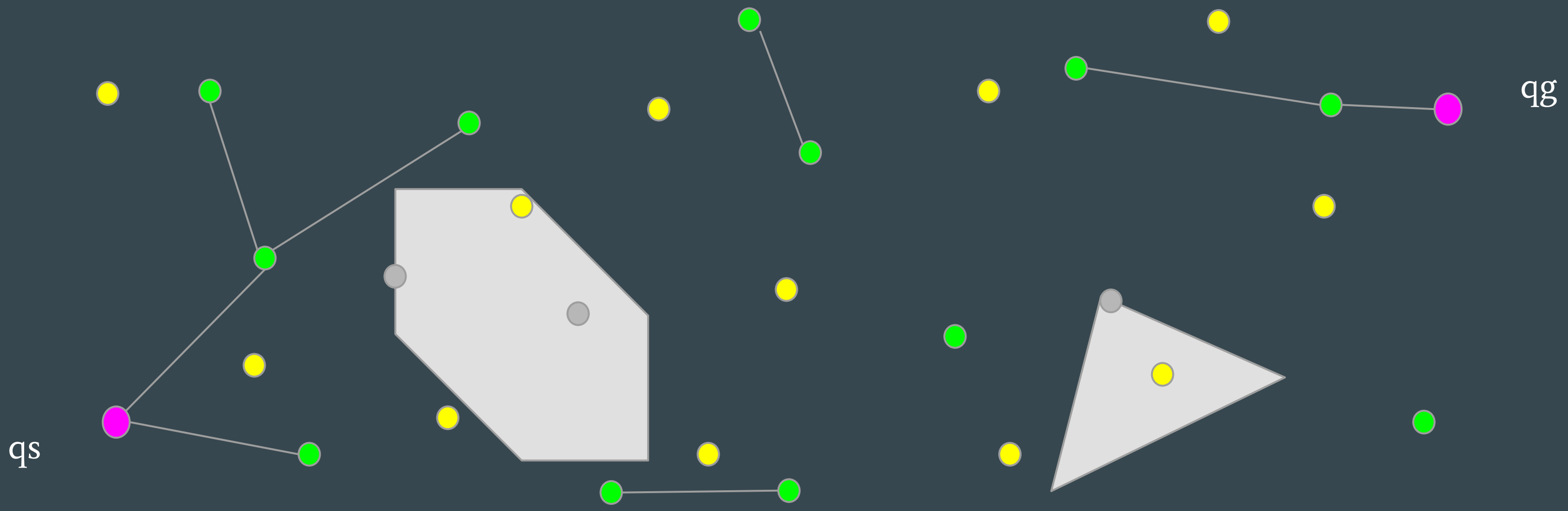
- Random sample of points in space
- Drop samples over obstacles

Path Planning: Probabilistic Roadmap



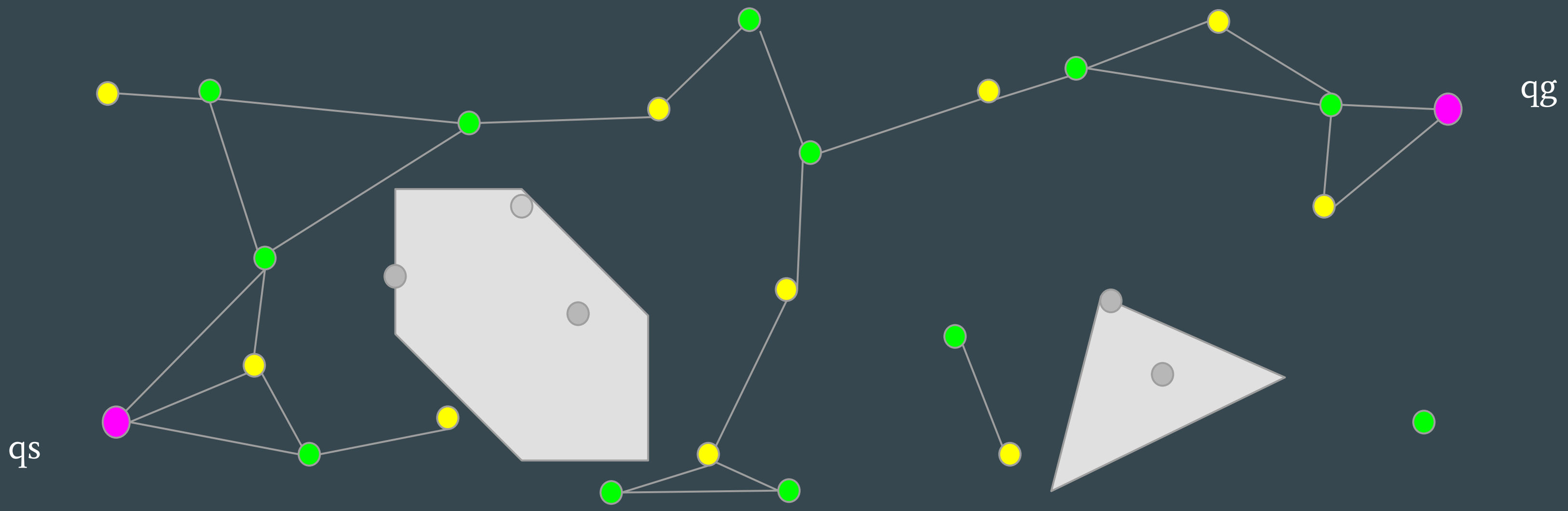
- Random sample of points in space
- Drop samples over obstacles
- Connect samples to k-nearest neighbors

Path Planning: Probabilistic Roadmap



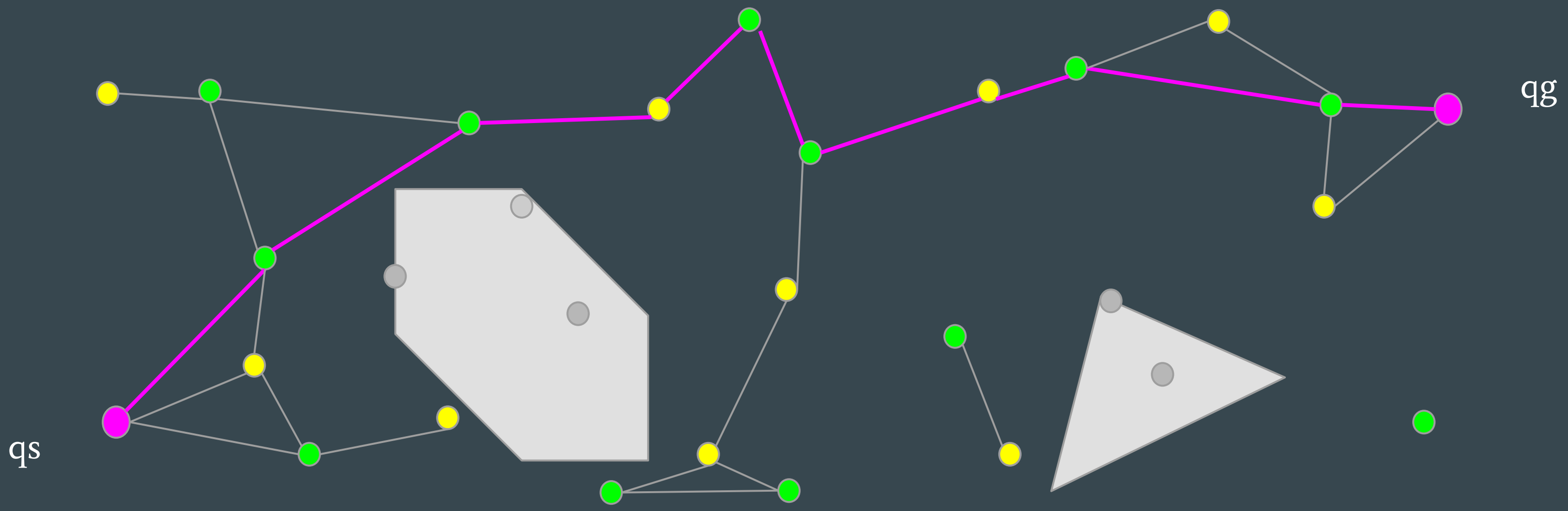
- Random sample of points in space
- Drop samples over obstacles
- Connect samples to k-nearest neighbors
- Sample more points until qs and qg are connected

Path Planning: Probabilistic Roadmap



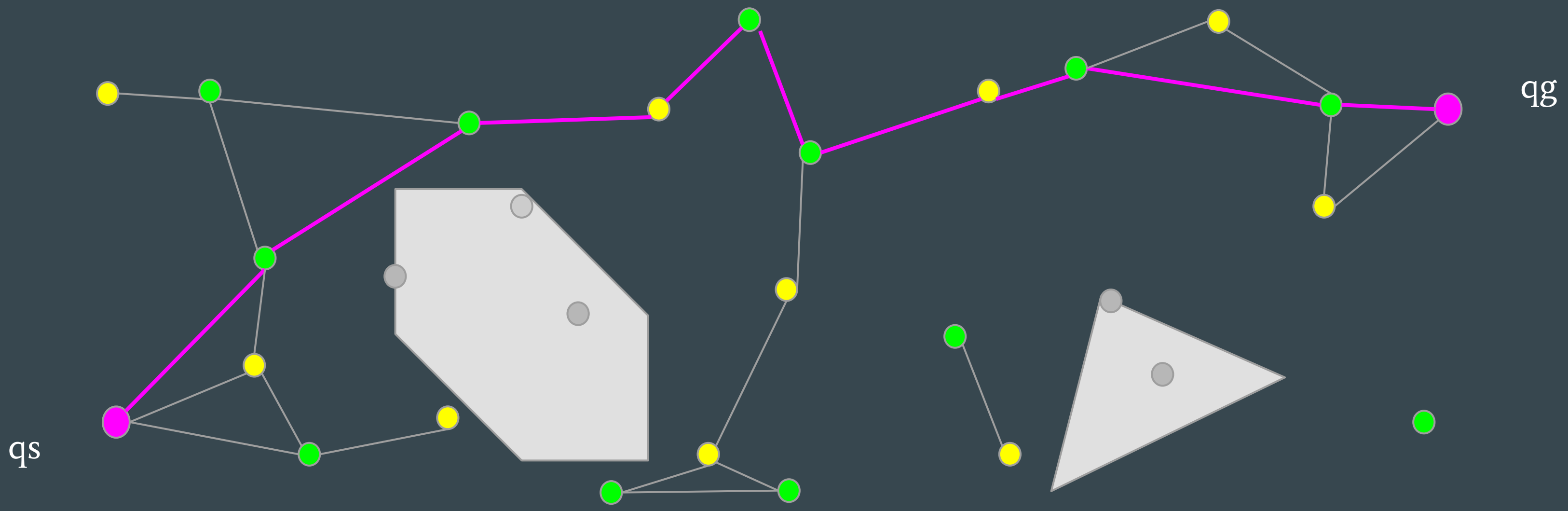
- Random sample of points in space
- Drop samples over obstacles
- Connect samples to k-nearest neighbors
- Sample more points until qs and qg are connected

Path Planning: Probabilistic Roadmap



- Random sample of points in space
- Drop samples over obstacles
- Connect samples to k-nearest neighbors
- Sample more points until qs and qg are connected

Path Planning: Probabilistic Roadmap



The path is non-optimal, how do you optimize it?

Searching in a Graph

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

Take Away

- Families of approaches to navigate world
  - Reactive
    - Local area and fast response
  - Model-based
    - Big picture and long paths
    - Build and searching graphs