Navigation in Unknown Environments

Navigation

- In robotics, ***navigation*** is the act of moving a robot from one place to another in a collision-free path.

- When navigating, robots either:

  I. head towards goal location(s) based on sensor input

  II. follow a fixed path (known in advance)

- Robot usually relies on local sensor information and updates its location/direction according to the *perceived* “best” choice that will lead to the goal.

- Assumes there is no map of the environment, otherwise the problem becomes that of path-planning.

Path Planning

- When a fixed path is provided on which to navigate, the path is usually computed (i.e., planned) beforehand.

- ***Path planning*** *is the act of examining known information about the environment and computing a path that satisfies one or more conditions.*
  - e.g., avoids obstacles, shortest, least turns, safest etc…

- Key to "good" path planning is efficiency
  - in real robots, optimal solution is not always practical
  - approximate solutions are often sufficient and desired.

Path Planning

- To accomplish complicated tasks, a mobile robot usually MUST pre-plan it’s paths.

- Many interesting problems are solved that make use of planned motion of the robot:
  - Efficient collision-free travel (e.g., shortest paths)
  - Environment coverage (e.g., painting, cleaning)
  - Guarding and routing (e.g., security monitoring)
  - Completion of various tasks etc…

- We will look first at ***goal-directed*** navigation in which the robot is trying to reach a goal location based on sensor readings.

Goal-Directed Navigation

- Approaches to goal-directed navigation vary depending on two important questions:
  - Are robot & goal locations (i.e., coordinates) known ?
    - if available, goal position given as a coordinate, otherwise the problem becomes one of searching.
    - robot would either maintain its own location as it moves (e.g., forward kinematics) or have this information provided externally (e.g., GPS system).
  - Are obstacles (i.e., locations and shape) known ?
    - if available, coordinates of all polygonal obstacle vertices would be given and known to the robot.
    - if unavailable, robot must be able to sense obstacles (sensing is prone to error and inaccuracies).

Goal-Directed Navigation

- Here is a summary of categories for navigating towards a goal location under various conditions:
  - **Goal Location "Unknown"**
    - Obstacles Unknown
      1. Behaviors (wandering, ball-seeking, wall following etc…)
    - Obstacles Known
      1. Search algorithms
      2. Coverage algorithms
  - **Goal Location "Known"**
    - Obstacles Unknown (i.e., local sensing information only)
      1. Reactive Navigation
    - Obstacles Known (i.e., global information available)
      1. Feature-Based Navigation
      2. Potential Field Navigation
      3. Roadmap-Based Planning

Already discussed

Won't be discussed

Discussed later

Discussed here

Won't be discussed

Discussed later

“Bug” Algorithms

- Simple navigation when goal location is known but obstacles locations are unknown (i.e., no map)
  - robot must know its location within the environment at all times

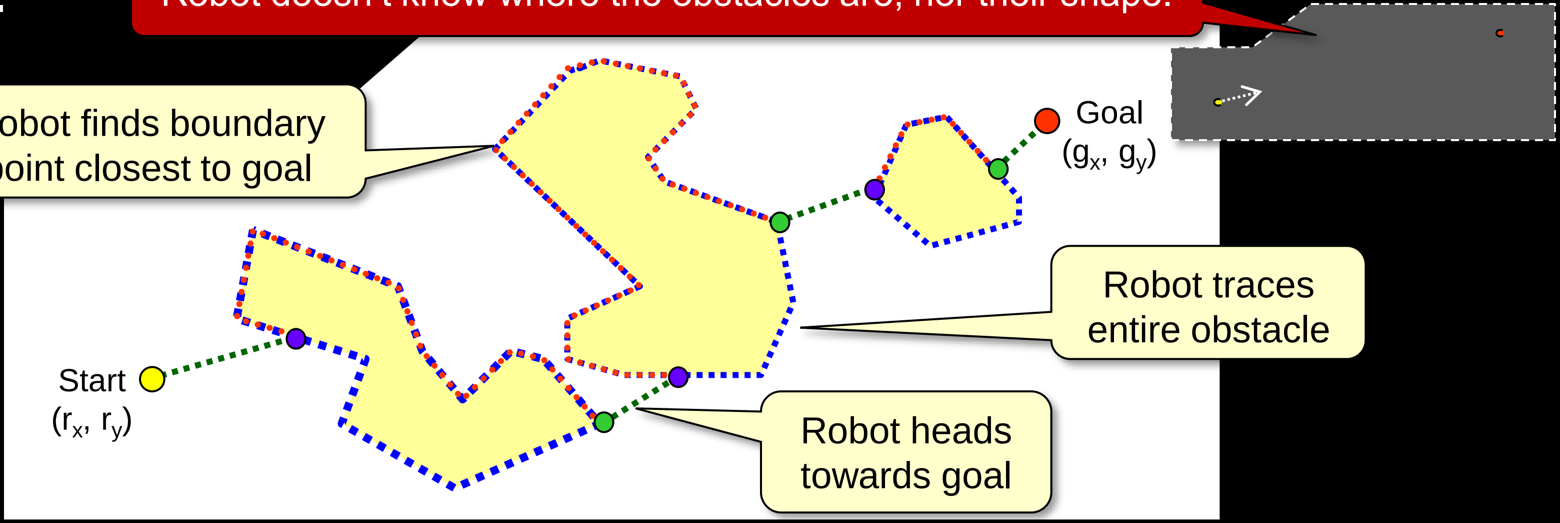
    (i.e., using forward kinematics, beacon location, grid est., gps etc..)
  - robot must have sensors to detect and follow obstacle boundaries
- There are three simple algorithms for this scenario:
  - Bug1
  - Bug2
  - Tangent Bug

Not discussed here

The Bug1 Algorithm

- Bug1 Strategy:
  - Move toward goal unless obstacle encountered, then go around obstacle and find its closest point to the goal.
  - Travel back to that closest point and move towards goal.
- Assumes robot knows goal location but is unable to see or detect it.

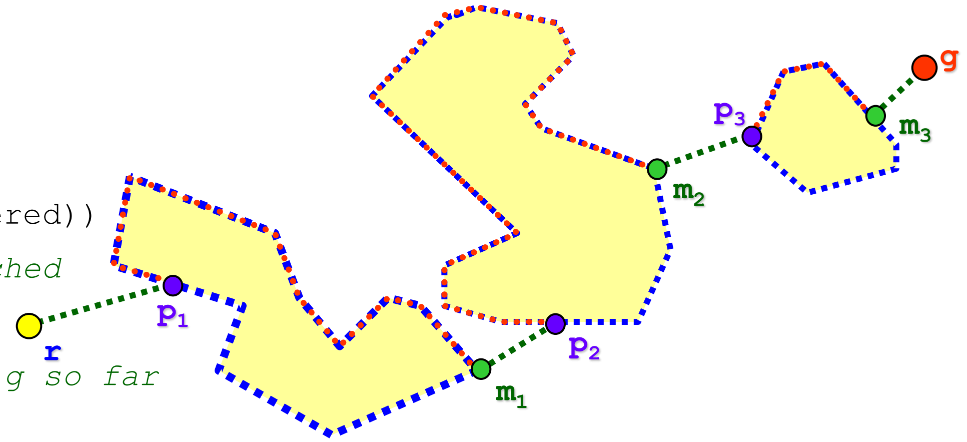
Robot doesn't know where the obstacles are, nor their shape.

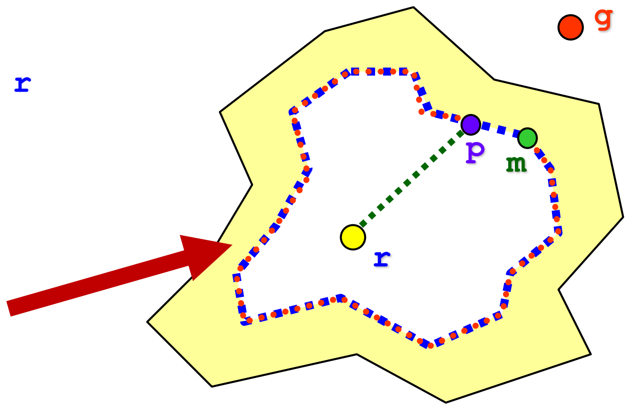

Robot finds boundary point closest to goal
Goal
(g_x, g_y)
Robot traces entire obstacle
Start
(r_x, r_y)
Robot heads towards goal

The Bug1 Algorithm

- Here is the pseudo code for the algorithm:

```
WHILE (TRUE)
   REPEAT
        Move from r towards g
        r = robot's current location
   UNTIL ((r == g) OR (obstacleIsEncountered))
   IF (r == g) THEN quit      // goal reached
   LET p = r        // contact location
   LET m = r        // location closest to g so far
   REPEAT
        Follow obstacle boundary
        r = robot's current location
        IF ((distance(r,g) < distance(m,g)) THEN m = r
   UNTIL ((r == g) OR (r == p))
   IF (r == g) THEN quit      // goal reached
   Move to m along obstacle boundary
   IF (obstacleIsEncountered at m in direction of g)
        THEN quit    // goal not reachable
ENDWHILE
```





The Bug1 Algorithm

- This algorithm:
  - always finds goal location (if it is reachable).
  - performs an exhaustive search for the "best" point to leave the obstacle and head towards the goal.

- If we denote the perimeter of an obstacle Obj_i as $perimeter(Obj_i)$, then the robot may travel a distance of:

$$|\overline{sg}| + 1.5 * [perimeter(Obj_1) + perimeter(Obj_2) + \ldots + perimeter(Obj_n)]$$

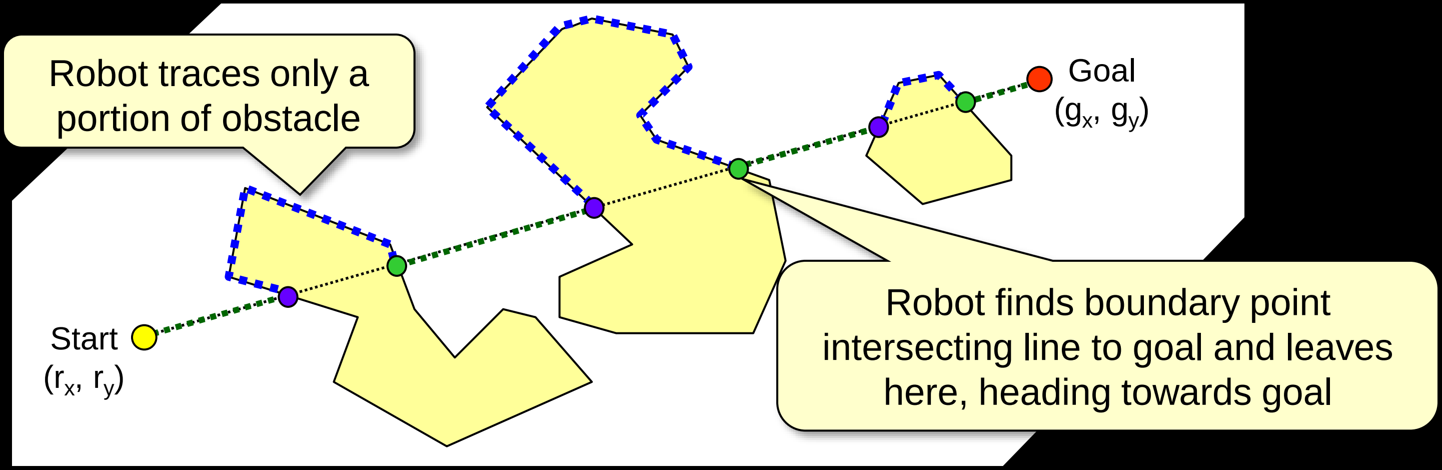
s = start location

Once around the obstacle to determine best position to leave from and up to ½ times around to get back to that position (since we can take the shorter of the two choices).

The Bug2 Algorithm

- A variation to this algorithm will allow the robot to avoid traveling ALL the way around the obstacles.

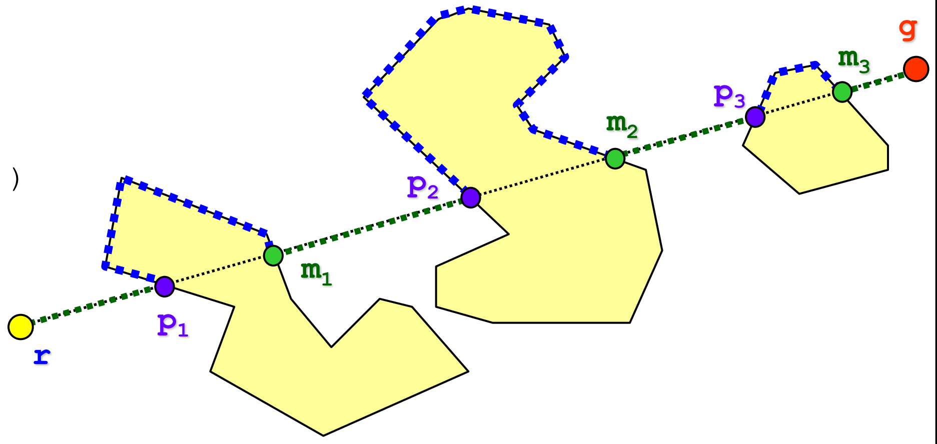
- Bug2 Strategy:
  - Move toward goal unless obstacle encountered, then go around obstacle.   Remember the line from where the robot encountered the obstacle to the goal and stop following when that line is encountered again.


Robot traces only a portion of obstacle
Goal
(g_x, g_y)
Start
(r_x, r_y)
Robot finds boundary point intersecting line to goal and leaves here, heading towards goal

The Bug2 Algorithm

- Here is the pseudo code for the algorithm:

```
WHILE (TRUE)
   LET L = line from r to g
   REPEAT
        Move from r towards g
        r = robot's current location
   UNTIL ((r == g) OR (obstacleIsEncountered))
   IF (r == g) THEN quit     // goal reached
   LET p = r        // contact location
   REPEAT
        Follow obstacle boundary
        r = robot's current location
        LET m = intersection of r and L
   UNTIL (((m is not null) AND (dist(m,g) < dist(p,g)) OR (r == g) OR (r == p))
   IF (r == g) THEN quit     // goal reached
   IF (r == p) THEN quit     // goal not reachable
ENDWHILE
```

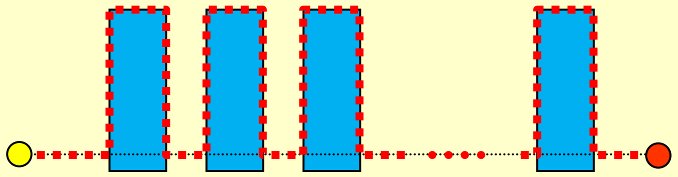


The Bug2 Algorithm

- This algorithm:
  - also always finds goal location (if it is reachable).
  - performs a "greedy" search for the "best" point to leave the obstacle and head towards the goal.
- The robot may travel a distance of:

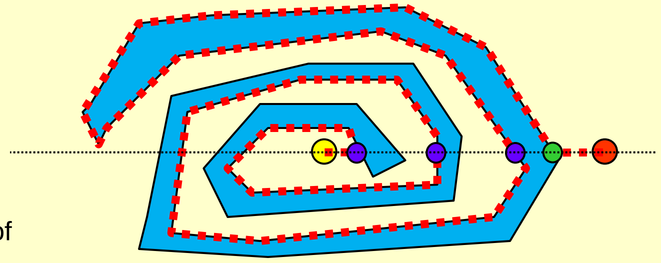
$$|\overline{sg}| + \mathbf{O}(\text{perimeter}(\text{obj}_1) + \text{perimeter}(\text{obj}_2) + \ldots + \text{perimeter}(\text{obj}_n))$$

In worst case, we choose the "wrong way to go around the obstacle, leading to almost a full perimeter traversal:



Amount of perimeter travel will depend on choice of left/right edge following:

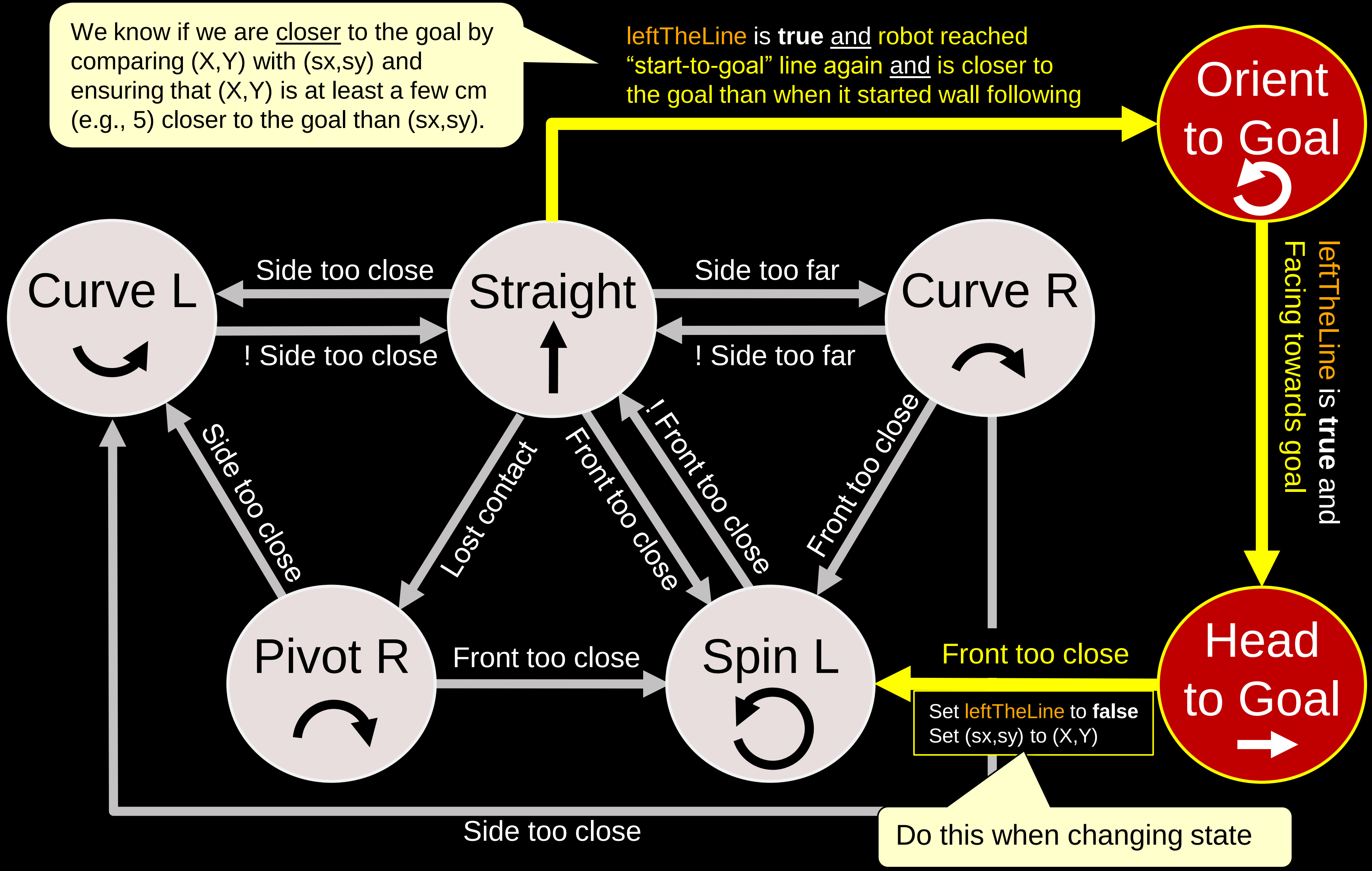
What happens if robot follows on its left side instead of right ?



The Bug2 Algorithm

- Bug2 algorithm is quicker than Bug1.
- The algorithms do have practical problems:
  - Assumes perfect positioning (not really possible)
  - Assumes error-free sensing (not ever possible)
  - Real robots have limited angular resolution
- Algorithms assume that robot could only detect obstacle upon contact or close proximity.
  - Can improve algorithm when robot is equipped with a 360° range sensor that determines distances to obstacles around it.
  - If you have time, look up the Tangent Bug algorithm.

Coding With State Machine


We know if we are closer to the goal by comparing (X,Y) with (sx,sy) and ensuring that (X,Y) is at least a few cm (e.g., 5) closer to the goal than (sx,sy).
leftTheLine is true and robot reached "start-to-goal" line again and is closer to the goal than when it started wall following
Orient to Goal
Curve L
Side too close
Straight
Side too far
Curve R
! Side too close
! Side too far
leftTheLine is true and Facing towards goal
Side too close
Lost contact
Front too close
! Front too close
Front too close
Pivot R
Front too close
Spin L
Front too close
Head to Goal
Set leftTheLine to false
Set (sx,sy) to (X,Y)
Side too close
Do this when changing state

Start the Lab …