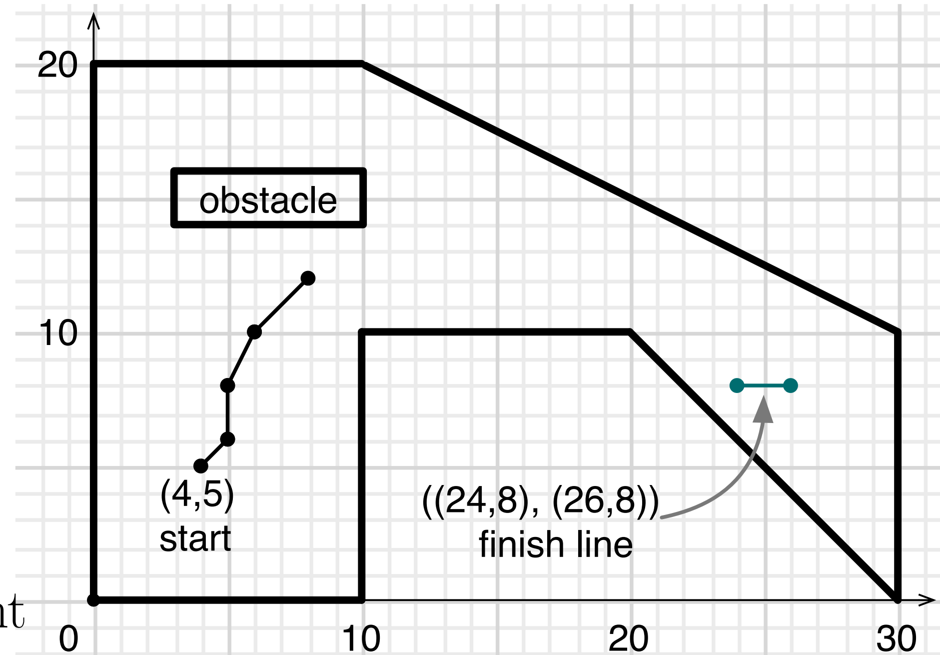
Project 2
Route-Finding for an Unreliable Vehicle

CMSC 421, Fall 2019

Last update October 14, 2019

- Due date: ?
- Late date (20% off): ?

Another kind of racetrack problem



- Robot vehicle, starting point, finish line, walls are the same as in Project 1

Differences:

(1) Vehicle's control system is unreliable

- May move to a slightly different location than you intended
- up to 1 unit in any direction

(2) You can make bigger changes in velocity

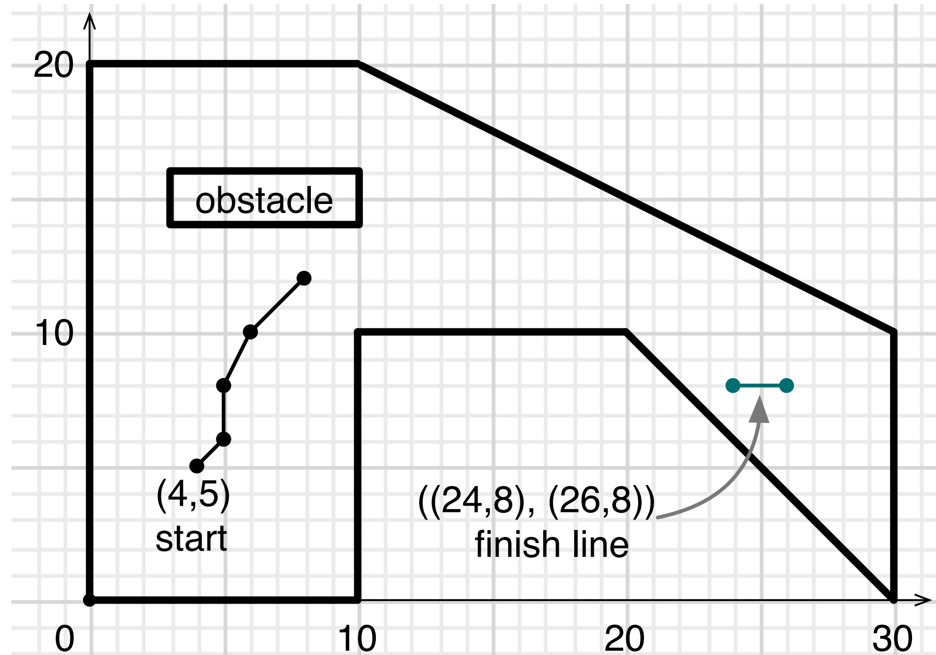
- Up to 2 units in any direction

(3) Don't need to stop exactly on the finish line

- OK to stop at distance ≤ 1

Moving the vehicle

- Current state $s = (p, z)$
  - location $p = (x, y)$, nonnegative integers
  - velocity $z = (u, v)$, integers



- You choose new velocity $z' = (u', v')$, where

  $$u' \in \{u,\ u \pm 1,\ u \pm 2\},$$
  $$v' \in \{v,\ v \pm 1,\ v \pm 2\}.$$

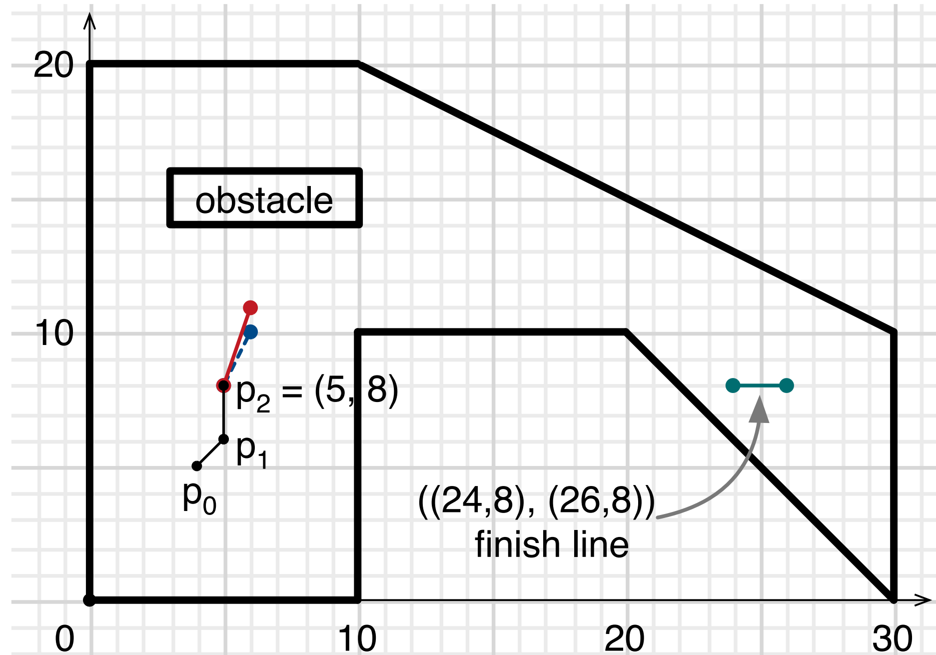
- If $z' \neq (0, 0)$, then the control system may make an error in your position
  - $e = (q, r)$, where $q, r \in \{-1, 0, 1\}$

- Vehicle moves to location $p' \;=\; p + z' + e \;=\; (x + u' + q,\ y + v' + r)$

- New state $s' = (p', z')$

Example

- State $s_2 = ((5,8),(0,2))$
  $$p_2, \quad z_2$$
- You choose
  $$z_3 = z_2 + (1,0) = (1,2)$$
- Control error $e_3 = (0,1)$
- New location $p_3 = p_2 + z_3 + e_3$
  $= (5,8) + (1,2) + (0,1)$
  $= (6,11)$

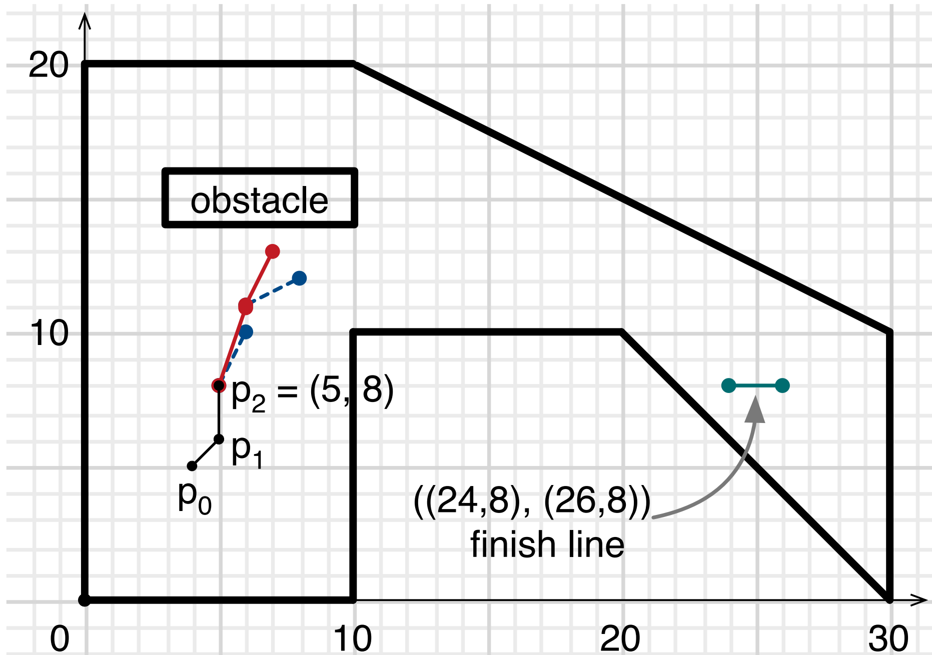


- New state $s_3 = (p_3, z_3) = ((6,11),(1,2))$

- The control error doesn't change velocity, just your position
  - Unrealistic, but it makes the problems easier to solve

Example

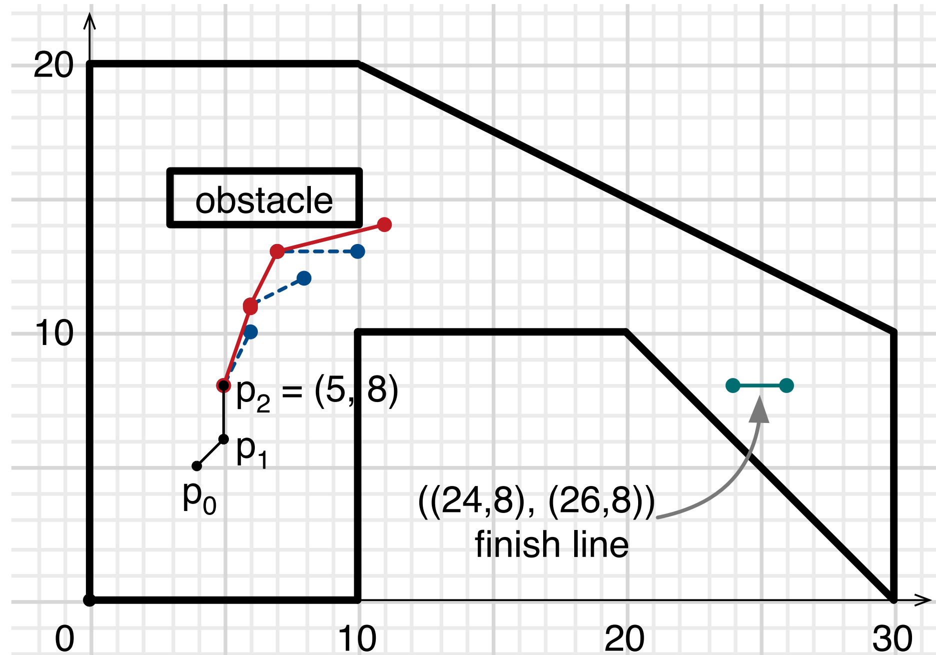
- State $s_3 = ((6, 11), (1, 2))$
  $\qquad\qquad\qquad p_3, \quad z_3$
- You choose
  $z_4 = z_3 + (1, -1) = (2, 1)$
- Control error $e_4 = (-1, 1)$
- New location $p_4 = p_3 + z_4 + e_4$
  $= (6, 11) + (2, 1) + (-1, 1)$
  $= (7, 13)$



- New state $s_4 = (p_4, z_4) = ((7, 13), (2, 1))$

Example

- State $s_4 = ((7, 13), (2, 1))$
  $\qquad\qquad p_4, \qquad z_4$
- You choose
  $z_5 = z_4 + (1, -1) = (3, 0)$
- Control error $e_5 = (1, 1)$
- New location $p_5 = p_4 + z_5 + e_5$
  $= (7, 13) + (3, 0) + (1, 1)$
  $= (11, 14)$

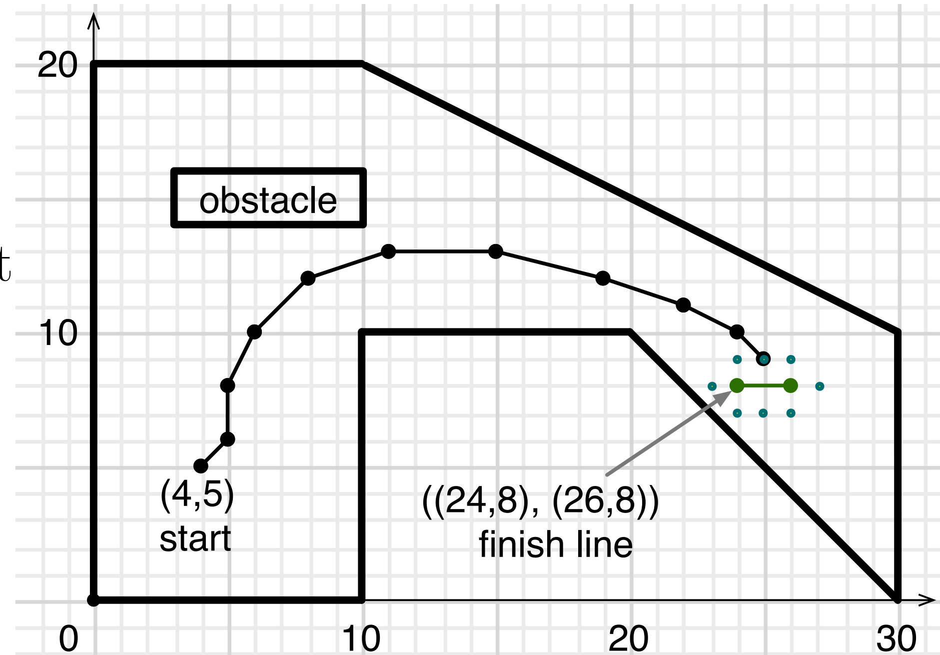


- New state $s_5 = (p_5, z_5) = ((11, 14), (3, 0))$

- Trajectory is *unsafe*
  - Would have crashed if e_5 were $(0, 1)$ or $(-1, 1)$
- Ideally, you want a strategy that will always keep you from crashing regardless of what control errors occur

Objective

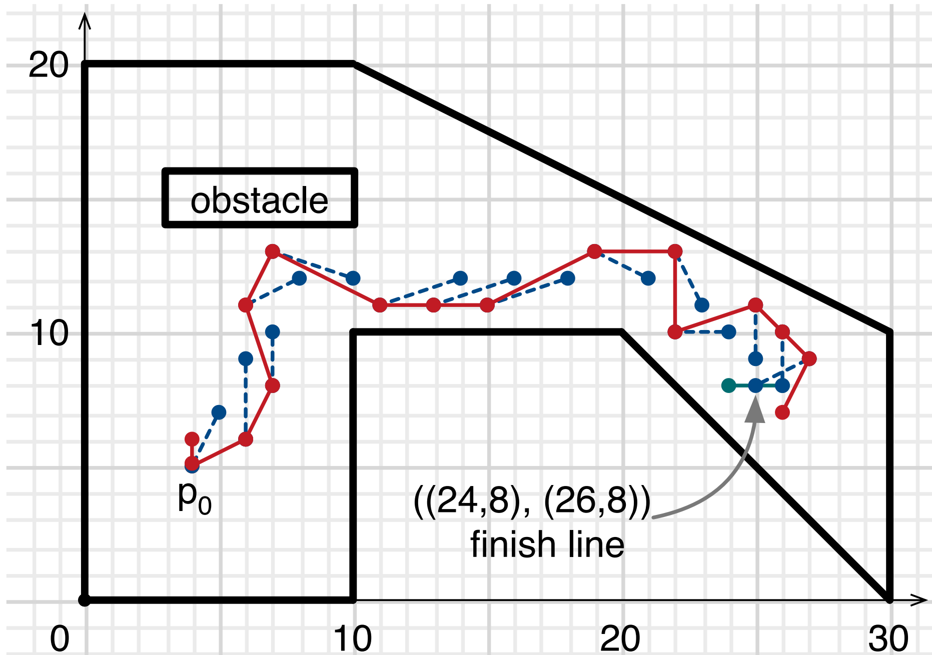
- Get to the finish line and stop
  - Might not be able to land exactly on the line
  - Control errors can prevent that
- OK to get to distance ≤ 1
- Need to stop
  - Last move needs to have velocity 0, as in Project 1



- Want to get there as quickly as possible without crashing, despite control errors

Strategy

- Pretend the control system is an opponent that's trying to make you crash
- Choose moves that will keep you from crashing, regardless of what it does
- Write a game-playing algorithm to do it move by move
  - as in chess, checkers, or go



How to do it

- One possibility: alpha-beta game-tree search
  - Limited-depth search, static evaluation function

- Another possibility: Monte Carlo rollouts
  - Problem: randomly generated paths are very unlikely to go to the goal
  - I don't think it will work very well

- Another idea: biased Monte Carlo rollouts
  - Generate paths randomly, but bias the moves toward good evaluation-function values
  - How well this will work, I have no idea

- No way to guarantee you won't crash

Game environment

- We'll give you a game environment for running racetrack problems
  - It will simulate turn-by-turn interactions with an opponent (the "unreliable control system")
- At each turn, it will call your program with three arguments
  - current state, finish line, list of walls
- Your program should print (to standard output) a sequence of recommendations for what velocity to use. Each recommendation should be a pair of integers (u, v) followed by a linebreak.

```
(2, 2)
(1, 3)
(1, 2)
(1, 2)
```

- Your program should keep searching for better and better recommendations
    - e.g., iterative deepening, or additional Monte Carlo rollouts
  - Shouldn't exit unless it has exhausted its search space

Game environment

- Game environment will let your program run for some amount of time t
  e.g., 500 or 1000 milliseconds
  - then it will kill your program and use the last velocity it recommended
- Next, the opponent will choose what error to use
  - $e = (q, r)$, where $q, r \in \{-1, 0, 1\}$
- Game environment will compute the new state, and check whether the game has ended
  - you crash $\Rightarrow$ you lose
  - you reach the finish line and your velocity is (0,0) $\Rightarrow$ you win
  - otherwise, game hasn't ended $\Rightarrow$ game environment will call your program again, with the new current state

Opponent

- We'll give you a simple opponent program
  - It will try to make you crash, but won't be very intelligent about it
- Warning: don't write a program that just tries to take advantage of the dumb opponent!
  - When we grade your program, we'll use a more intelligent opponent
- Need to choose moves that won't crash, no matter what the opponent does

Other comments

- You may use any of the code I gave you for Project 1, and any of the code you developed for Project 1
  - You can modify it if you wish
- Caveat: most of it won't be very useful
  - You'll need to write a game-tree-search algorithm and/or a Monte Carlo rollout algorithm
- You'll need a heuristic function
  - You can use the one you developed for Project 1
  - You can use any of the ones I gave you for Project 1
    - e.g., `h_walldist`
- Caveat: Will a heuristic function for Project 1 work well as a game-tree-search heuristic?
  - You might need to make modifications

Grading

- Evaluation criteria:
  - 35% correctness: – whether your algorithm works correctly, whether your submission follows the instructions
  - 15% programming style – see the following
    - Style guide: https://www.python.org/dev/peps/pep-0008/
    - Python essays: https://www.python.org/doc/essays/
  - 15% documentation
    - Docstrings at the start of the file and in each function; comments elsewhere
  - 35% on performance
    - Does your program crash? If so, then how frequently?
    - If it doesn't crash, then how many moves to reach the finish line?
    - Top n performers ($n \approx 3$ to 5) will get extra credit