Project 2
Route-Finding for an Unreliable Vehicle

## CMSC 421, Fall 2019

Last update October 16, 2019

- ▶ Due date: November 1
- ▶ Late date (20% off): November 3

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

obstacle

(4,5)
start

((24,8), (26,8))
finish line
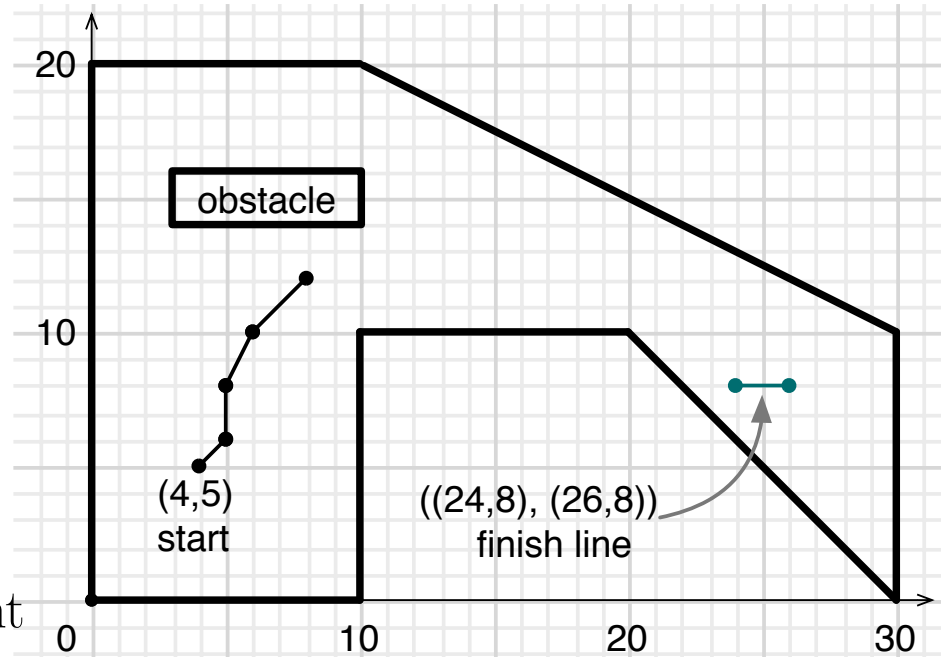
Moving the vehicle

- Current state $s = (p, z)$

  ▸ location $p = (x, y)$, nonnegative integers
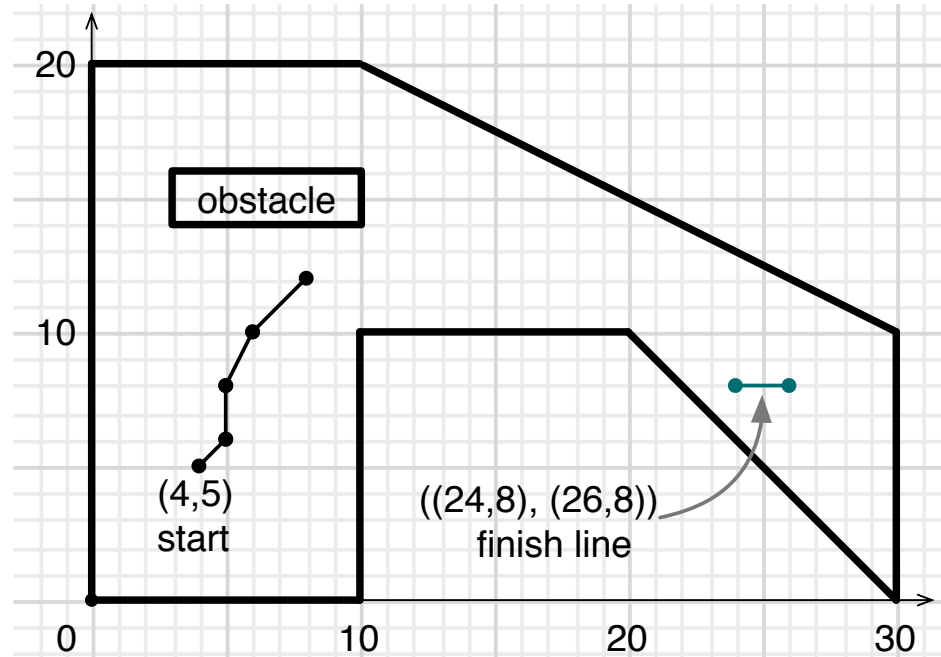
  ▸ velocity $z = (u, v)$, integers

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

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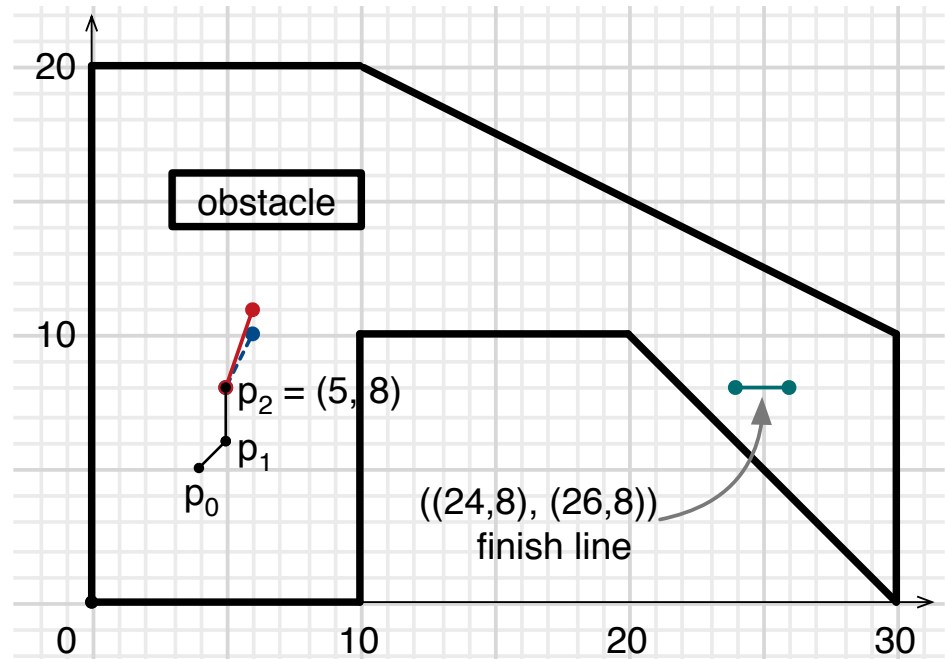$$v' \in \{v, \ v \pm 1, \ v \pm 2\}.$$

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

obstacle

(4,5)
start

((24,8), (26,8))
finish line

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 $s_3 = ((6, 11), (1, 2))$
  $\qquad\quad p_3, \qquad z_3$

- You choose
  $$z_4 = z_3 + (1, -1) = (2, 1)$$

- Control error $e_4 = (-1, 1)$

- New location $p_4 = p_3 + z_4 + e_4$
  $= (6, 11) + (2, 1) + (-1, 1)$
  $= (7, 13)$

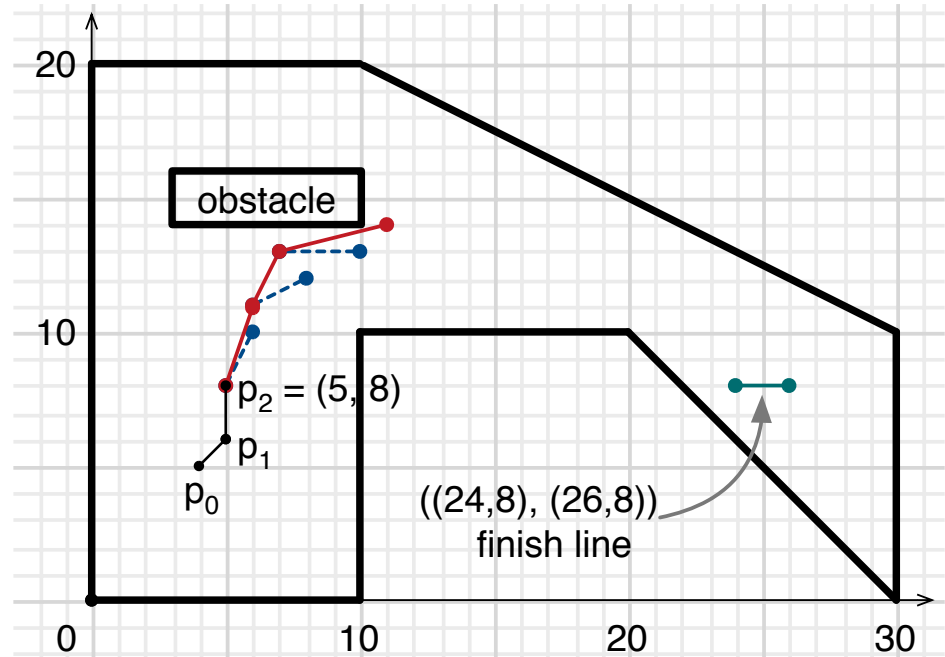- New state $s_4 = (p_4, z_4) = ((7, 13), (2, 1))$
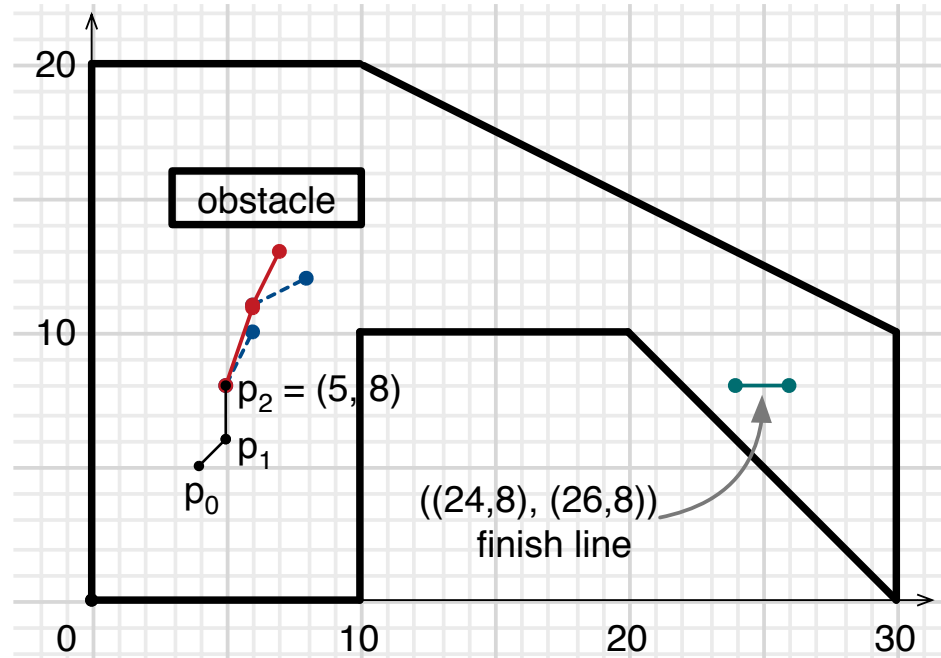
Example

- State $s_4 = ((7, 13), (2, 1))$
$$p_4, \qquad z_4$$

- You choose
$$z_5 = z_4 + (1, -1) = (3, 0)$$

- Control error $e_5 = (1, 1)$

- New location $p_5 = p_4 + z_5 + e_5$
$$= (7, 13) + (3, 0) + (1, 1)$$
$$= (11, 14)$$

obstacle

$p_2 = (5, 8)$

p_1

p_0

$((24,8), (26,8))$
finish line

- 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

obstacle

(4,5)
start

((24,8), (26,8))
finish line
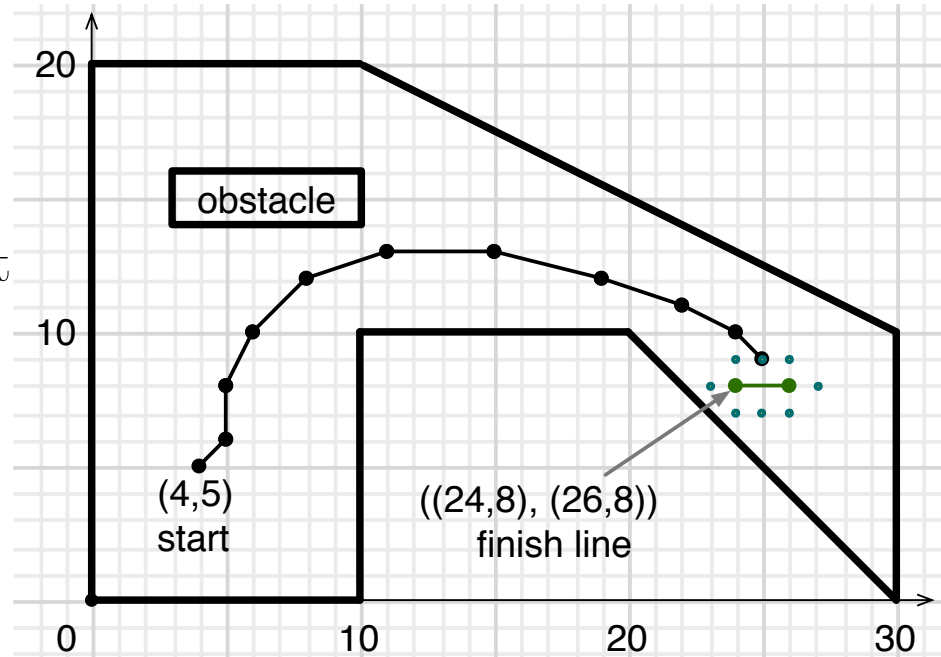
Strategy

- 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
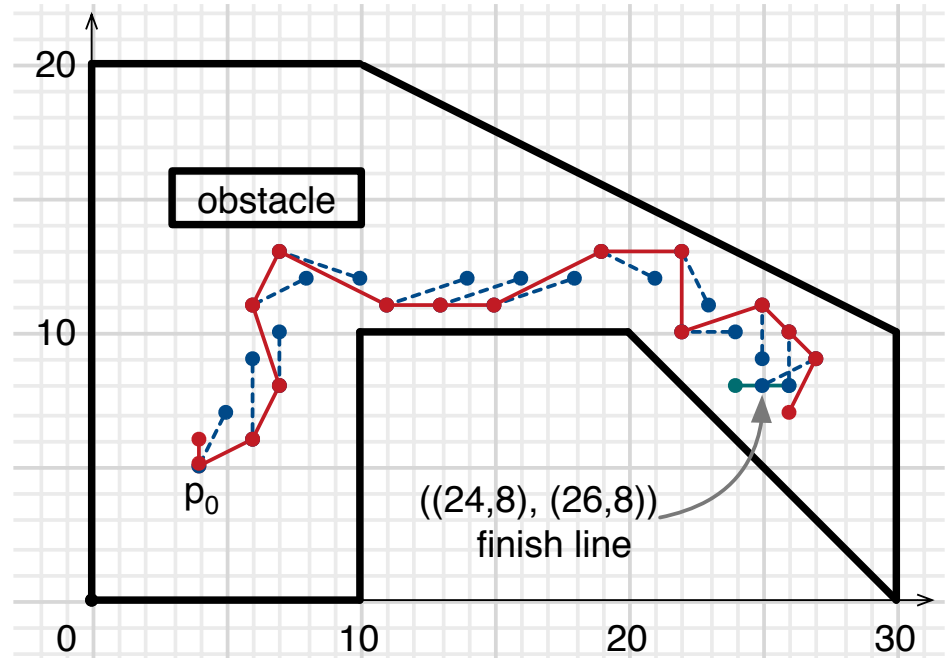
How to do it

- 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

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

What you need to submit

- You need to submit a file called `proj2.py` containing a program called `main`

- We'll give you a game environment for running it
  - ▶ It will simulate turn-by-turn interactions with the opponent
  - ▶ At each turn, it will run `proj2.main`(s, f, w)
    - $s = $ state, $f = $ finish line, $w = $ list of walls

- Your `proj2.main` program should print (to standard output) a sequence of choices for what velocity to use. Each choice should
be a pair of integers (u, v) followed by a linebreak.
  ```
  (2, 2)
  (1, 3)
  (1, 2)
  (1, 2)
  ```

- Keep searching for better and better recommendations
  - e.g., iterative deepening, or additional Monte Carlo rollouts

More about the game environment

- Game environment runs your `proj2.main` program as a separate process
  - ▸ Lets it run for 5 seconds, kills it, reads the last velocity it chose

- After getting your chosen velocity (u, v),
  it lets the opponent choose what error to use
  - ▸ $e = (q, r)$, where $q, r \in \{-1, 0, 1\}$

- It computes the new state, and checks 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

- If the game hasn't ended, it goes to the next turn
  - ▸ runs your `proj2.main` program again

Files I'll provide

- File on Piazza: `project2b_code.zip`
  - ▸ Not `project2_code.zip` – that version had an error in it

- `sample_probs` – modified version of the test problems from Project 1.
  - ▸ I removed or modified the ones that were obviously unsolvable.
  - ▸ Each problem is a list of the form `[name, p0, finish, walls]`
  - ▸ If a problem's dimensions are so small that the problem is unsolvable, you can call `double(`p`)` where p is the problem, to return a problem in which the x and y dimensions are both doubled.

- `opponents.py` – two simple opponent programs.
  - ▸ `opponent1` tries to head for the wall
  - ▸ `opponent0` makes moves at random
  - ▸ By default, `env.main` uses `opponent0`

Files I'll provide (continued)

- env.py – environment for running your proj2.py file.

Here's what env.main(problem) does:

1. If you have a proj2.initialize, it launches proj2.initialize(s, f, w), waits 5 seconds, and kills the process if it hasn't exited.
   - ▶ This is so you can compute some data to use in your proj2.main program
   - ▶ Your proj2.initialize should write the data to a file called data.txt; otherwise the data will be lost when the process exits

2. It repeats the following steps until you win or lose:
   - ▶ Launch proj2.main, wait 5 seconds, and kill the process
   - ▶ Read the last velocity (u, v) in choices.txt. If it isn't legal, return lose
   - ▶ If $(u, v) = (0, 0)$ and distance from finish line ≤ 1, return win.
   - ▶ Call the opponent to add an error to the velocity
   - ▶ Draw the move using turtle graphics
   - ▶ If the move crashes into a wall, return lose

Files I'll provide (continued)

- `proj2_example.py` – a deliberately stupid version of `proj2.py`

  ▶ Rename it to `proj2.py` if you want to use it with `env.py`.

  ▶ I provided it so you can see how `env.py` works before you start
    writing your own `proj2.py`.

  ▶ It can win if it's lucky, but usually it eventually crashes

  ▶ You'll need to write something that works much better

- Some files used by `proj2_example.py`

  ▶ `racetrack_example.py` – modified version of `racetrack` from Project 1

  ▶ `fsearch.py` and `tdraw.py` – same as in Project 1

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 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