Project 1: Vehicle Route-Finding

CMSC 421, Fall 2018

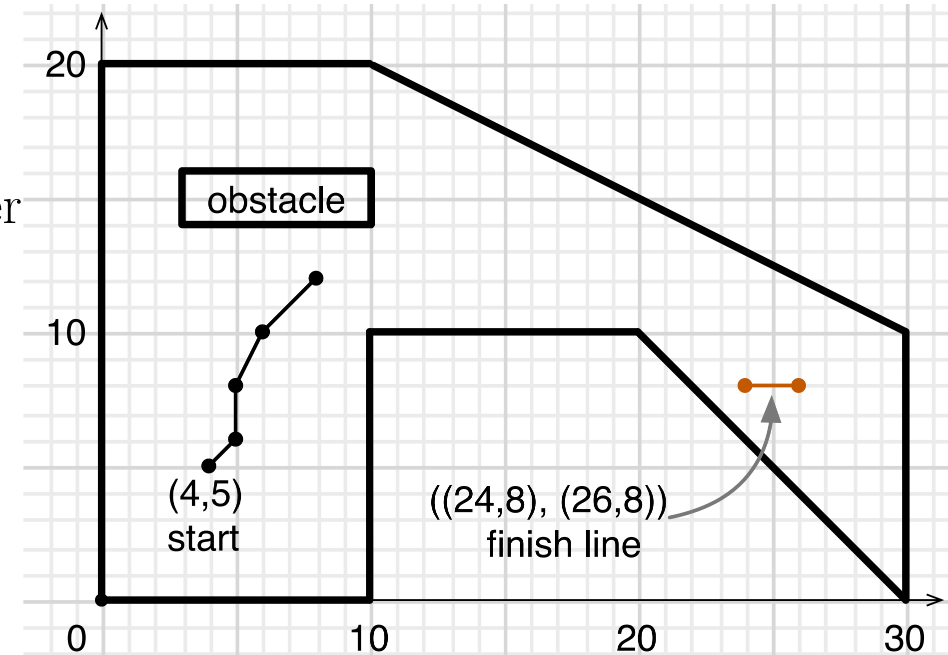
Last update September 13, 2018

- Due date: Sept 28, 11:59pm
- Late date (10% off): Oct 1, 11:59pm

- To be done individually (not in teams)

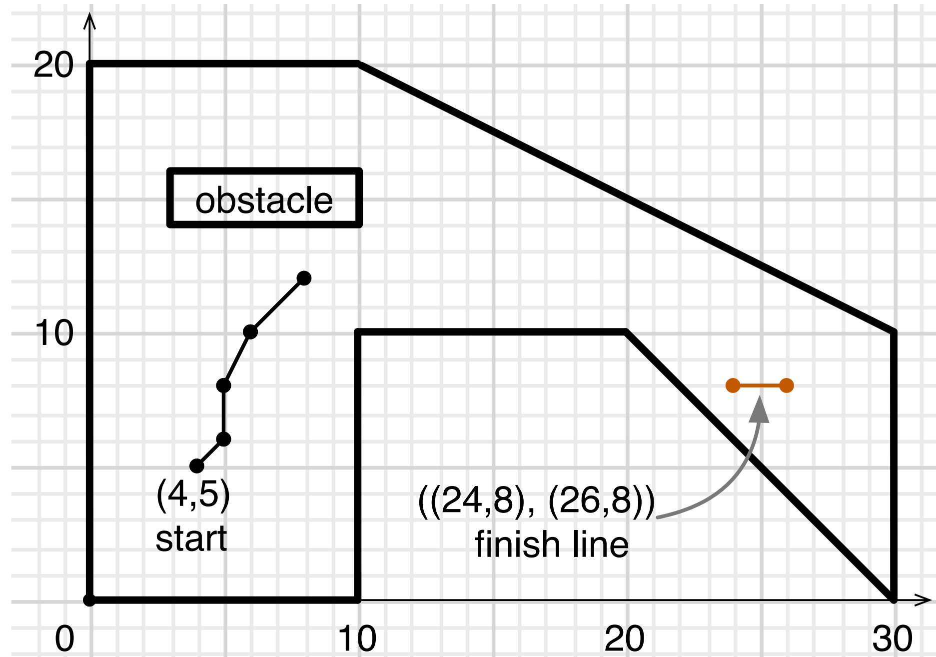
Problem domain

- Modified version of Racetrack
  - Invented in early 1970s
  - played by hand on graph paper



- 2-D polygonal region
  - Inside are a starting point, finish line, maybe obstacles
- All walls are straight lines
- All coordinates are nonnegative integers
- Robot vehicle begins at starting point, can make certain kinds of moves
- Want to move it to the finish line as quickly as possible
  - Without crashing into any walls
  - Need to come to a complete stop on the finish line

Moving the vehicle



- Before the i'th move, current state is
  $s_{i-1} = (p_{i-1}, z_{i-1})$
  - location $p_{i-1} = (x_{i-1}, y_{i-1})$, nonnegative integers
  - velocity $z_{i-1} = (u_{i-1}, v_{i-1})$, integers
- To move the vehicle
  - First choose a new velocity $z_i = (u_i, v_i)$, where

$$u_i \in \{u_{i-1} - 1, \quad u_{i-1}, \quad u_{i-1} + 1\}, \tag{1}$$

$$v_i \in \{v_{i-1} - 1, \quad v_{i-1}, \quad v_{i-1} + 1\}. \tag{2}$$

  - New location: $\quad p_i = (x_{i-1} + u_i, y_{i-1} + v_i)$
  - New state: $\quad s_i = (p_i, z_i)$

Example

- Initial state:

$$p_0 = (4,5)$$
$$z_0 = (0,0)$$
$$s_0 = (p_0, z_0) = ((4,5),(0,0))$$

- First move:

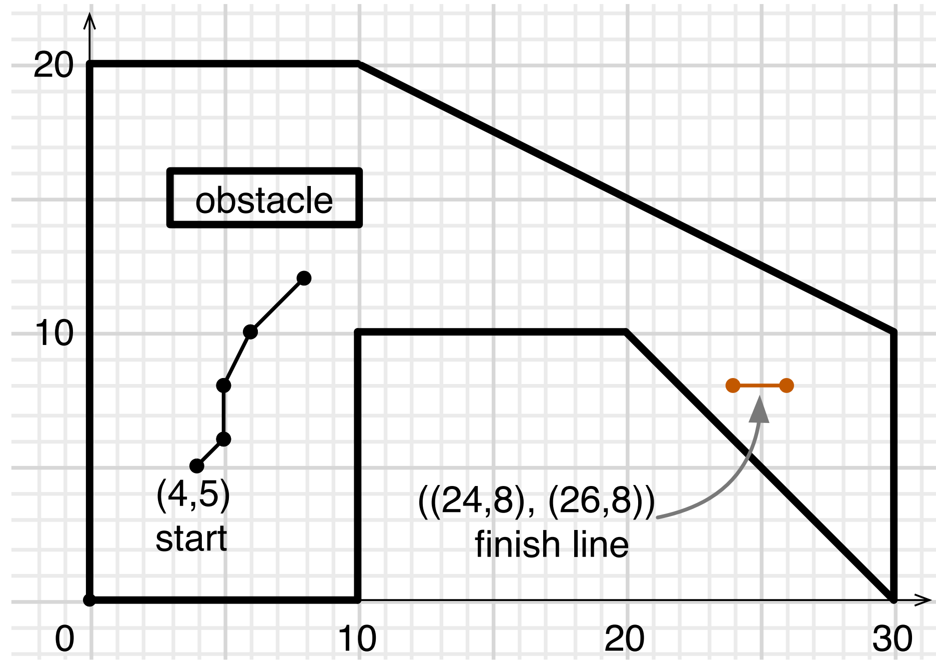
$$z_1 = (0,0) + (1,1) = (1,1)$$
$$p_1 = (4,5) + (1,1) = (5,6)$$
$$s_1 = ((5,6),\ (1,1))$$

- Second move:

$$z_2 = (1,1) + (-1,1) = (0,2)$$
$$p_2 = (5,6) + (0,2) = (5,8)$$
$$s_2 = ((5,8),\ (0,2))$$

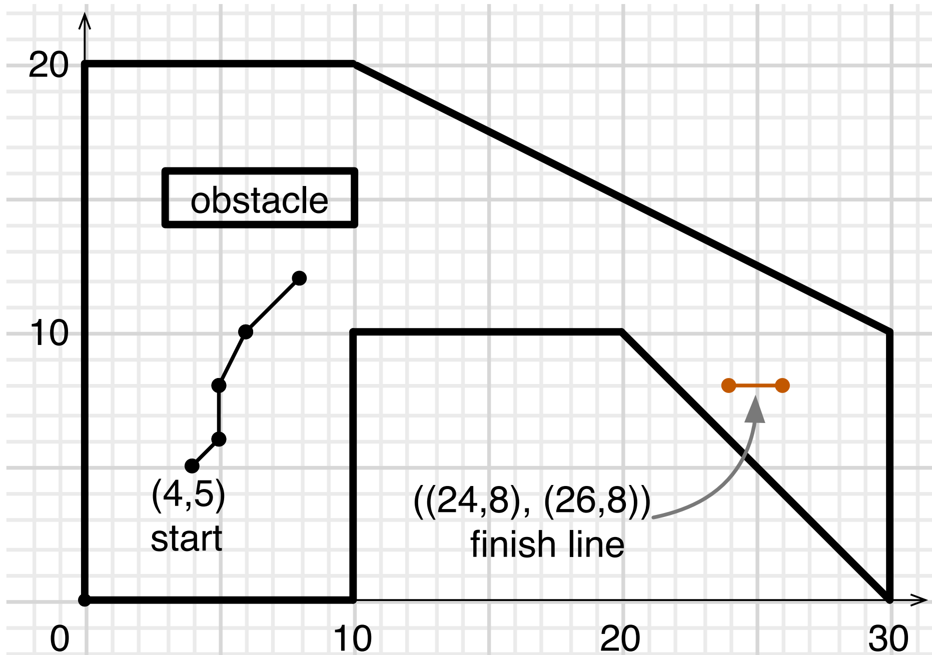
- Third move:

$$z_3 = (0,2) + (1,0) = (1,2)$$
$$p_3 = (5,8) + (1,2) = (6,10)$$
$$s_3 = ((6,10),\ (1,2))$$



Walls

- *edge*: a pair of points (p, q)
  - $p = (x, y)$, $q = (x', y')$
    - coordinates are nonnegative integers
- *wall*: an edge that the vehicle can't cross

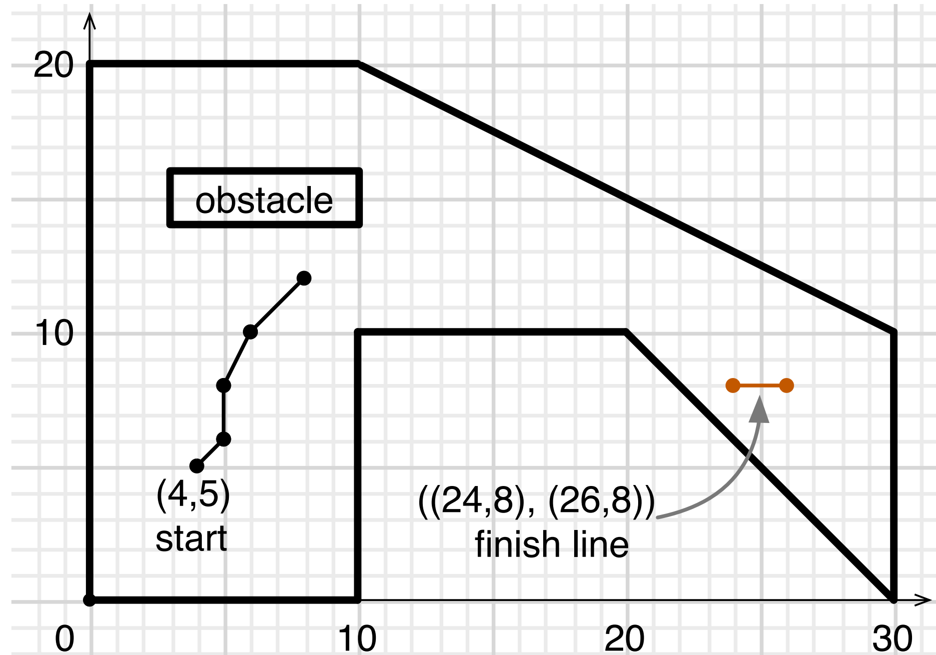


- List of walls in the example:

$$[((0,0),(10,0)),\quad ((10,0),(10,10)),\quad ((10,10),(20,10)),$$
$$((20,10),(30,0)),\quad ((30,0),(30,10)),\quad ((30,10),(10,20)),$$
$$((10,20),(0,20)),\quad ((0,20),(0,0)),\quad ((3,14),(10,14)),$$
$$((10,14),(10,16)),\quad ((10,16),(3,16)),\quad ((3,16),(3,14))]$$

Moves and paths

- *move*: an edge $m = (p_{i-1}, p_i)$
  - $p_{i-1} = (x_{i-1}, y_{i-1})$
  - $p_i = (x_i, y_i)$
  - represents change in location from time $i-1$ to time i

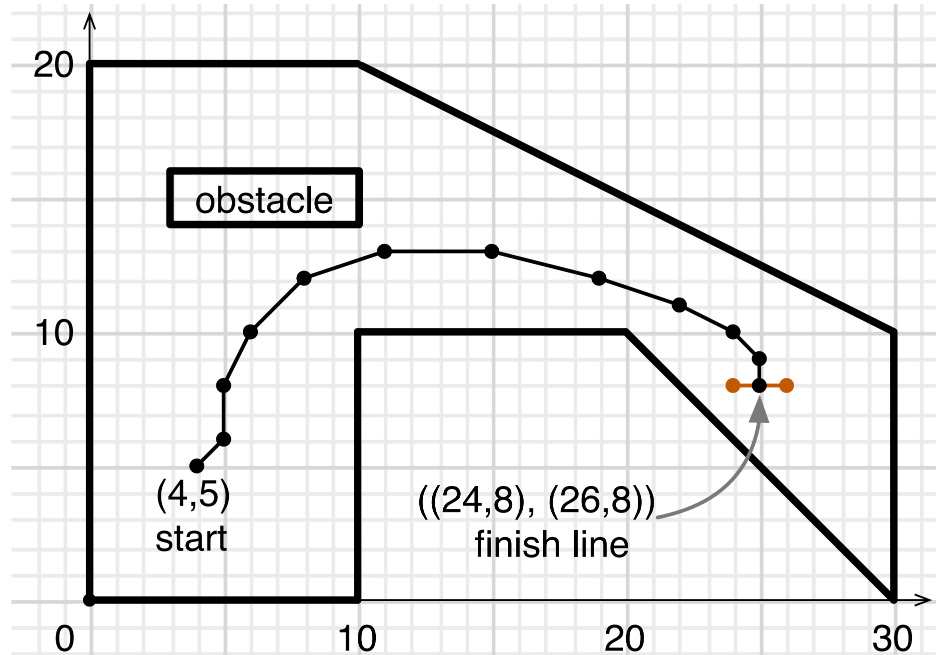


- Example:

$$m_1 = ((4,5),\ (5,6))$$
$$m_2 = ((5,6),\ (5,8))$$
$$m_3 = ((5,8),\ (6,10))$$
$$m_4 = ((6,10),\ (8,12))$$

- *path*: list of locations $[p_0, p_1, p_2, \ldots, p_n]$
  - represents sequence of moves $(p_0, p_1),\ (p_1, p_2),\ (p_2, p_3),\ \ldots,\ (p_{n-1}, p_n)$
  - Example: $[(4,5),\ (5,6),\ (5,8),\ (6,10),\ (8,12)]$
- If a move or path intersects a wall, it *crashes*, otherwise it is *safe*

Objective



- *Finish line*:
  - an edge $f = ((q, r), (q', r'))$
  - always horizontal or vertical
- Want to reach the finish line with as few moves as possible
- Find a path $[p_0, p_1, \ldots, p_n]$
  - p_n must be on the finish line
    - $\exists t,\ 0 \leq t \leq 1$, such that $p_n = t(q, r) + (1 - t)(q', r')$
  - Final velocity must be $(0, 0)$
    - Thus $p_{n-1} = p_n$

  Example: $[(4,5),\ (5,6),\ (5,8),\ (6,10),\ (8,12),\ (11,13),\ (15,13),$
  $(19,12),\ (22,11),\ (24,10),\ (25,9),\ (25,8),\ (25,8)]$

Things I'll provide

I'll post a zip archive that includes the following code (in Python 3.6):

- `fsearch.py` – domain-independent forward search algorithm
  - can do depth first, best first, uniform cost, A*, and GBFS
  - has hooks for calling a drawing package to draw search spaces
- `tdraw.py` – code to draw search spaces for racetrack problems
- `racetrack.py` – code to run `fsearch.py` on racetrack problems
- `maketrack.py` – Code to generate random racetrack problems
- `sample_probs.py` – Some racetrack problems I generated by hand
- `heuristics.py` – Some domain-specific heuristic functions

- `run_tests.bash` and `run_demo.bash` – customizable scripts to run experiments

Here are some details …

`fsearch.py`

Domain-independent forward-search algorithm

- Implementation of **Graph-Search-Redo**

- `main(s0, next_states, goal_test, strategy, h = None, verbose = 2, draw_edges = None)`
  - `s0` – initial state
  - `next_states`(s) – function that returns the possible next states after s
  - `goal_test`(s) – function that returns `True` if s is a goal state, else `False`
  - `strategy` – one of `'bf'`, `'df'`, `'uc'`, `'gbf'`, `'a*'`
  - `h`(s) – heuristic function, should return an estimate of $h^*(s)$
  - `verbose` – one of `0`, `1`, `2`, `3`, `4`
    - how much information to print out (see documentation in the file)
  - `draw_edges` – function to draw edges in the search space

`racetrack.py`

Code to run `fsearch.main` on racetrack problems

- `main(problem, strategy, h, verbose=0, draw=0, title='')`
  - `problem` – `[s0, finish_line, walls]`
  - `strategy` – one of `'bf'`, `'df'`, `'uc'`, `'gbf'`, `'a*'`
  - `h`(s, f, w) – heuristic function for racetrack problems
    - s = state, f = finish line, w = list of walls
    - `racetrack.py` converts this to the $h(s)$ function that `fsearch.main` needs
  - `verbose` – one of `0`, `1`, `2`, `3`, `4` (same as for `fsearch.py`)
  - `draw` – either `0` (draw nothing)
    or `1` (draw problems, node expansions, solutions)
  - `title` – a title to use at the top of the graphics window
    - default is the names of the strategy and heuristic
- Some subroutines that may be useful . . .

`racetrack.py` (continued)

- `intersect(e1,e2)` returns `True` if edges `e1` and `e2` intersect, `False` otherwise

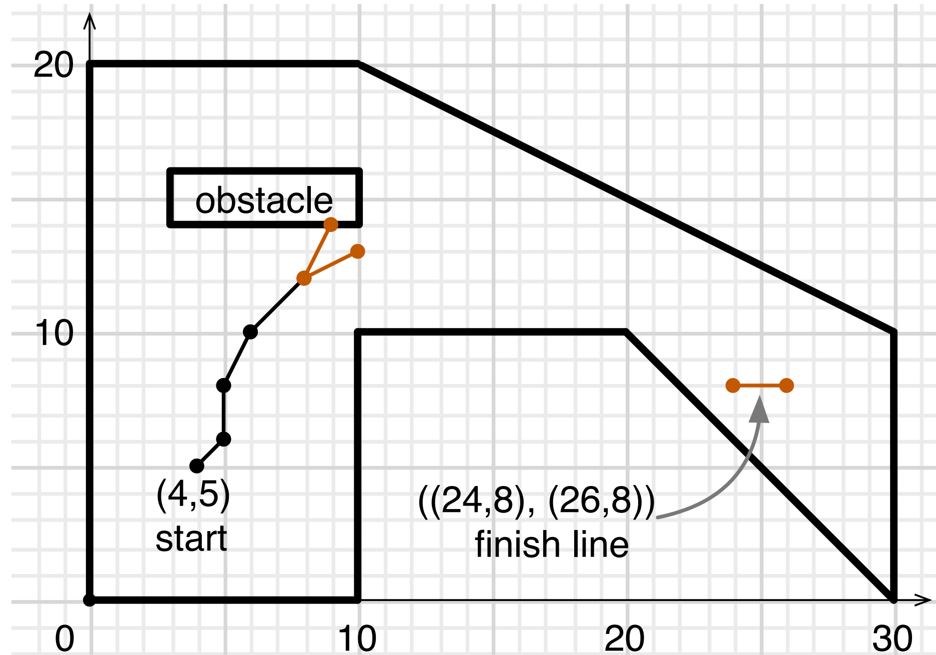
```
intersect([(0,0),(1,1)], [(0,1),(1,0)])  returns  True
intersect([(0,0),(0,1)], [(1,0),(1,1)])  returns  False
intersect([(0,0),(2,0)], [(0,0),(0,5)])  returns  True
intersect([(1,1),(6,6)], [(5,5),(8,8)])  returns  True
intersect([(1,1),(5,5)], [(6,6),(8,8)])  returns  False
```

Basic idea (except for some special cases)

- Suppose `e1` $= (p_1, p_1')$, `e2` $= (p_2, p_2')$
- Calculate the lines that contain the edges
  - $y = m_1 x + b_1;\ \ y = m_2 x + b_2$
- If $m_1 = m_2$ and $b_1 \neq b_2$ then parallel, don't intersect
- If $m_1 = m_2$ and $b_1 = b_2$ then collinear $\Rightarrow$ check for overlap
  - Does either edge have an endpoint that's inside the other edge?
- If $m_1 \neq m_2$ then calculate the intersection point p
  - The edges intersect if they both contain p

`racetrack.py` (continued)

- `crash(e,walls)`
  - `e` is an edge
  - `walls` is a list of walls
  - ▸ `True` if `e` intersects at least one wall in `walls`, else `False`

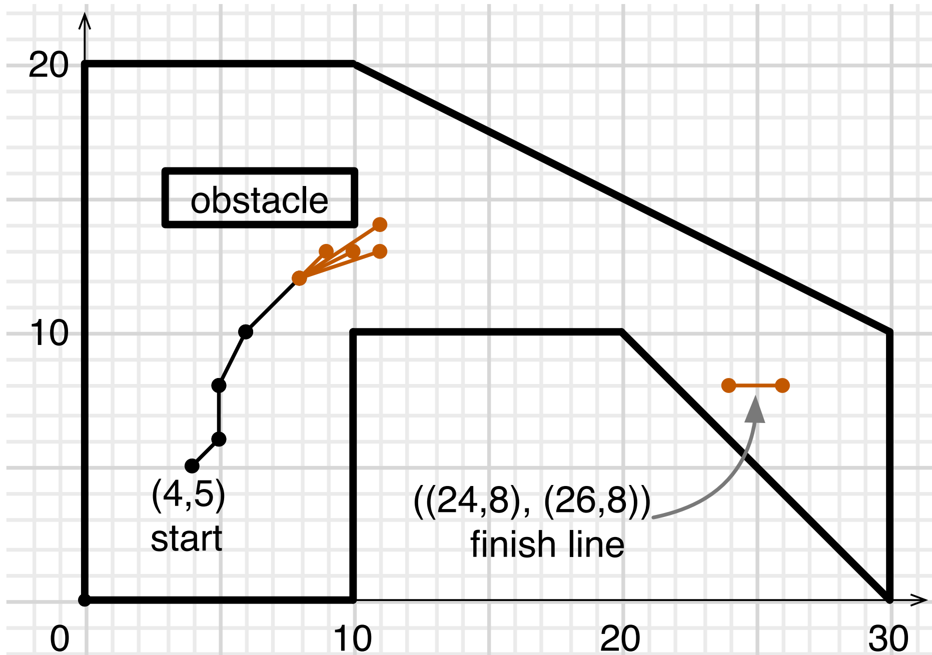


- Example:

```
crash([(8,12),(10,13)],walls)   returns  False
crash([(8,12),(9,14)],walls)   returns  True
```

`racetrack.py` (continued)

- `children(state,walls)`
  - state, list of walls
  - Returns a list $[s_1, s_2, \ldots, s_n]$
    - each s_i is a state that we can move to from `state` without crashing



- Example:
  - current state is $((8, 12), (2, 2))$
  - 9 possible states, 5 of them crash into the obstacle
  - `children(((8,12),(2,2)), walls)` returns
    `[((9,13),(1,1)), ((10,13),(2,1)), ((11,13),(3,1), ((11,14),(3,2))]`

heuristics.py

Three heuristic functions for the Racetrack domain:

- `h_edist`$(s, f, walls)$ returns the Euclidean distance from s to the goal
  - can go in the wrong direction because it ignores walls
  - can overshoot because it ignores the number of moves needed to stop
- `h_esdist`$(s, f, walls)$ is a modified version of `h_edist`
  - includes an estimate of how many moves it will take to stop
- `h_walldist`$(s, f, walls)$:

  The first time it's called, for each gridpoint that's not inside a wall it will cache a rough estimate of the length of the shortest path to the finish line. The computation is done by a breadth-first search going backwards from the finish line, one gridpoint at a time.

  On all subsequent calls, it will retrieve the cached value and add an estimate of how many moves will be needed to stop.

What to do

- Write a better heuristic function than `h_walldist`
  - Don't need to compute distances for *all* the gridpoints
    - I think it will work well to cache distances for a few "important" ones
      - e.g., points near the corners, midpoints between pairs of corners
  - Experiment to see what works best
    - Which points to cache?
    - What kind of distance to cache (Euclidean? taxicab?)
  - Performance
    - running time - probably can get a lot faster than `h_walldist`
    - A* and GBFS: length of solution, number of nodes generated
      - not sure if you can make these much better,
        but you probably don't want to make them worse ☺
- Don't just make minor modifications to `h_walldist`
  - You need to write something of your own

What to Submit

- One file, `proj1.py`
  - In the file, your heuristic function should be named `h_proj1`
- Submit it at the submit server, `submit.cs.umd.edu`

Grading

Evaluation criteria:

35% correctness: – whether your heuristic 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 the start of each function
- Comments elsewhere

35% performance

- A* and GBFS using your heuristic function:
- running time, length of solution path, number of nodes generated
- Top n performers ($n \approx 5$) will get extra credit