CMSC 722, Spring 2018, Project 1 (Revised):

Compare the FF heuristic with a domain-specific heuristic

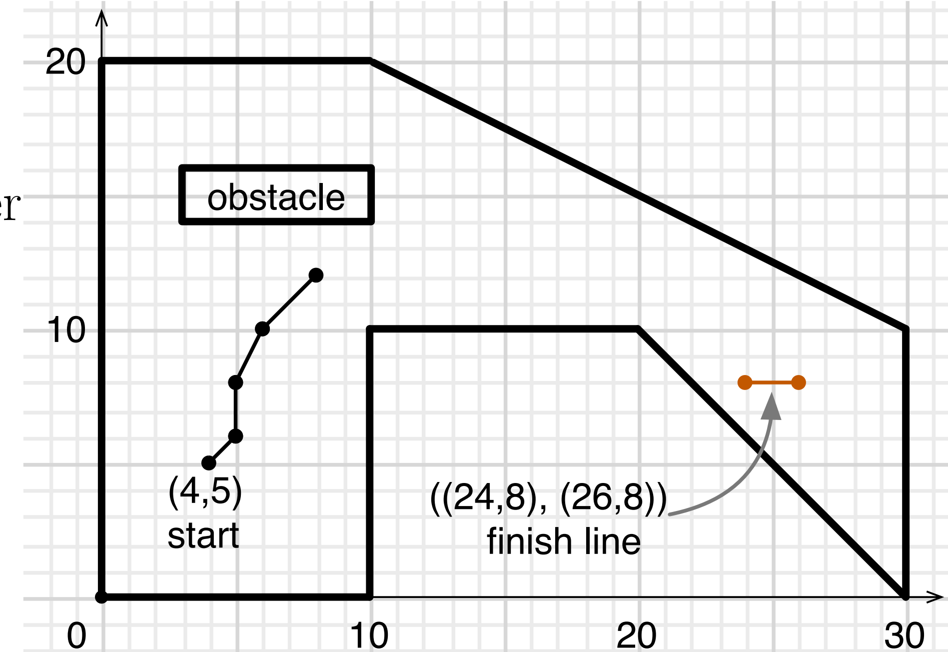
Last update February 28, 2018

- To be done individually (not in teams)

- Due date: March 14
- Late date (10% off): March 16

Problem domain

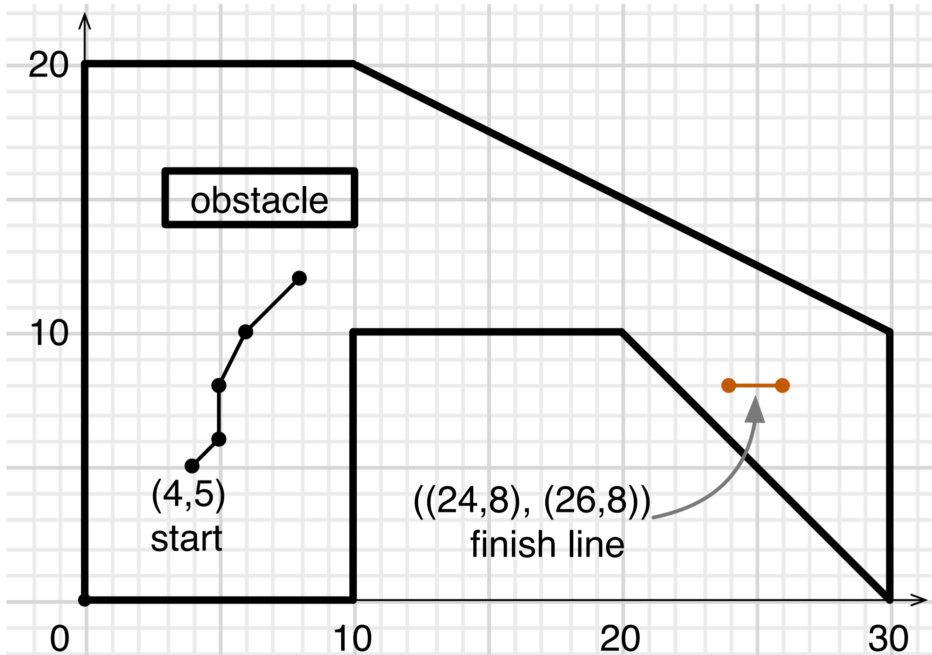
- Modified version of Racetrack
  - Invented in early 1970s
  - played by hand on graph paper



- 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

A domain-specific representation

- Later I'll discuss some state-variable representations



- Current state $s_{i-1} = (p_{i-1}, z_{i-1})$
  - 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: $p_i = (x_{i-1} + u_i, y_{i-1} + v_i)$
  - New state: $s_i = (p_i, z_i)$

Example

- Initial state:

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

- First move:

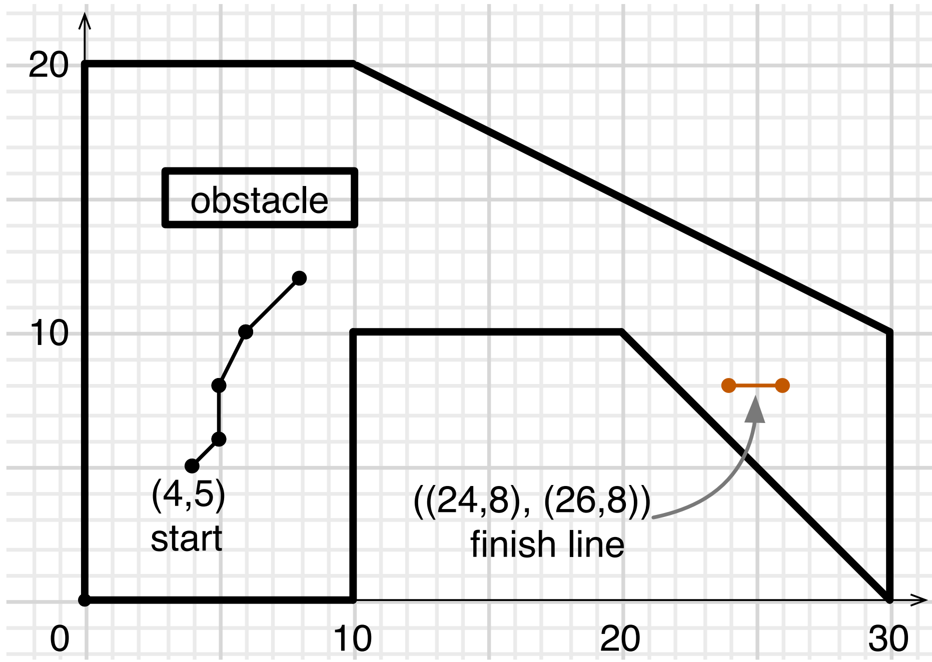
$$z_1 = (0,0) + (1,1) = (1,1)$$
$$p_1 = (4,5) + (1,1) = (5,6)$$
$$s_1 = ((5,6),\ (1,1))$$

- Second move:

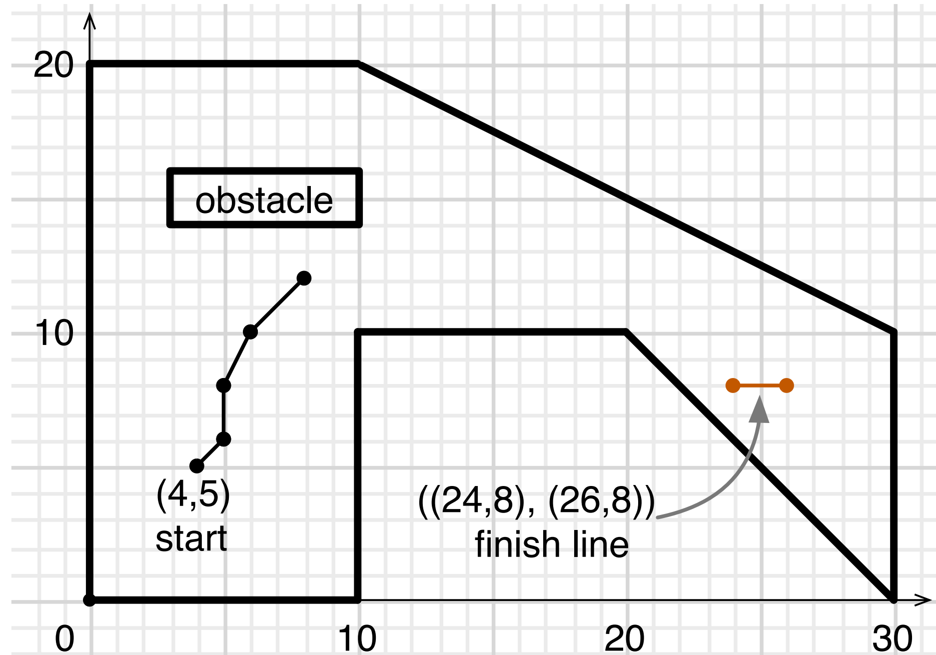
$$z_2 = (1,1) + (-1,1) = (0,2)$$
$$p_2 = (5,6) + (0,2) = (5,8)$$
$$s_2 = ((5,8),\ (0,2))$$

- 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

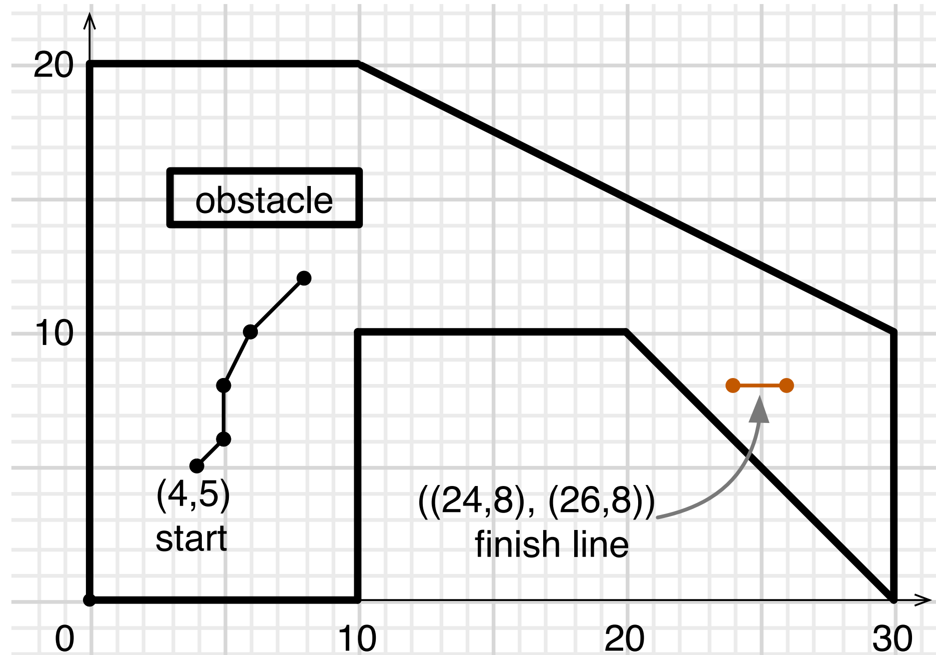


- *edge*: a pair of points (p, q)
  - $p = (x, y)$, $q = (x', y')$
    - coordinates are nonnegative integers
- *wall*: an edge that the vehicle can't cross
- List of walls in the example:

$$
\begin{array}{lll}
[((0,0),(10,0)), & ((10,0),(10,10)), & ((10,10),(20,10)), \\
((20,10),(30,0)), & ((30,0),(30,10)), & ((30,10),(10,20)), \\
((10,20),(0,20)), & ((0,20),(0,0)), & ((3,14),(10,14)), \\
((10,14),(10,16)), & ((10,16),(3,16)), & ((3,16),(3,14))]
\end{array}
$$

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



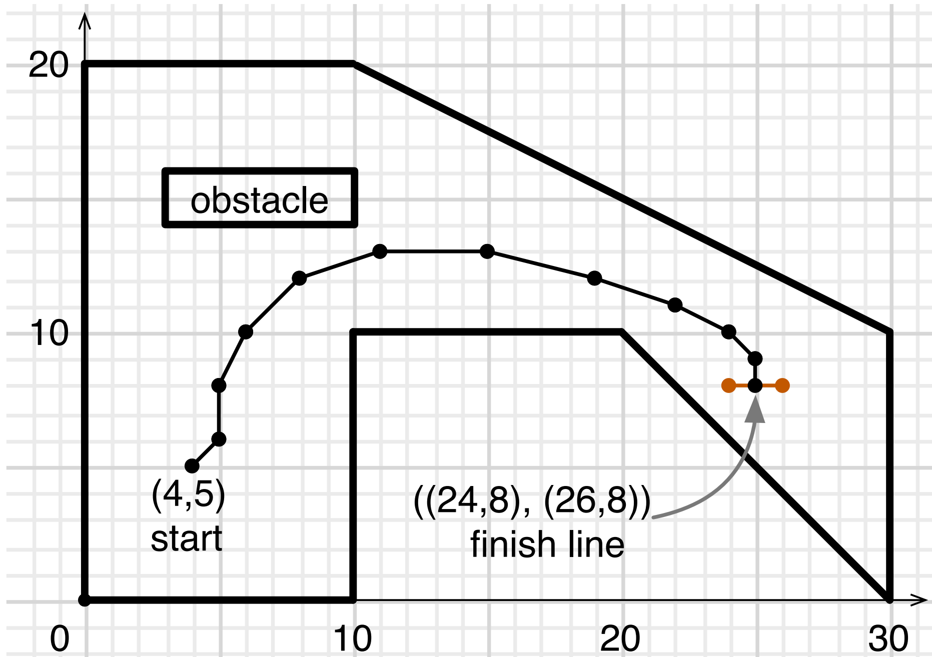
- Example:

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

- *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 as quickly as possible
  - 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: $[(4, 5),\ (5, 6),\ (5, 8),\ (6, 10),\ (8, 12),\ (11, 13),\ (15, 13),$
  $(19, 12),\ (22, 11),\ (24, 10),\ (25, 9),\ (25, 8),\ (25, 8)]$

Three state-variable representations

- Unlike the state-variable representations in the book,
  - value of a state variable can be an arbitrary data structure
  - preconditions, effects, goal tests, rigid conditions can be computational formulas
  - OK since we're doing forward search
- Representation 1: state variables `x`, `y` for location, and `u`, `v` for velocity
  - Each state variable's value is an integer
- Representation 2: state variables `p` and `z`: current location, current velocity
  - Each state variable's value is a pair of integers
- Representation 3: one state variable `s`: current state
  - Value is a 4-tuple of integers
- In each representation, the locations of the walls are rigid properties
  - just use the domain-specific representation for these

State-variable Representation 1

- state variables x, y for location, and u, v for velocity, values are integers
- action template:

  $\mathsf{move}(\delta_u, \delta_v)$ with $\text{Range}(\delta_u) = \text{Range}(\delta_v) = \{-1, 0, 1\}$

  pre: $\mathsf{safe}(\delta_u, \delta_v)$

  - where $\mathsf{safe}(\delta_u, \delta_v)$ is a procedure to test whether the edge $((\mathsf{x}, \mathsf{y}), (\mathsf{x} + \mathsf{u} + \delta_u, \mathsf{y} + \mathsf{v} + \delta_v))$ intersects any of the walls

  eff: $\mathsf{x} \leftarrow \mathsf{x} + \mathsf{u} + \delta_u,\ \ \mathsf{y} \leftarrow \mathsf{y} + \mathsf{v} + \delta_v,\ \ \mathsf{u} \leftarrow \mathsf{u} + \delta_u,\ \ \mathsf{v} \leftarrow \mathsf{v} + \delta_v$

- Initial state, if (x_0, y_0) is the starting point:
  - $s_0 = \{\mathsf{x} = x_0,\ \mathsf{y} = y_0,\ \mathsf{u} = 0,\ \mathsf{v} = 0\}$

- Goal test, if the finish line is $((q, r), (q', r'))$:
  - $\min(q, q') \le \mathsf{x} \le \max(q, q'),\ \ \min(r, r') \le \mathsf{y} \le \max(r, r'),\ \ \mathsf{u} = \mathsf{v} = 0$
  - OK since we're allowing the goal test to be an arbitrary formula

State-variable Representation 2

- state variables p, z for location and velocity, each is a pair of integers
- action template:

  $\mathsf{move}(\delta_u, \delta_v)$ with $\text{Range}(\delta_u) = \text{Range}(\delta_v) = \{-1, 0, 1\}$

  pre: $\mathsf{safe}(\delta_u, \delta_v)$

  - where $\mathsf{safe}(\delta_u, \delta_v)$ is a procedure to test whether the edge $(\mathsf{p}, \mathsf{p} + \mathsf{z} + (\delta_u, \delta_v))$ intersects any of the walls

  eff: $\mathsf{p} \leftarrow \mathsf{p} + \mathsf{z} + (\delta_u, \delta_v)$, $\mathsf{z} \leftarrow \mathsf{z} + (\delta_u, \delta_v)$

- Initial state, if (x_0, y_0) is the starting point:
  - $s_0 = \{\mathsf{p} = (x_0, y_0),\ \mathsf{z} = (0, 0)\}$

- Goal test, if the finish line is $((q, r), (q', r'))$:
  - $\min(q, q') \leq \mathsf{p}[0] \leq \max(q, q'),\ \min(r, r') \leq \mathsf{p}[1] \leq \max(r, r'),\ \mathsf{z} = (0, 0)$

State-variable Representation 3

- state variable s for current state, a 4-tuple of integers
- action template:

  $\mathsf{move}(\delta_u, \delta_v)$ with $\mathrm{Range}(\delta_u) = \mathrm{Range}(\delta_v) = \{-1, 0, 1\}$

  pre: $\mathsf{safe}(\delta_u, \delta_v)$

  - where $\mathsf{safe}(\delta_u, \delta_v)$ is a procedure to test whether the edge $((\mathsf{s}[0], \mathsf{s}[1]), (\mathsf{s}[0] + \mathsf{s}[2] + \delta_u, \mathsf{s}[1] + \mathsf{s}[3] + \delta_v))$ intersects any walls

  eff: $\mathsf{s} \leftarrow (\mathsf{s}[0] + \mathsf{s}[2] + \delta_u,\ \mathsf{s}[1] + \mathsf{s}[3] + \delta_v,\ \mathsf{s}[2] + \delta_u,\ \mathsf{s}[3] + \delta_v)$

- Initial state, if (x_0, y_0) is the starting point:
  - $s_0 = \{\mathsf{s} = (x_0, y_0, 0, 0)\}$

- Goal test, if the finish line is $((q, r), (q', r'))$:

$$\min(q, q') \le \mathsf{s}[0] \le \max(q, q'),\ \min(r, r') \le \mathsf{s}[1] \le \max(r, r'),\ \mathsf{s}[2] = \mathsf{s}[3] = 0$$

h^{FF} is better with some reps. than others

Suppose we start at $s_0 = ((1,1),(0,0))$, and choose $(u', v') = (1,1)$

- Representation 1:
  - $\gamma^+(s_0, a) = \{\mathsf{x} = 1,\ \mathsf{x} = 2,\ \mathsf{y} = 1,\ \mathsf{y} = 2,\ \mathsf{u} = 0,\ \mathsf{u} = 1,\ \mathsf{v} = 0,\ \mathsf{v} = 1\}$
    - 16 possible states
  - RPG will converge quickly but probably won't be very informative
- Representation 2:
  - $\gamma^+(s_0, a) = \{\mathsf{p} = (1,1),\ \mathsf{p} = (2,2),\ \mathsf{z} = (0,0),\ \mathsf{z} = (1,1)\}$
    - 4 possible states; I think this representation will work OK
- Representation 3:
  - $s_0 = \{\mathsf{s} = (1,1,0,0)\}$
  - $\gamma^+(s_0, a) = \{\mathsf{s} = (1,1,0,0), \mathsf{s} = (2,2,1,1)\}$
    - 2 possible states
  - RPG will do breadth-first search of the entire planning problem

Things I'll provide

I'll post a zip archive that includes the following:

- Domain-independent code (in Python 3.6):
  - `fsearch.py` – forward search algorithm
    - can do DFS, BFS, uniform-cost search, A*, and GBFS
    - has hooks for calling user-supplied code to draw search spaces
- Domain-specific code
  - `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` – a customizable `bash` script for running experiments

Here are some details ...

Contents of `fsearch.py`

Domain-independent forward-search algorithm

- `main(s0, next_states, goal_test, strategy,`
  `h = None, verbose = 2, draw_edges = None)`
  - `s0` – initial state, in whatever representation you're using, e.g,
    - for your FF heuristic, one of the state-variable representations
    - for the heuristic I give you, domain-specific representation
  - `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`
    - amount of verbosity in the output (see documentation in the file)
  - `draw_edges` – function to draw edges in the search space

Contents of `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 (look at the code for this; you'll need something similar for `next_state`)
  - `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 of the subroutines in `racetrack.py` may be useful ...

Contents of `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`

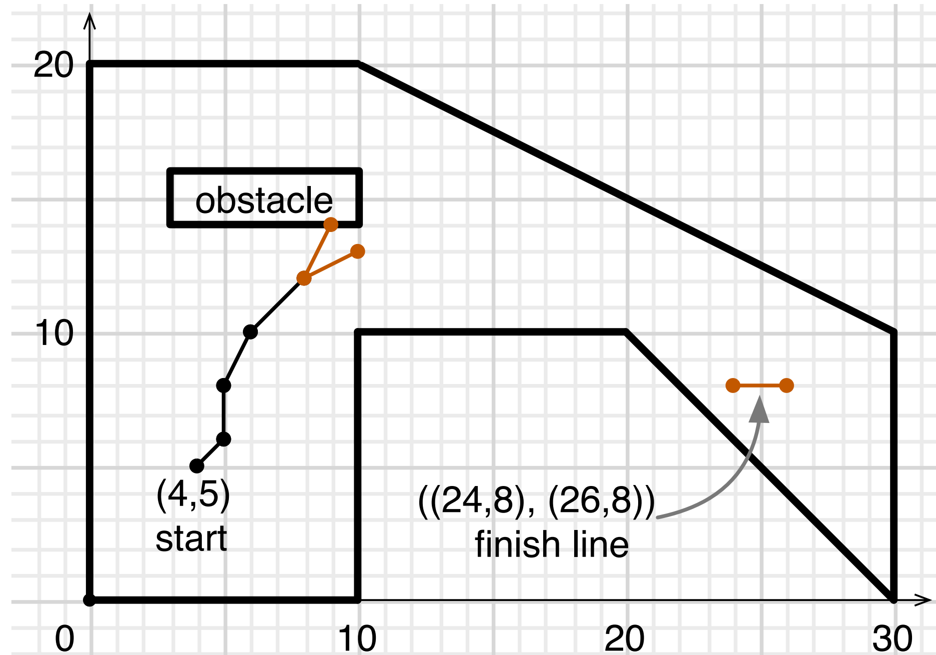
Basic idea (except for some special cases)

- 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
- If $m_1 \neq m_2$ then is intersection point contained in both edges?

- Can use this to implement the `goal_test` function for `fsearch.main`

Contents of `racetrack.py` (continued)

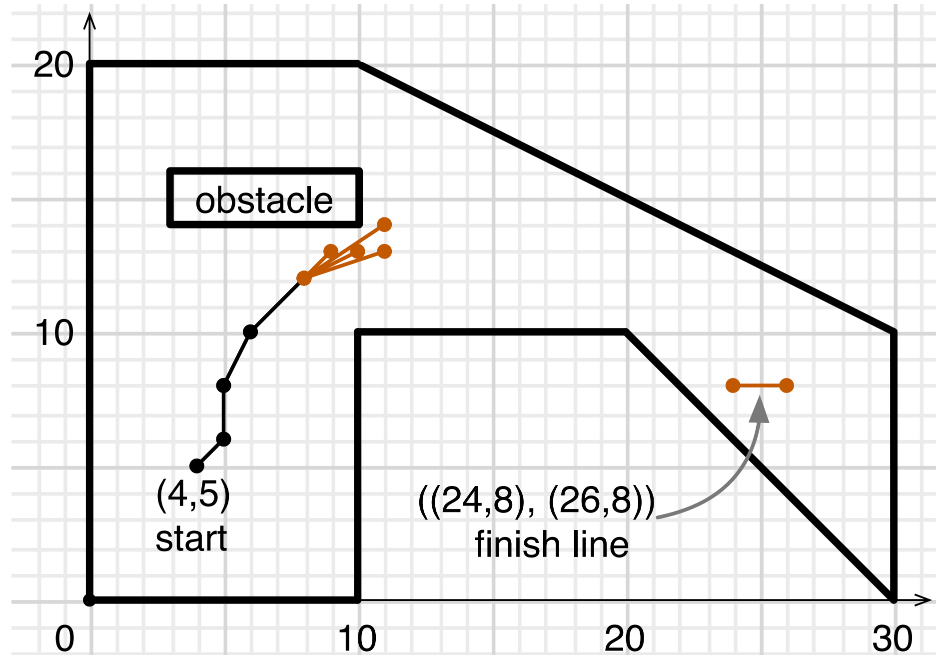
- `crash(e,walls)`
  - `e` is an edge
  - `walls` is a list of walls
  - ▸ `True` if `e` intersects a wall in `walls`, else `False`



- Example:

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

Contents of `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 go 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))]`
- Can use this to implement the `next_states` function for `fsearch.main`
  - That's a kluge since it's domain-specific, but I'll allow it anyway

Contents of `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 howmany 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 you need to do

1. Write a Python function `ff1`$(s, f, walls)$ that returns a minimal relaxed solution $\hat{\pi}$ for the racetrack problem $(s, f, walls)$.

Above, $\hat{\pi}$ is the same list $[\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k]$ that *would be computed* by HFF (Algorithm 2.3) in the book or the pseudocode on page 54 of my `chap2b.pdf` lecture slides, if $(s, f, walls)$ were represented in state variable representation 1. However, you aren't required to use that pseudocode, and you aren't required to use state-variable representation (at least, not explicitly). It's OK for you to write your own *ad hoc* algorithm, as long as it returns correct answers.

In the minimal relaxed solution $\hat{\pi}$ that your algorithm returns, each $\hat{a}_i$ should be a list of actions $\hat{a}_i = [a_{i1}, a_{i2}, \ldots]$ in which each action a_{ij} is represented by just giving its arguments, e.g., use the pair `(1,-1)` to represent the action `move(1,-1)`. Since all the actions are `move` actions, this representation is unambiguous.

2. Write a Python function `h1_ff1`$(s, g, walls)$ that calls `ff1`$(s, g, walls)$ to get the minimal relaxed solution $\hat{\pi}$, and returns the total number of actions in $\hat{\pi}$.

What you need to do (continued)

3. Write a Python function `ff2`$(s, f, walls)$ that has the same description as `ff1`$(s, f, walls)$, with the phrase "state variable representation 1" replaced by "state variable representation 2".

4. Write a Python function `h1_ff2`$(s, g, walls)$ that calls `ff2`$(s, g, walls)$ to get the minimal relaxed solution $\hat{\pi}$, and returns the total number of actions in $\hat{\pi}$.

Note: what "minimal" means

In the pseudocode for computing a minimal relaxed solution, if $\hat{a}_i$ is a minimal set of actions that r-achieves $\hat{g}_i$, this means there's no proper subset of $\hat{a}_i$ that r-achieves $\hat{g}_i$. It's OK if there's a smaller set of actions $\hat{a}'_i$ that r-achieves $\hat{g}_i$, as long as $\hat{a}'_i$ isn't a subset of $\hat{a}_i$.

There may be several minimal sets of actions that r-achieve $\hat{g}_i$, and they may have different sizes. Just take the first one you find; it doesn't matter whether or not it's the smallest one.

What you need to do (continued)

5. Do experiments to measure GBFS's performance as a function of problem size, using each of the following heuristics:
    - `h_ff1`
    - `h_ff2`
    - `h_esdist`
    - `h_walldist`

   ▸ Later we'll give some guidelines about what ranges of problem size to use

5. Write a report giving the results of your experiments
    - Include the plots and table described on the next two pages
    - For each plot or table, tell what you can conclude from it and why

   ▸ Format:
    - US letter paper, single column, 1-inch margins on all sides
    - Font size at least 11pt
    - At most 3 pages (or 4 if you do the extra-credit part)

Things to include in the report

(a) a semi-log plot that shows, for each heuristic, the total CPU time for GBFS as a function of problem size.

- Each data point: average of ≥ 10 randomly generated problems

(b) Same as (a), but instead of CPU time, show number of nodes generated

(c) A scatter plot for a subset of the Racetrack problems in `sample_probs.py` (I'll post a modified version of the file)

- For each problem, plot a point (x, y), where
  - x = number of nodes GBFS generates using `h_walldist`
  - y = number of nodes GBFS generates using `h_ff2`

(e) A table showing the exact numbers in the scatter plot:

  - Column 1: problem name
  - Column 2: number of nodes GBFS generates using `h_walldist`
  - Column 3: number of nodes GBFS generates using `h_ff2`

- The problems should be in the same order as in the `sample_probs.py` file

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 start of file and in each function; comments elsewhere
  - 35% on the report itself
    - Adequacy of your experiments, statistical significance, clarity of presentation, quality of conclusions
- Extra credit:
  - Develop a better (for `h_ff`) state-variable representation for the Racetrack domain, and include it in your experiments
  - Caveat: I'm not sure whether that's feasible