CS287-H AHRI HW2

In HW2 we will walk through Maximum Entropy inverse reinforcement learning, intent inference, and intent expression in a simple gridworld environment.

In [1]:
```
# Preliminaries
import numpy as np
from enum import IntEnum
import itertools as iter
import random
import matplotlib.pyplot as plt
import matplotlib
from scipy import stats
import math
import warnings
warnings.filterwarnings('ignore', category=UserWarning, module='matplotlib')
```

First, we will define a simple grid world environment with obstacles. In this world, each grid location can be either free or occupied by an obstacle. The agent starts at a specified start and has to end at one of the specified goal locations. It can move up, down, left, and right, as long as it doesn't leave the grid and it doesn't enter an occupied grid location.

In [2]:
```
# Defining the agent grid world.

class Actions(IntEnum):
    UP = 0
    DOWN = 1
    LEFT = 2
    RIGHT = 3

class AgentGridworld(object):
    """
    An X by Y gridworld class for an agent in an environment with obstacles.
    """
    Actions = Actions

    def __init__(self, X, Y, obstacles, start, goals):
        """
        Params:
            X [int] -- The width of this gridworld.
            Y [int] -- The height of this gridworld.
            start [tuple] -- Starting position specified in coords (x, y).
            goals [list of tuple] -- List of goal positions specified in coords
            obstacles [list] -- List of axis-aligned 2D boxes that represent
                obstacles in the environment for the agent. Specified in coords
                [[(lower_x, lower_y), (upper_x, upper_y)], [...]]
        """

        assert isinstance(X, int), X
        assert isinstance(Y, int), Y
        assert X > 0
        assert Y > 0
```

```
        # Set up variables for Agent Gridworld
        self.X = X
        self.Y = Y
        self.S = X * Y
        self.A = len(Actions)
        self.start = start
        self.goals = goals

        # Set the obstacles in the environment.
        self.obstacles = obstacles


    ###########################
    #### Utility functions ####
    ###########################

    def traj_construct(self, start, goal):
        """
        Construct all trajectories between a start and goal of the shortest ler
        Params:
            start [tuple] -- Starting position specified in coords (x, y).
            goal [tuple] -- Goal position specified in coords (x, y).
        Returns:
            trajs [list] -- Trajectories between start and goal in states (s).
        """
        trajs = []
        def recurse_actions(s_curr, timestep):
            # Recursive action combo construction. Select legal combinations.
            if timestep == T-1:
                if s_curr == s_goal:
                    trajs.append(list(traj))
            else:
                rand_actions = [a for a in Actions]
                random.shuffle(rand_actions)
                for a in rand_actions:
                    s_prime, illegal = self.transition_helper(s_curr, a)
                    if not illegal:
                        traj[timestep+1] = s_prime
                        recurse_actions(s_prime, timestep+1)

        s_start = self.coor_to_state(start[0], start[1])
        s_goal = self.coor_to_state(goal[0], goal[1])
        T = abs(start[0] - goal[0]) + abs(start[1] - goal[1]) + 1
        traj = [None] * T
        traj[0] = s_start
        recurse_actions(s_start, 0)

        return trajs

    def transition_helper(self, s, a):
        """
        Given a state and action, apply the transition function to get the next
        Params:
            s [int] -- State.
            a [int] -- Action taken.
        Returns:
            s_prime [int] -- Next state.
            illegal [bool] -- Whether the action taken was legal or not.
        """
        x, y = self.state_to_coor(s)
```

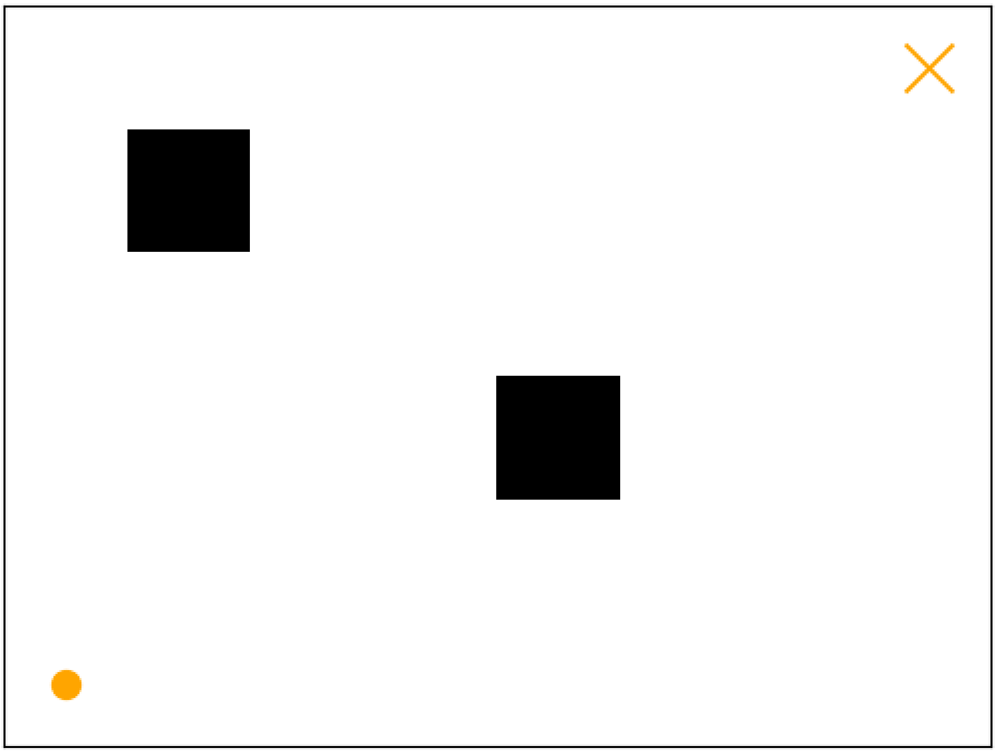
```
        assert 0 <= a < self.A

        x_prime, y_prime = x, y
        if a == Actions.LEFT:
            x_prime = x - 1
        elif a == Actions.RIGHT:
            x_prime = x + 1
        elif a == Actions.DOWN:
            y_prime = y + 1
        elif a == Actions.UP:
            y_prime = y - 1
        elif a == Actions.UP_LEFT:
            x_prime, y_prime = x - 1, y - 1
        elif a == Actions.UP_RIGHT:
            x_prime, y_prime = x + 1, y - 1
        elif a == Actions.DOWN_LEFT:
            x_prime, y_prime = x - 1, y + 1
        elif a == Actions.DOWN_RIGHT:
            x_prime, y_prime = x + 1, y + 1
        elif a == Actions.ABSORB:
            pass
        else:
            raise BaseException("undefined action {}".format(a))

        illegal = False
        if x_prime < 0 or x_prime >= self.X or y_prime < 0 or y_prime >= self.Y
            illegal = True
            s_prime = s
        else:
            s_prime = self.coor_to_state(x_prime, y_prime)
            if self.is_blocked(s_prime):
                illegal = True
        return s_prime, illegal

    def get_action(self, s, sp):
        """
        Given two neighboring waypoints, return action between them.
        Params:
            s [int] -- First waypoint state.
            sp [int] -- Next waypoint state.
        Returns:
            a [int] -- Action taken.
        """
        x1, y1 = self.state_to_coor(s)
        x2, y2 = self.state_to_coor(sp)

        if x1 == x2:
            if y1 == y2:
                return Actions.ABSORB
            elif y1 < y2:
                return Actions.DOWN
            else:
                return Actions.UP
        elif x1 < x2:
            if y1 == y2:
                return Actions.RIGHT
            elif y1 < y2:
                return Actions.DOWN_RIGHT
            else:
                return Actions.UP_RIGHT
```

```
        else:
            if y1 == y2:
                return Actions.LEFT
            elif y1 < y2:
                return Actions.DOWN_LEFT
            else:
                return Actions.UP_LEFT

    def is_blocked(self, s):
        """
        Returns True if s is blocked.
        By default, state-action pairs that lead to blocked states are illegal.
        """
        if self.obstacles is None:
            return False

        # Check against internal representation of boxes.
        x, y = self.state_to_coor(s)
        for box in self.obstacles:
            if x >= box[0][0] and x <= box[1][0] and y >= box[1][1] and y <= bo
                return True
        return False

    def visualize_grid(self):
        """
        Visualize the world with its obstacles.
        """
        self.visualize_demos([])

    def visualize_demos(self, demos):
        """
        Visualize the world with its obstacles and given demonstration.
        """
        # Create world with obstacles on the map.
        world = 0.5*np.ones((self.Y, self.X))

        # Add obstacles in the world in opaque color.
        for obstacle in self.obstacles:
            lower = obstacle[0]
            upper = obstacle[1]
            world[upper[1]:lower[1]+1, lower[0]:upper[0]+1] = 1.0

        fig1, ax1 = plt.subplots()
        plt.imshow(world, cmap='Greys', interpolation='nearest')

        # Plot markers for start and goal
        plt.scatter(self.start[0], self.start[1], c="orange", marker="o", s=10(
        for goal in self.goals:
            plt.scatter(goal[0], goal[1], c="orange", marker="x", s=300)

        # Plot demonstrations
        for t, demo in enumerate(demos):
            demo_x = []
            demo_y = []
            for s in demo:
                x, y = self.state_to_coor(s)
                demo_x.append(x)
                demo_y.append(y)
            step = t/float(len(demos)+1)
            col = ((1*step), (0*step), (0*step))
```

```
        plt.plot(demo_x,demo_y, c=col)

    plt.xticks(range(self.X), range(self.X))
    plt.yticks(np.arange(-0.5,self.Y+0.5),range(self.Y+1))
    ax1.set_yticklabels([])
    ax1.set_xticklabels([])
    ax1.set_yticks([])
    ax1.set_xticks([])
    ax = plt.gca()
    plt.minorticks_on
    ax.grid(True, which='both', color='black', linestyle='-', linewidth=2)
    plt.show(block=False)


#################################
# Conversion functions
#################################
# Helper functions convert between state number ("state") and discrete coor
#
# State number ("state"):
# A state `s` is an integer such that 0 <= s < self.S.
#
# Discrete coordinates ("coor"):
# `x` is an integer such that 0 <= x < self.X. Increasing `x` corresponds t
# `y` is an integer such that 0 <= y < self.Y. Increasing `y` corresponds t
#
#################################

def state_to_coor(self, s):
    """
    Params:
        s [int] -- The state.
    Returns:
        x, y -- The discrete coordinates corresponding to s.
    """
    assert isinstance(s, int)
    assert 0 <= s < self.S
    y = s % self.Y
    x = s // self.Y
    return x, y

def coor_to_state(self, x, y):
    """
    Convert discrete coordinates into a state, if that state exists.
    If no such state exists, raise a ValueError.
    Params:
        x, y [int] -- The discrete x, y coordinates of the state.
    Returns:
        s [int] -- The state.
    """

    x, y = int(x), int(y)
    if not(0 <= x < self.X):
        raise ValueError(x, self.X)
    if not (0 <= y < self.Y):
        raise ValueError(y, self.Y)

    return (x * self.Y) + (y)
```

Let's create a 6x8 grid world with 2 obstacles. For now, let's set the start in the lower left corner and a goal in the upper right corner.

In [3]:
```
# Build grid world.
sim_height = 6
sim_width = 8
obstacles = [[[1,1], [1,1]] , [[4,3],[4,3]]]
start = [0, 5]
goals = [[7, 0]]
gridworld = AgentGridworld(sim_width, sim_height, obstacles, start, goals)
gridworld.visualize_grid()
```



Something that will become very important later on is a set of all of the feasible trajectories of fixed length between the start and goal, $\xi \in \Xi$:

In [4]:
```
SG_trajs = gridworld.traj_construct(start, goals[0])
```

Simulated Human

Now let's create a simulated human to produce some demonstrations that our agent will learn from. The human has access to the grid world, and, therefore, to the trajectories $\xi \in \Xi$ that are feasible in the grid world, but we need to implement a way of selecting amongst those trajectories. We need two components for this: a **cost function**, and an **observation model**.

The Cost Function

The cost function $C : \Xi \rightarrow \mathbb{R}$ maps trajectories to a real-numbered score and defines how much the human prefers some trajectories in the grid world over the others. To make computation tractable, this cost function is typically parameterized by some *weight vector* θ : C_θ. While the cost could operate directly on the state trajectory, it is more common (also for tractability's sake) to write it as a function of *features* $\Phi : \Xi \rightarrow \mathbb{R}^n$ - aspects of behavior that the person might care about. For example, the feature vector of a trajectory $\Phi(\xi)$ could be the trajectory's total distance to an obstacle, its jerk, its average x coordinate, etc. In class, we saw the cost written as a linear combination of features $C_\theta = \theta^T \Phi$, and we'll use this same cost structure in the homework.

We can think of θ as a parameter that determines how much the human prioritizes certain features over others. We will play with different weight vectors θ later on and see how that affects the demonstrations produced. Before we do this, we need to define what features Φ matter to the simulated human we're creating. The implementation below already defines some example features; **fill in the `dist_to_obstacles` and `dist_to_goals` functions**, which should implement an Euclidean distance to all obstacles and goals in the environment, respectively.

In [5]:
```
#################################
#### Featurization functions ####
#################################

####### Obstacles Feature #######

def obstacles_feature(traj):
    """
    Compute obstacle feature values for all obstacles and the entire trajectory
    The obstacle feature consists of distance from the obstacle.
    Params:
        traj [list] -- The trajectory.
    Returns:
        obstacle_feat [list] -- A list of obstacle features per obstacle.
    """
    obstacle_feat = np.zeros(len(gridworld.obstacles))
    for s in traj:
        obstacle_feat += np.asarray(dist_to_obstacles(s))
    return obstacle_feat.tolist()

def dist_to_obstacles(s):
    """
    Compute distance from state s to the obstacles in the environment.
    Params:
        s [int] -- The state.
    Returns:
        distances [list] -- The distance to the obstacles in the environment.
    """
    x, y = gridworld.state_to_coor(s)
    distances = []

    # Compute delta x and delta y in distance from obstacle
    for obstacle in gridworld.obstacles:
        # YOUR CODE HERE
        distances.append(math.dist([x,y], [obstacle[0][0],obstacle[1][1]]))
```

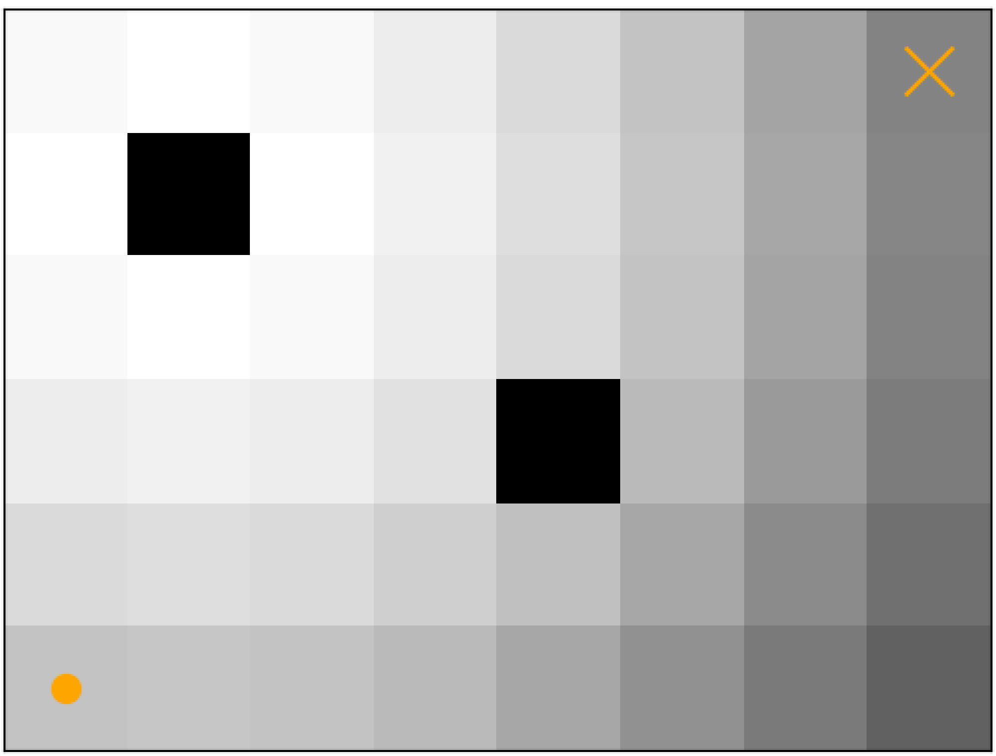
```python
    return distances

####### Goal Feature #######

def goals_feature(traj):
    """
    Compute goal feature values for all goals and the entire traj.
    The goal feature consists of distance from the obstacle.
    Params:
        traj [list] -- The trajectory.
    Returns:
        goal_feat [list] -- The distance to the goals in the environment.
    """
    goal_feat = np.zeros(len(gridworld.goals))
    for s in traj:
        goal_feat += np.asarray(dist_to_goals(s))
    return goal_feat.tolist()

def dist_to_goals(s):
    """
    Compute distance from state s to the goal in the environment.
    Params:
        s [int] -- The state.
    Returns:
        distance [float] -- The distance to the goals in the environment.
    """
    x, y = gridworld.state_to_coor(s)
    distances = []

    for goal in gridworld.goals:
        # YOUR CODE HERE
        distances.append(math.dist([x,y], [goal[0],goal[1]]))
    return distances

####### Coordinate Features #######

def average_x_feature(traj):
    """
    Compute average x feature value for the entire trajectory.
    Params:
        traj [list] -- The trajectory.
    Returns:
        avgx_feat [float] -- The average x feature value for entire traj.
    """
    x_coords = [gridworld.state_to_coor(s)[0] for s in traj]
    return np.mean(x_coords)

def average_y_feature(traj):
    """
    Compute average y feature value for the entire trajectory.
    Params:
        traj [list] -- The trajectory.
    Returns:
        avgy_feat [float] -- The average y feature value for entire traj.
    """
    y_coords = [gridworld.state_to_coor(s)[1] for s in traj]
    return np.mean(y_coords)


####### Utils #######
```

```python
def featurize(traj, feat_list, scaling_coeffs=None):
    """
    Computes the user-defined features for a given trajectory.
    Params:
        traj [list] -- A list of states the trajectory goes through.
    Returns:
        features [array] -- A list of feature values.
    """
    features = []
    for feat in range(len(feat_list)):
        if feat_list[feat] == 'goals':
            features.extend(goals_feature(traj))
        elif feat_list[feat] == 'obstacles':
            features.extend(obstacles_feature(traj))
        elif feat_list[feat] == 'avgx':
            features.append(average_x_feature(traj))
        elif feat_list[feat] == 'avgy':
            features.append(average_y_feature(traj))
    if scaling_coeffs is not None:
        for feat in range(len(features)):
            features[feat] = (features[feat] - scaling_coeffs[feat]["min"]) / (
    return np.asarray(features)

def feat_scale_construct(feat_list):
    """
    Construct scaling constants for the features available.
    """
    # First featurize all trajectories with non-standard features.
    Phi_nonstd = np.array([featurize(xi, feat_list) for xi in SG_trajs])

    # Compute scaling coefficients depending on what feat_scaling is
    scaling_coeffs = []
    for Phi in Phi_nonstd.T:
        min_val = min(Phi)
        max_val = max(Phi)
        coeffs = {"min": min_val, "max": max_val}
        scaling_coeffs.append(coeffs)
    return scaling_coeffs

def visualize_feature(feat_vals, idx):
    """
    Visualize the world with its obstacles and given demonstration.
    """
    # Create world with obstacles on the map.
    world = np.ones((gridworld.Y, gridworld.X))
    for s in range(gridworld.S):
        x, y = gridworld.state_to_coor(s)
        world[y][x] = feat_vals[s][idx]

    # Add obstacles in the world in opaque color.
    for obstacle in gridworld.obstacles:
        lower = obstacle[0]
        upper = obstacle[1]
        world[upper[1]:lower[1]+1, lower[0]:upper[0]+1] = 10.0

    fig1, ax1 = plt.subplots()
    plt.imshow(world, cmap='Greys', interpolation='nearest')

    # Plot markers for start and goal
```

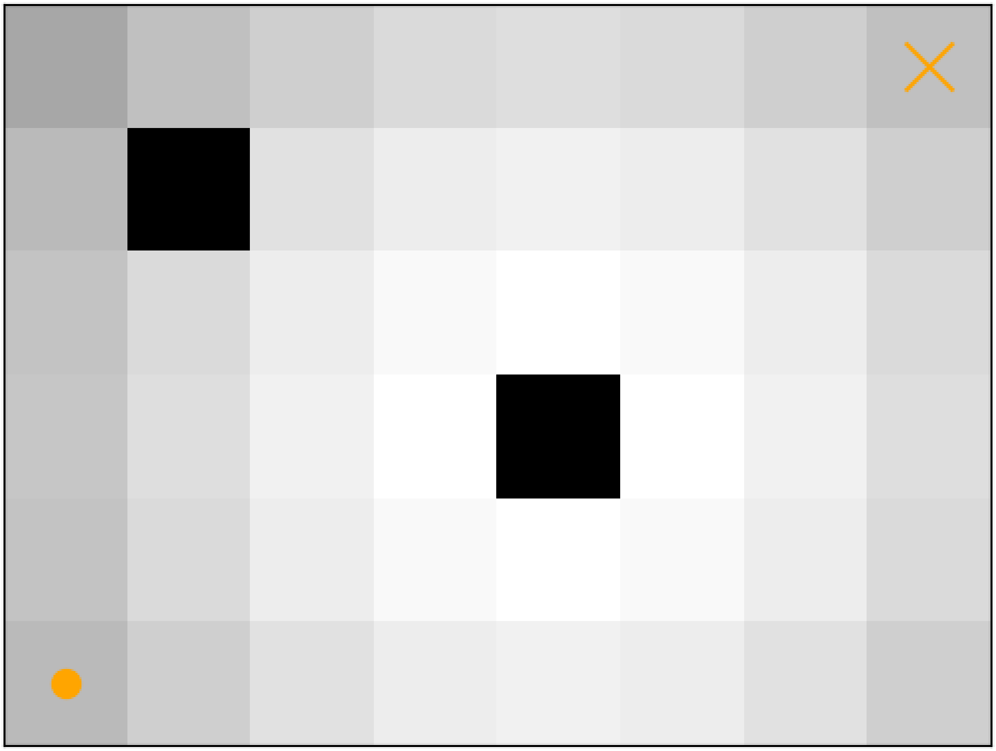
```
    plt.scatter(gridworld.start[0], gridworld.start[1], c="orange", marker="o",
    for goal in gridworld.goals:
        plt.scatter(goal[0], goal[1], c="orange", marker="x", s=300)

    plt.xticks(range(gridworld.X), range(gridworld.X))
    plt.yticks(np.arange(-0.5,gridworld.Y+0.5),range(gridworld.Y+1))
    ax1.set_yticklabels([])
    ax1.set_xticklabels([])
    ax1.set_yticks([])
    ax1.set_xticks([])
    ax = plt.gca()
    plt.minorticks_on
    ax.grid(True, which='both', color='black', linestyle='-', linewidth=2)
    plt.show(block=False)
```

Now let's visualize each feature to make sure your implementation is correct.

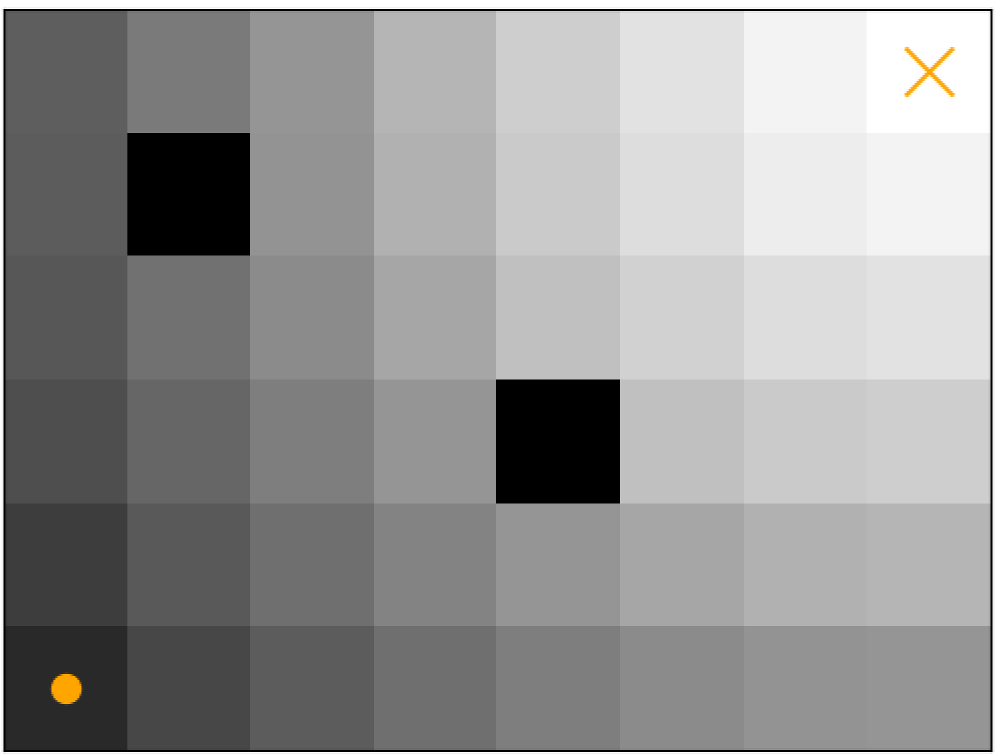
In [6]:
```
# Left obstacle. You should see the cells get darker the farther they are from
feat_list = ["obstacles"]
state_feat_vals = [featurize([s], feat_list) for s in range(gridworld.S)]
visualize_feature(state_feat_vals, 0)
```



In [7]:
```
# Right obstacle. You should see the cells get darker the farther they are from
feat_list = ["obstacles"]
state_feat_vals = [featurize([s], feat_list) for s in range(gridworld.S)]
visualize_feature(state_feat_vals, 1)
```



In [8]:
```
# Goal. You should see the cells get darker the farther they are from the goal.
feat_list = ["goals"]
state_feat_vals = [featurize([s], feat_list) for s in range(gridworld.S)]
visualize_feature(state_feat_vals, 0)
```



The Observation Model

In class, we learned about the Maximum Entropy IRL observation model - the Boltzmann model - where trajectories are chosen in proportion to their exponentiated negative cost:

$$P(\xi \mid \theta, \beta) = \frac{e^{-\beta\theta^T\Phi(\xi)}}{\int e^{-\beta\theta^T\Phi(\bar{\xi})}d\bar{\xi}} \quad .$$

Here, β is an inverse temperature parameter that controls how rational or noisy the human is when giving demonstrations: high βs produce demonstrations closer to optimal, while lower ones produce noisier demonstrations.

We will now implement the Boltzmann noisy rationality model for a given θ and β. **Fill in the `observation_model` function below**, assuming that $\Phi(\xi)$ and $\Phi(\bar{\xi}), \forall\bar{\xi} \in \Xi$ are passed in already as arguments.

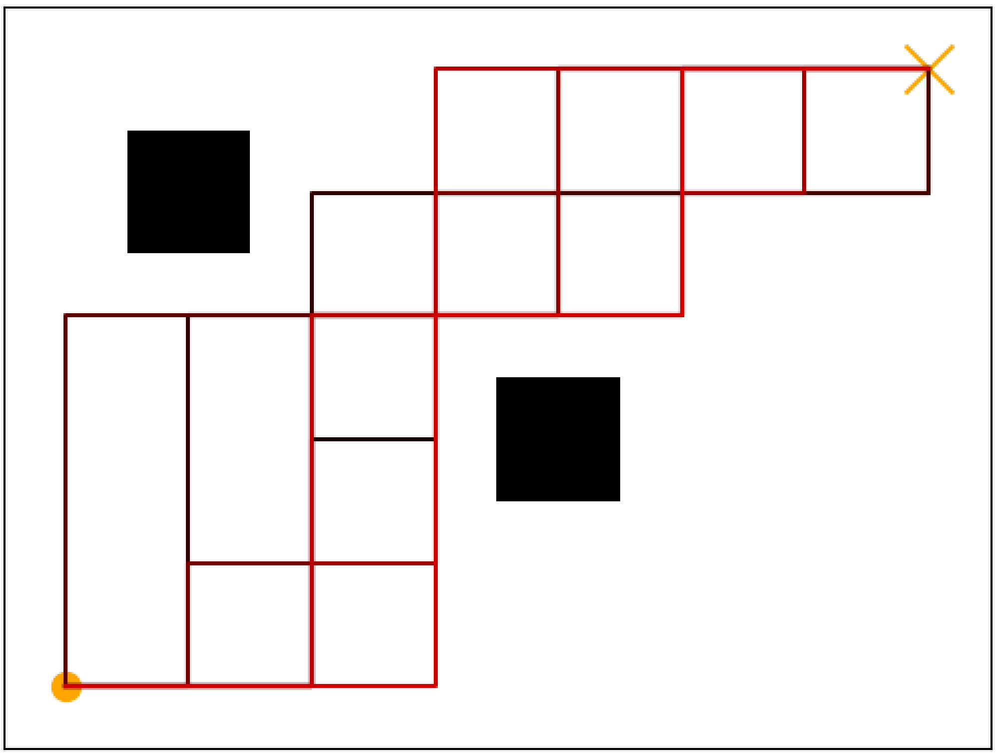
In [9]:
```
def observation_model(Phi_xi, Phi_xibar, theta, beta):
    """
    Finds observation model for given demonstrated features, using initialized
    Params:
        Phi_xi [array] -- The cost features for an observed trajectory.
        Phi_xibar [list] -- A list of the cost features for all trajectories ir
        theta [list] -- The preference parameter.
        beta [float] -- The rationality coefficient.
    Returns:
        P_xi_bt [float] -- P(xi | theta, beta)
    """
    # YOUR CODE HERE
    numerator = np.exp(-beta*np.dot(theta, Phi_xi))
    denominator = sum([np.exp(-beta*np.dot(theta, b)) for b in Phi_xibar])
    P_xi_bt = numerator / denominator
    return P_xi_bt
```

Given the cost model and the observation model, the simulated human can now sample demonstrations. The function below does this by generating the feature vector for all $\xi \in \Xi$, computing the Boltzmann probability for all of them, and sampling according to those probabilities.

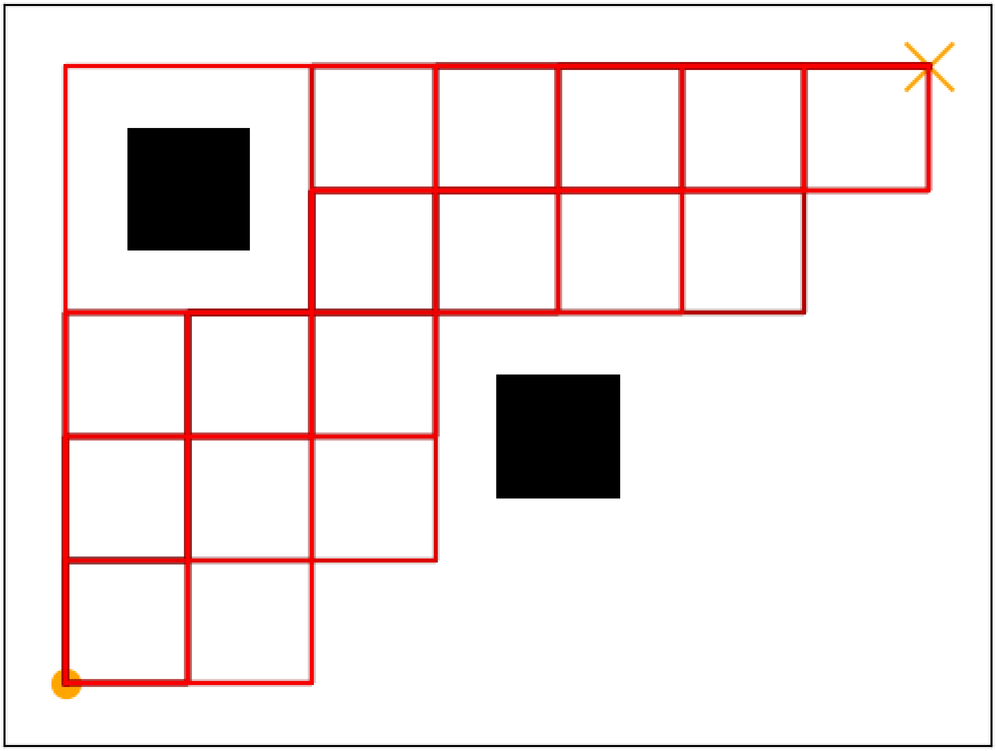
In [10]:
```
def sample_demonstrations(theta, beta, samples):
    """
    Sample <samples> demonstrations for a given theta and beta.
    Params:
        theta [list] -- The preference parameter.
        beta [float] -- The rationality coefficient.
        samples [int] -- Number of demonstrations to be sampled.
    """
    # Generate feature values for all trajectories in the gridworld.
    Phi_xibar = [featurize(xi, feat_list, scaling_coeffs) for xi in SG_trajs]

    # Create the xi observation model for all trajectories.
    P_xi = [observation_model(Phi, Phi_xibar, theta, beta) for Phi in Phi_xibar

    # Sample <samples> trajectories using this distribution.
    traj_idx = np.random.choice(len(P_xi), samples, p=P_xi)

    # Return trajectories given by traj_idx
    return [SG_trajs[i] for i in traj_idx]
```

Now we're ready to create the simulated human and sample demonstrations. Let's say our human cares about the distances to the obstacles in the environment, and let's see how varying θ and β changes their demonstrations.

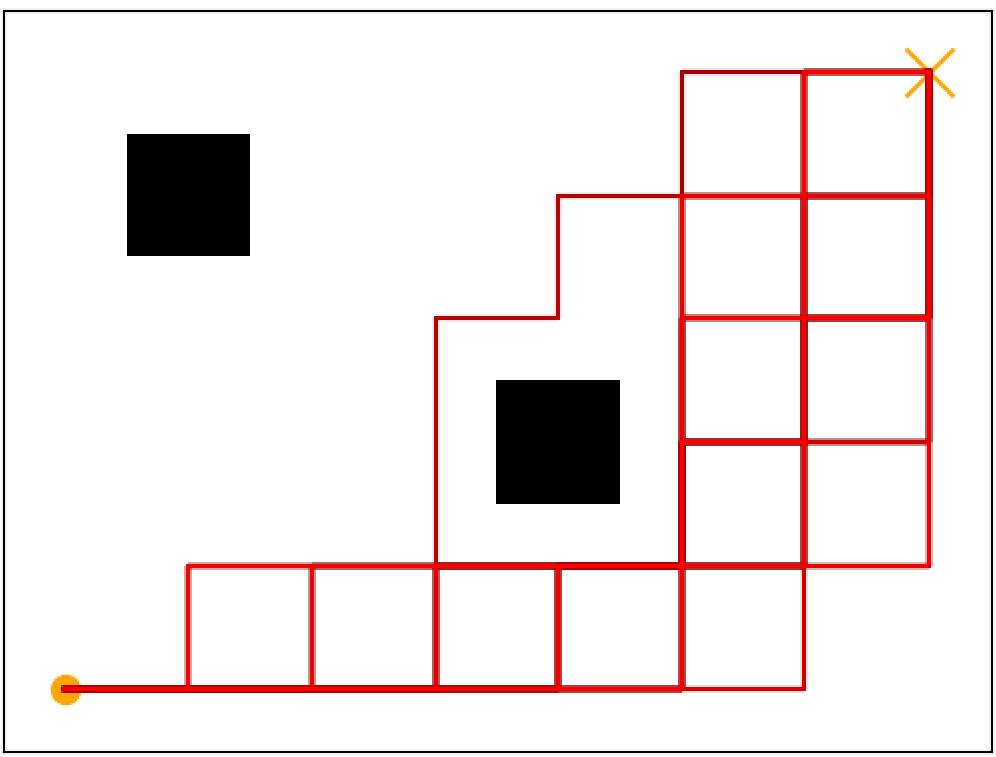
In [11]:
```
# Define parameters for simulated human.
feat_list = ["obstacles"]
num_features = 2
scaling_coeffs = feat_scale_construct(feat_list) # Used for normalizing feature
real_theta = np.array([1.0, 1.0])
real_beta = 10.0
num_demos = 10

# Wants to stay close to both obstacles.
demos = sample_demonstrations(real_theta, real_beta, num_demos)
gridworld.visualize_demos(demos)
```



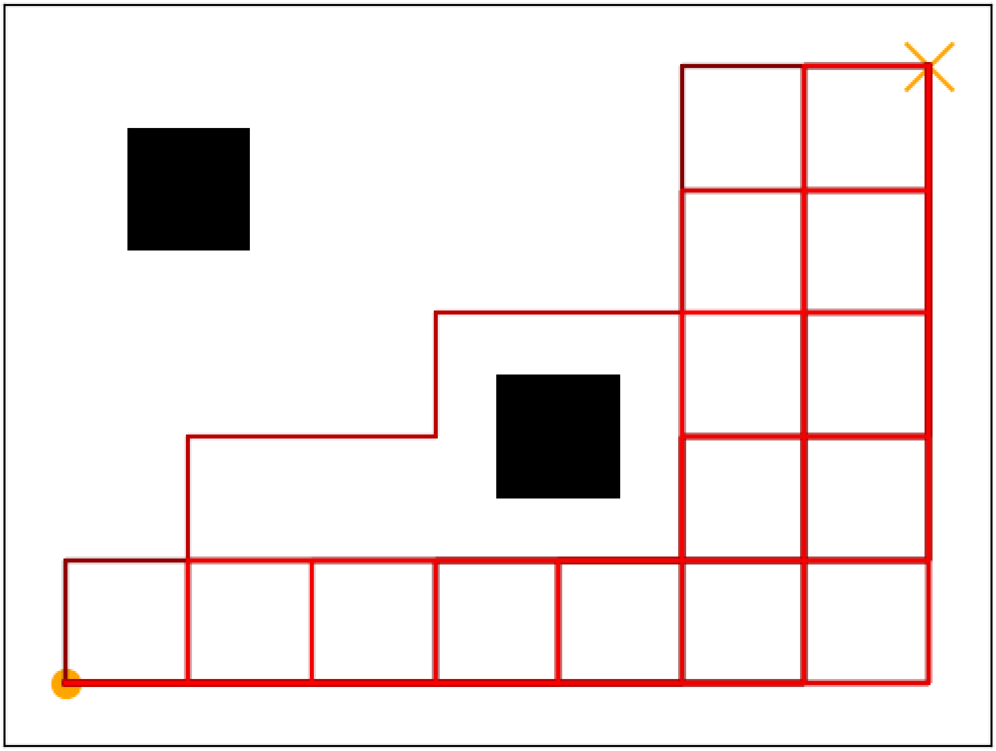
In [12]:
```
# Wants to stay close to left obstacle, doesn't care about the other one.
demos = sample_demonstrations(np.array([1.0, 0.0]), 10.0, 100)
gridworld.visualize_demos(demos)
```



In [13]:
```
# Wants to stay far from the left obstacle, close to the other one.
demos = sample_demonstrations(np.array([-1.0, 1.0]), 10.0, 100)
gridworld.visualize_demos(demos)
```

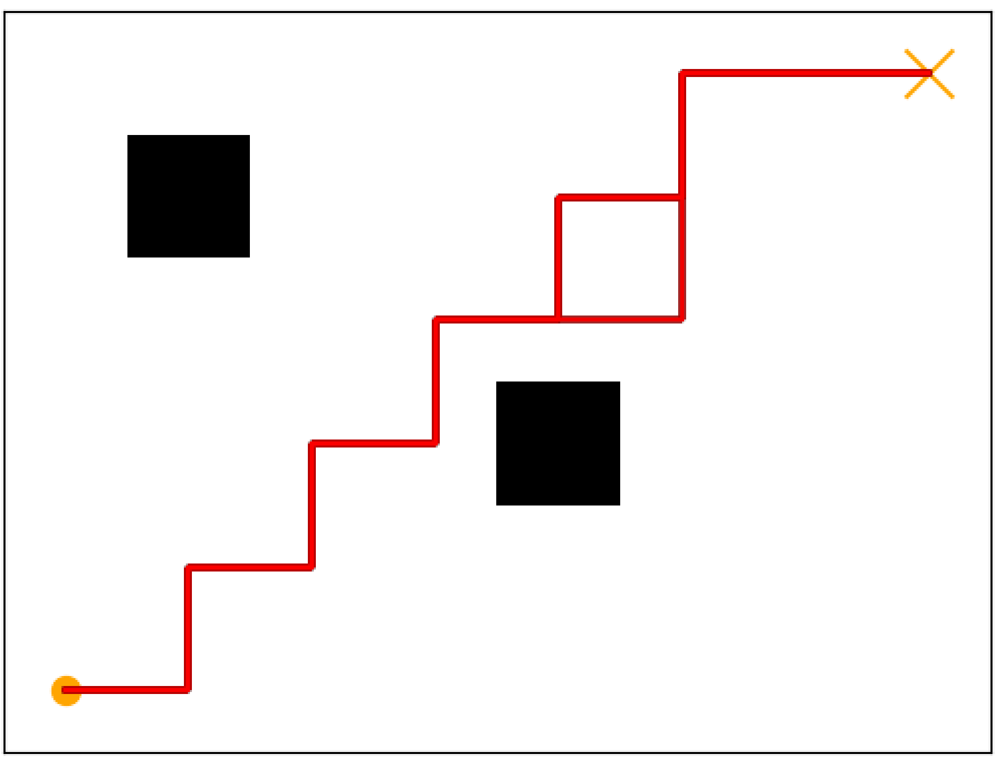


In [14]:
```
# Wants to stay far from the left obstacle, doesn't care about the other one.
demos = sample_demonstrations(np.array([-1.0, 0.0]), 10.0, 100)
gridworld.visualize_demos(demos)
```



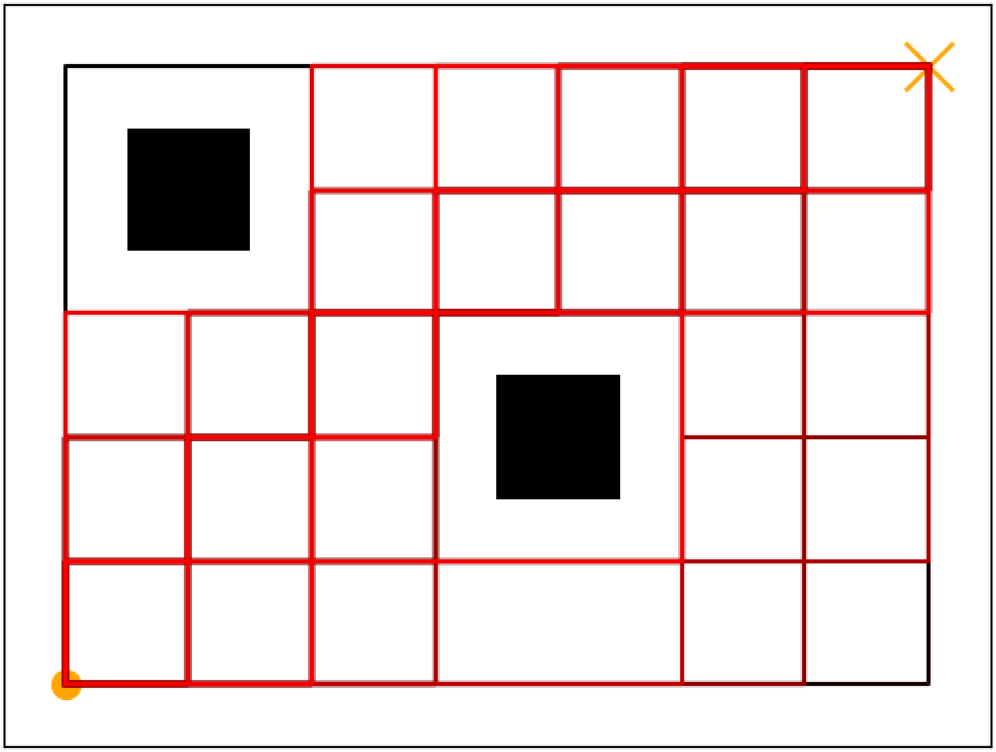
Change the β parameter below to create a human that almost always produces the optimal trajectory (pick a β that doesn't make the Python floating point system overflow, i.e. $|\beta| \leq 1000$).

In [15]:
```
near_optimal_beta = 1000
demos = sample_demonstrations(np.array([1.0, 1.0]), near_optimal_beta, 100)
gridworld.visualize_demos(demos)
```



Now **change the β parameter below** to create a human that essentially picks trajectories randomly (pick a β that doesn't make the Python floating point system overflow, i.e. $|\beta| \leq 1000$).

In [16]:
```
random_beta = 0
demos = sample_demonstrations(np.array([1.0, 1.0]), random_beta, 100)
gridworld.visualize_demos(demos)
```



Learning from Demonstrations

Boltzmann Rational Human Model

Now that we have some demonstrations from the simulated human, we want to infer just by observing their behavior how they chose that particular motion, i.e. what θ and β parameters they used to generate demonstrations. This is the typical Inverse Reinforcement Learning (IRL) problem, which seeks to explain an observed demonstration by uncovering the demonstrator’s unknown objective function. We're going to implement Bayesian IRL to do so:

$$P(\theta, \beta \mid \xi) = \frac{P(\xi \mid \theta, \beta) P(\theta, \beta)}{\int_{\bar{\theta}} \int_{\bar{\beta}} P(\xi \mid \bar{\theta}, \bar{\beta}) P(\bar{\theta}, \bar{\beta}) d\bar{\theta} d\bar{\beta}} \quad .$$

From the discussion in class, remember that Bayesian IRL is intractable because the double integral is infeasible to compute over continuous spaces. To make it tractable, we discretize the possible set of θ and β values.

In [17]:
```
# Inference parameters
theta_vals =  [-1.0, 0.0, 1.0]
betas = [0.1, 0.3, 1.0, 3.0, 10.0, 30.0]
thetas = list(iter.product(theta_vals, repeat=num_features))
if (0.0,)*num_features in thetas:
    thetas.remove((0.0,)*num_features)
thetas = [w / np.linalg.norm(w) for w in thetas]
thetas = set([tuple(i) for i in thetas])
thetas = [list(i) for i in thetas]
```

Given the discrete sets of possible θ and β values, as well as the already computed feature vectors for N demonstrations, **fill in the `inference` function below** to perform discrete Bayesian inference. We provide a uniform prior over θ and β that you should use.

Hint 1: the probability of multiple demonstrations is the product of the probability of each demonstration.

Hint 2: use the `observation_model` function you wrote earlier.

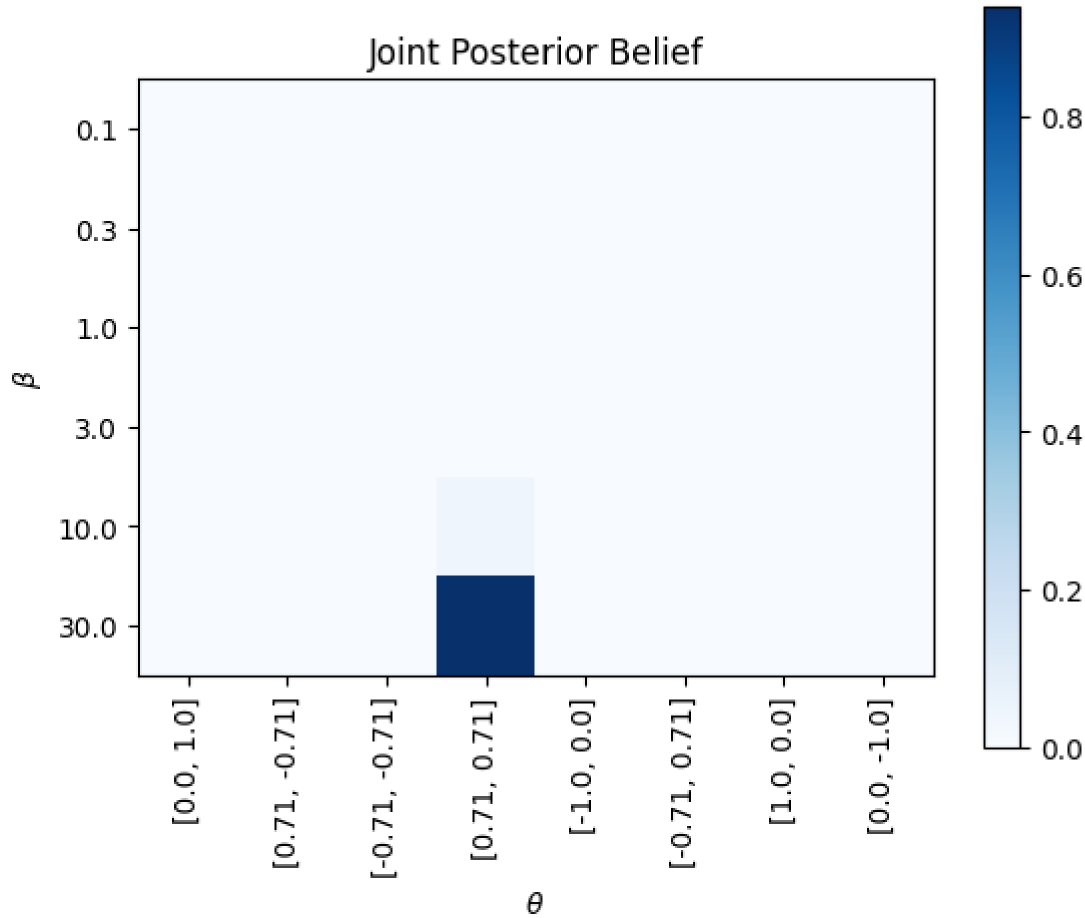
In [18]:
```
def inference(Phi_xis, thetas, betas):
    """
    Performs inference from given demonstrated features, using initialized mode
    Params:
        Phi_xis [list] -- A list of the cost features for observed trajectories
        thetas [list] -- Possible theta vectors.
        betas [list] -- Possible beta values.
    Returns:
        P_bt [array] -- Posterior probability P(beta, theta | xi_1...xi_N)
    """
    prior = np.ones((len(betas), len(thetas))) / (len(betas) * len(thetas))

    # Generate feature values for all trajectories in the gridworld.
    Phi_xibar = [featurize(xi, feat_list, scaling_coeffs) for xi in SG_trajs]

    P_bt = prior.copy()

    for Phi_xi in Phi_xis:
        # Compute likelihoods.
        likelihoods = np.zeros_like(prior)
        for b, beta in enumerate(betas):
            for t, theta in enumerate(thetas):
                likelihoods[b, t] = observation_model(Phi_xi, Phi_xibar, theta,

        # Multiply with prior and normalize.
        P_bt *= likelihoods
        P_bt /= np.sum(P_bt)

    return P_bt

##################################
#### Visualization functions ####
##################################

def visualize_posterior(prob, thetas, betas):
    # matplotlib.rcParams['font.sans-serif'] = "Arial"
    # matplotlib.rcParams['font.family'] = "Arial"
    # matplotlib.rcParams.update({'font.size': 15})
```

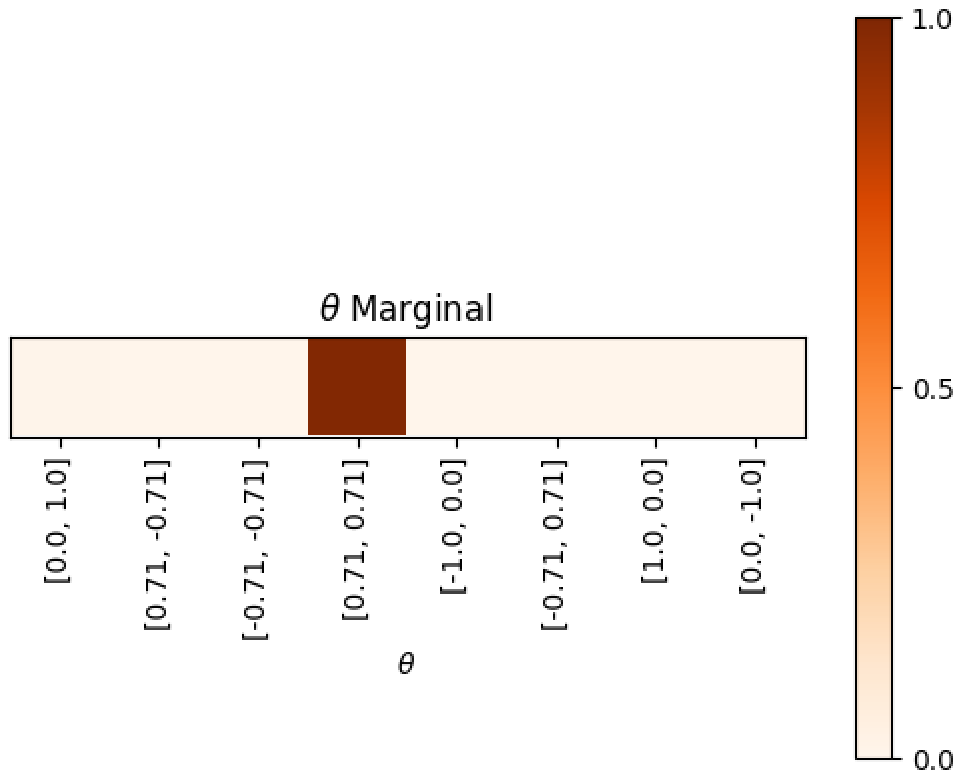
```
    plt.figure()
    plt.imshow(prob, cmap='Blues', interpolation='nearest')
    plt.colorbar()
    plt.clim(0, None)

    weights_rounded = [[round(i,2) for i in j] for j in thetas]
    plt.xticks(range(len(thetas)), weights_rounded, rotation = 'vertical')
    plt.yticks(range(len(betas)), betas)
    plt.xlabel(r'$\theta$')
    plt.ylabel(r'$\beta$')
    plt.title("Joint Posterior Belief")
    plt.show()

def visualize_marginal(marg, thetas):
    # matplotlib.rcParams['font.sans-serif'] = "Arial"
    # matplotlib.rcParams['font.family'] = "Arial"
    # matplotlib.rcParams.update({'font.size': 15})

    plt.figure()
    plt.imshow([marg], cmap='Oranges', interpolation='nearest')
    plt.colorbar(ticks=[0, 0.5, 1.0])
    plt.clim(0, 1.0)

    weights_rounded = [[round(i,2) for i in j] for j in thetas]
    plt.xticks(range(len(thetas)), weights_rounded, rotation = 'vertical')
    plt.yticks([])
    plt.xlabel(r'$\theta$')
    plt.title(r'$\theta$ Marginal')
    plt.show()
```

Now let's test your inference function. First, for a simulated human with high rationality coefficient β.

In [19]:

```
# Define parameters for simulated human with high rationality, who wants to sta
feat_list = ["obstacles"]
num_features = 2
scaling_coeffs = feat_scale_construct(feat_list) # Used for normalizing feature
real_theta = np.array([1.0, 1.0])
real_beta = 30.0
num_demos = 10

# Generate demonstrations.
demos = sample_demonstrations(real_theta, real_beta, num_demos)

# Generate feature values for all demonstrations.
Phi_xis = [featurize(xi, feat_list, scaling_coeffs) for xi in demos]

# Perform and visualize inference.
posterior = inference(Phi_xis, thetas, betas)
visualize_posterior(posterior, thetas, betas)
visualize_marginal(sum(posterior), thetas)
```

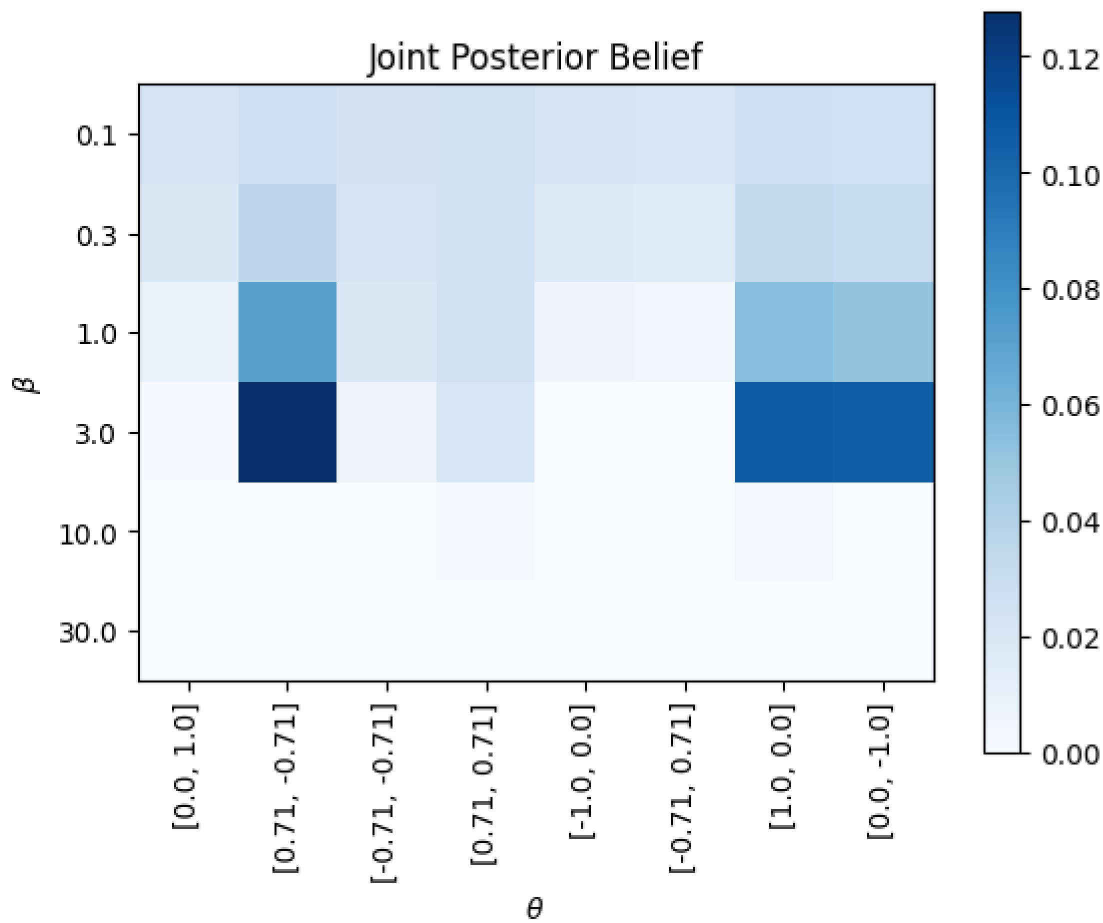


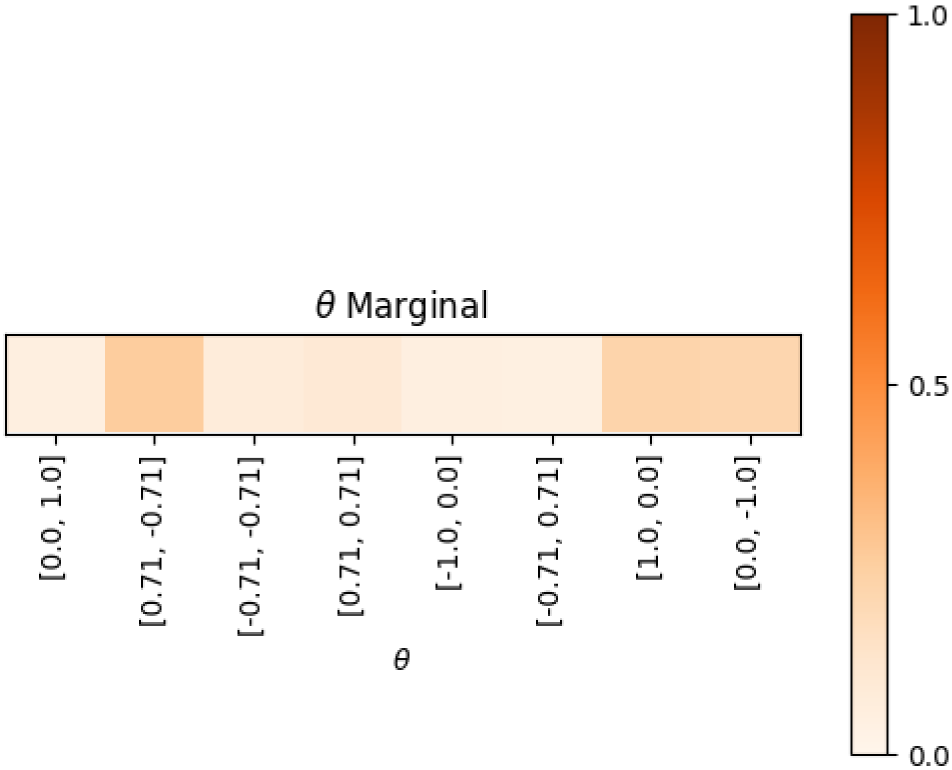


Now, for a noisier human. Try running the following cell multiple times to see what happens. What do you notice?

Answer: I noticed that in the noisy human case, the joint posterior belief of β and θ is less concentrated and more spread out compared to the rational human model, as the random actions make it more challenging to infer the true values of β and θ. The marginal distributions of θ is also less concentrated and more spread out, indicating less confidence in the inferred preferences.

In [20]:
```
# Generate demonstrations with low rationality.
demos = sample_demonstrations(real_theta, 0.1, num_demos)

# Generate feature values for all demonstrations.
Phi_xis = [featurize(xi, feat_list, scaling_coeffs) for xi in demos]

# Perform and visualize inference.
posterior = inference(Phi_xis, thetas, betas)
visualize_posterior(posterior, thetas, betas)
visualize_marginal(sum(posterior), thetas)
```



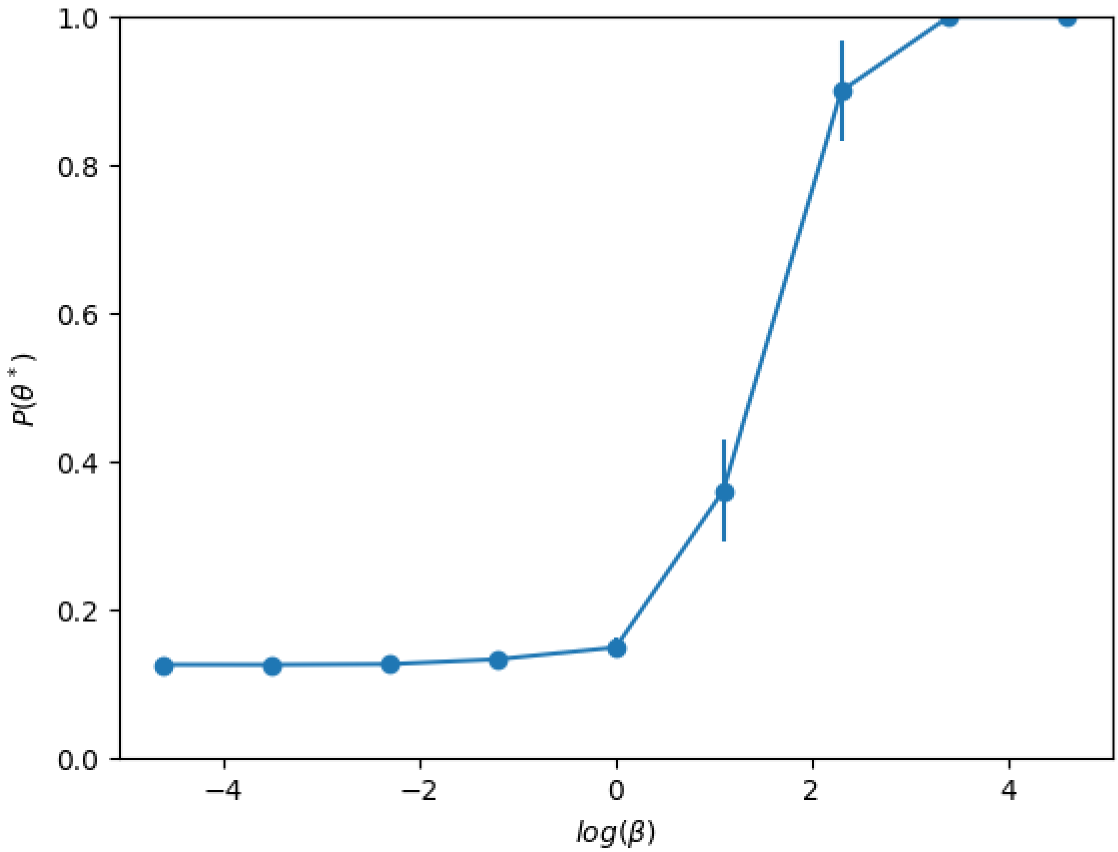


Now we're going to see how inference quality varies with the simulated human's rationality parameter β. **Fill in `plot_inferred_with_beta`** to plot how $P(\theta^* \mid \xi_1, \ldots, \xi_N)$, the marginal posterior probability of the true weight parameter θ^*, varies as the simulated human's β values change.

In [21]:
```
def plot_inferred_with_beta(xs, num_sims, real_theta, num_demos):
    """
    Plots P(theta^* | xi_1...xi_N) varies with beta.
    Params:
        xs [list] -- A list of the beta values for the simulated human.
        num_sims [int] -- Number of different demo samplings to run per simulat
        real_theta [list] -- True weight parameter theta^*.
        num_demos [int] -- Number of demonstrations sampled for every simulatic
    Returns:
        P_bt [array] -- Posterior probability P(beta, theta | xi_1...xi_N)
    """
    ys = []
    for x in xs:
        sims = []
        # Use this to index into the marginal for the true theta.
        true_idx = thetas.index((real_theta/np.linalg.norm(real_theta)).tolist(
        for _ in range(num_sims):
            # YOUR CODE HERE
            demos = sample_demonstrations(real_theta, x, num_demos)
            Phi_xis = [featurize(xi, feat_list, scaling_coeffs) for xi in demos
            P_bt = inference(Phi_xis, thetas, [x])
            sims.append(P_bt[0, true_idx])
        ys.append(sims)
    ys_mean = np.mean(ys, axis=1)
    ys_std = stats.sem(ys, axis=1)
    plt.errorbar(np.log(xs), ys_mean, ys_std, marker='o')
    plt.xlabel(r'$log(\beta$)')
```

```
    plt.ylabel(r'$P(\theta^*$)')
    plt.ylim(0, 1)
```

In [22]:
```
plot_inferred_with_beta([0.01, 0.03, 0.1, 0.3, 1.0, 3.0, 10.0, 30.0, 100.0], 10
```



Learning with a Misspecified Human Model

In the above examples, we used data that was generated from a precisely Boltzmann-rational human model. Of course, real people aren't exactly like this. What happens if we apply inverse RL when the human model is misspecified?

To explore this case, let's consider one way people can differ from the Boltzmann-rational model: myopia (short-sightedness). Often, people do not consider the long-term effects of their decisions as much as the short term ones. In our case, we can represent this by assuming people only consider the first half of each possible trajectory when deciding which one to take. That is,

$$P_{\text{myopic}}(\xi \mid \theta, \beta) = \frac{e^{-\beta\theta^T \Phi_{\text{myopic}}(\xi)}}{\int e^{-\beta\theta^T \Phi_{\text{myopic}}(\bar{\xi})} d\bar{\xi}} \quad ,$$

where $\Phi_{\text{myopic}}(\xi)$ consists of the features for only the first half of the trajectory ξ.

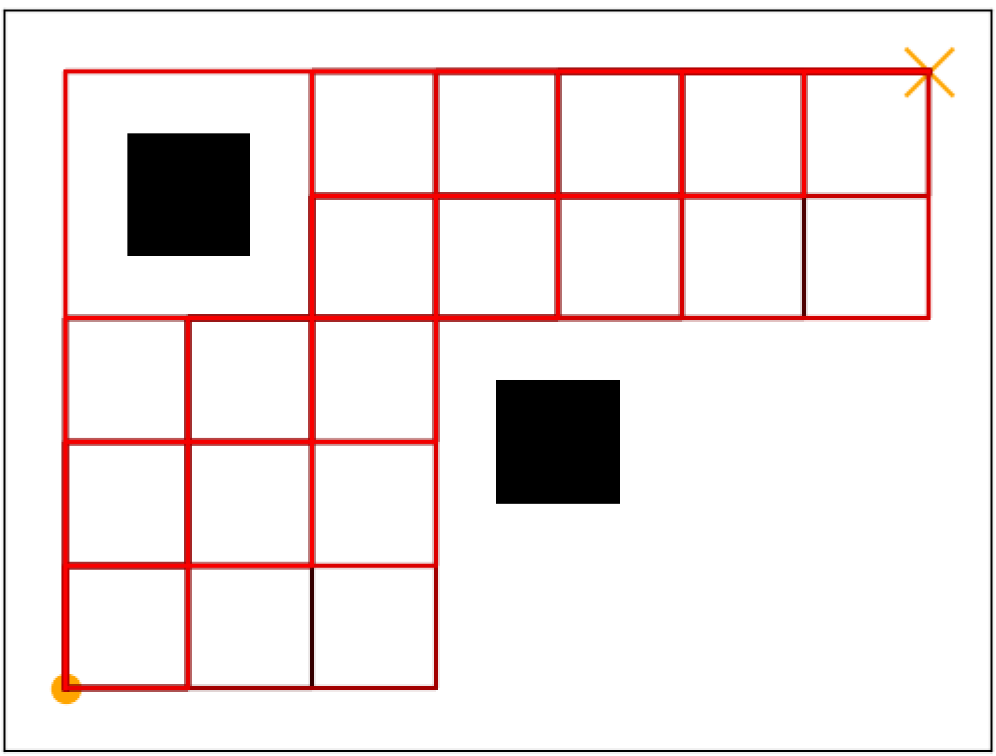
Let's start by implementing a function to generate data from the myopic human model:

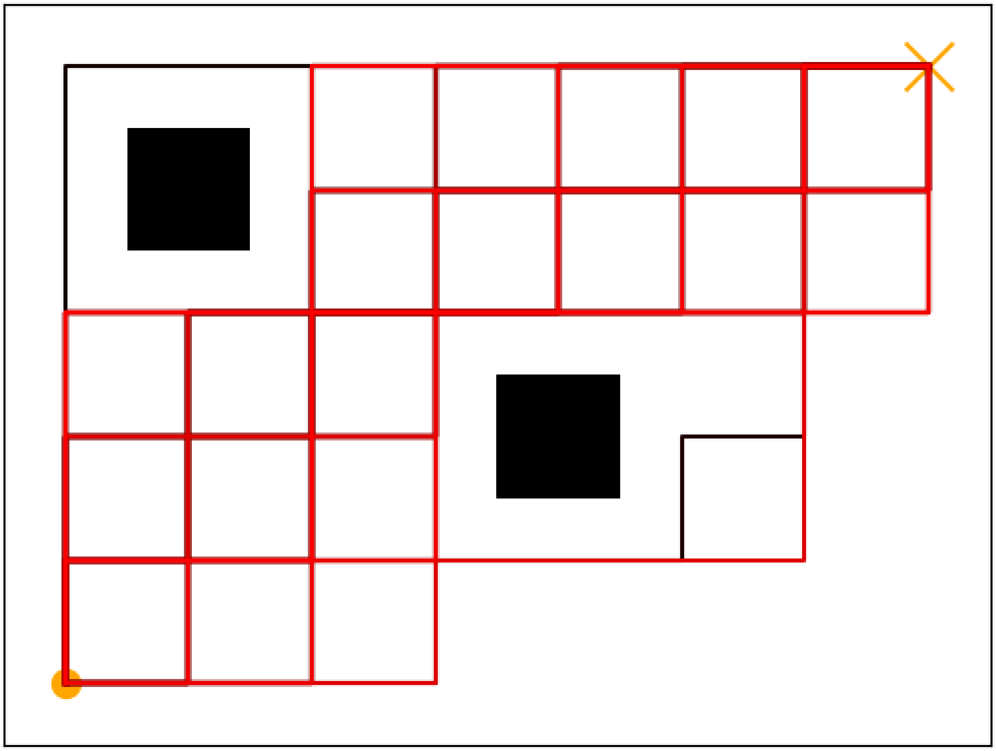
In [23]:
```
def sample_myopic_demonstrations(theta, beta, samples):
    """
```

```
    Sample <samples> myopic demonstrations for a given theta and beta.
    Params:
        theta [list] -- The preference parameter.
        beta [float] -- The rationality coefficient.
        samples [int] -- Number of demonstrations to be sampled.
    """
    # Generate feature values for all trajectories in the gridworld.
    Phi_xibar = [featurize(xi[:len(xi)//2], feat_list, scaling_coeffs) for xi i

    # Create the xi observation model for all trajectories.
    P_xi = [observation_model(Phi, Phi_xibar, theta, beta) for Phi in Phi_xibar

    # Sample <samples> trajectories using this distribution.
    traj_idx = np.random.choice(len(P_xi), samples, p=P_xi)

    # Return trajectories given by traj_idx
    return [SG_trajs[i] for i in traj_idx]
```

To explore the difference between the myopic and non-myopic humans, you can try running the cell below a couple of times. The top plot shows non-myopic trajectories, while the bottom will show myopic trajectories.

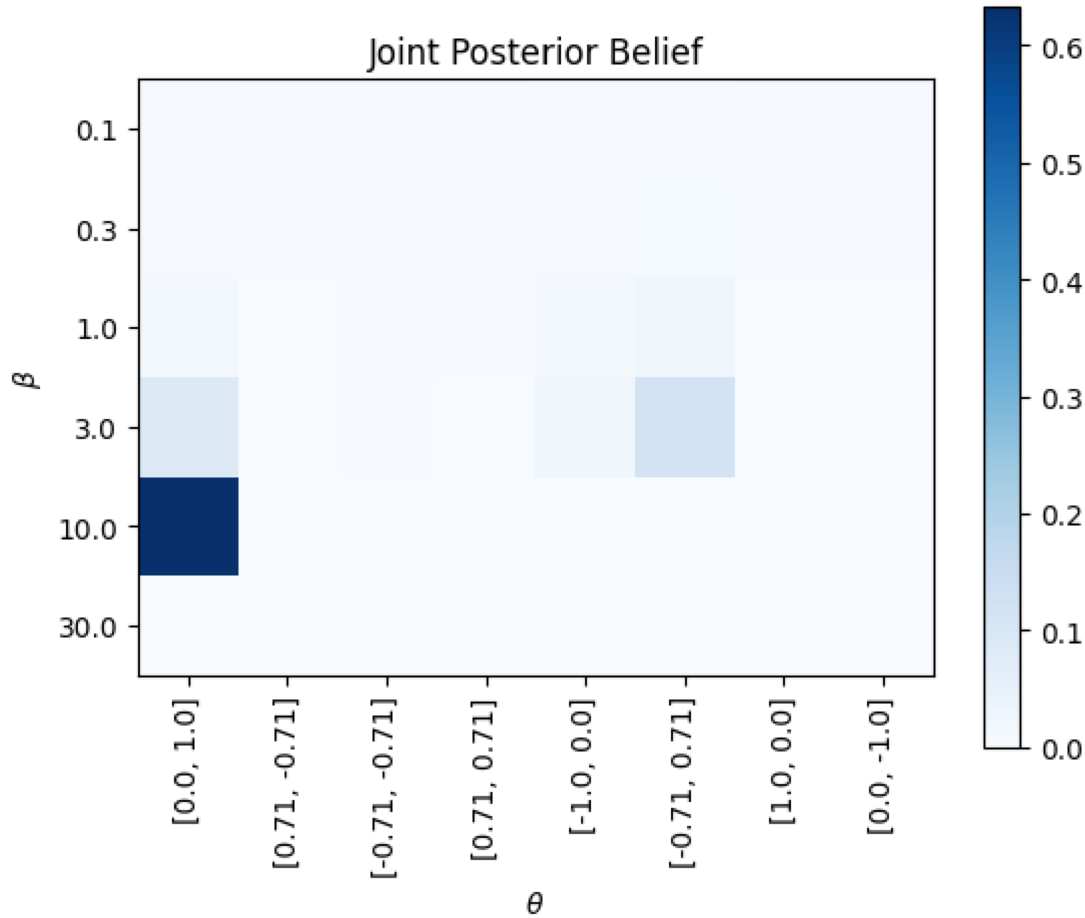
In [24]:
```
# Wants to stay close to left obstacle, doesn't care about the other one.
demos = sample_demonstrations(np.array([1.0, 0.0]), 10.0, 100)
gridworld.visualize_demos(demos)

demos = sample_myopic_demonstrations(np.array([1.0, 0.0]), 10.0, 100)
gridworld.visualize_demos(demos)
```

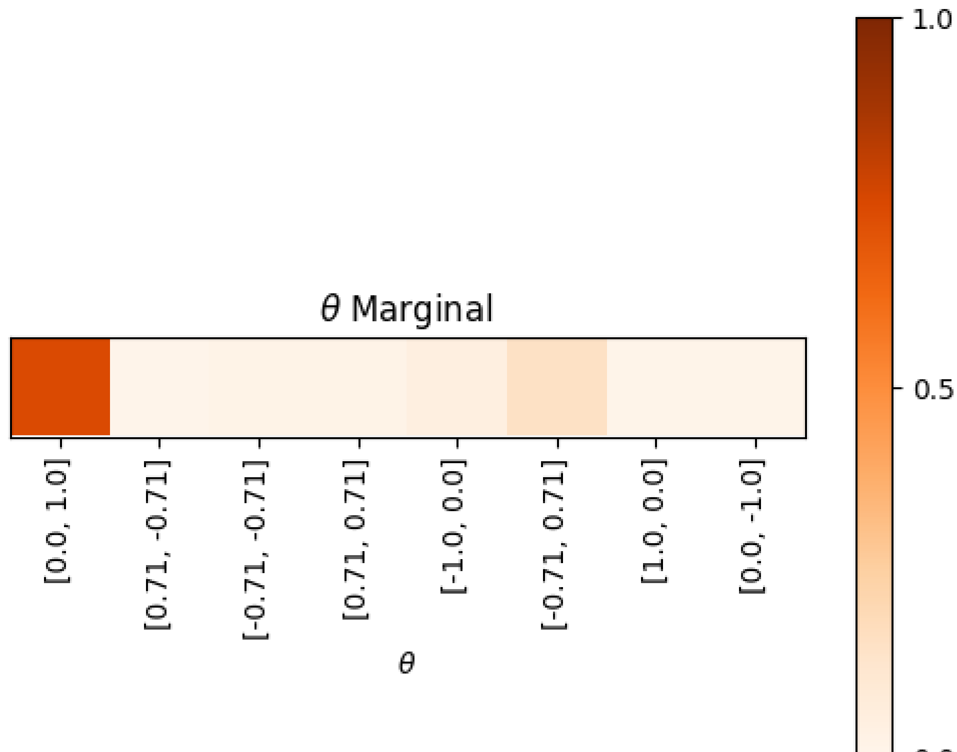




Now, let's see what happens if we use IRL to try and estimate the cost parameters of the myopic human:

In [25]:
```
feat_list = ["obstacles"]
num_features = 2
scaling_coeffs = feat_scale_construct(feat_list) # Used for normalizing feature
real_theta = np.array([0.0, 1.0])
real_beta = 30.0
num_demos = 10

# Generate myopic demonstrations.
demos = sample_myopic_demonstrations(real_theta, real_beta, num_demos)

# Generate feature values for all demonstrations.
Phi_xis = [featurize(xi, feat_list, scaling_coeffs) for xi in demos]

# Perform and visualize inference.
posterior = inference(Phi_xis, thetas, betas)
visualize_posterior(posterior, thetas, betas)
visualize_marginal(sum(posterior), thetas)
```





How does IRL perform when the human is myopic? What happens to the inferred preference parameter? Why?

Answer: IRL algorithms typically assume that humans are rational agents who optimize their preferences over the entire trajectory. However, when people are myopic, they only optimize over a limited horizon, and this discrepancy can lead to inaccurate preference parameter estimation. From the above graphs, we can see that the output is still relevantly concentrated at a few points with decent accuracy but is not as confident or accurate as the total rational human model.

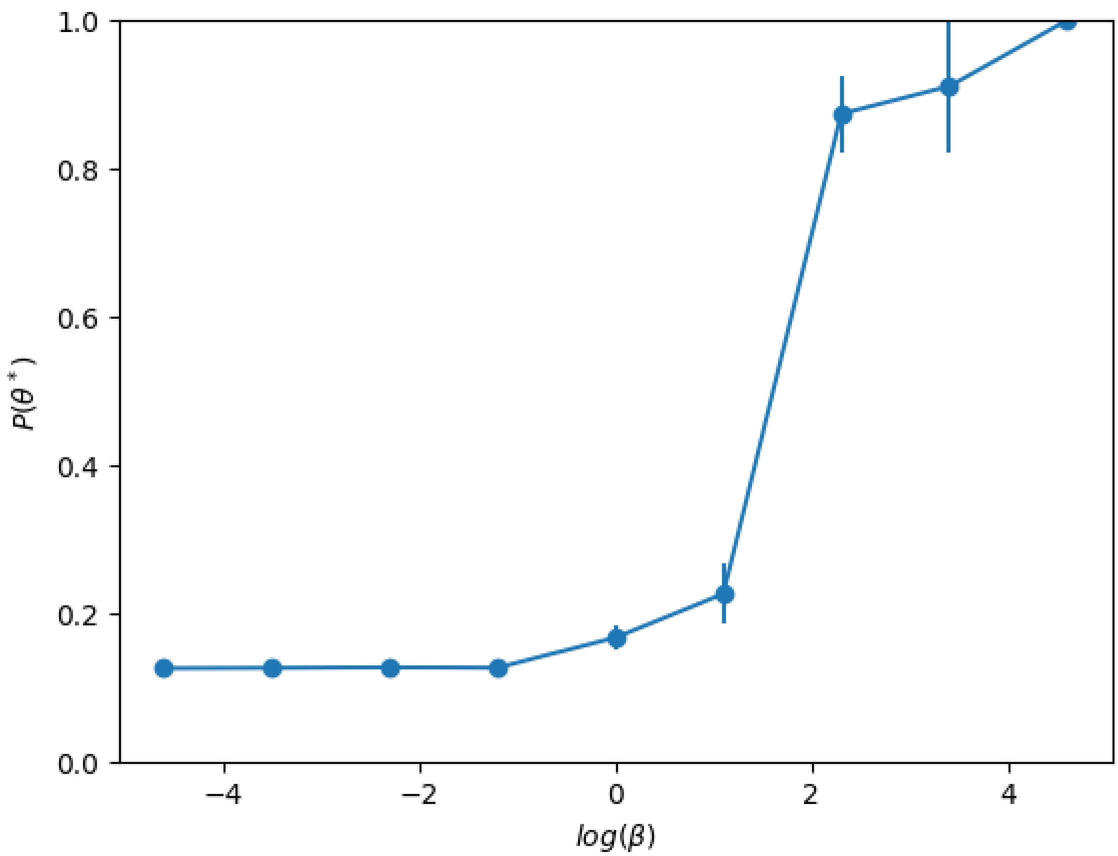
Now, write a function similar to `plot_inferred_with_beta` (or modify it) to plot the posterior probability of the true theta as a function of beta for myopic demonstrations. You can use the `sample_myopic_demonstrations` function to generate the demonstrations. How does this compare to the non-myopic case?

Answer: In the myopic case, inference quality tends to be lower than that of the non-myopic case with the same rationality parameter beta, especially with higher rationality parameter values. In general, the non-myopic approach is better at handling a range of rationality levels, as it models the human's consideration of future rewards more accurately. However, as the rationality parameter decreases, the inference quality of both approaches tends to degrade.

In [26]:

```
# YOUR PLOTTING CODE
def plot_inferred_with_beta(xs, num_sims, real_theta, num_demos):
    """
    Plots P(theta^* | xi_1...xi_N) varies with beta.
    Params:
        xs [list] -- A list of the beta values for the simulated human.
        num_sims [int] -- Number of different demo samplings to run per simulat
        real_theta [list] -- True weight parameter theta^*.
        num_demos [int] -- Number of demonstrations sampled for every simulatio
    Returns:
        P_bt [array] -- Posterior probability P(beta, theta | xi_1...xi_N)
    """
    ys = []
    for x in xs:
        sims = []
        # Use this to index into the marginal for the true theta.
        true_idx = thetas.index((real_theta/np.linalg.norm(real_theta)).tolist(
        for _ in range(num_sims):
            # YOUR CODE HERE
            demos = sample_myopic_demonstrations(real_theta, x, num_demos)
            Phi_xis = [featurize(xi[:len(xi)], feat_list, scaling_coeffs) for x
            P_bt = inference(Phi_xis, thetas, [x])
            col_sum = 0
            for row in P_bt:
              col_sum += row[true_idx]
            sims.append(col_sum)
        ys.append(sims)
    ys_mean = np.mean(ys, axis=1)
    ys_std = stats.sem(ys, axis=1)
    plt.errorbar(np.log(xs), ys_mean, ys_std, marker='o')
    plt.xlabel(r'$log(\beta$)')
    plt.ylabel(r'$P(\theta^*$)')
    plt.ylim(0, 1)
```

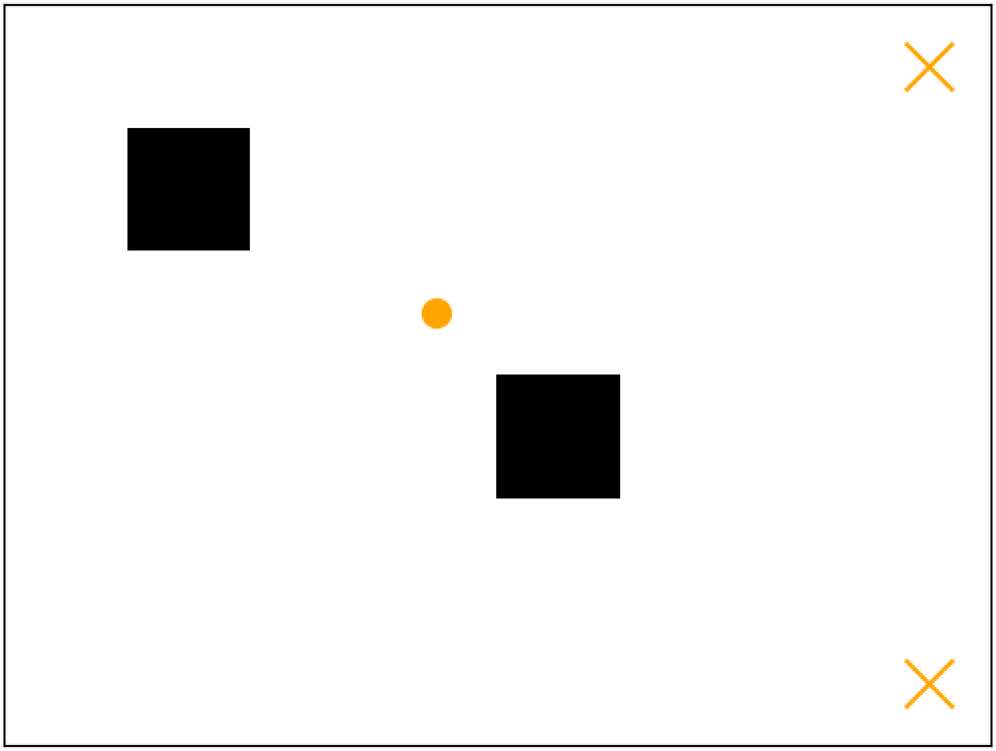
In [27]:
```
plot_inferred_with_beta([0.01, 0.03, 0.1, 0.3, 1.0, 3.0, 10.0, 30.0, 100.0], 10
```



Goal Inference

We're now going to switch gears to intent inference. Instead of caring about obstacles, our agent is presented with 2 possible goals in the grid world. After seeing a partial trajectory $\xi_{S\to Q}$ from the start S to an intermediate state Q, we want to compute the probability the trajectory is headed to either goal G.

In [28]:
```
# Build gridworld.
sim_height = 6
sim_width = 8
obstacles = [[[1,1], [1,1]] , [[4,3],[4,3]]]
start = [3, 2]
goals = [[7, 0], [7, 5]]
feat_list = ["goals"]
gridworld = AgentGridworld(sim_width, sim_height, obstacles, start, goals)
gridworld.visualize_grid()
```



What we want is $P(G \mid \xi_{S\to Q})$ for every goal $G \in \mathcal{G}$. From class, recall the following:

$$P(G \mid \xi_{S\to Q}) = \frac{P(\xi_{S\to Q} \mid G)P(G)}{\sum_{\bar{G}} P(\xi_{S\to Q} \mid \bar{G})P(\bar{G})} \quad ,$$

and

$$P(\xi_{S\to Q} \mid G) = \frac{e^{-C_G(\xi_{S\to Q})} \int_{\xi_{Q\to G}} e^{-C_G(\xi_{Q\to G})} d\xi_{Q\to G}}{\int_{\xi_{S\to G}} e^{-C_G(\xi_{S\to G})} d\xi_{S\to G}} \quad .$$

Let's define the cost function C_G as the cumulative distance from the goal G. Using the above formulae, **implement the `goal_inference` function below**, which takes in the partial trajectory $\xi_{S\to Q}$ and the goal set $\mathcal{G}$. We provide a uniform prior over goals.

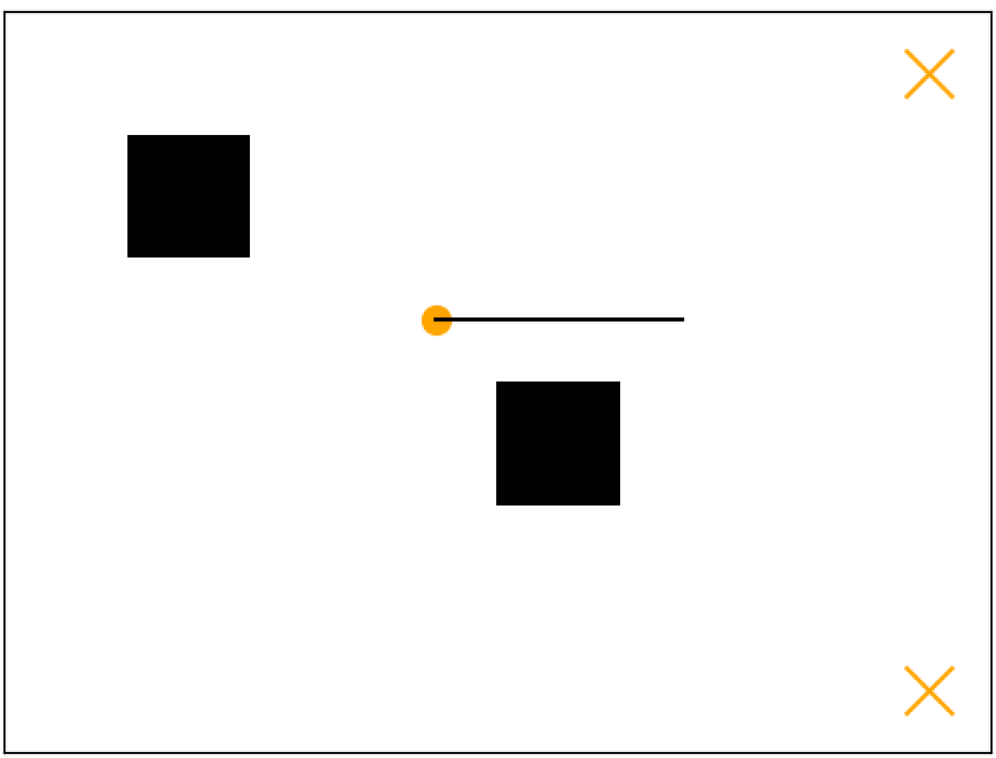
In [29]:
```
def goal_inference(traj, goals):
    """
    Performs goal inference from given partial trajectory.
    Params:
        traj [list] -- The partial trajectory xi_SQ.
        goals [list] -- List of goals.
    Returns:
        P_g [array] -- Posterior probability P(G | xi_SQ)
    """
    prior = np.ones(len(goals)) / len(goals)
    P_g = np.zeros(len(goals))
    Phi_xi = featurize(traj, feat_list) # Cost from S to Q, under both G1 and G
    for i in range(len(goals)):
        SG_trajs = gridworld.traj_construct(start, goals[i])
        QG_trajs = gridworld.traj_construct(gridworld.state_to_coor(traj[-1]),
        Phi_xiSG = [featurize(xi, feat_list)[i] for xi in SG_trajs] # Distances
        Phi_xiQG = [featurize(xi, feat_list)[i] for xi in QG_trajs] # Distances
```

```
        # YOUR CODE HERE
        P_g[i] = prior[i] * np.exp(-Phi_xi[i]) * sum([np.exp(-x) for x in Phi_x

    P_g /= sum(P_g)
    return P_g


def visualize_inference(prob, goals):
    matplotlib.rcParams['font.sans-serif'] = "Arial"
    matplotlib.rcParams['font.family'] = "Times New Roman"
    matplotlib.rcParams.update({'font.size': 15})

    plt.figure()
    plt.imshow([prob], cmap='Oranges', interpolation='nearest')
    plt.colorbar(ticks=[0, 0.5, 1.0])
    plt.clim(0, 1.0)

    plt.xticks(range(len(goals)), goals, rotation = 'vertical')
    plt.yticks([])
    plt.xlabel('Goals')
    plt.title("Inference")
    plt.show()
```

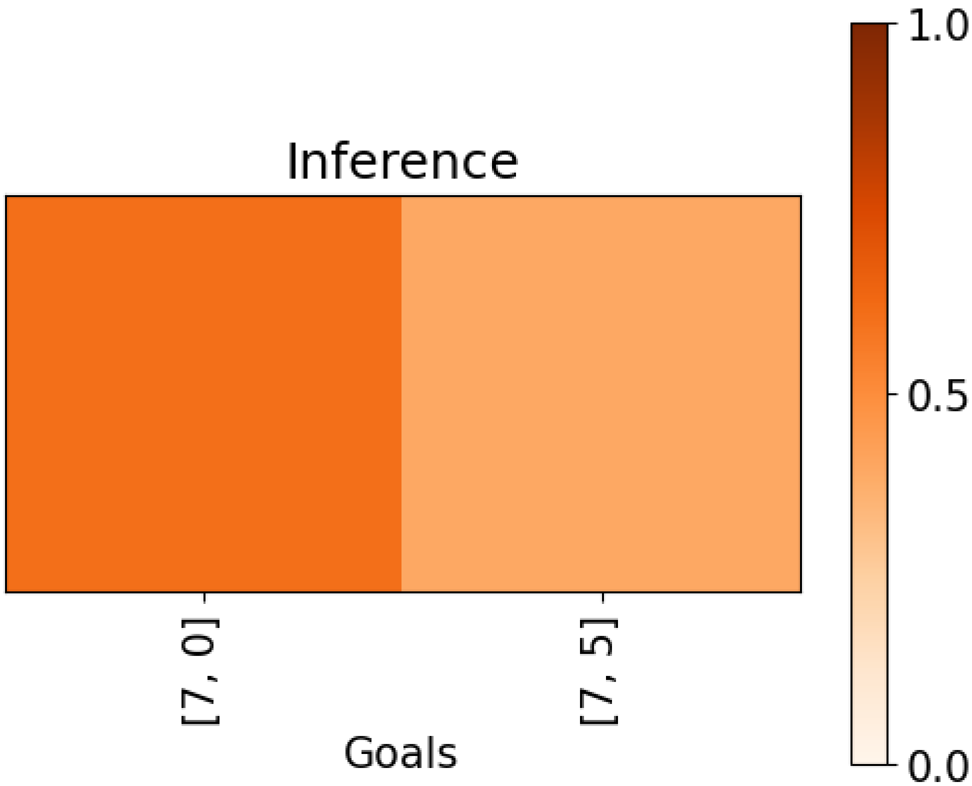
Let's test your implementation for the following 3 partial trajectories.

In [30]:

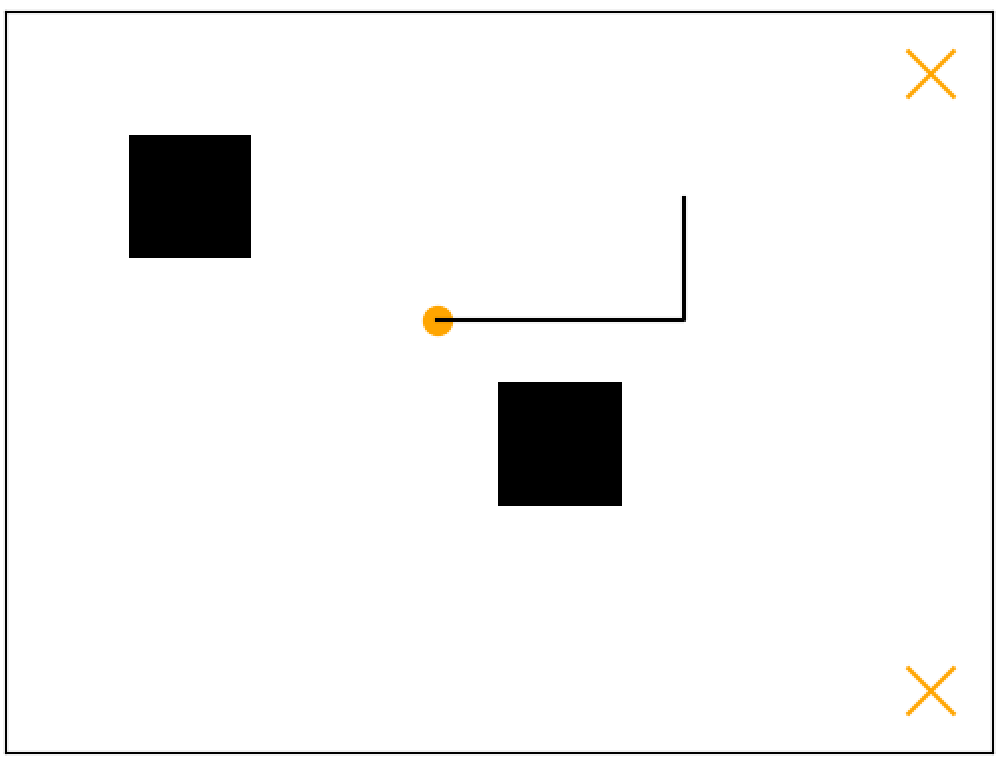
```
traj = [(3,2), (4,2), (5,2)]
demo = [gridworld.coor_to_state(x, y) for (x,y) in traj]
gridworld.visualize_demos([demo])
posterior = goal_inference(demo, goals)
visualize_inference(posterior, goals)
```



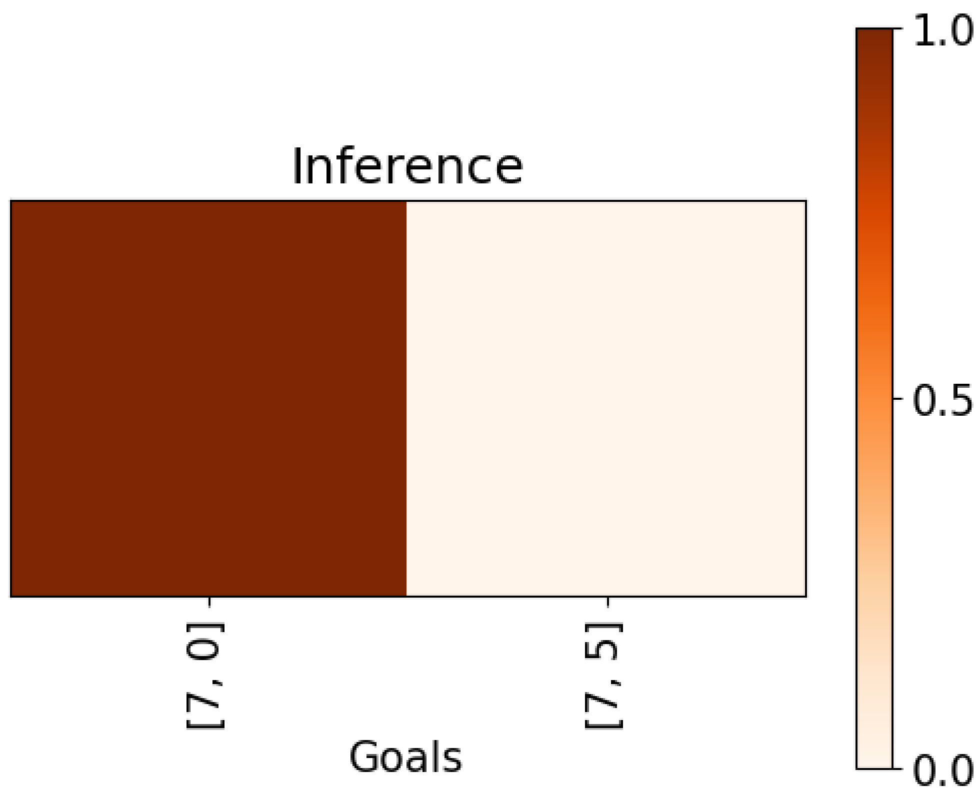
```
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
```



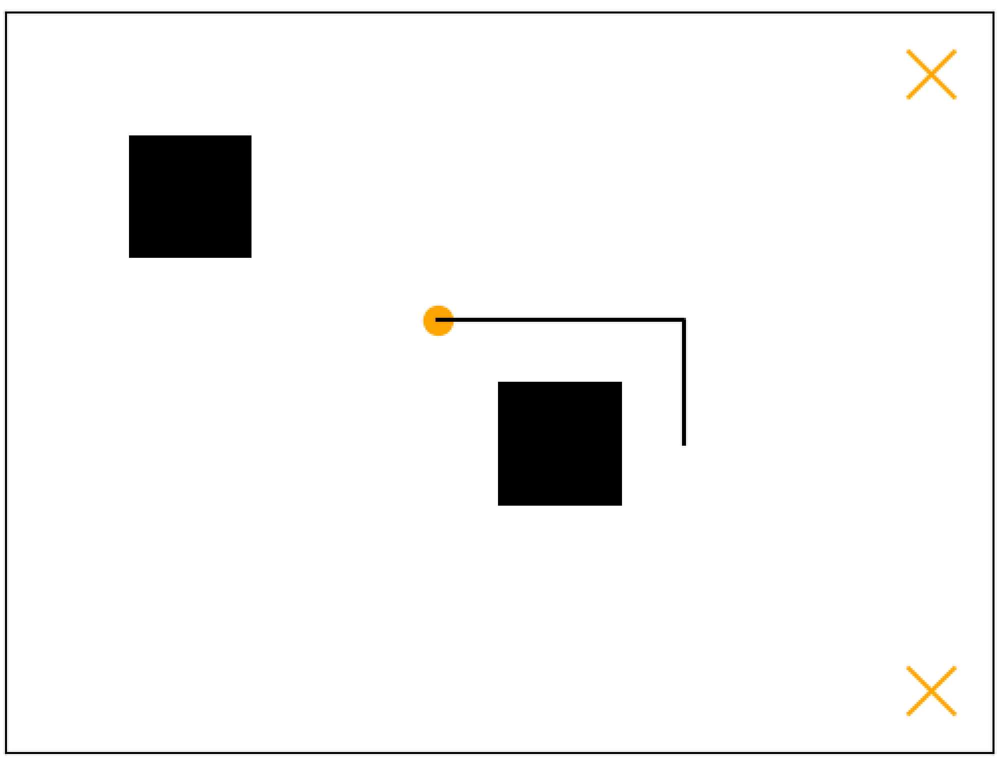
In [31]:
```
traj = [(3,2), (4,2), (5,2), (5,1)]
demo = [gridworld.coor_to_state(x, y) for (x,y) in traj]
gridworld.visualize_demos([demo])

posterior = goal_inference(demo, goals)
visualize_inference(posterior, goals)
```



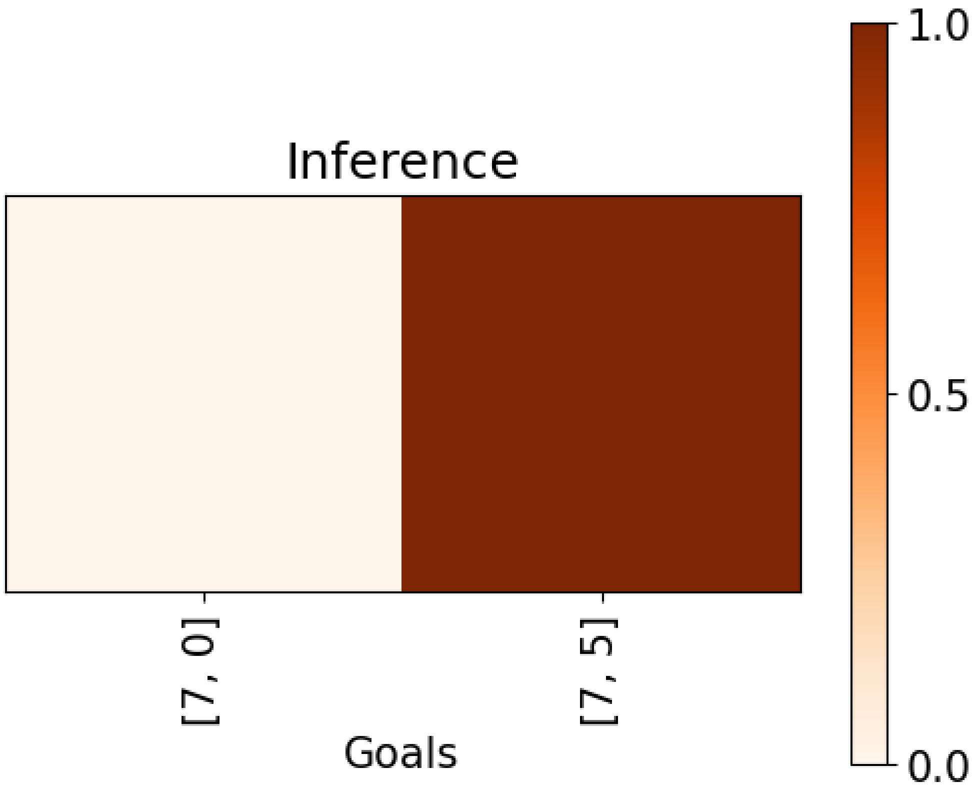
```
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
```



In [32]:
```
traj = [(3,2), (4,2), (5,2), (5,3)]
demo = [gridworld.coor_to_state(x, y) for (x,y) in traj]
gridworld.visualize_demos([demo])

posterior = goal_inference(demo, goals)
visualize_inference(posterior, goals)
```



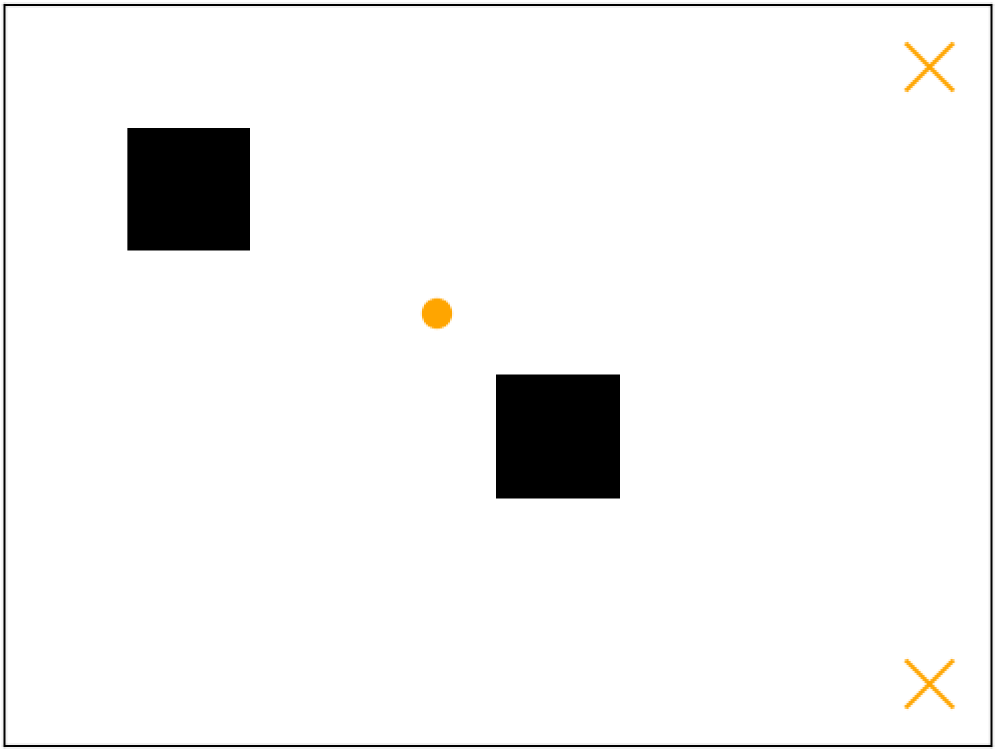
```
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
WARNING:matplotlib.font_manager:findfont: Font family 'Times New Roman' not fo
und.
```



Expressing Intent

So far we've been inferring what a person's intent is from their choice of trajectories. Now, we will reverse the roles and instead focus on the robot generating trajectories that express its intent to the human.

In [33]:
```
# Build grid world.
sim_width = 8
sim_height = 6
obstacles = [[[1,1], [1,1]] , [[4,3],[4,3]]]
start = [3, 2]
goals = [[7, 0], [7, 5]]
feat_list = ["goals"]
gridworld = AgentGridworld(sim_width, sim_height, obstacles, start, goals)
gridworld.visualize_grid()

# Build possible trajectories for each goal.
SG_trajs = [gridworld.traj_construct(start, goals[0]), gridworld.traj_construct
```



In the above grid world, we want to produce the robot trajectory that maximizes *predictability* to either goal. That is, we want to produce the trajectory that is most likely for a particular goal:

$$\xi^{pred} = \max_{\xi} P(\xi \mid G) = \frac{e^{-C_G(\xi)}}{\int e^{-C_G(\bar{\xi})} d\bar{\xi}} \quad .$$

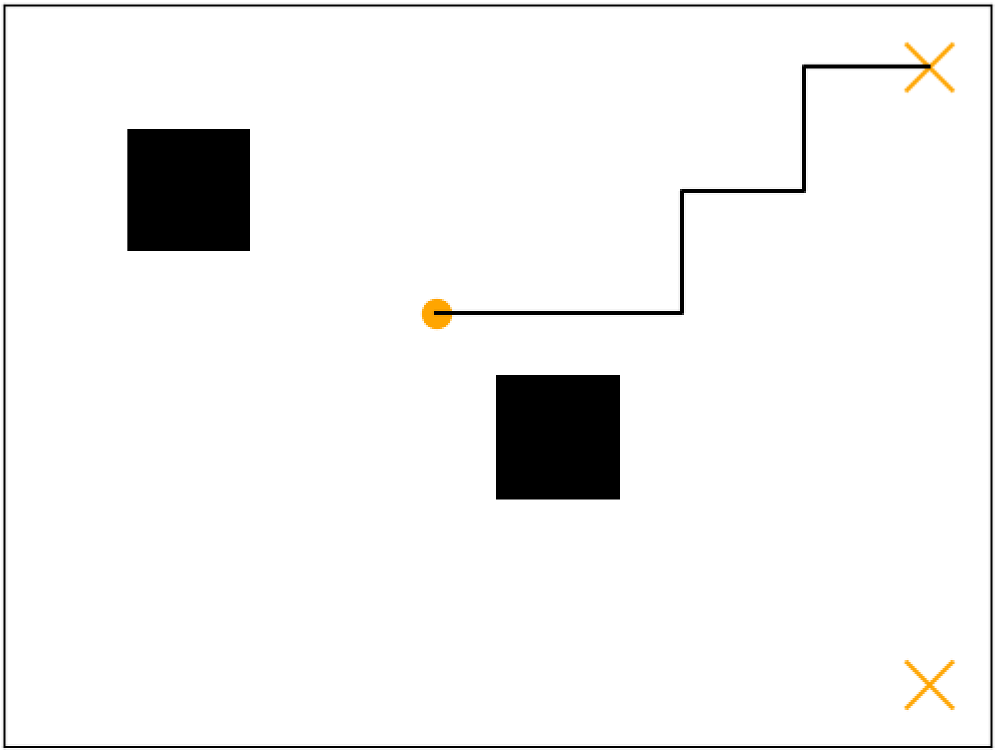
Fill in the `predictable_trajectory` function below, that maximizes predictability with respect to a goal given by `goal_idx`.

In [34]:
```
def predictable_trajectory(goal_idx):
    """
    Compute trajectory that maximizes predictability.
    Params:
        goal_idx [int] -- The goal w.r.t. we want to maximize predictability
    Returns:
        pred_traj [list] -- The trajectory that maximizes predictability.
    """

    # Generate feature values for all trajectories in the gridworld.
    Phi_xiSG = [featurize(xi, feat_list)[goal_idx] for xi in SG_trajs[goal_idx]

    # YOUR CODE HERE
    # You want to get the index pred_idx of the most predictable trajectory.
    probs = np.zeros(len(SG_trajs[goal_idx]))

    for i in range(len(SG_trajs[goal_idx])):
      probs[i] = np.exp(-Phi_xiSG[i])

    probs /= sum(probs)
    pred_idx = np.argmax(probs)
```

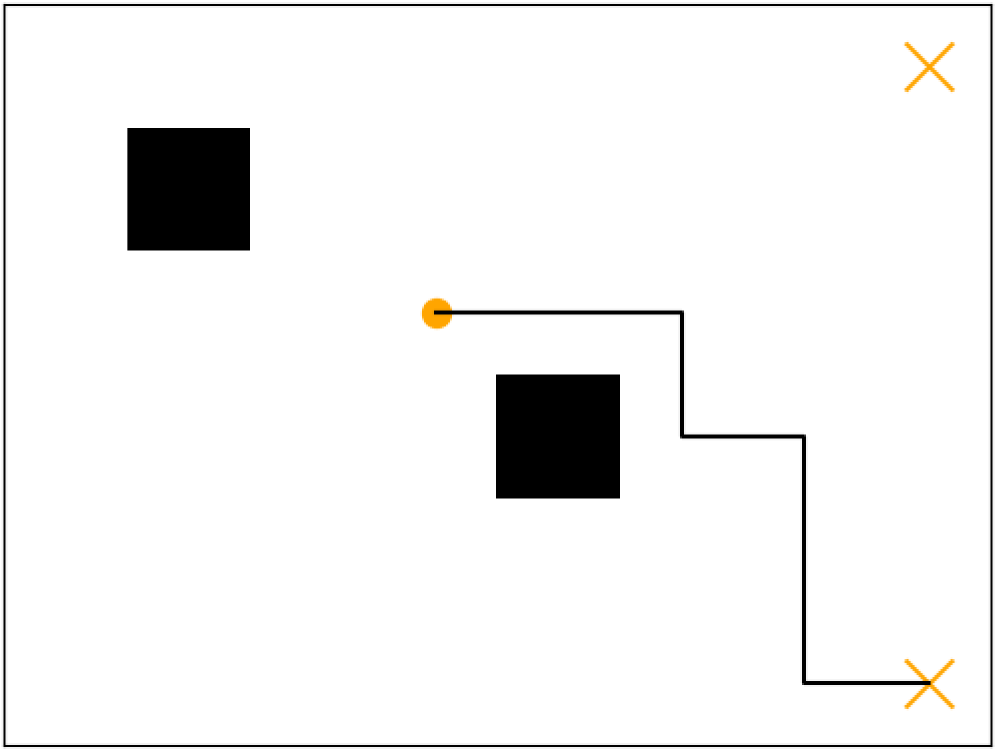
```
    pred_traj = SG_trajs[goal_idx][pred_idx]
    return pred_traj
```

Now let's test your code below for each goal.

In [35]:
```
traj = predictable_trajectory(0)
gridworld.visualize_demos([traj])
```



In [36]:
```
traj = predictable_trajectory(1)
gridworld.visualize_demos([traj])
```



In class, we also discussed robot trajectories that are *legible* for an intent. Legible trajectories are trajectories such that the person would easily be able to distinguish the robot's intent early on:

$$\xi^{leg} = \max_{\xi} P(G \mid \xi) = \frac{P(\xi \mid G)P(G)}{\sum_{\bar{G}} P(\xi \mid \bar{G})P(\bar{G})} \quad .$$

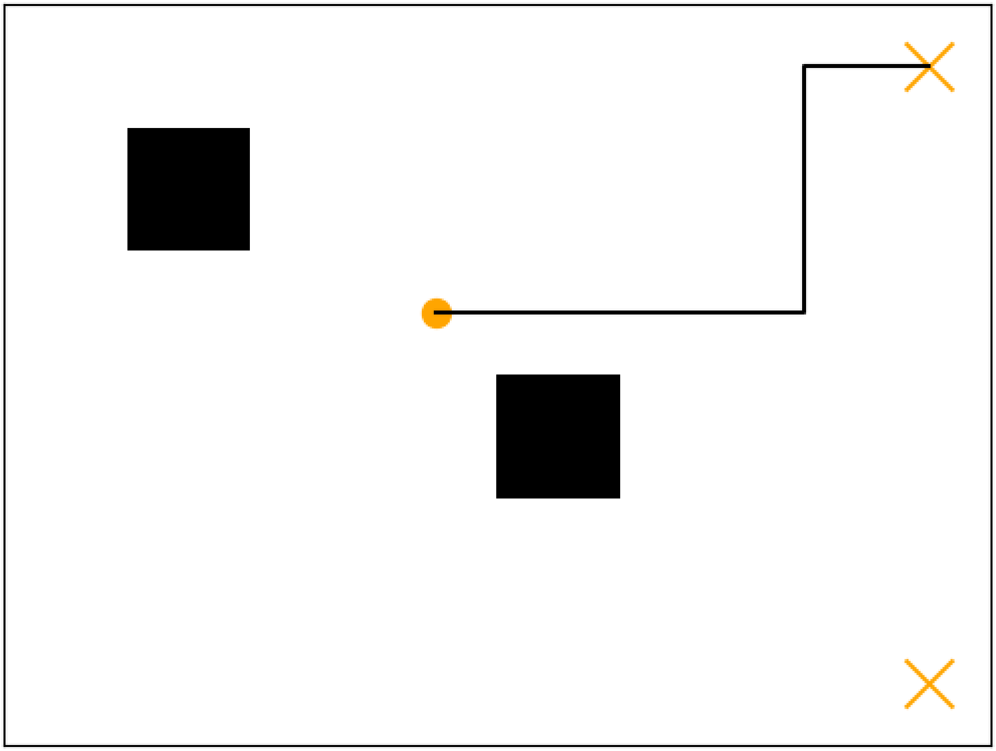
Fill in the `legible_trajectory` function below, that maximizes legibility with respect to a goal given by `goal_idx`.

In [37]:
```
def legible_trajectory(goals, goal_idx):
    """
    Compute trajectory that maximizes legibility.
    Params:
        goals [list] -- List of goals in the environment.
        goal_idx [int] -- The goal w.r.t. we want to maximize predictability
    Returns:
        leg_traj [list] -- The trajectory that maximizes predictability.
    """
    # max_xi P(g | xi)
    prior = np.ones(len(goals)) / len(goals)

    Phi_xiSGs = []
    C_xiSGs = []
    for i in range(len(goals)):
        # Generate feature values for all trajectories in the gridworld.
        Phi_xiSG = [featurize(xi, feat_list)[i] for xi in SG_trajs[i]]
        C_xiSG = [-Phi for Phi in Phi_xiSG]
        Phi_xiSGs.append(Phi_xiSG)
        C_xiSGs.append(C_xiSG)

    # YOUR CODE HERE
```

```python
    # You want to get the index leg_idx of the most legible trajectory.
    probs = np.zeros(len(goals))

    for i in range(len(goals)):
      probs[i] = prior[i] * np.exp(C_xiSG[i])

    probs /= sum(probs)
    leg_idx = np.argmax(probs)


    leg_traj = SG_trajs[goal_idx][leg_idx]
    return leg_traj


def legible_trajectory(goals, goal_idx):
    """
    Compute trajectory that maximizes legibility.
    Params:
        goals [list] -- List of goals in the environment.
        goal_idx [int] -- The goal w.r.t. we want to maximize predictability
    Returns:
        leg_traj [list] -- The trajectory that maximizes predictability.
    """
    # max_xi P(g | xi)
    prior = np.ones(len(goals)) / len(goals)

    Phi_xiSGs = []
    C_xiSGs = []
    for i in range(len(goals)):
        # Generate feature values for all trajectories in the gridworld.
        Phi_xiSG = [featurize(xi, feat_list)[i] for xi in SG_trajs[i]]
        C_xiSG = [-Phi for Phi in Phi_xiSG]
        Phi_xiSGs.append(Phi_xiSG)
        C_xiSGs.append(C_xiSG)

    # YOUR CODE HERE
    # You want to get the index leg_idx of the most legible trajectory.
    probs = np.zeros(len(SG_trajs[goal_idx]))
    for i in range(len(SG_trajs[goal_idx])):
        num = np.exp(-Phi_xiSGs[goal_idx][i]) * prior[goal_idx]
        den = sum([np.exp(-Phi_xiSGs[j][i]) * prior[j] for j in range(len(goals
        probs[i] = num / den

    leg_idx = np.argmax(probs)
    leg_traj = SG_trajs[goal_idx][leg_idx]
    return leg_traj
```

Now let's test your code below for each goal.

In [38]:
```python
traj = legible_trajectory(goals, 0)
gridworld.visualize_demos([traj])
```



In [39]:
```
traj = legible_trajectory(goals, 1)
gridworld.visualize_demos([traj])
```