Markov Decision Process (MDP) for this problem:

State Space (S): The state is a tuple that includes the positions of the human and the robot, the position of the goal, the positions of the obstacles, and the status of the tether (whether it’s taut or not).

Action Space (A): The robot can move in any direction in the 2D space, or choose to stay still. These actions could be discretized for computational feasibility.

Transition Function (T): The transition function would be determined by the physics of the environment. When the robot chooses an action, it changes its own position, potentially changes the status of the tether (if it moves far enough from the human), but does not directly change the position of the human. The human’s movement could be considered as part of the environment’s dynamics.

Reward Function ®: The robot gets a positive reward when the human reaches the goal and negative reward for bumping into obstacles. There could also be a small negative reward at each time step to encourage the robot to guide the human to the goal as quickly as possible.

Discount Factor (γ): This factor determines the importance of future rewards. A value of 0 makes the agent short-sighted by only considering current rewards, while a value approaching 1 makes it strive for a long-term high reward.

This MDP formulation allows us to use reinforcement learning algorithms to find the optimal policy for the robot to guide the human to the goal.

To solve the MDP as per the approach in the paper, we would use **Offline Reinforcement Learning** (RL) with an emphasis on learning influence strategies from human-human interaction data. Here’s a high-level overview of the steps:

1. **Data Collection**: Collect data of human-human interactions in the simulated environment. One player acts as the robot and tries to guide the other player (the human) towards the goal using the tether. The human player only sees a direction arrow when the tether is pulled to its full extent. Record these interactions as the dataset for training the RL agent.
2. **Preprocessing**: Preprocess the collected data to form suitable state-action-reward-next state tuples (S, A, R, S’). The state would include the positions of the human, robot, goal, and obstacles, and the status of the tether. The action would be the movement direction of the robot.
3. **Model Definition**: Define a model that can learn from this data. This could be a deep neural network with the state as input and the Q-values for all actions as output.
4. **Training**: Use an offline RL algorithm to train the model on the collected data. The aim is to learn a policy that can influence the human’s actions to guide them safely to the goal. The training process involves iteratively updating the model parameters to minimize the difference between the predicted Q-values and the target Q-values (calculated using the rewards and next state Q-values).
5. **Policy Extraction**: After training, extract the policy from the model. For each state, the action with the highest Q-value is chosen as the policy’s action.
6. **Evaluation**: Evaluate the learned policy in the simulated environment to ensure it can guide the human safely and efficiently to the goal.

This approach allows us to leverage the power of RL to learn from human-human interaction data, and to adapt to changes in the human’s latent strategies, such as goals or plans. It’s important to note that this is a high-level overview, and the actual implementation would involve more detailed steps and considerations. For example, the choice of the neural network architecture, the offline RL algorithm, and the hyperparameters would all be crucial to the success of the project.

Remember, RL involves a lot of hyperparameter tuning and iterative refinement to get good performance. It’s also important to note that the performance of the learned policy should be thoroughly tested and validated before deployment in a real-world scenario.

Offline Reinforcement Learning (RL) problem:

1. **Bellman Equation**: The value of a state under a policy is the expected immediate reward plus the expected value of the next state, discounted by a factor γ. This can be written as:

V^{\pi}(s) = \mathbb{E}\_{a \sim \pi, s' \sim P}[r(s,a) + \gamma V^{\pi}(s') | s]

1. **Q-function**: The Q-function (or action-value function) under a policy is the expected return for taking action a in state s and thereafter following policy π. It can be written as:

Q^{\pi}(s,a) = \mathbb{E}\_{s' \sim P, a' \sim \pi}[r(s,a) + \gamma Q^{\pi}(s',a') | s, a]

1. **Optimal Q-function**: The optimal Q-function obeys the Bellman optimality equation, which can be written as:

Q^{\*}(s,a) = \mathbb{E}\_{s' \sim P}[r(s,a) + \gamma \max\_{a'} Q^{\*}(s',a') | s, a]

1. **Policy Extraction**: Once we have the optimal Q-function, we can extract the optimal policy by choosing the action that gives the maximum Q-value for each state:

\pi^{\*}(s) = \arg\max\_{a} Q^{\*}(s,a)

1. **Loss Function for Q-Learning**: The loss function for Q-learning is typically the mean squared error between the current Q-values and the target Q-values:

\mathcal{L}(\theta) = \mathbb{E}\_{s,a,r,s'}[(Q(s,a;\theta) - y)^2]

wherey = r + \gamma \max\_{a'} Q(s',a';\theta)is the target Q-value, and\thetaare the parameters of the Q-network.

These equations form the basis of many RL algorithms, including Q-Learning and Deep Q-Network (DQN), which could be used to solve the MDP in the guide dog robot problem. Note that in practice, these equations are often implemented with function approximators (like neural networks) to handle large state and action spaces. Also, the expectations are usually approximated with samples in the case of model-free RL.