Tutorial on Using DESPOT

Table of Contents

1. Overview

DESPOT[1] is an anytime online POMDP planning algorithm. To use our solver package, the user first need to represent the POMDP in one of the following two ways:

- specify the POMDP in the POMDPX format as described in the POMDPX documentation, or
- specify a deterministic simulative model [1] for the POMDP in C++ according to the `DSPOMDP` interface included in the solver package (Section 3).

Once a model is specified, the user can follow simple routines to use DESPOT to solve the POMDP (Section 2).

Which type of model is better? A POMDPX model requires relatively less programming, and some domain-independent bounds are provided to guide the policy search in DESPOT. However, POMDPX can only be used to represent POMDPs which are not very large, and an exact representation of the POMDP is needed. A C++ model may require more programming, but it comes with the full flexibility of integrating user's domain knowledge into the policy search process. In addition, it can represent extremely large problems, and only a black-box simulator - rather than an exact representation of the POMDP - is needed. To enjoy the full power of DESPOT, a C++ model is encouraged.

In this tutorial, we will work with a very simple POMDP problem. We first explain with illustration how DESPOT can solve a POMDP given its C++ model, and then explain how to code a C++ model for a POMDP. For how to use DESPOT to solve a POMDP specified in the POMDPX format, see README.txt in the DESPOT package.

2. Coding a C++ Model

We explain and illustrate how a deterministic simulative model of a POMDP can be specified according to the `DSPOMDP` interface. The ingredients are the following:

- representation of states, actions and observations,
- the deterministic simulative model,
- functions related to beliefs and starting states,
- bound-related functions, and
- memory management functions.

We shall start with the minimal set of functions that need to be implemented in a C++ model (Section 2.2), and then explain how to implement additional functions which can be used to get better performance (Section 2.3).

2.1. Problem

We will be using a simplified version of the *RockSample* problem [2] as our running example. It is the same as the example used in the POMDPX documentation, but its description is included for the sake of completeness. It models a rover on an exploration mission and it can achieve rewards by sampling rocks in its immediate area. Consider a map of size 1 x 3 as shown in Figure 1, with one rock at the left end and the terminal state at the right end. The rover starts off at the center and its possible actions are A = {*West*, *East*, *Sample*, *Check*}.

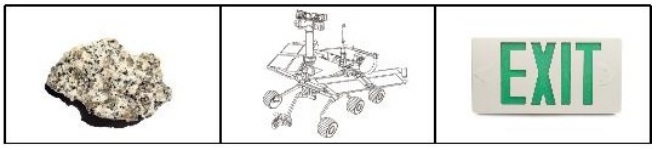

Figure 1. The 1 x 3 *RockSample* problem world.

As with the original version of the problem, the *Sample* action samples the rock at the rover's current location. If the rock is good, the rover receives a reward of 10 and the rock becomes bad. If the rock is bad, it receives a

penalty of −10. Moving into the terminal area yields a reward of 10. A penalty of −100 is imposed for moving off the grid and sampling in a grid where there is no rock. All other moves have no cost or reward. The *Check* action returns a noisy observation from $O = \{Good, Bad\}$.

2.1.1 Using C++ Models

DESPOT can also be used to solve a POMDP specified in C++ according to the `DSPOMDP` interface in the solver package. Assume for now that a C++ model for the *RockSample* problem has been implemented as a class called `SimpleRockSample`, then the following code snippet shows how to use DESPOT to solve it.

Listing 1. Code snippet for running simulations using DESPOT

```
DSPOMDP* model = new SimpleRockSample();
ScenarioLowerBound* lower_bound = model->CreateScenarioLowerBound();
ScenarioUpperBound* upper_bound = model->CreateScenarioUpperBound();

Solver* solver = new DESPOT(model, lower_bound, upper_bound);

Random random;
for (int run = 0; run < 100; run ++) {
    State* start_state = model->CreateStartState();
    Belief* belief = model->InitialBelief(start_state);
    solver->belief(belief);

    for (int step = 0; step < 90; step ++) {
        OBS_TYPE obs; double reward;

        int action = solver->Search();
        bool terminal = model->Step(*state, random.NextDouble(), action, reward, obs);

        if(terminal) break;

        solver->Update(action, obs);
    }
}
```

In fact, the user only need to write something like the first line in the above code as the remaining snippet is already included the solver package. Go to the root directory of the solver package. Put the model files, say simple_rock_sample.h and simple_rock_sample.cpp, at src/problems. Open src/main.cpp, add the lines for including header files and initializing the model:

Listing 2. Adding a model to the solver package.

```
...
#include "src/problems/simple_rock_sample.h"
...
DSPOMDP* InitializeModel(string problem, option::Option* options) {
    DSPOMDP* model = NULL;
    ...
    } else if (problem == "simplerocksample") {
        model = new SimpleRockSample();
    }
    ...
    return model;
}
...
```

Recompile the solver, and then simulations can be done using the following command

Listing 3. Command for running simulations on the added model.

```
bin/despot -q simplerocksample --runs <N> [OPTION]...
```

The above command performs N simulations. Each simulation consists of iterations of online search and action execution. The search is done with default discount factor of 0.95 and 500 scenarios. The initial state is randomly drawn from the initial belief, and the simulation is terminated when a terminal state is encountered or after a default of 90 steps. There are various command line options that can be used, including specifying the discount factor, the maximum number of simulation steps, the search horizon. Run *bin/despot* for available options with brief descriptions.

By default, the software prints out some useful information on the simulations for the user. For example, the software prints out the initial world state, and at each step, it also prints out the action, current world state, observation, observation probability, one step reward, current total discounted reward and current total undiscounted reward. These outputs can be silenced by using the command line option **--silence**. When coding a model, these outputs are pretty useful for checking correctness. In addition, more outputs can be generated using the **--verbosity** or **-v** option. There are 6 different levels of verbosity: NONE, WARN, ERROR, INFO, DEBUG, VERBOSE, which correpond to values from 0 to 5 respectively. The default verbosity level is 0.

2.2. Essentials

The following code snippet shows the essential functions in the `DSPOMDP` interface. **Note that we do not cover functions that have to be implemented (pure virtual) but are not required by the solver to work correctly, such as the `PrintAction` and `PrintState` in the `DSPOMDP` class.**

Listing 4. Essential functions in the DSPOMDP interface

```
class DSPOMDP {
public:
```

```
    /* Returns total number of actions.*/
    virtual int NumActions() const = 0;

    /* Deterministic simulative model.*/
    virtual bool Step(State& state, double random_num, int action,
        double& reward, OBS_TYPE& obs) const = 0;

    /* Functions related to beliefs and starting states.*/
    virtual double ObsProb(OBS_TYPE obs, const State& state, int action) const = 0;
    virtual Belief* InitialBelief(const State* start, string type = "DEFAULT") const = 0;
    virtual State* CreateStartState(string type = "DEFAULT") const = 0;

    /* Bound-related functions.*/
    virtual double GetMaxReward() const = 0;
    virtual ValuedAction GetMinRewardAction() const = 0;

    /* Memory management.*/
    virtual State* Allocate(int state_id, double weight) const = 0;
    virtual State* Copy(const State* particle) const = 0;
    virtual void Free(State* particle) const = 0;
};
```

The following declaration of the `SimpleRockSample` class implements the above `DSPOMDP` interface. It is the same as the `DSPOMDP` interface except that the functions are no longer pure virtual, and a `MemoryPool` object is declared for memory management. We explain the functions and their implementation in detail in the following paragraphs.

Listing 5. Declaration of the SimpleRockSample class

```
class SimpleRockSample : public DSPOMDP {
public:
    /* Returns total number of actions.*/
    int NumActions() const;

    /* Deterministic simulative model.*/
    bool Step(State& state, double random_num, int action,
        double& reward, OBS_TYPE& obs) const;

    /* Functions related to beliefs and starting states.*/
    double ObsProb(OBS_TYPE obs, const State& state, int action) const;
    Belief* InitialBelief(const State* start, string type = "DEFAULT") const;
    State* CreateStartState(string type = "DEFAULT") const;

    /* Bound-related functions.*/
    virtual double GetMaxReward() const;
    virtual ValuedAction GetMinRewardAction() const;

    /* Memory management.*/
    State* Allocate(int state_id, double weight) const;
    State* Copy(const State* particle) const;
    void Free(State* particle) const;
    int NumActiveParticles() const;

private:
    mutable MemoryPool<SimpleState> memory_pool_;
};
```

2.2.1. States, Actions and Observations

A state is required to be represented as an instance of the `State` class or its subclass. The generic state class inherits `MemoryObject` for memory management, as will be discussed more later. It has two member variables: `state_id` and `weight`. The former is useful when dealing with simple discrete POMDPs, and the latter is used when using the `State` object to represent a weighted particle.

Listing 6. The generic state class

```
class State : public MemoryObject {
public:
    int state_id;
    double weight;

    State(int _state_id = -1, double _weight = 0.0) :
        state_id(_state_id),
        weight(_weight) {
    }

    virtual ~State() {
    }
};
```

For `SimpleRockSample`, we can actually use the generic state class to represent its states by mapping each state to an integer, but we define customized state class so as to illustrate how this can be done.

Listing 7. The state class for `SimpleRockSample`

```
class SimpleState : public State {
public:
    int rover_position; // takes value 0, 1, 2 starting from the leftmost grid
    int rock_status; // indicates whether the rock is good

    SimpleState() {
    }

    SimpleState(int _rover_position, int _rock_status) :
        rover_position(_rover_position),
        rock_status(_rock_status) {
```

```
    }

    ~SimpleState() {
    }
};
```

Actions are represented as consecutive integers of `int` type starting from 0, and the user is required to implement the `NumActions()` function which returns the total number of actions.

Listing 8. Implementation of `NumActions()` for `SimpleRockSample`.

```
int SimpleRockSample::NumActions() const {
    return 4;
}
```

For the sake of readability, we shall use an `enum` to represent actions for `SimpleRockSample`

Listing 9. Action enum for `SimpleRockSample`

```
enum {
    A_SAMPLE = 0,
    A_EAST = 1,
    A_WEST = 2,
    A_CHECK = 3
};
```

Observations are represented as integers of type `uint64_t`, which is also named as `OBS_TYPE` using `typedef`. Unlike the actions, the set of observations need not be consecutive integers. For `SimpleRockSample`, we use an `enum` to represent the observations as well.

Listing 10. Observation enum for `SimpleRockSample`

```
enum {
    O_BAD = 0,
    O_GOOD = 1,
};
```

2.2.2. Deterministic Simulative Model

A deterministic simulative model for a POMDP is a function $g(s, a, r) = \langle s', o \rangle$ such that when random number r is randomly distributed in $[0,1]$, $\langle s', o \rangle$ is distributed according to $P(s', o \mid s, a)$. The deterministic simulative model is implemented in the `Step` function. `Step` function takes a state s and an action a as the inputs, and simulates the real execution of action a on state s, then outputs the resulting state s' and the corresponding reward. The argument names are self-explanatory, but note that

- there is a single `State` object which is used to represent both s and s',
- the function returns true iff executing a on s results in a terminal state.

Listing 11. A deterministic simulative model for SimpleRockSample

```
bool SimpleRockSample::Step(State& state, double rand_num, int action,
        double& reward, OBS_TYPE& obs) const {
    SimpleState& simple_state = static_cast<SimpleState&>(state);
    int& rover_position = simple_state.rover_position;
    int& rock_status = simple_state.rock_status;

    if (rover_position == 0) {
        if (action == A_SAMPLE) {
            reward = (rock_status == R_GOOD) ? 10 : -10;
            obs = O_GOOD;
            rock_status = R_BAD;
        } else if (action == A_CHECK) {
            reward = 0;
            obs = (rock_status == R_GOOD) ? O_GOOD : O_BAD;
        } else if (action == A_WEST) {
            reward = -100;
            obs = O_GOOD;
            rover_position = 2;
        } else {
            reward = 0;
            obs = O_GOOD;
            rover_position = 1;
        }
    } else if (rover_position  == 1) {
        if (action == A_SAMPLE) {
            reward = -100;
            obs = O_GOOD;
            rover_position = 2;
        } else if (action == A_CHECK) {
            reward = 0;
            obs =  (rand_num > 0.20) ? rock_status : (1 - rock_status);
        } else if (action == A_WEST) {
            reward = 0;
            obs = O_GOOD;
            rover_position = 0;
        } else {
            reward = 10;
            obs = O_GOOD;
            rover_position = 2;
        }
    } else {
        reward = 0;
        obs = O_GOOD;
    }

```

```
    return rover_position == 2;
}
```

2.2.3. Beliefs and Starting States

Our solver package supports arbitrary belief representation: The user can implement their own belief representation by implementing the `Belief` interface, which is only required to support sampling of particles, and belief update.

```
class Belief {
public:
  Belief(const DSPOMDP* model);

  virtual vector<State*> Sample(int num) const = 0;
  virtual void Update(int action, OBS_TYPE obs) = 0;
};
```

You can customize your own belief representation by implementing your belief class inheriting from the abstract `Belief` class (see 2.3.1 Custom Belief). Alternatively, you can use the `ParticleBelief` class already implemented in the solver package.

`ParticleBelief` class implements SIR (sequential importance resampling) particle filter, and inherits from `Belief` class. It is used as the default belief. To use `ParticleBelief` class, the only thing you need to do is to implement the `ObsProb` function. The `ObsProb` function is required in `ParticleBelief` for belief update. It implements the observation function in a POMDP, that is, it computes the probability of observing `obs` given current state `state` resulting from executing an action `action` in previous state.

Listing 12. Observation function for `SimpleRockSample`

```
double SimpleRockSample::ObsProb(OBS_TYPE obs, const State& state, int action) const {
    if (action == A_CHECK) {
        const SimpleState& simple_state = static_cast<const SimpleState&>(state);
        int rover_position = simple_state.rover_position;
        int rock_status = simple_state.rock_status;
        if (rover_position == LEFT) {
            return obs == rock_status;
        } else if (rover_position == MIDDLE) {
            return (obs == rock_status) ? 0.8 : 0.2;
        }
    } else {
        return obs == O_GOOD;
    }
}
```

The following code shows how the initial belief for `SimpleRockSample` can be represented by `ParticleBelief`. This example does not use the arguments, but in general, one can use `start` to pass partial information about the starting state to the initial belief, and use `type` to select different types of initial beliefs (such as uniform belief, or skewed belief), where `type` is specified using the command line option **--belief** or **-b**, with a value of "DEFAULT" if left unspecified.

Listing 13. Initial belief for `SimpleRockSample`

```
Belief* SimpleRockSample::InitialBelief(const State* start, string type) const {
        vector<State*> particles;

        if (type == "DEFAULT" || type == "PARTICLE") {
            SimpleState* good_rock = static_cast<SimpleState*>(Allocate(-1, 0.5));
            good_rock->rover_position = 1;
            good_rock->rock_status = 1;
            particles.push_back(good_rock);

            SimpleState* bad_rock = static_cast<SimpleState*>(Allocate(-1, 0.5));
            bad_rock->rover_position = 1;
            bad_rock->rock_status = 0;
            particles.push_back(bad_rock);

            return new ParticleBelief(particles, this);
        } else {
            cerr << "Unsupported belief type: " << type << endl;
            exit(1);
        }
}
```

The `CreateStartState` function is used to sample starting states in simulations. The starting state is generally sampled from the initial belief, but it may be sampled from a different distribution in some problems. Users may use the argument `type` to choose how the starting state is sampled.

Listing 14. Sample a starting state from the initial belief for `SimpleRockSample`

```
State* SimpleRockSample::CreateStartState(string type) const {
  return new SimpleState(1, Random::RANDOM.NextInt(2)));
}
```

2.2.4. Bound-related Functions

The heuristic search in DESPOT needed to be guided by upper and lower bounds on the infinite-horizon value that can be obtained on a set of scenarios. The `DSPOMDP` interface requires implementing the `GetMinRewardAction` function and the `GetMaxReward` function to construct the simplest such bounds.

The `GetMinRewardAction` function returns *(a, v)*, where *a* is an action with largest minimum reward when it is executed, and *v* is its minimum reward. The minimum infinite-horizon value that can be obtained on a set of scenarios with total weight *W* is then bounded below by *Wv / (1 - γ)*, where *γ* is the discount factor.

Listing 15. Implementation of `GetMinRewardAction` for `SimpleRockSample`

```
ValuedAction SimpleRockSample::GetMinRewardAction() const {
    return ValuedAction(A_EAST, 0);
}
```

The `GetMaxReward` function returns the maximum possible reward R_{max} in a step, and the maximum infinite-horizon value that can be obtained on a set of scenarios with total weight *W* is then bounded above by $W R_{max} / (1 - \gamma)$, where *γ* is the discount factor.

Listing 16. Implementation of `GetMaxReward` for `SimpleRockSample`

```
double SimpleRockSample::GetMaxReward() const {
    return 10;
}
```

2.2.5 Memory Management

DESPOT requires the creation of many `State` objects during the search. The creation and destruction of these objects are expensive, so they are done using the `Allocate`, `Copy`, and `Free` functions to allow users to provide their own memory management mechanisms to make these operations less expensive. We provide a solution method based on the memory management technique in David Silver's implementation of the POMCP algorithm. The idea is to create new `State` objects in chunks (instead of one at a time), and put objects in a free list for recycling when they are no longer needed (instead of deleting them). The following code serves as a template of how this can be done.

Listing 17. Memory management functions for `SimpleRockSample`.

```
State* SimpleRockSample::Allocate(int state_id, double weight) const {
    SimpleState* state = memory_pool_.Allocate();
    state->state_id = state_id;
    state->weight = weight;
    return state;
}

State* SimpleRockSample::Copy(const State* particle) const {
    SimpleState* state = memory_pool_.Allocate();
    *state = *static_cast<const SimpleState*>(particle);
    state->SetAllocated();
    return state;
}

void SimpleRockSample::Free(State* particle) const {
    memory_pool_.Free(static_cast<SimpleState*>(particle));
}
```

2.3. Improving Performance

Accurate belief tracking and good bounds are important for getting good performance. An important feature of the DESPOT software package is the flexibility that it provides for defining custom beliefs and custom bounds. This will be briefly explained below.

2.3.1 Custom Belief

The solver package can work with any belief representation implementing the abstract `Belief` interface. A concrete belief class need to implement two functions: the `Sample` function returns a number of particles sampled from the belief, and the `Update` function updates the belief after executing an action and receiving an observation. To allow the solver to use a custom belief, create it using the `InitialBelief` function in the `DSPOMDP` class. See src/problems for examples.

```
class Belief {
public:
    Belief(const DSPOMDP* model);

    virtual vector<State*> Sample(int num) const = 0;
    virtual void Update(int action, OBS_TYPE obs) = 0;
};
```

2.3.2 Custom Bounds

The lower and upper bounds mentioned in the previous section are non-informative and generally only works for simple problems. This section gives a brief explanation on how users can create their own bounds. We shall focus on the lower bounds. Creating an upper bound is similar, and examples can be found in the code in examples/cpp_models directory.

A new type of lower bound is defined as a child class of the `ScenarioLowerBound` class shown in Listing 20. A `ScenarioLowerBound` object computes a lower bound for the infinite-horizon value of a set of weighted scenarios (as determined by the particles and the random number streams) given the action-observation history. The first action that need to be executed in order to achieve the lower bound value is also returned together with the value, using a `ValuedAction` object. The random numbers used in the scenarios are reprensented using a `RandomStreams` object.

Listing 20. The `ScenarioLowerBound` interface

```
class ScenarioLowerBound {
protected:
    const DSPOMDP* model_;

public:
    ScenarioLowerBound(const DSPOMDP* model);

    /**
     * Returns a lower bound to the maximum total discounted reward over an
     * infinite horizon for the weighted scenarios.
     */
    virtual ValuedAction Value(const vector<State*>& particles,
        RandomStreams& streams, History& history) const = 0;
};
```

You can customize your own lower bound by implementing your lower bound class inheriting from the abstract `ScenarioLowerBound` class. Alternatively, you can use `ParticleLowerBound` or `Policy` that already implemented in the solver package.

`ParticleLowerBound` and `Policy` are two types of lower bounds inheriting from `ScenarioLowerBound` class. A `ParticleLowerBound` simply ignores the random numbers in the scenarios, and computes a lower bound for the infinite-horizon value of a set of weighted particles given the action-observation history. A `Policy` defines a policy mapping from the scenarios/history to an action, and runs this policy on the scenarios to obtain a lower bound. The random number streams only has finite length, and a `Policy` uses a `ParticleLowerBound` to estimate a lower bound on the scenarios when all the random numbers have been consumed.

Listing 18 shows the interface of `ParticleLowerBound`. To use `ParticleLowerBound`, you need to implement `Value` function shown below.

Listing 18. The `ParticleLowerBound` interface

```
class ParticleLowerBound : public ScenarioLowerBound {
public:
    ParticleLowerBound(const DSPOMDP* model);

    /**
     * Returns a lower bound to the maximum total discounted reward over an
     * infinite horizon for the weighted particles.
     */
    virtual ValuedAction Value(const vector<State>& particles) const = 0;
};
```

Listing 19 shows the interface of `Policy`. To use `Policy`, you need to implement `Action` function shown below.

Listing 19. Code snippet from the `Policy` class.

```
class Policy : public ScenarioLowerBound {
public:
    Policy(const DSPOMDP* model, ParticleLowerBound* bound, Belief* belief = NULL);
    virtual ~Policy();

    virtual int Action(const vector<State*>& particles,
        RandomStreams& streams, History& history) const = 0;
};
```

As an example of a `Policy`, the following code implements a simple fixed-action policy for `SimpleRockSample`.

Listing 20. A simple fixed-action policy for `SimpleRockSample`.

```
class SimpleRockSampleEastPolicy : public policy {
    public:
        SimpleRockSampleEastPolicy(const DSPOMDP* model, ParticleLowerBound* bound)
            : Policy(model, bound){}

        int Action(const vector<State*>& particles,
                RandomStreams& streams, History& history) const {
            return 1; // move east
        }
};
```

You can read the code in examples/cpp_models directory for more lower bound classes implemented for other tasks, for example, `PocmanSmartPolicy`, the policy implemented for `Pocman` task.

After you have implemented the lower bound class, you need to add your lower bound to the solver. The `DSPOMDP` interface allows user-defined lower bounds to be easily added by overriding the `CreateScenarioLowerBound` function in the `DSPOMDP` interface. The default implementation of `CreateScenarioLowerBound` only supports the creation of the `TrivialParticleLowerBound`, which returns the lower bound as generated using `GetMinRewardAction`.

Listing 21. `DSPOMDP` code related to supporting user-defined lower bounds.

```
class DSPOMDP {
public:
    virtual ScenarioLowerBound* CreateScenarioLowerBound(string name = "DEFAULT",
      string particle_bound_name = "DEFAULT") {
        if (name == "TRIVIAL" || name == "DEFAULT") {
```

```
            scenario_lower_bound_ = new TrivialParticleLowerBound(this);
        } else {
            cerr << "Unsupported scenario lower bound: " << name << endl;
            exit(0);
        }
    }
};
```

The following code adds this lower bound to `SimpleRockSample` and sets it as the default scenario lower bound.

Listing 22. Adding `SimpleRockSampleEastPolicy`.

```
ScenarioLowerBound* SimpleRockSample::CreateScenarioLowerBound(string name = "DEFAULT",
  string particle_bound_name = "DEFAULT") {
    if (name == "TRIVIAL") {
        scenario_lower_bound_ = new TrivialParticleLowerBound(this);
    } else if (name == "EAST" || name == "DEFAULT") {
        scenario_lower_bound_ = new SimpleRockSampleEastPolicy(this,
          new TrivialParticleLowerBound(this));
    } else {
        cerr << "Unsupported lower bound algorithm: " << name << endl;
        exit(0);
    }
}
```

Once a lower bound is added, and the solver package is recompiled, then users can choose to use it when running bin/despot by setting the **-l** option or the **--plb** option. For example, both of the following commands use `SimpleRockSampleEastPolicy`.

```
bin/despot -q simplerocksample --runs 100
bin/despot -q simplerocksample --runs 100 -l EAST
```

3. Other Examples

See examples/cpp_models for more model examples. We implemented the cpp models for Tiger [3], Rock Sample [2], Pocman [4], Tag [5], and many other tasks in examples/cpp_models.

4. References

[1] A. Somani and N. Ye and D. Hsu and W.S. Lee. DESPOT: Online POMDP Planning with Regularization. In Advances In Neural Information Processing Systems, 2013.
[2] T. Smith and R. Simmons. Heuristic Search Value Iteration for POMDPs. In Proc. Uncertainty in Artificial Intelligence, 2004.
[3] Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial Intelligence 101(1) (1998).
[4] Silver, D., Veness, J.: Monte-Carlo planning in large POMDPs. In: Advances in Neural Information Processing Systems (NIPS) (2010).
[5] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In Proc. Int. Jnt. Conf. on Artificial Intelligence, pages 477-484, 2003.