Tutorial on Using DESPOT

Table of Contents

1. Overview

DESPOT[1] is an anytime online POMDP planning algorithm. It performs heuristic search in a sparse belief tree conditioned under a set of sampled "scenarios". Each scenario considered in DESPOT comprises a sampled starting state, referred to as a "particle" in this tutorial, together with a stream of random numbers to determinize future transitions and observations. To use our solver package, the user first needs to represent the POMDP in one of the following ways:

- specify the POMDP in 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 DESPOT solver package (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. The C++ model requires more programming, but it comes with the full flexibility of integrating the 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. First we introduce the POMDP problem itself and explain how DESPOT can solve it given its C++ model (Section 2.1). Then we explain how to code a C++ model from scratch including the essential functions (Section 2.2) and optional ones that may make the search more efficient (Section 2.3). Finally, Section 3 gives references to other example problems.

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, such as constructing intial belief
- 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 use a simplified version of the *RockSample* problem [2] as our running example. The complete C++ model of this example can be found in examples/cpp_models/**simple_rock_sample**. Note that the examples/cpp_models/rock_sample folder contains a more complex version of the RockSample problem. The RockSample POMDP models a rover on an exploration mission. The rover can achieve rewards by sampling rocks in its current 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 $A = \{West, East, Sample, Check\}$.

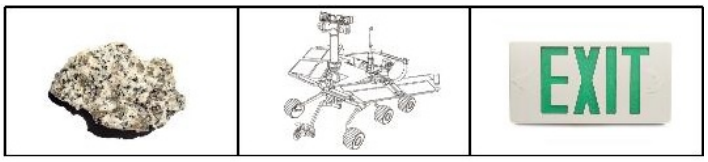

Figure 1. The 1 x 3 *RockSample* problem world.

As with the original version of the problem, the rover knows exactly its own location and the rock's location, but it is unaware of the status of the rock (good or bad). It can execute the *Check* action to get observations of the status ($O = \{Good, Bad\}$), and its observation is correct with probability 1 if the rover is at the rock's location, 0.8 otherwise. 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 and terminates the task. Moving off the grid and sampling in a grid where there is no rock result in a penalty of −100, and terminate the task. All other moves have no cost or reward.

2.1.1 Using C++ Models

DESPOT can 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

```
 1  class TUI: public SimpleTUI {//TUI: text user interface
 2  public:
 3    TUI() {
 4    }
 5
 6    DSPOMDP* InitializeModel(option::Option* options) {
 7      DSPOMDP* model = new SimpleRockSample();
 8      return model;
 9    }
10
11    void InitializeDefaultParameters() {
12    }
13  };
14
15  int main(int argc, char* argv[]) {
16    return TUI().run(argc, argv);
17  }
```

To solve other problems, for example *Tiger* [3], the user may implement the `DSPOMDP` interface as a class called `Tiger`. Then the user only needs to change Line 7 in Listing 1 to:
`DSPOMDP* model = new Tiger();`
If there are paremeters to set for the problem, they can be specified in `InitializeDefaultParameters()` (Line 11 in Listing 1).

2.2. Essential functions

The following code snippet shows the essential functions in the `DSPOMDP` interface. It also shows some displaying functions, which are used for debugging and are not required by the solver to work correctly. We will only discuss the essential functions.

Listing 2. Essential functions in the DSPOMDP interface

```
class DSPOMDP {
public:
    /* ======= Essential Functions: ========*/

    /* 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 pure virtual functions are also required to be implemented.
    However, they are not required by the solver to work correctly.
    Those functions only output some information for debugging.
    Hence we won't discuss them in this tutorial*/

    /* ======= Display Functions: ========*/

    /* Prints a state. */
    virtual void PrintState(const State& state, std::ostream& out = std::cout)const = 0;
    /*  Prints an observation. */
    virtual void PrintObs(const State& state, OBS_TYPE obs, std::ostream& out = std::cout)const = 0;
    /* Prints an action. */
    virtual void PrintAction(int action, std::ostream& out = std::cout) const = 0;
    /* Prints a belief. */
    virtual void PrintBelief(const Belief& belief, std::ostream& out = std::cout)const = 0;
    /* Returns number of allocated particles. */
    virtual int NumActiveParticles() const = 0;
};
```

The following declaration of the `SimpleRockSample` class implements the `DSPOMDP` interface above. The code is the same as the interface except that the functions are no longer pure virtual, and a `MemoryPool` object is declared for memory management. In the following we will discuss each function and its implementation in detail.

Listing 3. 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.*/
    double GetMaxReward() const;
    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

The state, action and observation spaces are three basic components of a POMDP model.

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, which will be discussed 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 the `State` object represents a weighted particle.

Listing 4. 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 could actually use the generic state class to represent its states by mapping each state to an integer, but we define customized state class to illustrate how this can be done.

Listing 5. 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 simply returns the total number of actions.

Listing 6. Implementation of `NumActions()` for `SimpleRockSample`.

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

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

Listing 7. 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 does not need to be consecutive integers. Note that both actions and observations need to be represented as or mapped into integers due to some implementation constrains. For `SimpleRockSample`, we use an `enum` to represent the observations.

Listing 8. 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) = <s', o>* such that when random number *r* is randomly distributed in *[0,1]*, *<s', o>* is distributed according to *P(s', o | 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'*, the corresponding reward and observation. 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 if and only if executing *a* on *s* results in a terminal state.

Listing 9. 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 == LEFT) {
        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;
            // when the rover at LEFT, its observation is correct with probability 1
            obs = (rock_status == R_GOOD) ? O_GOOD : O_BAD;
        } else if (action == A_WEST) {
            reward = -100;
            // moving does not incur observation, setting a default observation
            // note that we can also set the default observation to O_BAD, as long
            // as it is consistent.
            obs = O_GOOD;
            return true; // Moving off the grid terminates the task.
        } else { // moving EAST
            reward = 0;
            // moving does not incur observation, setting a default observation
            obs = O_GOOD;
            rover_position = MIDDLE;
        }
    } else if (rover_position  == MIDDLE) {
        if (action == A_SAMPLE) {
            reward = -100;
            // moving does not incur observation, setting a default observation
            obs = O_GOOD;
            return true; // sampling in the grid where there is no rock terminates the task
        } else if (action == A_CHECK) {
            reward = 0;
            // when the rover is at MIDDLE, its observation is correct with probability 0.8
            obs =  (rand_num > 0.20) ? rock_status : (1 - rock_status);
        } else if (action == A_WEST) {
            reward = 0;
            // moving does not incur observation, setting a default observation
            obs = O_GOOD;
            rover_position = LEFT;
        } else { //moving EAST to exit
            reward = 10;
            obs = O_GOOD;
            rover_position = RIGHT;
        }
    }
    if(rover_position == RIGHT) return true;
    else return false;
}
```

2.2.3. Beliefs and Starting States

Our solver package supports arbitrary belief representations: The user may use a custom belief representation by implementing the Belief interface, which only needs to support sampling of particles, and updating the belief.

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

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

See 2.3.1 Custom Belief for further details on implementing a custom belief class.

As an alternative to implementing an own belief class, one may use the ParticleBelief class available in the solver package. The 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 function to be implemented is 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 10. 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) {
            // when the rover at LEFT, its observation is correct with probability 1
            return obs == rock_status;
        } else if (rover_position == MIDDLE) {
            // when the rover at MIDDLE, its observation is correct with probability 0.8
            return (obs == rock_status) ? 0.8 : 0.2;
        }
    }

    // when the actions are not A_CHECK, the rover does not receive any observations.
    // assume it receives a default observation with probability 1.
    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 parameter start, 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 11. Initial belief for SimpleRockSample

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

        if (type == "DEFAULT" || type == "PARTICLE") {
            //Allocate() function allocates some space for creating new state;
            SimpleState* good_rock =static_cast<SimpleState*>(Allocate(-1, 0.5));
            good_rock->rover_position = MIDDLE;
            good_rock->rock_status = O_GOOD;
            particles.push_back(good_rock);

            SimpleState* bad_rock =static_cast<SimpleState*>(Allocate(-1, 0.5));
            bad_rock->rover_position = MIDDLE;
            bad_rock->rock_status = O_BAD;
            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 12. Sample a starting state from the initial belief for `SimpleRockSample`

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

2.2.4. Bound-related Functions

The heuristic search in DESPOT is guided by upper and lower bounds on the discounted 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 (uninformative bounds).

The `GetMinRewardAction` function returns *(a, v)*, where *a* is an action with largest minimum immediate reward when it is executed, and *v* is its minimum immediate reward. This can be comupted by first finding the immediate reward for each action *a* in the worst case, i.e. the minimum immediate reward for each *a*. The `GetMinRewardAction` function should then return the largest of these worst case immediate reward values, and the corresponding action.

In the simple rock sample problem, the worst case for executing the *Sample* action is when the agent is not on a rock, where the minimum immediate reward of *Sample* is -100. In the worst case, executing *West* action causes a penalty of -100 when the agent is in the left grid and would go off the grid by moving west. The minimum immediate reward of *East* is 0 and the minimum immediate reward of *Check* is 0. The largest minimum immediate reward is 0, and the corresponding action is *East* or *Check*. We may choose either of them, i.e., *(East, 0)* or *(Check, 0)*.

DESPOT uses these values to bound the minimum discounted infinite-horizon value that can be obtained on a set of scenarios. When the weight of the scenarious is *W*, the minimum discounted infinite-horizon value is bounded by $Wv / (1 - \gamma)$, where γ is the discount factor. (There is no need to implement this bound, it is included in DESPOT.)

Listing 13. Implementation of `GetMinRewardAction` for `SimpleRockSample`

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

The `GetMaxReward` function returns the maximum possible immediate reward R_{max}. Unlike `GetMinRewardAction`, there is no need to return the corresponding action. DESPOT then bounds the maximum discounted infinite-horizon value that can be obtained on a set of scenarios with total weight *W* by $W R_{max} / (1 - \gamma)$, where γ is the discount factor.

Listing 14. 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 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. We have implemented the memory management class. To use it the user only needs to implement the following three functions.

Listing 15. 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. Optional Functions

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 needs 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 the `FullChainBelief` class in examples/cpp_models/chain for an example.

```
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 Section 2.2.4 are non-informative and generally only work for simple problems. This section gives a brief explanation on how users can create their own lower bounds. Creating an upper bound can be done similarly. Examples can also be found in the code in examples/cpp_models directory. Note that only GetMaxReward() and GetMinRewardAction() functions are required to be implemented if one does not want to use custom bounds. However, it is highly recommended to use bounds based on domain knowledge as it often improves performance significantly.

A new type of lower bound is defined as a child class of the `ScenarioLowerBound` class shown in Listing 16. The user needs to implement the `Value` function that 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 needs 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 represented by a `RandomStreams` object.

Listing 16. 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;
};
```

The user can customize the lower bound by implementing an own lower bound class inheriting directly from the abstract `ScenarioLowerBound` class. Alternatively, we also provide two custom lower bound classes, `ParticleLowerBound` and `Policy`, that inherit from `ScenarioLowerBound` and are already implemented in the solver package.

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. Listing 17 shows the interface of `ParticleLowerBound`. To use `ParticleLowerBound`, one needs to implement the `Value` function shown below.

Listing 17. 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;
};
```

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 Policy. To use Policy, one needs to implement Action function shown below.

Listing 18. 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 19. A simple fixed-action policy for SimpleRockSample.

```
class SimpleRockSampleEastPolicy : public policy {
    public:
        enum { // action
            A_SAMPLE = 0, A_EAST = 1, A_WEST = 2, A_CHECK = 3
        };
        SimpleRockSampleEastPolicy(const DSPOMDP* model, ParticleLowerBound* bound)
            : Policy(model, bound){}

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

Other examples for implementing lower bound classes can be found in examples/cpp_models. For example, PocmanSmartPolicy implements a policy for the Pocman [4] task.

After implementing the lower bound class the user needs to add it 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 20. 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 21. 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 package is recompiled, the user can choose to use it by setting the **-l** option when running the package. For example, both of the following commands use SimpleRockSampleEastPolicy for a task package named simple_rs.

```
./simple_rs --runs 100
./simple_rs --runs 100 -l EAST
```

We refer to /doc/Usage.txt file for the usage of command line options.

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. It is highly recommended to check these examples to gain a better understanding on the possible implementations of specific model components.

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.