Symbolic Planning in Belief Space

by Ciara L. Kamahele-Sanfratello

S.B., Computer Science and Engineering M.I.T., 2015

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2015

Author: _____

Department of Electrical Engineering and Computer Science
May 22, 2015

Certified by: _____

Leslie Pack Kaelbling, Panasonic Professor of Computer Science and Engineering
Department of Electrical Engineering and Computer Science
May 22, 2015

Accepted by: _____

Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

Symbolic Planning in Belief Space
by Ciara L. Kamahele-Sanfratello
Submitted to the Department of Electrical Engineering and Computer Science

May 22, 2015
In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

SASY (Scalable and Adjustable SYmbolic) Planner is a flexible symbolic planner which searches for a satisfying plan to a partially observable Markov decision process, or a POMDP, while benefiting from advantages of classical symbolic planning such as compact belief state expression, domain-independent heuristics, and structural simplicity. Belief space symbolic formalism, an extension of classical symbolic formalism, can be used to transform probabilistic problems into a discretized and deterministic representation such that domain-independent heuristics originally created for classical symbolic planning systems can be applied to them. SASY is optimized to solve POMDPs encoded in belief space symbolic formalism, but can also be used to find a solution to general symbolic planning problems. We compare SASY to two other POMDP solvers, SARSOP and POMDPX_NUS, and define a new benchmark domain called Elevator.

Acknowledgements

This thesis would not have been possible without the mentorship and guidance of my wonderful advisor, Leslie Pack Kaelbling. Thank you for being so reliable and always making time for me! And thank you so much for the invaluable ideas, feedback, and especially the encouragement you provided at every step along the way.

I would also like to thank everyone who helped me use and explore their planners, particularly Ye Nan, Amanda Coles, Derek Long, and Maria Fox.

Tim Mickel helped me name SASY Planner and Miles Steele thought of the awesome story behind my benchmark domain – thanks!

I would like to thank Chris Terman for making my extended stay at MIT possible – I would not have realized I could start my M.Eng. at the time I did if not for you, and being a TA for 6.004 was an absolutely enjoyable way to fund my research.

Thanks to Brandon Vasquez for letting me bounce ideas off you, staying up late with me, and always being there for me.

Finally, I would like to thank Leo Gruenschloss for reading and editing my thesis, and especially for your love and support throughout the whole journey.

Contents

1 Introduction

Imagine that you are planning to prepare a meal: you envision collecting several food items from around the kitchen, cooking some of them for varying amounts of time, and cleaning the dishes when you are done. There are probably many parts of your plan that you cannot know for certain in advance, such as the exact cooking time of each recipe. Some parts of your plan depend on the success of other tasks, and there are many ways that your plan could go wrong: you might accidentally overcook food, realize you have forgotten to buy some items, or drop and break a dish or two. It is very difficult, if not impossible, to anticipate everything that could happen, and making a plan in advance that explores all possible outcomes and details a backup plan for every conceivable failure is clearly infeasible.

Now, imagine a robot trying to solve this same problem, while simultaneously dealing with limited computation power, faulty sensors, and imperfect tools for achieving its goals. In order to enable robots to make resilient plans for circumstances like these, we can model the dynamics of the situation as a Markov decision process and use online and offline planning systems to help the robot make decisions and adapt to unforeseen situations along the way.

2 Background

In this section, we explain Markov decision processes, partial observability, and value iteration to provide a foundation for the later discussion of existing planners and ideas for a new planner.

2.1 Markov Decision Processes

A Markov decision process is characterized by states, actions, transitions, and rewards. The states in a Markov decision process are all possible configurations of the world. Applying an action to a particular state of the world triggers a transition from one state to another; especially desirable state transitions can be associated with rewards.

The belief state in a Markov decision process represents the current estimate of the state of the world. In a completely observable Markov decision process, there are a fixed number of discrete world states that the belief state transitions between. In a partially observable

Markov decision process, or a POMDP, the belief state is a continuous distribution over all possible states.

Transitions in an MDP or a POMDP can be probabilistic; thus, applying an action to a state might result in one of several different new states. The transition distribution maps state and action pairs to a distribution over all possible new states that can be obtained by applying the action to the state.

While state transitions are not completely observable, we can sometimes make other kinds of observations to give us a better idea of what state we might have transitioned to. Similar to the transition distribution, the observation distribution maps state and action pairs to a distribution over possible observations that can come from applying the action to the state.

2.2 POMDP Value Iteration

In order to solve a POMDP, we want to find a sequence of actions that will bring us from an initial state to a desirable terminal state while maximizing reward along the way; the actions we choose will depend on the observations we get. We can decide which action to take depending on our current belief state and the last observation we received using a policy. A policy can be thought of as a directed graph where each node represents a belief state and action pair, and each edge represents a possible observation. An optimal policy shows which action to take from each belief state to earn the maximal reward.

Figure 1 shows an example of a simple policy graph describing the optimal behavior of a mobile robot which is trying to move from location 0 to location 1. The robot's move actuator works with probability 0.8 and the robot's look sensor is correct with probability 0.9. The states are labelled with the robot's belief state about its location. The robot initially believes itself to be in location 0 or 1 with equal probability. The robot tries to move from location 0 to location 1. It uses its sensor to see whether it moved successfully, and either tries to move again or moves into the terminal state depending on the observation it receives.

When solving a POMDP, we can make two very important assumptions. The first assumption is that there are a finite number of actions and observations. This means that from a given state, there are a finite number of states that we can transition to. The second assumption is that the next state depends only upon the current state, the chosen action, and the received observation; since each belief state is a continuous distribution over states, and because a belief update deterministically transitions to a new belief state for a given initial belief state,

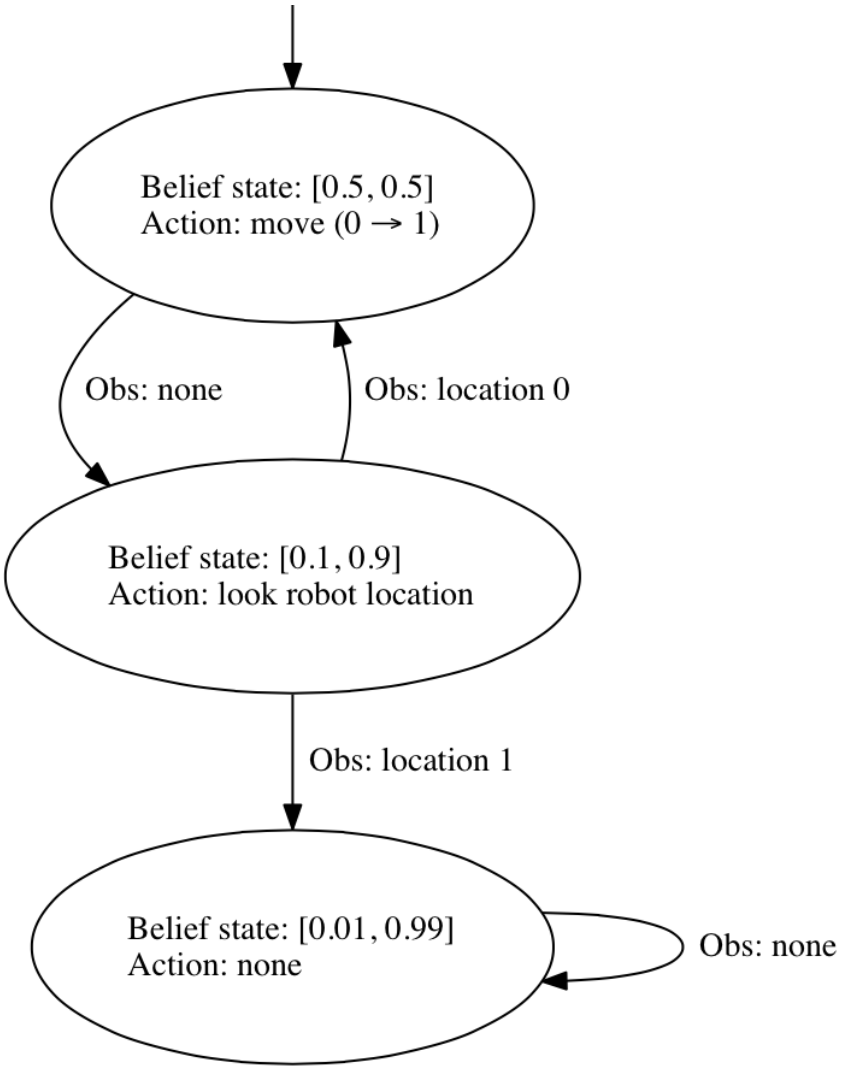

Figure 1: Graph describing the optimal policy for a mobile robot trying to move from location 0 to location 1. Nodes are labelled with possible belief states and the action to take for such a belief state. Edges are labelled with observations received from taking actions. Belief state: $[p_0, p_1]$ means that the robot believes itself to be in location 0 with probability p_0 and in location 1 with probability p_1.

action, and observation, the belief state at any moment in time is all we need to encapsulate the entire history of actions since the initial state.

Another important consideration when computing an optimal policy is the horizon, or how far into the future we need to plan for. If we use a finite horizon, then the agent assumes it will be forced to terminate its actions after a given number of steps. If the horizon is infinite, the agent assumes it will live forever. Because it is quite unlikely we will always know the exact horizon size to use, we can encourage the agent to make more greedy choices in the near future by using an infinite horizon with a discounting factor. With a discounting factor, rewards are discounted geometrically with $0 < \gamma \leq 1$, meaning that the reward the agent receives in step n is multiplied by γ^n. An infinite horizon with a discounting factor of γ is analogous to a finite horizon of size $\frac{1}{1-\gamma}$.

To find an optimal policy for a finite-horizon POMDP, we can compute a *value function*. A value function specifies for each state the expected discounted future reward of starting in that state and acting optimally, so a greedy policy can easily be derived from it. To find the best path through the state space to achieve our goals, we consider the set of actions we can take from the initial state. We can assign a value to each action by taking an expectation over possible observations and their associated reward for each action. We then choose the action with the maximum value. Because we said that the number of belief states we can transition to is finite, a value function for a finite-horizon POMDP can always be represented as a finite set of vectors. The vector with the largest dot product with our belief state determines the value of the belief state and the optimal action to take.

By repeating this process, we can iteratively compute the optimal finite-horizon value function for increasingly large horizons. As we grow this finite horizon very large, we can approximate the infinite-horizon discounted value function. From this approximate value function, we can compute an approximately optimal policy for a discounted infinite-horizon POMDP.

3 Contemporary POMDP Solvers

Value iteration allows us to compute an approximately optimal policy at the cost of scaling quite poorly for two outstanding reasons: the curse of dimensionality and the curse of history.

The curse of dimensionality refers to a problem inherent in solving POMDPs with a very

large number of possible states. The dimensionality of the belief state vector used by value iteration is equal to the number of states of the POMDP.

The curse of history arises as we grow our finite horizon larger and larger to approximate an infinite-horizon discounted value function. The number of possible histories that value iteration must consider is exponential in the length of the horizon.

Due to the curses of dimensionality and history, finding an optimal solution to a POMDP is highly intractable. To mitigate these challenges, current state-of-the-art POMDP solvers use a variety of insights to simplify the problem, including clever sampling of points from the belief space to make searches smaller and faster, online anytime algorithms, and approximate state estimation.

3.1 SARSOP

The SARSOP (Successive Approximations of Reachable Space under Optimal Policies) POMDP solver uses a sampling of points from the belief space. The key insight of SARSOP is to begin with the subset of the belief space which can be reached from a given initial state, and then iteratively converge to the subset of the belief space that can be reached through only optimal sequences of actions [2]. The subset of the belief space reachable from the initial state, $R(b_0)$, may be much larger than the subset reachable through only optimal sequences of actions, $R^*(b_0)$. If we can find $R^*(b_0)$, it is much easier to find an approximation for the value of an optimal policy compared to trying to directly approximate the value of an optimal policy from $R(b_0)$ or finding an optimal policy in $R(b_0)$ and calculating its value.

SARSOP maintains an upper and lower bound on the optimal value function. It samples a set of reachable points in the belief space, starting from the initial state. Then, it chooses some nodes to back up. Backing up a node entails propagating information from all of the children of a node up to the node, and then either changing the lower bound on the value function accordingly or discontinuing exploration of the node. SARSOP continues this process until it has driven the lower bound sufficiently close to the upper bound, or until the time limit is reached.

SARSOP uses several other techniques to avoid sampling from regions of the belief space that are unlikely to be optimal. SARSOP does not sample in regions disqualified by the current upper and lower bounds on the value function because these regions are unlikely to contain an optimal policy. As SARSOP continues to sample further from the initial state, sampling

paths are increasingly less likely to be paths reachable by only optimal actions. For this reason, SARSOP attempts to keep sampling paths as short as possible and only chooses to sample deeply in a very selective way that allows it to continually improve the lower bound on the value function. If a deep sampling path does not continue to significantly improve the lower bound on the value function, the path is terminated. Additionally, SARSOP uses a technique called belief point pruning where it prunes alpha-vectors if they are dominated by other alpha-vectors over the optimally reachable subset of the belief space rather than by alpha-vectors over the entire belief space.

3.2 POMCP

POMCP (Partially Observable Monte-Carlo Planning) is an algorithm for online POMDP planning [3]. It uses a modified Monte-Carlo tree search to find the next action to take from a given belief state, and it performs a Monte-Carlo update of the belief state which only requires a black box simulator of the POMDP. This allows POMCP to tackle problems whose domains are too complicated to have time to consider all possible state transitions, and to work on problems with domains too large to be able to update the entire explicit probability distribution.

Monte-Carlo tree search explores a search tree with a node for every state. Each node has a value for each action explored from that state. Values are assigned by taking the average value of many simulations made by starting at that node's state and taking that action. Each simulation has two parts: a tree policy and a rollout policy. The tree policy is used to search within the tree, for example a greedy policy in basic MCTS or the UCT algorithm [4], and the rollout policy is used to select the actual action to take. After each simulation, the first state visited in the rollout policy is added to the search tree. Following the execution of each action, the belief state is updated using an unweighted particle filter rather than a large, explicit belief update.

PO-UCT extends the UCT algorithm to be able to work under partial observability by making each node in the search tree represent a history rather than a state. The value of each node represents the value of the history. Simulations are modified to start from a state sampled from the belief state and to proceed with a first stage, similar to the tree policy, which tries to maximize the value of the history, and then with a second stage, similar to the rollout policy, where the first history encountered is added to the search tree.

The complete POMCP algorithm uses PO-UCT to perform simulations, and when the search

finishes, the action with the greatest value is executed. Then, an observation and state transition are sampled from the black box simulator. The belief state is updated using a particle filter. If the belief state is correct, the POMCP algorithm will converge to the optimal policy for any finite horizon POMDP.

3.3 DESPOT

The DESPOT (Determinized Sparse Partially Observable Tree) algorithm uses sampling to reduce the search space in the same spirit as POMCP [5]. However, DESPOT is able to avoid the worst-case performance of POMCP by evaluating policies on a small set of sampled scenarios. The search algorithm looks for a promising policy in the DESPOT starting from the current belief state, executes one step of the policy, and repeats.

A DESPOT is like a sparsely-sampled belief tree – it contains only nodes reachable through the execution of all possible policies under a set of sampled scenarios. To construct a DESPOT, a deterministic simulative model is applied to the set of sampled scenarios from an initial belief state for all possible sequences of actions. This process results in a tree that is similar to a standard belief tree, but with some of the observation branches removed.

The B-DESPOT (Basic DESPOT) algorithm is used to evaluate a policy under the set of sampled scenarios. First, the policy is simulated under all the scenarios and the the total discounted reward of the policy is found. Then, a belief tree search is used to find a policy that performs well under the set of sampled scenarios.

Because B-DESPOT chooses the best action at every node of the DESPOT, it can sometimes overfit a policy to the set of sampled scenarios so that it ends up performing very well for the scenarios but not well in general. R-DESPOT (Regularized DESPOT) regularizes to balance the estimated performance of a policy under the set of sampled scenarios with the size of the policy. Essentially, if a subtree from any internal node in the DESPOT becomes too large, the search is terminated from that node onwards and a simple default policy is used.

AR-DESPOT (Anytime Regularized DESPOT) uses a heuristic search and branch-and-bound pruning to partially construct a DESPOT. Then, it searches for a policy in the DESPOT that maximizes regularized utility.

3.4 POMDPX_NUS

POMCP converges quickly in cases when its optimistic action selection strategy is successful, but it has very poor worst-case performance. Because DESPOT uses a belief tree built only from a set of sampled scenarios, it has very good worst-case performance and also good performance in practice. POMDPX_NUS combines the two planners by running both DESPOT and POMCP at the same time on a problem, and then switches to only running the planner which performs better on the problem after a sufficient amount of results have been gathered.

4 Benchmark Domains

In this section we define a new POMDP benchmark domain called Elevator. We also discuss the merits of a widely-used benchmark domain called RockSample, and explore the performance of SARSOP on modified versions of RockSample.

4.1 Elevator

Elevator is a new POMDP benchmark domain. In an instance of Elevator, a robot moves multiple elevators in a single elevator shaft from one set of floors to another set of floors. Since the elevators share a shaft, only one elevator can be on a floor at a time, and the elevators are unable to move past each other within the shaft. The robot runs along the wall of the elevator shaft, giving it the ability to move the elevators within the shaft without taking up space in the shaft. The robot is equipped with a clamp which can attach to and release elevators. The robot has several actions which it can choose from: the robot can move up and down, attach to and release an elevator, and use its sensors to observe the floor each elevator is currently on, the elevator it is currently attached to, and its position in the shaft.

An instance of Elevator is defined by the number of floors, the number of elevators, and the initial and goal configurations of the elevators. The robot always knows the number of floors and elevators, but is uncertain about its own position, the elevator it is attached to, and the positions of the elevators in the shaft. When the robot tries to move, it occasionally is not successful and does not move at all. Similarly, attaching to and releasing elevators is not always successful. When the robot does manage to attach to an elevator, it may not

be attached to the elevator it expects. The robot can become more certain of its position and the configuration of the elevators in the shaft by using look actions. Once the robot thinks that all elevators have been moved to their goal floors, it moves above the top floor of the building. A more detailed description of the Elevator benchmark domain is included in Appendix A.

4.2 RockSample

RockSample is a widely-used benchmark domain for POMDP solvers. It models a robot equipped with a long-range sensor, exploring a region with rocks to potentially sample depending on the quality of the rock.

"An instance of RockSample with map size $n \times n$ and k rocks is described as RockSample$[n, k]$. The POMDP model of RockSample$[n, k]$ is as follows. The state space is the [Cartesian] product of $k + 1$ features: Position $= \{(1, 1), (1, 2), ..., (n, n)\}$, and k binary features $\text{RockType}_i = \{\text{Good}, \text{Bad}\}$ that indicate which of the rocks are good. There is an additional terminal state, reached when the rover moves off the right-hand edge of the map. The rover can select from $k + 5$ actions: North, South, East, West, Sample, $\text{Check}_1, \ldots, \text{Check}_k$. The first four are deterministic single-step motion actions. 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 (indicating that nothing more can be gained by sampling it). If the rock is bad, it receives a penalty of -10. Moving into the exit area yields reward 10. All other moves have no cost or reward. Each Check_i action applies the rover's long-range sensor to rock i, returning a noisy observation from $\{\text{Good}, \text{Bad}\}$. The noise in the long-range sensor reading is determined by the efficiency η, which decreases exponentially as a function of Euclidean distance from the target. At $\eta = 1$, the sensor always returns the correct value. At $\eta = 0$, it has a 50/50 chance of returning Good or Bad. At intermediate values, these behaviors are combined linearly. The initial belief is that every rock has equal probability of being Good or Bad" [6].

It is important to note that the RockSample benchmark problem is technically a mixed observability MDP because the state variable corresponding to the robot's position is completely observable and all state transitions are deterministic. The robot knows its initial position and the initial position of all the rocks, and always moves successfully. The robot's sensor determines the quality of the rocks probabilistically. If the robot is at a distance d from the rock it is trying to sense, the accuracy of the sensor is e^{-d}. This means that if the robot is in the same spot as the rock, its sensor works perfectly, but if the robot is only

one location away from the rock, its sensor is correct with probability 0.68. This strongly encourages the robot to be in the same location as a rock before sampling it in order to acquire useful information, and sampling a rock from the same location always removes all ambiguity about its quality.

4.3 SARSOP Performance on RockSample

Many POMDP solvers are able to solve RockSample at impressive speeds with high average rewards. In the standard version of RockSample, the robot's sensor has an efficiency of e^{-d}. Unsurprisingly, this drastic penalty for using the sensor at any meaningful distance from a rock strongly discourages the robot from using the sensor from anywhere but directly on top of a rock. Additionally, because of the high prior probability of a rock being good (0.5), it makes sense for the robot to always try sensing every rock in the map before exiting.

The standard configuration of the RockSample problem therefore results in many solvers quickly computing roughly the same solution for every instance of the problem: the robot chooses an optimal walk to the right edge of the map that passes through all locations that contain a rock. The robot checks each rock from directly on top of it along the way, and pauses to sample if the rock is observed to be good. In these cases, the problem is essentially reduced to a fully-observable MDP.

To explore the other types of plans that solvers might come up with, we experimented with varying the robot's sensor efficiency, prior belief of the quality of rocks, and rewards. We found that these changes strongly influence the solution found by SARSOP.

When the sensor efficiency is fixed at a constant 0.75, the computation time of SARSOP greatly increases. The robot performs all checks from far away because it has no incentive to move towards a rock before sensing it. Depending on the map size, the robot continues to sample each rock until it has received sufficiently more good or bad observations. If it receives more good observations, it goes to sample the rock, and if it receives more bad observations, it skips the rock.

With a sensor efficiency that decreases linearly by 0.1 for each distance unit between the robot and a rock, the computation time of SARSOP decreases. Instead of walking through all rock locations, the robot checks a rock several times from afar, and then either decides to go sample the rock or ignore it entirely. Before sampling the rock, the robot sometimes checks the rock again from closer to ensure that the rock is actually good.

RockSample	(1, 2)	(1, 2)	(1, 2)	(5, 5)	(5, 5)	(5, 5)	(7,8)	(7,8)	(7,8)
Sensor efficiency (η)	e^{-d}	0.75	$1 - 0.1d$	e^{-d}	0.75	$1 - 0.1d$	e^{-d}	0.75	$1 - 0.1d$
Mean time (s)	0.01	0.01	0.01	600	600	229	600	600	600
Upper bound (u_f)	-	-	-	19.857	19.539	-	24.852	24.317	24.713
Lower bound (l_f)	-	-	-	19.237	16.917	-	21.054	17.011	21.180
Gap ratio	-	-	-	0.03	0.14	-	0.17	0.35	0.15
Alpha vectors	4	19	5	4155	7601	4328	4590	4824	5097

Table 1: Comparison of SARSOP planning for RockSample domains with varying sensor efficiency. SARSOP was given a maximum of 600 seconds to plan offline. If SARSOP did not fully close the gap between the upper and lower bounds in this time, the final upper and lower bounds are given, as well as the gap ratio, $g = \frac{u_f - l_f}{(u_f + l_f)/2}$ where u_f and l_f are the final upper and lower bounds respectively. The number of alpha vectors roughly approximates the length and branching factor of the resulting plan, and is proportional to the length of the output file.

Lowering the prior belief of the quality of the rocks also increases the computation time for SARSOP and encourages the robot to check rocks from a distance, or even completely disregard rocks that are too far out of the way. Giving a penalty to movement produces similar results to changing the sensor efficiency to a linearly decreasing function of distance.

5 Idea for a Scalable POMDP Planner

While contemporary POMDP planners excel on the existing set of benchmarks, there are not many which can solve domains with continuous belief states and perform well with increasing domain size. We present a design for a POMDP planner which can solve larger-scale continuous domains through determinization, approximation, and the insightful application of heuristics used in symbolic planning.

For the remainder of this chapter, we will use a simple robot manipulation domain to explain ideas. In our example domain, the robot can move left and right above a one-dimensional world. The one dimensional world contains objects which the robot can grasp. If the robot moves to a new location while holding an object, the object moves to the new location as well.

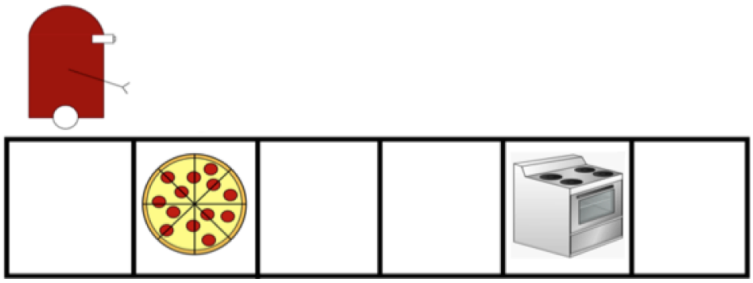

Figure 2: Depiction of the robot in our example domain. The robot can move left and right above a one-dimensional world with 6 locations and 2 objects. The robot is equipped with an arm which can reach down into the world and grasp an object.

5.1 Classical Symbolic Planning

In classical symbolic planning, states are represented as the conjunction of discrete variables called *fluents*. In the example domain in Figure 2, the state is a set of fluents describing the location of the robot, the object which it is holding, and the locations of objects within the world:

State = {robot loc = 0, holding = none, obj 0 loc = 1, obj 1 loc = 4}

Actions are defined by preconditions and effects. The preconditions of an action are a set of fluents that must be true in a state in order to apply the action to the state. The effects of an action are split into two sets of fluents: an add list and a delete list. If an action is applied to a state, the fluents in the add list are added to the state's set of fluents and the fluents in the delete list are removed. A state satisfies a goal state if the state contains all fluents present in the goal state. Movement and grasping in our example domain can be described using the following actions:

```
Move(l, l'):
  pre:    robot loc = l
          holding   = none
  add:    robot loc = l'
  delete: robot loc = l

Pick(o, l):
  pre:    robot loc = l
          o loc     = l
          holding   = none
  add:    holding   = o
  delete: holding   = none
```

When the robot executes Move from location l to location l', it must start in location l and be holding nothing. If both of these conditions are satisfied, the robot will move to location l' and will no longer be in location l. When the robot executes Pick, it must be holding nothing and attempting to pick up an object in the same location as itself. If both of these conditions are satisfied, the robot will be holding the object and will no longer be holding nothing.

If we apply Move(0, 1) to initial state {robot loc = 0, holding = none, obj 0 loc = 1, obj 1 loc = 4}, we achieve the resulting state {robot loc = 1, holding = none, obj 0 loc = 1, obj 1 loc = 4}. If we apply Pick(0, 1) to our new state, we achieve the resulting state {robot loc = 1, holding = 0, obj 0 loc = 1, obj 1 loc = 4}. If our goal state were simply {holding = 0}, our new state would satisfy the requirements of our goal state.

5.2 Belief Space Planning

The simplicity of symbolic planning makes it very attractive in terms of designing search algorithms and domain-independent heuristics. A drawback of symbolic planning is that the state of the world is discretized and fully observable at all times; it is intended to be used to define and solve problems with deterministic actions, such as MDPs.

In order to apply symbolic planning to POMDPs, we can create a discretized state representation and determinized transitions. Each variable of the state in a POMDP is represented as a probability distribution over all possible values of the variable. To convert a POMDP state representation into a form that is compatible with symbolic planning, we can add a *belief probability* to each fluent. The belief probability of a fluent represents a lower bound on the the extent to which we believe the fluent to be true in the state. To achieve the most accurate resulting state, we always make the belief probability assignments as stringent as possible, i.e. each fluent is assigned to the largest possible belief probability.

We define B(variable, value, p) to mean Pr[variable = value] $\geq p$. The MDP state {robot loc = 1, holding = 0, obj 0 loc = 1, obj 1 loc = 4} could be converted to the belief space state {B(robot loc, 1, 1.0), B(holding, none, 1.0), B(obj 0 loc, 1, 1.0), B(obj 1 loc, 4, 1.0)}. If the robot is unsure whether it is in location 0 or location 1, its state might instead be {B(robot loc, 0, 0.2), B(robot loc, 1, 0.8), B(holding, none, 1.0), B(obj 0 loc, 1, 1.0), B(obj 1 loc, 4, 1.0)}. This means that the robot believes itself to be in location 0 with probability at least 0.2 and in location 1 with probability at least 0.8. Note that fluents with a belief probability of 0 are implicit but omitted to keep the representation of the belief state compact.

Satisfying a goal state in our new scheme means that all the fluents in the goal state are present in the satisfying state with belief probability greater than or equal to that of the corresponding fluents in the goal state. The belief states {B(robot loc, 1, 0.9)} and {B(robot loc, 1, 1.0)} would both satisfy the goal state {B(robot loc, 1, 0.8)}. Goal states represent a bound on the minimum certainty that must be achieved for each fluent.

Once we have discretized our POMDP state, we must convert the actions of the POMDP into a deterministic form that is compatible with symbolic planning. In a POMDP, executing an action results in a probabilistic state transition and observation distribution. In symbolic planning, actions can always be executed as long as the preconditions of the actions are satisfied by the state. So, the effects of an action in symbolic planning must be deterministic and implicitly assume something about the observation received in order to be able to update the state correctly.

For actions without an associated observation, such as moving or picking up an object, we modify the belief probabilities assigned to each fluent in the add list of the action based on the belief probabilities of fluents in the preconditions and delete list. The Move and Pick operators can be converted as follows, where move_p and pick_p represent the probability of success for the robot's movement and grasping actuators respectively:

```
Move(l, l'):
  pre:    B(robot loc, l, x)
          B(robot loc, l', y)
          B(holding, none, z)
  add:    B(robot loc, l', x · z · move_p + y)
          B(robot loc, l, x · (1 − z · move_p))
  delete: B(robot loc, l, x)
          B(robot loc, l', y)

Pick(o, l):
  pre:    B(robot loc, l, w)
          B(o loc, l, x)
          B(holding, none, y)
          B(holding, o, z)
  add:    B(holding, o, y · w · x · pick_p + z)
          B(holding, none, y · (1 − w · x · pick_p))
  delete: B(holding, none, y)
          B(holding, o, z)
```

Upon execution of Move(l, l'), the robot will successfully move to location l' if it starts in location l, is holding nothing, and the move actuator is successful. The prior probability of the robot being in location l' will increase by the probability of of these three fluents being true. The probability of the robot remaining in location l will be the probability that the robot started in location l but at least one of the other requisite fluents was not true. Upon execution of Pick(o, l), the robot will successfully grasp the object if it is in the same location as the object and it is holding nothing. The probabilities of the robot holding the object and holding nothing are adjusted similarly to the probabilities in Move.

For actions with a set of possible observations, executing an action must make an assumption about which observation will be received. We can add a new operator, Look, to our example domain to demonstrate how to convert an action with a probabilistic observation. The robot can use Look to check whether the robot is in a specific location. If the robot's sensor detects that the robot is in the location it returns a success observation, and if the sensor detects that the robot is not in the location it returns a failure observation. For simplicity, the robot

assumes that it will always receive a success observation from its sensor.

Look(l):
  pre: B(robot loc, l, x)
  add: B(robot loc, l, $\frac{x \cdot \texttt{look\_p}}{x \cdot \texttt{look\_p} + (1-x) \cdot (1 - \texttt{look\_p})}$)
  delete: B(robot loc, l, x)

Let look_p represent the accuracy of the robot's sensor. If the robot executes Look(l) and receives a success observation, it is the case that either the robot is in location l and the sensor was correct (true positive) or the robot is not in location l and the sensor was not correct (false positive). So, because of the assumption of receiving a success observation, the probability that the robot actually is in location l is the probability of the first case.

In order to discourage the robot from choosing actions that assume a highly unlikely observation, we can associate a cost with actions based on the likelihood of the assumed observation. In the case of the Look action, the likelihood of receiving a success observation is $p = x \cdot \text{look\_p} + (1 - x) \cdot (1 - \text{look\_p})$. In order to penalize a Look action which is extremely unlikely to succeed, we can give Look actions a cost inversely related to p such as $\frac{1}{p}$ or $-\log(p)$.

Converting a POMDP into a belief space symbolic planning problem lets us retain the useful aspects of the symbolic scheme while remedying its limitations. A planner and heuristics designed for symbolic planning can be applied to our belief space scheme as long as the operators are adjusted accordingly.

5.3 Relationship between POMDP and SSP MDP

Many POMDPs have both positive and negative rewards. It is useful to be able to convert POMDPs into SSP (Stochastic Shortest Path) MDPs with only positive costs so that they can be solved using common forward searching algorithms and heuristics. We can do this conversion by first turning a POMDP with positive and negative rewards into a POMDP with only negative rewards by adding a constant negative offset to all rewards in the POMDP. As long as the offset is constant over all rewards in the POMDP, the optimal policy will not change. Once we have a POMDP with negative rewards only, we can convert it into a belief space MDP with the process outlined in the previous section. Finally, we can negate all rewards in the belief space MDP and interpret them as costs in an SSP MDP. A solution to the resulting SSP MDP will also solve the original POMDP.

6 Implementation

In this section, we describe heuristics used in classical symbolic planning. Then, we discuss the difficulties we encountered with using existing planners in belief space, and explain our idea for a flexible and scalable symbolic planner which addresses these problems.

6.1 Fast-Forward Planner and Heuristics

The Fast-Forward planner is derived from the Heuristic Search Planner (HSP) algorithm designed by Bonet, Loerincs, and Geffner [7], which uses a forward search directed by a heuristic function. The HSP system calculates its heuristics by relaxing the original planning problem to only apply the add lists and not the delete lists of actions. A planning graph is built by applying all actions to the initial state and then repeatedly applying all actions to all subsequent child states. The *weight* of a fluent is equal to the level of the graph at which the fluent first appears. The weight of all fluents present in the initial state is 0.

The H_Max heuristic function is the largest weight of any fluent in the goal state. The H_Add heuristic function, also referred to as H_HSP, is the sum of the weights of all the fluents in the goal state. A weakness of the HSP system is that this guiding heuristic can produce large overestimates because it will double-count action edges for fluents that are achieved simultaneously by the same action.

In order to alleviate the problems with the HSP heuristic function, the Fast-Forward planner uses a modified heuristic function that is similar in spirit but more often underestimating [8]. The planner uses the Graphplan algorithm to build a planning graph, alternating fluent and action layers until the goal state is achieved. Then, it extracts a relaxed plan by backtracking and counting the exact number of action edges used to achieve the goal state. This relaxed plan extraction yields a heuristic estimating fewer actions than H_Add since it recognizes when two actions share a precondition or sub-goal in the planning graph. Figure 3 illustrates a case where H_Add and H_FF both overestimate the number of actions required and differ from each other.

To use the Fast-Forward planner in the continuous domain of POMDPs, we can encode our belief space planning formalism in PDDL, the input language of many existing symbolic planners. In order to discretize the POMDP belief space, we specify a finite set of non-overlapping probability buckets, each representing a discrete probability interval, and in total covering the range [0.0, 1.0]. We assign each fluent of the POMDP state to one of the

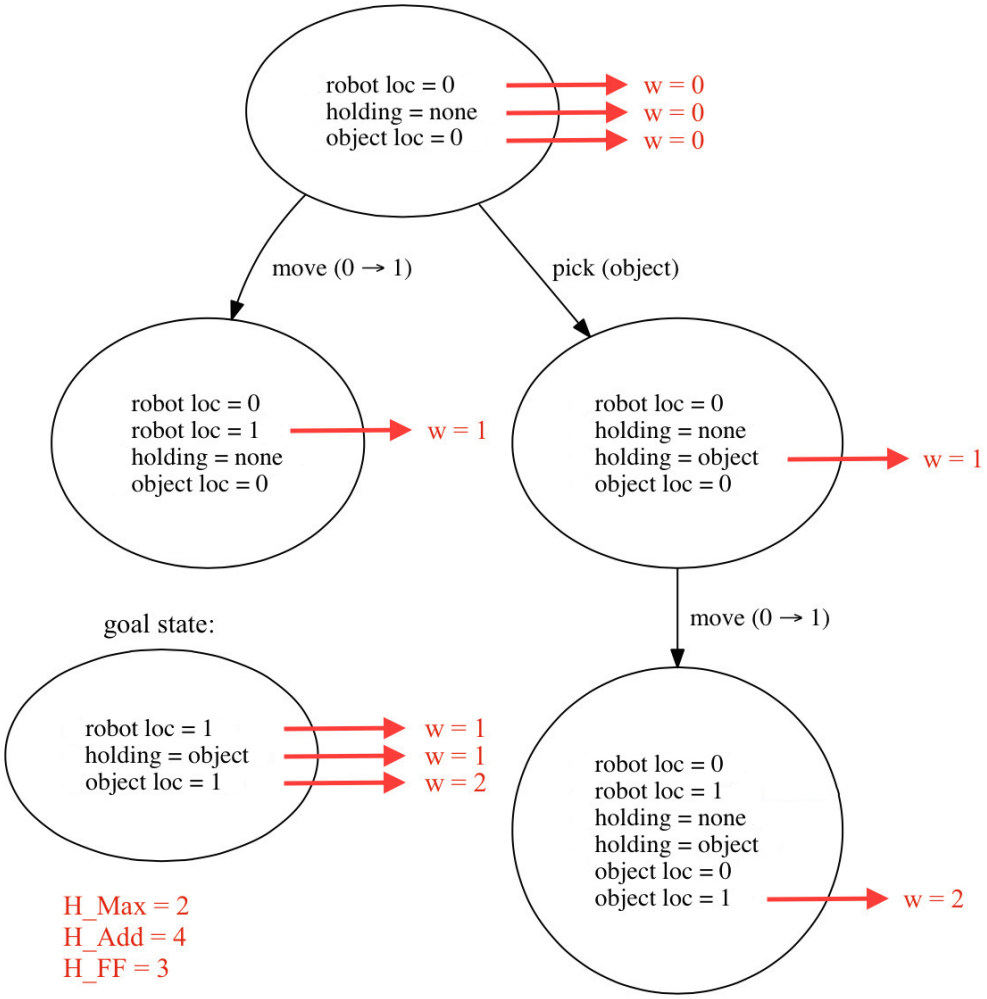

Figure 3: The three fluents existing in the initial state are assigned a weight of 0. The fluents robot loc = 1 and holding = object can both be achieved after executing just one action from the initial state, so they are both assigned a weight of 1. The fluent object loc = 1 can be achieved after executing two actions from the initial state, so it is assigned a weight of 2. The H_Max value is 2 because the maximally weighted fluent in the goal state has a weight of 2. The H_Add value is 4 because the sum of the weights of the three fluents in the goal state is 4. The H_FF value is 3 because holding = object and object loc = 1 share the pick(object) edge, so the total number of unique edges necessary to achieve all three fluents in the goal state is 3.

buckets based on its belief probability. In order for a fluent to be assigned to a bucket, it must be contained in the probability interval represented by the bucket. So, for example, assigning a fluent to the bucket representing [0.7, 0.8) means that the fluent is true with a probability of at least 0.7 and less than 0.8. To avoid overestimation of probabilities, the lower interval bound is used for updating belief probabilities after applying an action.

The way we select our probability buckets will greatly affect the ability of the Fast-Forward planner to choose an intelligent plan. If the buckets are not chosen with a fine enough granularity around the probability range the planner will be working in, the planner might not be able to execute some actions that are necessary for it to take.

Once we have chosen our set of probability buckets, we can design actions that enforce the requisite probability relationships by adding a relationship fluent to the preconditions of each action. The relationship fluent will determine which sets of fluents can be used in the effects of the action, and all possible relationship fluents must be enumerated in the initial domain. The following listing shows how we can convert our Move action into a discretized PDDL action.

```
(:action move
   :parameters
     (?from-loc
      ?to-loc
      ?from-loc-p
      ?to-loc-p
      ?holding-none-p
      ?new-from-loc-p
      ?new-to-loc-p)
   :precondition
     (and (location ?from-loc)
          (location ?to-loc)
          (robot-loc-p ?from-loc ?from-loc-p)
          (robot-loc-p ?to-loc ?to-loc-p)
          (holding-none-p ?holding-none-p)
          (move-relationship ?from-loc-p ?to-loc-p ?holding-none-p
                             ?new-from-loc-p ?new-to-loc-p))
   :effect
     (and (robot-loc-p ?from-loc ?new-from-loc-p)
          (robot-loc-p ?to-loc ?new-to-loc-p)
          (not (robot-loc-p ?from-loc ?from-loc-p))
          (not (robot-loc-p ?to-loc ?to-loc-p)))
)
```

Move acts upon several fluents: two locations, from-loc and to-loc, the probability that the robot is holding nothing, holding-none-p, the original probabilities that the robot is in these locations, from-loc-p and to-loc-p, and the new probabilities that the robot is in these locations, new-from-loc-p and new-to-loc-p. The Fast-Forward planner requires everything to be discrete and cannot evaluate numeric formulas. So, the initial PDDL domain specification must contain all possible legal combinations of from-loc-p, to-loc-p, holding-none-p, new-from-loc-p, and new-to-loc-p to be enumerated under move-relationship fluents according to the same equations we used in the belief space action.

The preconditions of Move check that from-loc and to-loc are locations, that the robot is in from-loc with probability from-loc-p and in to-loc with probability to-loc-p, and that the robot is holding nothing with probability holding-none-p. Additionally, the preconditions enforce that all the probabilities are connected under an existing move-relationship fluent. If they are, the robot updates its belief probabilities about from-loc and to-loc to new-from-loc-p and new-to-loc-p respectively.

6.2 Metric-FF

We are able to use our discretized POMDP state and action formulation as input to an extension to the Fast-Forward planner, Metric-FF [9]. Metric-FF uses the Fast-Forward planner to make a satisfying, but not necessarily optimal, plan while simultaneously minimizing a metric that can be manipulated by actions.

When Metric-FF chooses to take an action, it automatically applies add and delete lists. In a POMDP there is a chance that actions can fail, so simply applying the effects of the chosen action is would be synonymous to assuming that the action was successful. In order to accurately use Metric-FF with our model of a POMDP, we let Metric-FF choosing an action symbolize Metric-FF choosing an action-observation pair in the POMDP. We always assume the observation received is the observation which would correspond to the action being successful, and we adjust the add and delete lists of our actions to account for this assumption. In order to prevent Metric-FF from choosing actions which are very unlikely to succeed, we give each action cost a constant component based on the action taken as well as a variable component which is inversely related to the likelihood of receiving the assumed observation.

Using this approach, we were able to solve small instances of the Elevator problem with Metric-FF. Unfortunately, however, the PDDL problem specification for Metric-FF is limited

to 10,000 lines. While this might seem like a lot of space for a problem specification, all relevant facts for the domain must be detailed as well, including all possible relationship fluents for every action. For a complex problem, such as the Elevator, we were limited to using on the order of 10 probability buckets for each problem instance. Ultimately, this resolution was not sufficient to be able to plan for the larger and more complicated instances we are interested in.

6.3 POPF

POPF is a forward-chaining temporal planner which is able to solve linear equations [10]. After the difficulties we encountered with Metric-FF, we thought that using a planner that could handle equations would alleviate the issues associated with having to enumerate all probability relationships ahead of time.

In our PDDL problem specifications to POPF, we converted all probability values to logarithmic values. Because the probability values we use are between 0 and 1, the corresponding logarithmic values we used were almost all negative. When two probability values needed to be multiplied or divided, we could simply add or subtract them from each other. To implement other nonlinear functions, such as the cost function, we used piecewise linear equations to approximate the original function.

With this system, we were able to plan for small instances of a simplified version of the Elevator problem. However, we were limited in the types of actions we could implement because actions with seemingly dependent assignments were flagged as potentially cyclic and prohibited by POPF. We also encountered issues while using equations with negative numbers. Despite finding workarounds for these obstacles, as the required plan length increased to 8 or more actions, the state space became too large for POPF to search within a reasonable amount of time. We eventually tried using another similar planner similar to POPF, described in [11], but we encountered similar problems to those we had with POPF. Ultimately, we decided that it would be easiest to make our own planner with the ability to handle arbitrary numeric functions.

6.4 SASY Planner

Our POMDP solver has two main components: SASY (Scalable and Adjustable SYmbolic) Planner, to choose an action to take, and a filter, to update our belief state after taking

an action. Additionally, we have a simple simulator to keep track of the true state of the world. At each step, we give a discretized version of our current belief state to SASY, and use it to generate a plan to follow. We execute actions according to the plan, and we use the filter to update our belief state based on the observation we receive from the simulator when executing each action. If an observation we receive from the simulator ever differs from the observation assumed by SASY, we give a discretized version of our new belief state to SASY and replan.

SASY performs a forward search from an initial belief state until it reaches a goal state. The search traverses the SSP MDP resulting from a conversion of the belief space MDP corresponding to the original POMDP. The conversion process is discussed earlier in Section 5.3. The underlying belief space MDP assumes a success observation for every action. Each action has a base cost of 1, and actions with assumed observations augment this base cost with a variable component that is inversely related to the probability of the assumed observation.

To traverse the search tree, SASY employs a priority queue which holds the nodes to be expanded next. For each belief state it removes from the priority queue, SASY considers all possible actions that can be taken. Each possible new belief state resulting from executing an action is added to the priority queue, which sorts nodes by cost. The search terminates once the first goal state has been found and returns the path it took to reach that goal state. The cost of a node is equal to the cost so far to reach the node's state plus the expected heuristic cost to reach the goal state from the node.

SASY can efficiently compute H_Max and H_Add for a node by repeatedly applying all possible actions to the node and finding the maximally weighted fluent and the sum of the weights of all the goal fluents (see Section 6.1). It is more difficult for SASY to compute H_FF because SASY must perform a forward search with an underestimating heuristic (e.g. H_Max) from the node being evaluated to the goal state to find the exact number of edges necessary to reach the goal state. A simple backtrace to find unique edges, like in the classic computation of the heuristic, often generates a gross overestimate. This is because many earlier actions have almost always modified the belief probabilities of fluents that SASY uses again for later actions. Since it is not trivially clear whether these belief probabilities actually needed to be modified in order for SASY to achieve the goal state with later actions, it is computationally expensive to decide whether two goal fluents shared edges on the way to the goal.

SASY attempts to minimize the repeated exploration of very similar nodes by using a mod-

ified comparison function when deciding whether to add a node to the priority queue. If a node with fluents and belief probabilities very similar to a those of a previously visited node is encountered, SASY will not add the node to the queue. This strategy is very helpful when exploring continuous space because it prevents SASY from repeatedly searching down similar paths which do not lead to a goal state.

SASY is very flexible – SASY can be used to solve problems in classical symbolic planning formalism in addition to problems expressed as a belief space MDP. The user defines operators which can act upon states in any way that the user is able to encode in C++. Explicit operator definition allows the user to limit the number of nodes SASY searches by making intelligent choices about what successor nodes to return for each state and operator combination.

SASY also has two adjustable input parameters which give the user more control over the way the search is performed. The first parameter is a weight which controls how greedy SASY is when sorting nodes in the priority queue. The cost of a node is calculated as $w \cdot (\text{cost so far}) + (1 - w) * (\text{expected cost to goal state})$, where w is the weight parameter. Given a weight of 0.5, SASY will perform an A^* search. Given a weight of 1, SASY will perform a uniform cost search, and given a weight of 0, SASY will perform a very greedy search. We have found SASY to perform best with a weight of 0.35 – this is a more greedy search than A^*, without rendering the history of a node insignificant. If SASY is taking a long time to perform a search, the solver will terminate SASY and replan with a lower weight. The weight is closely related to the *robustness* of the plan that SASY will produce – the lower the weight, the less robust the plan will be. If the weight is too low, the plan SASY makes will execute many actions in a row that do not give an observation before checking the outcome of the chain of actions by executing a series of Look actions. So, if any action in the chain fails in the simulator, the simulator will continue to execute the rest of the chain before executing a Look action that will cause it to replan.

The second parameter SASY has is a threshold to consider multiple nodes at the top of the queue rather than just one. For a threshold of $\epsilon \geq 0$, SASY considers all nodes with a cost so far within ϵ of the topmost node. From this set of nodes, SASY expands the node with the lowest H_Max estimate. SASY uses H_Max for this step because H_Max is the cheapest and fastest heuristic to compute, and because H_Max is always an admissible heuristic. Larger values of epsilon generally result in fewer expanded nodes overall, but too large of values for epsilon can slow the search significantly due to the increased overhead each time SASY chooses a new node from the queue. We have found that using either 0 or the base action cost as epsilon both work well. Using $\epsilon = 0$ only requires SASY to consider nodes with the

minimal cost so far, and setting ϵ equal to the base action cost requires SASY to consider nodes within at most one action of the cheapest node.

6.5 SASY, SARSOP, POMDPX_NUS Performance on Elevator

	Elevator	(1, 4)	(2, 8)
SASY	Mean planning time (s)	1.848	-
	Mean discounted reward	48.793 ± 14.567	-
	Mean number of replans	1.35	-
SARSOP	Planning time (s)	1.968	600
	Mean discounted reward	48.055	24.730
	95% reward confidence interval	(41.302, 54.809)	(21.424, 28.035)
	Upper bound (u_f)	-	23.594
	Lower bound (l_f)	-	23.566
	Gap ratio	-	0.001
	Alpha vectors	1884	17466
POMDPX_NUS	Mean planning time (s)	10.034	28.137
	Mean discounted reward	51.531 ± 2.689	11.651 ± 3.076

Table 2: Comparison of SASY, SARSOP, and POMDPX_NUS planning for Elevator domains. A maximum of 600 seconds of online and offline planning time was given. If SARSOP did not fully close the gap between the upper and lower bounds in this time, the final upper and lower bounds are given, as well as the gap ratio, $g = \frac{u_f - l_f}{(u_f + l_f)/2}$ where u_f and l_f are the final upper and lower bounds respectively. The mean rewards are calculated over 20 simulations, each running for a maximum of 500 steps.

It is apparent from Table 2 that relatively small instances of the Elevator domain are already difficult to solve. None of the planners were able to make a plan for the Elevator(3, 12) problem instance – the extremely large input size and state space of the problem cause both SARSOP and POMDPX_NUS to run out of memory. SASY is able to make an initial plan for Elevator(3, 12) problem instances given very greedy search parameters, but the plan is never robust enough to bring the simulator to a goal state, and subsequent replans have enough uncertainty in the initial belief state to slow SASY significantly.

It is especially challenging for SARSOP to make an offline plan for the Elevator domain. Due to the large amount of uncertainty inherent in the Elevator domain, and because of the many places where the robot can make a mistake or see a misleading observation, it takes SARSOP a long time to solve most problem instances of the Elevator domain. POMDPX_NUS is the only one of the evaluated planners that performs well on the Elevator(2, 8) domain. As of now, SASY can reliably solve Elevator(1, 4) problem instances, even with lots of uncertainty

in the initial belief state.

6.6 Conclusion

An important strength of SASY is that adding unused objects and locations to a problem instance do not greatly affect the planning time. As long as SASY does not need to manipulate the distractor objects or visit the extra locations, SASY will not explore additional search paths related to them or add many extra fluents to its belief state. Another strength of SASY, as explained in the previous section, is that SASY can accept arbitrary numeric formulas as part of the domain specification. SASY does not require the transition and observation distribution for domains to be explicitly enumerated as do many planners that require a specific input file format.

SASY can consistently make a plan for Elevator(2, 8) problem instances, but unfortunately the plans it makes are almost never reliable enough to guide the simulator to a goal state. One feature of SASY that can detract from the robustness of larger plans it makes is that SASY assumes a success observation for all of its Look actions. The ability to assume a success observation for a Look action is important when parts of the belief state become very uncertain. If SASY were only able to assume a success observation if the prior probability of the variable were greater than a fixed threshold, SASY would not be able to observe success for any Look action on a uniformly distributed variable in a large enough domain. However, the fact that SASY can only observe success even in the most unlikely of cases can cause belief states in the search to greatly differ from the state of the world when the plan runs in the simulator.

Another attribute of SASY that will greatly affect its planning behavior is the function it uses to assign costs to actions. Currently, SASY uses a constant cost for actions without an observation, and a constant base cost augmented by a variable determinized cost for actions with observations. The variable determinized cost for an observation action is $c = -\log_2(p)$ where p is the probability of receiving the assumed observation. For shorter plans, this cost function works very well – SASY is readily discouraged from assuming observations that are very unlikely. However, when the plan length begins to grow very long, SASY will choose to execute a Look action that is expensive and unlikely to succeed in lieu of performing a series of cheaper actions. Additionally, SASY will often produce plans which are not robust if the planner weight is set too greedily. For small domains, this behavior is unattractive but still possible to recover from. For larger domains, however, this behavior leads to the simulator never reaching a goal state.

6.7 Future Work

A promising idea to remedy the problems caused due to only receiving success observations is to make it possible for SASY to receive failure observations using randomized rounding corresponding to the prior probability of the variable. We plan to explore different observation schemes such as these in future work. We would also like to experiment with different cost functions that relate the cost of actions with assumed observations to the trajectory of the plan so far. Using a cost function that encourages sensing early and often, rather than all at the end of the plan, would make it more likely that the simulator will detect actuator failures and cause SASY to replan much sooner. We hope that these changes will enable SASY to solve much larger instances of problems than it currently can.

7 References

[1] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. Artificial Intelligence 101(1-2), pages 99-134. 1998.

[2] H. Kurniawati, D. Hsu, and W. S. Lee. SARSOP: Efficient Point-based POMDP Planning by Approximating Optimally Reachable Belief Spaces. In Proc. Robotics: Science and Systems. 2008.

[3] D. Silver and J. Veness. Monte-Carlo Planning in Large POMDPs. In Advances in Neural Information Processing Systems (NIPS). 2010.

[4] L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo Planning. In 15th European Conference on Machine Learning, pages 282-293. 2006.

[5] A. Somani, N. Ye, D. Hsu, and W. S. Lee. DESPOT: Online POMDP Planning with Regularization. In Advances In Neural Information Processing Systems, pages 1772-1780. 2014.

[6] T. Smith and R. Simmons. Heuristic Search Value Iteration for POMDPs. In Proc. Uncertainty in Artificial Intelligence, pages 520-527. 2004.

[7] B. Bonet, G. Loerincs, and H. Geffner. A Robust and Fast Action Selection Mechanism for Planning. In Proceedings of the Fourteenth National Conference on Artificial Intelligence, pages 714-719. 1997.

[8] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research 14, pages 253-302. 2001.

[9] J. Hoffmann. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. Journal of Artificial Intelligence Research 20, pages 291-341. 2003.

[10] A. J. Coles, A. I. Coles, M. Fox, and D. Long. Forward-Chaining Partial-Order Planning. In Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010). 2010.

[11] A. J. Coles and A. I. Coles. PDDL+ Planning with Events and Linear Processes. In Proceedings of the Twenty Fourth International Conference on Automated Planning and Scheduling (ICAPS-14). 2014.

[12] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, Englewood Cliffs, NJ. 1995.

A Elevator Domain and Problem Specification

An elevator domain with e elevators and f floors is described as Elevator(e, f). A problem instance of Elevator must also include an initial and a goal world configuration which specify the locations of the robot and the elevators.

The state space of the POMDP model of Elevator(e, f) is the Cartesian product of several vectors. The first is a vector of length f which is a probability distribution of the robot's location over the floors of the building. The second is a vector of length $e + 1$ which is a probability distribution over which elevator the robot is currently grasping, with the last element in the vector being the probability that the robot is not grasping any elevator. The third is a two-dimensional vector: the outer vector is length e, and each inner vector is a length f probability distribution for the location of elevator e over the floors of the building.

The robot can select from six actions: Move(f, f'), Pick(e, f), Place(e, f), LookRobot(f), LookHand(e), and LookElevator(e, f). All of the robot's actuators and sensors have an accuracy of 0.9. Move(f, f') attempts to move the robot from floor f to floor f'. Floors f and f' must be adjacent. If the robot is holding nothing, or if the robot is holding an elevator and there is not an elevator on floor f', the probability the robot successfully moves is 0.9. If the robot is holding an elevator when it moves, the elevator will move with the robot. Pick(e, f) attempts to pick up elevator e at floor f. If the robot is at floor f and elevator e is at floor f, the probability the robot successfully picks up the elevator is 0.9. Place(e, f) attempts to place elevator e at floor f. If the robot is at floor f and is holding elevator e, the probability the robot successfully places elevator e at floor f is 0.9. Move, Pick, and Place always return a null observation regardless of whether or not the action was successful.

LookRobot(f) looks for the robot on floor f. If the robot is on floor f, the robot will receive a success observation with probability 0.9. If the robot is not on floor f, the robot will receive a success observation with probability 0.1. Otherwise, the robot will receive a failure observation. LookHand(e) looks for elevator e in the hand of the robot. If the robot is holding elevator e, the robot will receive a success observation with probability 0.9. If the robot is not holding elevator e, the robot will receive a success observation with probability 0.1. Otherwise, the robot will receive a failure observation. LookElevator(e, f) looks for elevator e on floor f. If elevator e is on floor f, the robot will receive a success observation with probability 0.9. If elevator e is not on floor f, the robot will receive a success observation with probability 0.1. Otherwise, the robot will receive a failure observation.

Each action the robot takes has a cost of 1. When the robot believes each elevators to be in its goal configuration with probability 0.9, the robot moves above the top floor of the building. Moving above the top floor of the building yields reward 100 if the robot has successfully reconfigured the elevators or reward 0 if the robot has not successfully reconfigured the elevators.