Learning and Discovery in Dynamical Systems with Hidden State

Milena Scaccia

Supervised by Prakash Panangaden, Joelle Pineau, and Doina Precup

August 15, 2007

Abstract

We consider the problem of learning in dynamical systems with hidden state. This problem is deemed challenging due to the fact that the state is not completely visible to an outside observer. We explore a candidate algorithm, which we call the Merge-Split algorithm, for learning deterministic automata with observations. This is based on the work of Gavalda et al(2006) which approximates a given Hidden Markov Model (HMM) with a learned Probabilistic Deterministic Finite Automaton (PDFA).

1 Introduction

Learning and planning under uncertainty is one of the main aspects of modern AI research. This problem has a considerable application in the field of robotics since robots must cope with environments that are partially observable, stochastic, dynamic and continuous. In order for a robot to reason correctly about its environment, it must be provided with a good internal representation of it. Learning the internal representation of a partially observable system is deemed difficult due to the state being hidden to an outside observer. Instead, there are observables that partly identify the state. From information we are able to retrieve, given the observables, we want to learn a model that approximates the partially observable system and that will enable us to make good predictions on future action effects.

A good representation should contain enough information for the robot to make the right decisions and its internal state variables should correspond to the natural state variables of the physical world. In the AI literature, there is much discussion about choosing a good state representation to work with. In robotics, for example, standard notions are the x,y position of a robot at any given time, or the angle of its joints (in the case of an arm robot). This choice of representation is not something that is set in stone, but it is wondered whether the obvious choice is always the best choice.

The problem of learning a good internal representation has received a lot of attention in recent research but current solutions are suboptimal. This paper presents an attempt to find an algorithm that constructs an automaton that best represents data generated by a partially observable system.

We explore a candidate algorithm, which we call the Merge-Split algorithm, for learning deterministic automata with observations. This is based on Gavalda's et al.'s GKPP algorithm(2006) which approximates a Hidden Markov Model (HMM) with a Probabilistic Deterministic Finite Automaton (PDFA).

We demonstrate how the algorithm works with different inputs and explore the limitations of the algorithm as well as the advantages it may have over other learning algorithms. For now, we only consider the Merge-Split algorithm in the deterministic case, but we are primarily interested in finding a solution for inferring hidden state in probabilistic environments such as Partially Observable Probabilistic Automata (POPA). We wish to extend Merge-Split to work in such stochastic environments. This will better apply to real-world problems since the real world is essentially a probabilistic environment.

2 Background

In this section, we review the definitions of various types of partially observable systems.

2.1 Partially Observable Systems

We use the flip automaton in Mealy form [2] to illustrate three types of partially observable systems: Deterministic Kripke Automata (DKA), Deterministic Automata with Stochastic Observations (DASOs), and Partially Observable Probabilistic Automata (POPAs).

Definition 2.1 *A **deterministic Kripke automaton** (DKA) is a quintuple*

$$\mathcal{K} = \{S, A, O, \delta : S \times A \to S, \gamma : S \to 2^O\}.$$

where S is a set of states, A is a set of actions, O is a set of observations, δ is a deterministic transition function and γ is an deterministic observation function associated with the states.

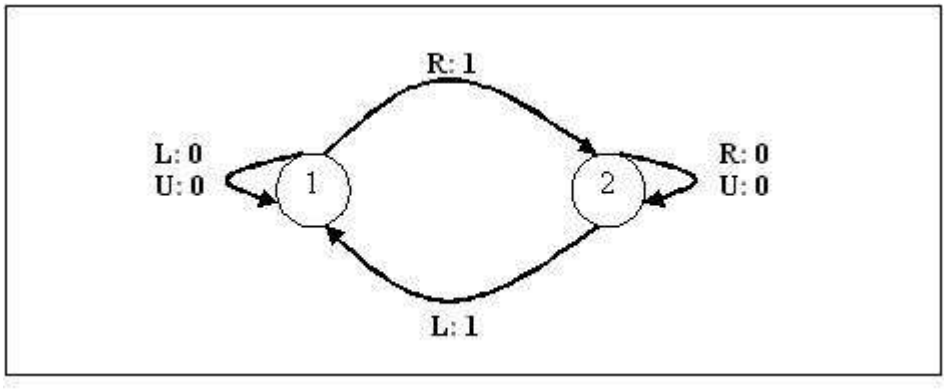


Figure 1: Deterministic Kripke Flip Automaton

Definition 2.2 *A **deterministic automaton with stochastic observations** (DASO) is a quintuple*

$$\mathcal{J} = (S, A, O, \delta : S \times A \to S, \gamma : S \times O \to [0, 1])$$

where S is a set of states, A is a set of actions, O is a set of observations, δ is a deterministic transition function and γ is an observation function such that $\gamma(s, \omega)$ is the probability of observing ω in state s.

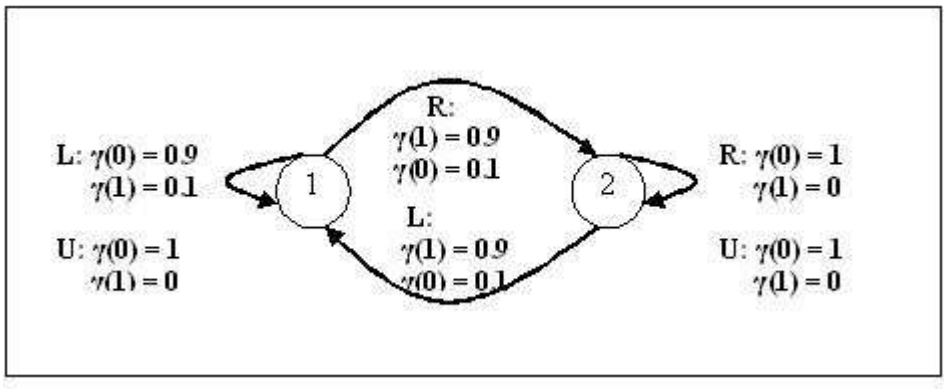


Figure 2: Deterministic Flip Automaton with Stochastic Observations

Definition 2.3 *A **partially observable probabilistic automaton** (POPA) is a quintuple*

$$\mathcal{H} = (S, A, O, \tau : S \times A \times S \to [0,1], \gamma : S \times O \to [0,1])$$

where S is a set of states, A is a set of actions, O is a set of observations, $\tau(s,a,\cdot)$ defines a probability distribution on possible final states and $\gamma(s,\omega)$ is the probability of observing ω in state s.

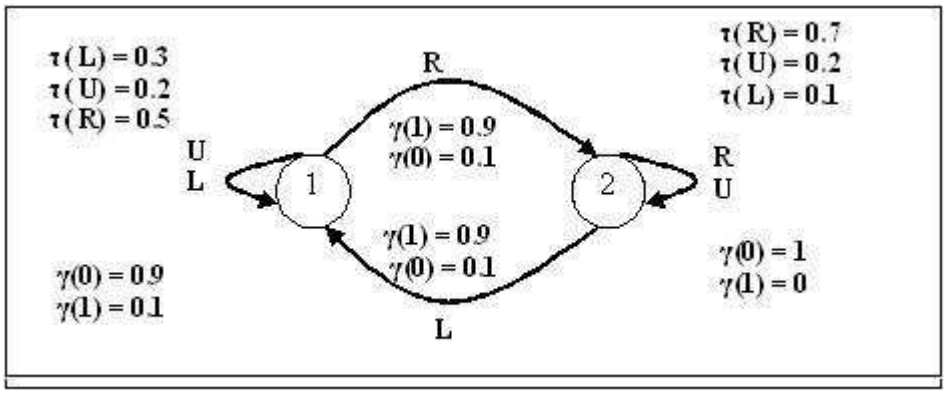


Figure 3: Partially Observable Probabilistic Flip Automaton

Most of the work we have done revolved around the DKA, but we wish to extend results to probabilistic systems such as DASOs and POPAs since the real world is essentially a probabilistic world.

2.2 HMMs and PDFAs

In this section, we define two systems used in the GKPP algorithm. Recall that the GKPP algorithm learns a Probabilistic Deterministic Finite Automaton that can approximate a Hidden Markov Model.

Definition 2.4 *A **Probabilistic Deterministic Finite Automaton (PDFA)** is a tuple (S, Σ, T, O, s_0) where S is a finite set of states, Σ is a finite set of observations, $T : S \times \Sigma \to S$ is the transition function, $O : S \times \Sigma \to [0,1]$ defines the emission probability of each observation on each state, $O(s,\sigma) = P(\sigma_t = \sigma | s_t = s)$ and $s_o \in S$ is the initial state. Note that we do not work with actions, but only with observations in PDFAs. Observations are what allow for transitions from one state to the other. Once an observation had been selected, the transition to a new state, s', is deterministic and given by $T(s,\sigma)$.*

*In a **probabilistic nondeterministic finite automaton(PNFA)**, the transition function is stochastic: $T : S \times \sigma \times S \to [0,1]$, and $T(s,\sigma,s') = P(s_{t+1} = s' | s_t = s, \sigma_t = \sigma)$.*

In both PDFAs and PNFAs, a special symbol marks the end of a string.

Definition 2.5 *A **Hidden Markov Model (HMM)** is a tuple (S, Σ, T, O, b_0) where S is a finite set of states, Σ is a finite set of observations, $T(s,s') = P(s_{t+1} = s' | s_t = s)$ is the transition probability, $O(s,\sigma) = P(\sigma_t = \sigma | s_t = s)$ is the emission probability of each observation on each state, and finally, $b_0(s) = P(s_0 = s)$ is the initial state distribution. Given an observation trajectory $\sigma_1, ...\sigma_k$, emitted by a known HMM, we can recursively estimate the probability distribution over states at any time b_{t+1}, by Bayesian updating as follows: $b_{t+1}(s) = \sum_{s' \in S} b_t(s') O(s', \sigma_t) t(s', s)$.*

In the paper PAC-Learning of Markov Models with Hidden State, Gavalda et al. demonstrated that every HMM can be transformed into an equivalent PNFA which can in turn be approximated by a PDFA. (Note that all the systems are finite.) This shows that finite-size approximation of an HMM by a PDFA is always possible. This can be used in the context of PAC-learning where the goal of learning is to find a model that approximates the true probability distribution over observation trajectories.

Ron at al.(2005) proposed that PAC-learning of PDFAs is possible if we restrict to the class of PDFAs that are acyclic and have *distinguishability criterion* between states.

Definition 2.6 *A PDFA has* ***distinguishability*** μ *if for any two states* s *and* s'*, the probability distributions over observation trajectories starting at* s *and* s'*,* $\mathcal{P}_s$ *and* $\mathcal{P}_{s'}$*, differ by at least* μ*:* $m(\mathcal{P}_s, \mathcal{P}_{s'}) \geq \mu, \forall s, s' \in S$*, where* m *is a measure of the difference between two probability distributions.*

3 GKPP Algorithm

Given sequences of observations, $\sigma_1, ...\sigma_n$. the GKPP algorithm constructs an automaton which can generate this data with high probability. The algorithm maintains a list of "safe states" and "candidate states", the safe states being states we decide to include in the graph constructed.

Let D denote the set of training trajectories (sequences of observations). The algorithm works as follows:

It takes as input the parameters δ, n and μ, where δ is the desired confidence, n is the upper bound on the number of states desired in the model, and μ is a lower bound on the distinguishability between any two states.

We begin by initializing a single safe state $S = \{s_0\}$ representing the initial state of the target machine. We also initialize a set of candidate states $s_0\sigma \ \forall \sigma \in \Sigma$. Also, a multiset D_s, or $D_{s\sigma}$ is associated with each safe and candidate state, respectively. The multiset of a state will contain the suffixes of all training trajectories that pass through it.

For each sequence of observations, d $= \sigma_0, ...\sigma_k$, we traverse the graph by matching each observation to a state until we reach a candidate state $s\sigma$ or until we have exhausted all observations in d, in which case we proceed to the next sequence of observations. Note that in the case where we find a candidate state $s\sigma_i$ (which occurs when all transitions up to σ_{i-1} lead to a safe state s and there is no transition out of s with observation σ_i), we add the sub-trajectory $\{\sigma_{i+1}, ...\sigma_k\}$ to the multiset $D_{s\sigma_i}$.

In the induction step, the algorithm will either:

1. Promote a candidate state to a safe state
   • if the training data suggests that there is a significant difference between the probability distribution over trajectories observed from the candidate state and any safe state, then the candidate state will be promoted.
2. Merge a candidate state with an existing safe state
   • If there is not a significant difference between the probability distributions over trajectories observed from the candidate state and a safe state, then the candidate state will be merged with the existing safe state.
3. Retain it as a candidate state.
   • If there is not enough information to decide, then we retain the state as a candidate state.

The algorithm first judges whether there is enough information to promote or merge a candidate state correctly. A candidate state will not be retained when it is declared large.

Definition 3.1 *A candidate state* $s\sigma$ *is declared* ***large*** *when:*

$$|D_{s\sigma}| \geq \frac{3(1+\mu/4)}{(\mu/4)} ln \frac{2}{\delta'} \qquad (2)$$

$$\textit{where } \delta' = \frac{\delta\mu}{2(n|\Sigma|+2)}$$

This is what we call the largeness condition.

Let $|D_s(\sigma)|$ denote the number of trajectories starting with observation σ in multiset D_s and note that $|D_s(\sigma)|/|D_s|$ can be regarded as an empirical approximation of the probability that trajectory d will be observed starting from state s.

If the candidate state is not retained, and if there exists a safe state s' such that for every trajectory d,

$$|\frac{|D_{s\sigma}(d)|}{|D_{s\sigma}|} - \frac{|D_{s'}(d)|}{D_{s'}}| \leq \mu/2, \qquad (3)$$

then we merge $s\sigma$ and s'. In this case, candidate state $s\sigma$ is removed and a σ transition from s tos' is added. In addition, $|D_s(d)|$ is increased by $|D_{s\sigma}(d)|$.

On the other hand, if for every s', there exists a d such that

$$|\frac{|D_{s\sigma}(d)|}{|D_{s\sigma}|} - \frac{|D_{s'}(d)|}{|D_{s'}|}| > \mu/2 \qquad (4),$$

then $s\sigma$ is promoted to a new safe state. In this case, we add a σ transition from s to $s\sigma$, and add candidate states $s\sigma\sigma'$ for every observation $\sigma' \in \Sigma$. All trajectories in $D_{s\sigma}$ are distributed appropriately to the new candidate states.

The final step of the algorithm entails transforming the constructed graph into a PDFA. Every safe state will become a state of the automaton, the set of observations Σ remains the same, and the probabilistic observation function is computed as follows:

$$O(s,\sigma) = \frac{|D_{s'}(\sigma)|}{\sum_{\sigma' \in \Sigma} |D_s(\sigma')|}(1 - (|\Sigma| + 1)\gamma) + \gamma, \qquad (5)$$

where $\gamma < \frac{1}{|\Sigma|+1}$.

Finally, we take care of any remaining candidate state $s\sigma$ by looking for a safe state s' that is the most similar to it, add a σ transition from s to s' (the transition function is $T(s,\sigma) = s\sigma$), and we compute the observation probability as defined above.

The resulting automaton will be a PDFA that can be used to approximately compute the probabilities of different trajectories in the HMM.

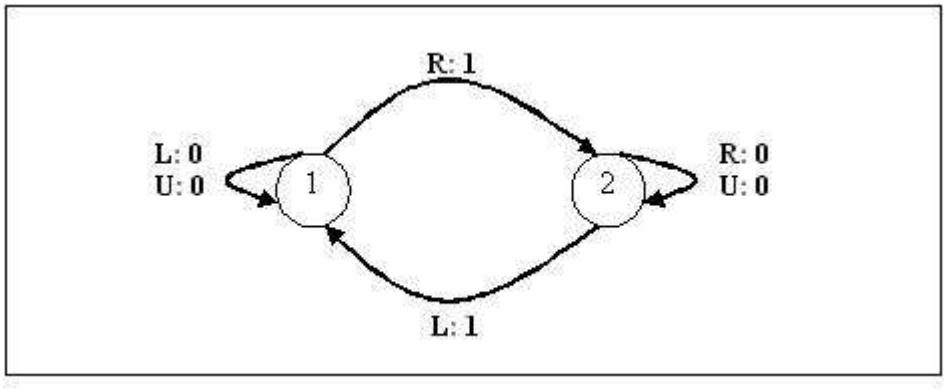
4 The Merge-Split Algorithm

The Merge-Split algorithm is a derivation of the GKPP algorithm. It uses the same concept as GKPP: it uses both state splitting and merging operations. Unlike GKPP, it works with observations *and* actions, and it works in deterministic environments. It does not consider the distinguishability parameter, which is specific to a probability distribution.

The basic idea of the Merge-Split algorithm is as follows:
• Two nodes are merged is their sets of possible future transitions are identical
• If a node cannot be merged with any other existing node, it will be further expanded (split).

4.1 Flip Automaton Example

We revisit the flip automaton which we use to demonstrate a step-by-step sample run of the Merge-Split algorithm.



Flip Automaton (as seen in Figure 1)

Step 1: The Merge-Split algorithm begins by initializing a single null state, from which one is allowed to take any action/observation sequence. (Note that the set of numbers next to a node in the graph indicate the possible states we can be in at this point. In this case we can be in either state 1 or state 2, since the null node indicates no actions were taken yet).

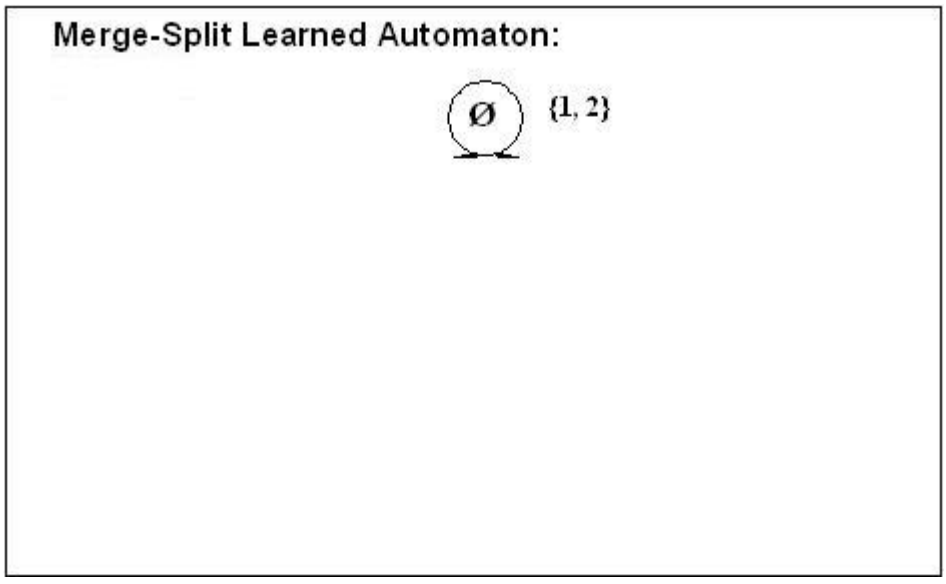


Figure 4: Step 1 of the Merge-Split Algorithm initializes a single null state

Step 2: For every possible transition, create a child node.

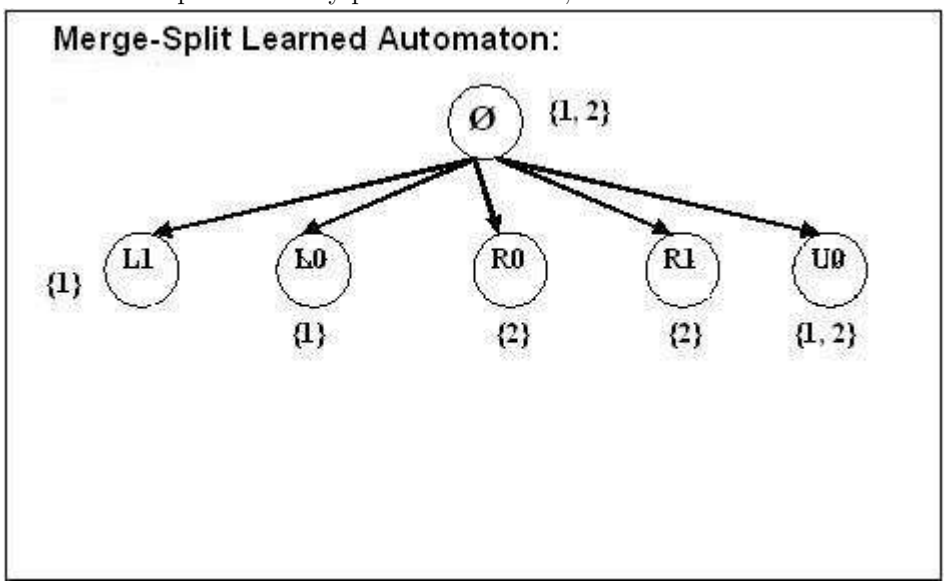


Figure 5: Merge-Split creates a child node for every possible transition from the null state.

Step 3: Merge nodes whose sets of future possible transitions are identical. For example, taking an L0 action is the same as taking an L1 action. Both actions will take us to state 1. The future trajectories of nodes L0 and L1 will thus be identical, so the two nodes are merged.

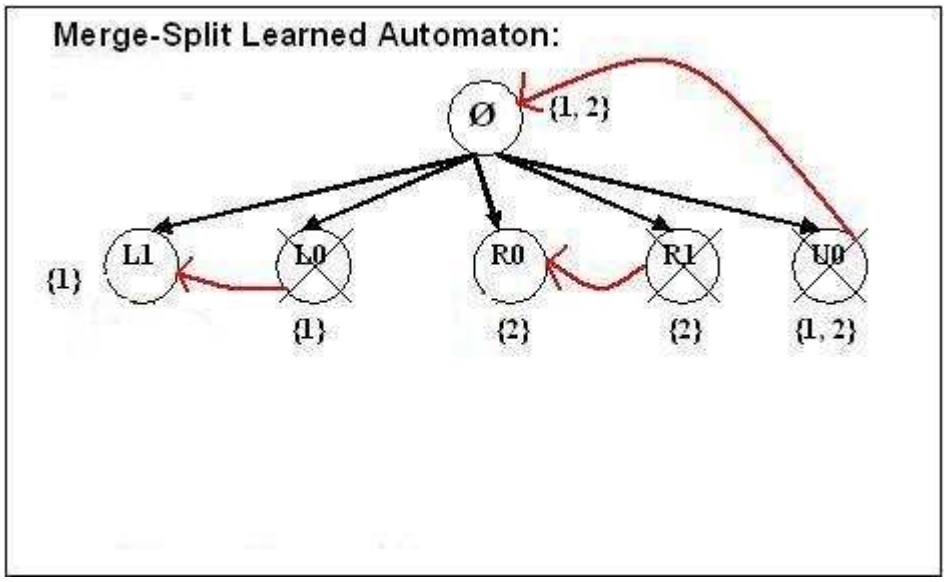


Figure 6: Merge-Split merges redundant nodes

Step 4: Nodes that did not get merged with other existing nodes will be expanded an additional step.

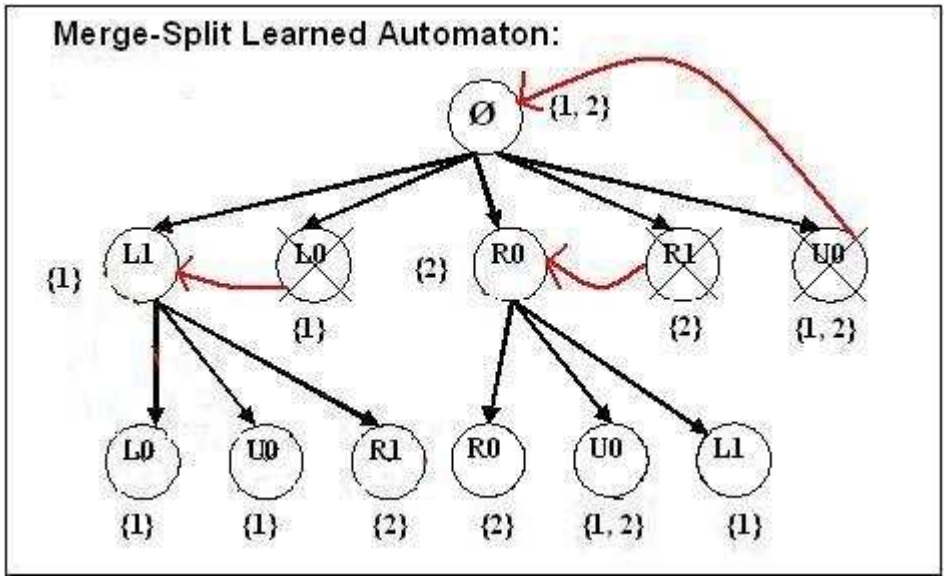


Figure 7: Merge-Split expands nodes which did not previously get merged

Step 5: The following is the final automaton learned by the Merge-Split algorithm.

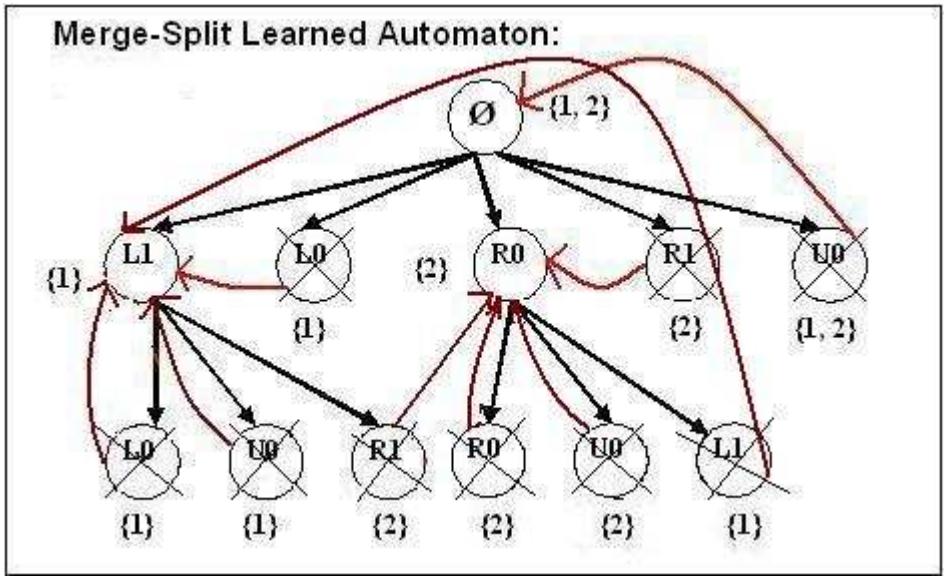


Figure 8: Merge-Split merges redundant nodes (as in Figure 6)

We can clean up our diagram of the learned automaton by removing crossed out nodes and adding appropriate labels and transition arrows to the remaining nodes:

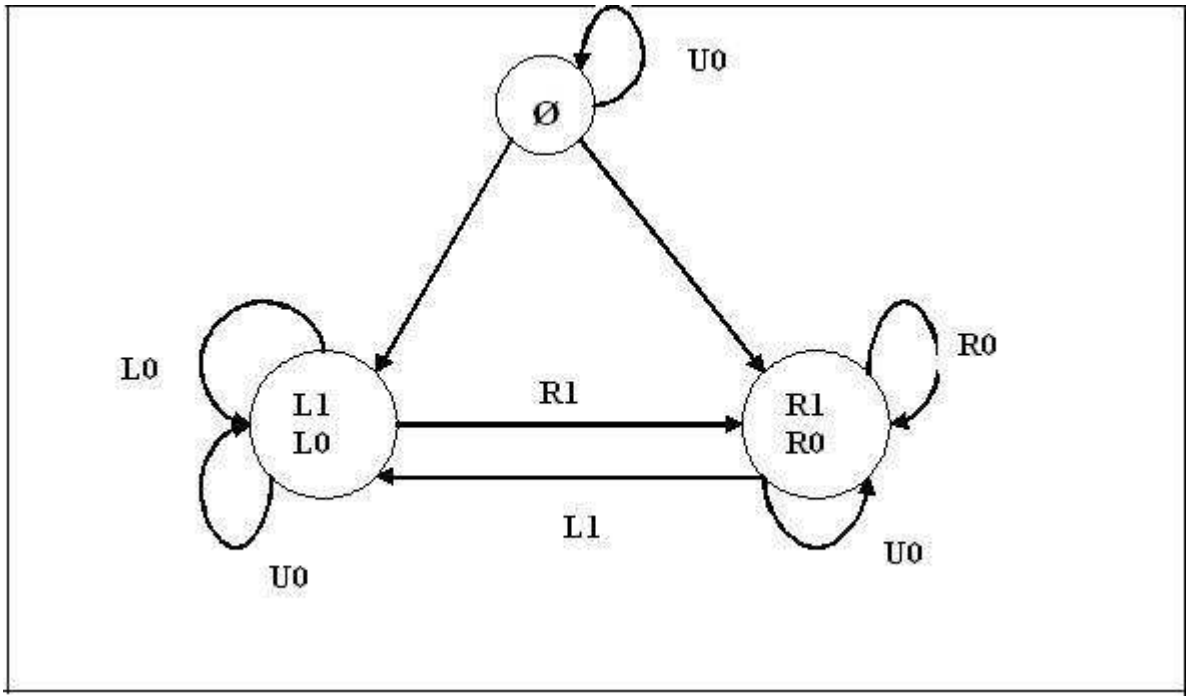


Figure 9: The final automaton learned by the Merge-Split algorithm w.r.t. the Flip Automaton

The resulting automaton closely resembles the original machine, and allows us to correctly predict future action effects!

4.2 Additional Examples

While Merge-Split learned a model that closely resembled the original machine in the flip-automaton, there exist examples where Merge-Split may learn a model that resembles the minimal representation of the original machine.

The following is a 6-state *DKA* [3], in Mealy form.

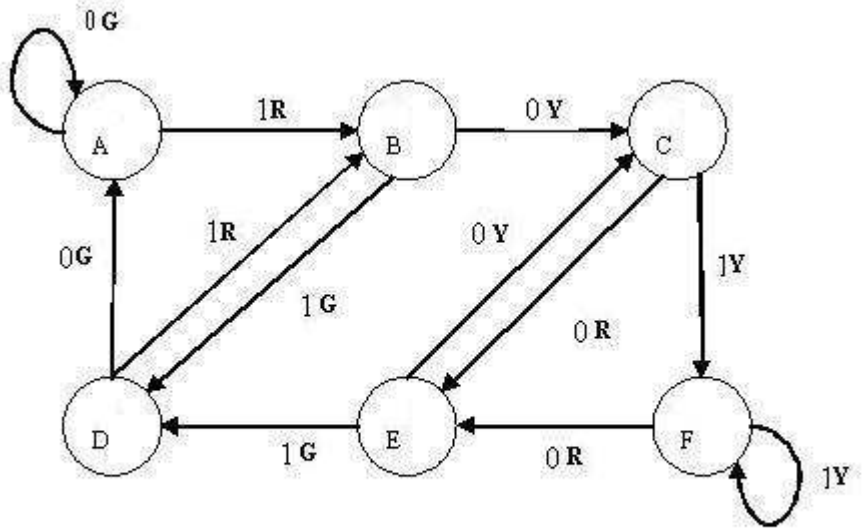


Figure 10: A 6-state Deterministic Kripke Automaton in Mealy form

Merge-Split learns the following model.

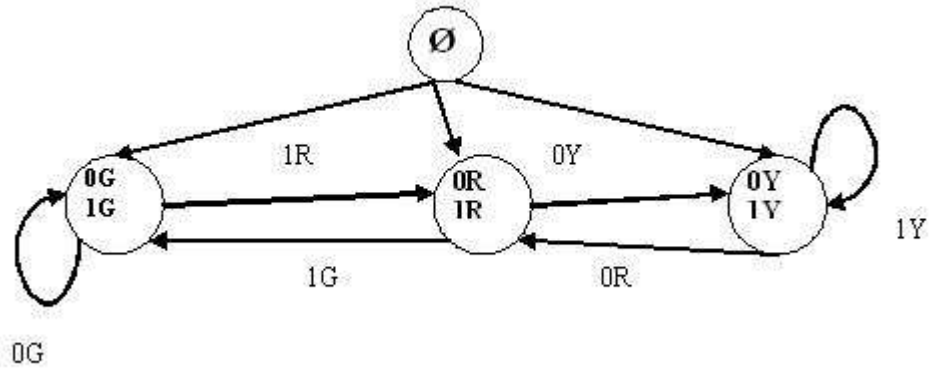


Figure 11: Learned Merge-Split Automaton

The original machine was not minimized in this case, but Merge-Split learned a model that closely resembles its minimal representation. This led us to conjecture that Merge-Split always learns the minimal model.

But we discovered that Merge-Split may not always learn the minimal representation. Consider the following 7-state float/reset automaton [4].

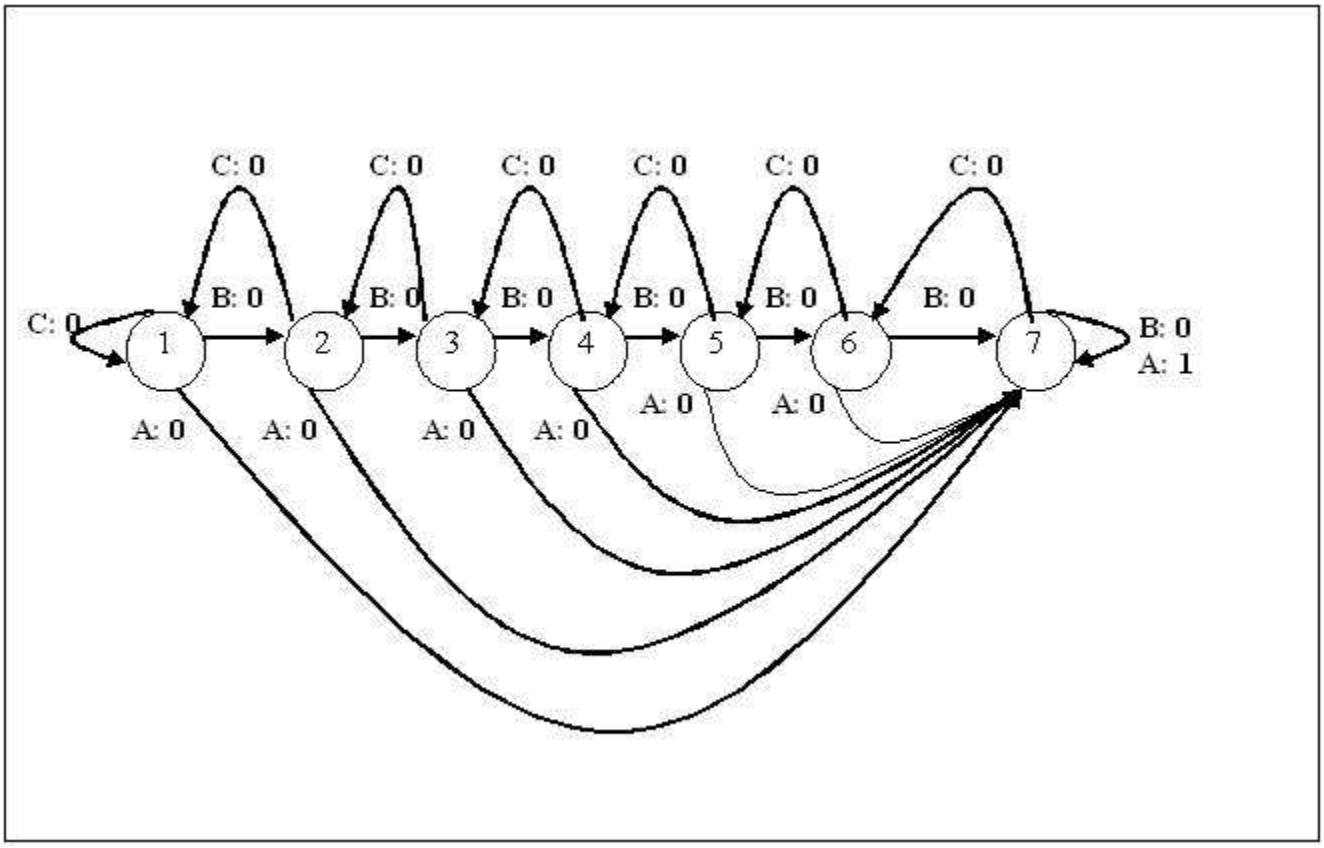


Figure 12: 7-state Float/Reset Automaton

The Merge-Split algorithm will learn a 28-state model, which is well over the number of states in the original automaton.

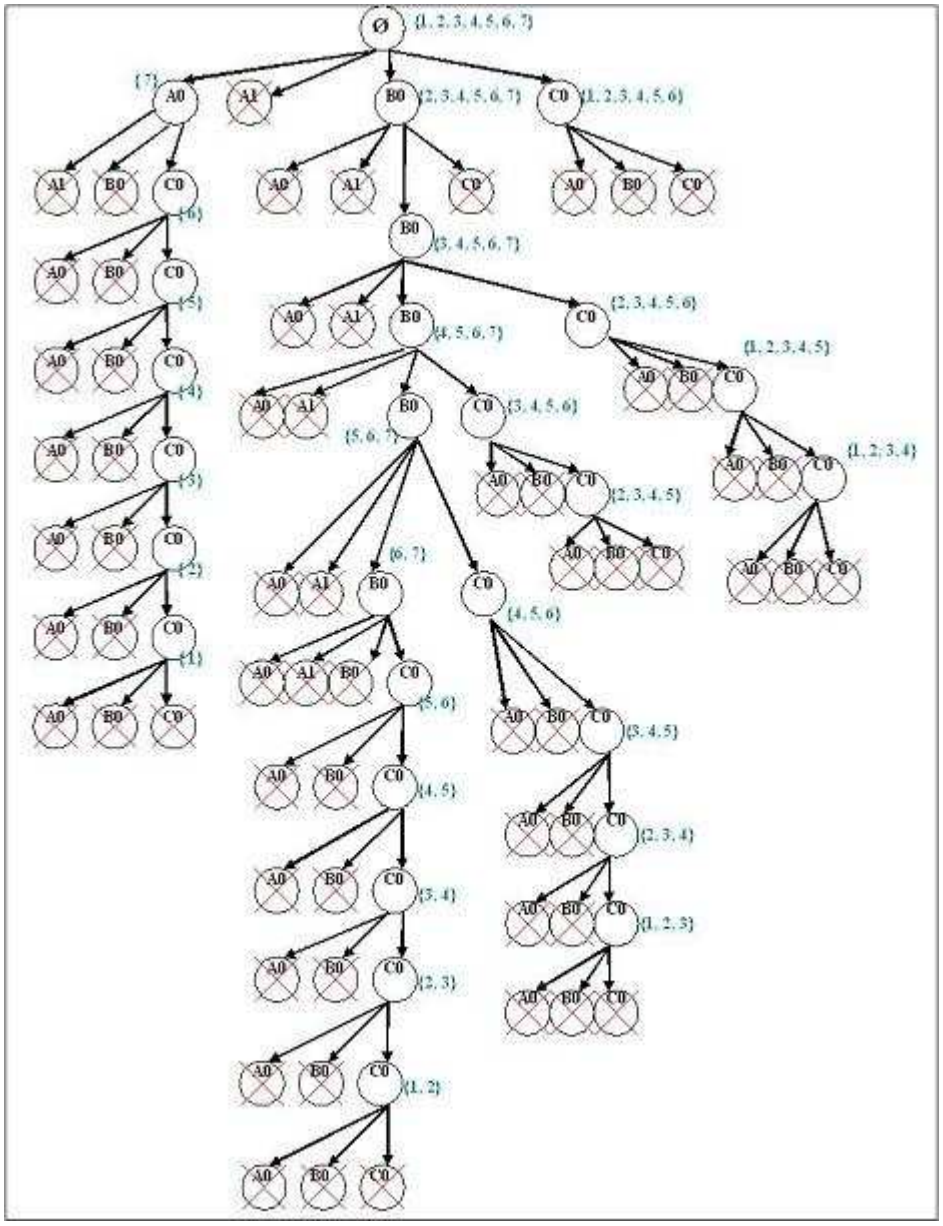


Figure 13: The Merge-Split Learned Automaton w.r.t. the 7-state Float-Reset Automaton

Merge-Split works by looking at the set of future transitions of a node. The reason the algorithm repeatedly expands nodes in the float/reset case is because it identifies differences in the futures of the nodes. For instance, starting from the B0 node, if we look at a 5-step extension of the node, i.e. B0B0B0B0B0B0,we note that our future possible transitions are {B0, A1, C0}, whereas up to a 4-step extension will give us the transition set {B0, A0, A1, C0}. Hence we cannot merge B0B0, B0B0B0, B0B0B0B0, or B0B0B0B0B0 with the node B0, because this would not reflect the difference in future transitions encountered.

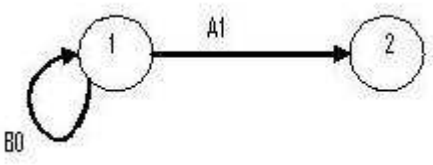
4.3 Deterministic Case vs. Stochastic Case

There are several differences in the difficulties that may arise when learning the structure of a deterministic system versus that of a probabilistic system. In fact, there is much discussion devoted to which of the two cases is more difficult to handle. Intuitively, one would gather that it is easier to deal with the deterministic case, but the following demonstrates why it may be quite challenging.

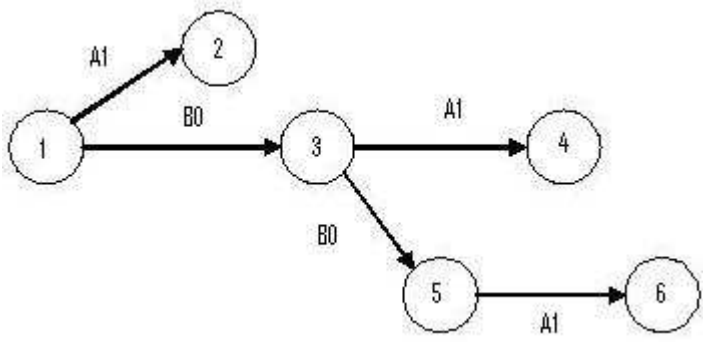
What is particular about the deterministic case is that we cannot assign distributions to the transitions, and we cannot determine with what probability the machine can satisfy an experiment. We can only determine what transitions are possible.

Figure 14 illustrates the challenges faced when learning the structure of a deterministic system. If the current data suggests that our model should satisfy the sequences a1, b0a1 and b0b0a1, then at this point, we do not know whether we should create additional states to satisfy the data, or whether our model should loop on action b0 (i.e. merge the nodes that satisfy b0). Both looping and non-looping models fit the current data.

Learning the structure of a deterministic transition system involves more than producing a model that merely memorizes the current data (as in Figure 14(b)). A model that merely memorizes what was seen so far will be "shafted" by any longer sequence. Notice that the model in Figure 14(b) will require modifications if the data suggests that the sequence b0b0b0a1 should also be satisfied. This is why it is a good idea to generalize the model.



(a) Simple two-state model



(b) A model that memorizes the data

Figure 14: Models (a) and (b) both satisfy a1, b0a1, b0b0a1, but (a) can satisfy b0b0b0a1, while (b) cannot.

Our Merge-Split algorithm overcomes this difficulty by exploring the sets of *all* possible future transitions of each node. In the Float-Reset automaton example (Figure 12), Merge-Split was able to recognize that the learned model (Figure 13) should not merge all the nodes satisfying B0. It was able to detect a difference between the 4^{th} and 5^{th}-step extensions of the B0 node.

Given that Merge-Split works efficiently in the deterministic case, we aim to extend it to the stochastic case and hope for comparable results.

GKPP can in a certain sense be thought of as the probabilistic version of Merge-Split. Unlike GKPP, Merge-Split works with transition systems consisting of actions and observations whereas GKPP works for HMMs, in which case we do not have actions but only observations that account for transitions between states. We want to generalize GKPP to work for the case of actions in order to be able to infer hidden state in probabilistic environments such as POPAs.

To extend the Merge-Split algorithm to the stochastic case, we must reintroduce the largeness condition and the distinguishability parameter in order to determine if there is a significant difference between the probability distributions over trajectories observed from any two states. The trajectories will now consist of both actions and observations, hence it is necessary to adjust the largeness condition to take actions into consideration. This was not yet worked on, but we do not expect it to be a difficult task.

5 Future Work

Although our Merge-Split algorithm allows us to correctly predict future action effects in deterministic environments, the constructed automaton is not minimal. We wish to refine Merge-Split so that it learns the minimal representation of a given partially observable system. We want to do this in order to connect it with the notion of the double-dual representation [3]. The double-dual representation is the minimal version of the original machine with deterministic transition structure and no hidden state. Its states can be thought of as "bundles" of predictions for experiments, making this representation hold the promise of better learning and planning algorithms.

As described, we also aim to extend the Merge-Split algorithm to work in probabilistic environments in order to better apply it to real-world problems. Achieving these goals will bring us a step closer to reaching our long-term goal of finding an algorithm that learns the most succinct automaton consistent with the data produced by a partially observable system.

6 References

[1] R.Gavalda, P.Keller, J.Pineau, D.Precup. "PAC-Learning of Markov Models with Hidden State" In *Proceedings of ECML, 2006.*

[2] M.P.Holmes, C.L.Isbell. "Looping Suffix Tree-Based Inference of Partially Observable Hidden State". In *Proceedings of ICML, 2006.*

[3]C.Hundt, P.Panangaden, J.Pineau, D.Precup, M.Dinculescu. "Duality of State and Observations". *Research supported by NSERC and CFI, 2006.*

[4] M.L.Littman, R.S. Sutton, S.Singh. "Predictive Representations of State". *AT&T Labs–Research, 2001.*