Hierarchical Reasoning with Probabilistic Programming

Brian E. Ruttenberg
Charles River Analytics
625 Mt Auburn St.
Cambridge MA 02140
bruttenberg@cra.com

Matthew P. Wilkins
Applied Defense Solutions
10440 Little Patuxent Pkwy #600
Columbia, MD 21044
mwilkins@applieddefense.com

Avi Pfeffer
Charles River Analytics
625 Mt Auburn St.
Cambridge MA 02140
apfeffer@cra.com

Abstract

Hierarchical representations are common in many artificial intelligence tasks, such as classification of satellites in orbit. Representing and reasoning on hierarchies is difficult, however, as they can be large, deep and constantly evolving. Although probabilistic programming provides the flexibility to model many situations, current probabilistic programming languages (PPL) do not adequately support hierarchical reasoning. We present a novel PPL approach to representing and reasoning about hierarchies that utilizes *references*, enabling unambiguous access and referral to hierarchical objects and their properties.

Introduction

Tracking, detecting and identifying resident space objects (RSOs) is an important task for the US government and other space agencies. Any of the estimated 500,000 objects greater than $1\,cm$ orbiting the earth can severely impair or destroy vital satellites upon collision (NASA 2013). Yet positively identifying the type, orbit and function of an RSO (e.g., a communications satellite in low earth orbit) is a difficult task given the limited and uncertain observations available and the extensive number of orbital objects.

One approach to manage the identification of RSOs is to utilize a space object taxonomy to classify RSOs (Früh et al. 2013; Wilkins et al. 2013). Similar to classical Linnean taxonomy in biology, this hierarchical representation of RSOs can organize space objects into different functional or physical categories that increase in specificity at lower levels of the hierarchy. In uncertain conditions (e.g., poor telescope observations), this representation has the advantage that identifying an object's label with low specificity is better than an incorrect identification; labeling a remote sensing satellite as an "earth observation" satellite instead of "remote sensing" is much more useful than a "navigation" label. This hierarchical classification process requires reasoning about the label of an unknown RSO given some specific observable attributes.

Effective hierarchical representation and reasoning on RSOs, and in other domains, raises a number of challenges: (1) Hierarchies can be large and deep. (2) Hierarchies can be evolving; it should be possible to add labels or attributes to existing hierarchies. (3) In some domains, hierarchies do not form a tree but exhibit convergence. (4) There may be interactions between hierarchies; for example, a satellite hierarchy might interact with a hierarchy of lift vehicles. (5) There may be multiple instances of a hierarchy in the same situation, such as several RSOs clustered in one location. While ad hoc solutions can be provided for many of these challenges, a general–purpose solution that can help create new hierarchical reasoning systems would be beneficial.

Probabilistic programming has the potential to provide such a general-purpose solution, as probabilistic programming languages (PPL) have reduced the time and effort needed to build and use large, complex and highly relational probabilistic models (Goodman 2013). PPLs provide native probabilistic constructs and methods that encode the model complexity in the language, facilitating a more natural model building procedure. Finally, because the probabilistic constructs are interwoven into the language, arbitrary data structures can be incorporated into models, and the object–oriented nature of some PPLs provides a natural means to express hierarchical models.

Unfortunately, current PPLs have limited support for hierarchical generative processes and reasoning about posterior distributions of entities from the hierarchy. A typical encoding requires the definition of a large conditional probability distribution (CPD) for attributes over all labels in the hierarchy. Such a CPD is impractical for large hierarchies, and lacks the modularity needed to add new labels to the hierarchy. This process also cannot easily support domains that contain multiple hierarchies.

In this paper, we demonstrate that with the addition of several novel constructs, the full power, flexibility and richness of PPLs can be applied to hierarchical models. We extend the Figaro PPL (Pfeffer 2012), where hierarchical domains can be naturally encoded as a probabilistic object–oriented class hierarchy. However, there are several technical challenges that must be overcome, such as unambiguously identifying random variables in different inherited classes, allowing different attribute sets for classes, and consistent application of evidence. We overcome these challenges by using Figaro's *reference* system, which allows us to apply evidence to a hierarchical model that is conditioned upon the labels of an en-

tity. We demonstrate that we can build a large RSO hierarchy and effectively reason using the hierarchy for identification of real satellites.

Background

Hierarchical structures can be found in many domains, such as social networks (Gupte et al. 2011). In vision tasks, hierarchical models such as latent Dirichlet allocation (LDA) (Blei, Ng, and Jordan 2003) are frequently used to model the hierarchical generation of features, regions, or objects in an image (Bakhtiari and Bouguila 2010; Fei-Fei and Perona 2005). Once learned, the hierarchy can be used to classify an object into labels with specific semantic meaning, or general labels if such specificity is not possible (Sadovnik and Chen 2012; Marszalek and Schmid 2007). This type of hierarchical reasoning and classification can be found in other domains as well, such as text (Sun and Lim 2001) or audio (Burred and Lerch 2003) classification.

Probabilistic programming has recently developed as a potential solution to the problems associated with the creation of large and complex models. Models created with these languages tend to be highly modular, reusable, and can be reasoned with using a built–in suite of algorithms that work on any model created in the language. These languages take many different forms; some are functional languages (IBAL (Pfeffer 2001), Church (Goodman et al. 2008)), imperative (FACTORIE (McCallum, Schultz, and Singh 2009)), embedded in an existing language (Infer.NET (Microsoft Research 2013), Figaro) or implement their own domain specific language (BLOG (Milch et al. 2007)).

There has, however, been little effort expended so far to support hierarchical reasoning in PPLs. Hierarchical representations are implicitly supported in these languages via traditional conditional dependence. While PPLs provide flexible means to create chains of conditional distributions, the size of these hierarchies can make such a method even impractical for PPLs, and does not address situations where entities that share a common label may contain varied attributes. Work by Kuo et al. (2013) and prior work on hierarchical relational models (Newton and Greiner 2004) has tried to address some of these issues for ontology–based models. These works mostly proposed representations and reasoning on the model when relationships in the ontology are dependent on attributes that may not exist in some entities. While these works presented some solutions to these problems, they do not provide a PPL framework for reasoning about hierarchies.

Figaro Programming Language

Figaro is an open source, object–oriented, functional programming language implemented in Scala. In Figaro, models are Scala objects that define a stochastic or non–stochastic process. The basic unit in Figaro is the `Element[V]` class, parameterized by V, the output type of the element. An element is basically a unit that contains some value based on a random process. For example, `Geometric(0.9)` is an `Element[Int]` that produces an integer output according to a geometric process whose parameter is 0.9. Elements can be passed to other elements as arguments, and a program typically consists of some number of element definitions.

One essential element in Figaro, the `Chain` element, captures sequencing of a probabilistic model by chaining the generation of one element with the generation of a subsequent element that depends on the first element. In Bayesian network terms, a `Chain`'s first argument represents a distribution over the parent of a node, while the second argument is essentially a conditional probability distribution over the `Chain`'s output type given the parent. Using `Chain`, many elements can be nested together, allowing a user to quickly create complex conditional probability distributions.

Hierarchies

We first formally define a probabilistic hierarchy, and then subsequently provide examples of several problems with hierarchical representation in PPLs.

Formal Definition

Let $\mathbb{L} = l_1, \ldots, l_n$ be a set of n category labels. A hierarchy $\mathcal{H}$ is a partial ordering of $\mathbb{L}$ that defines the hierarchical structure of the labels, where l_n is the top label in the hierarchy, and $l_i \leq l_j$ means that l_i is a sub–label of l_j. We denote a global set of m attributes $\mathbb{A} = \{a_1, \ldots, a_m\}$ for all labels in the hierarchy. Each label l_i, however, contains only a subset, $\mathbb{A}_i \subseteq \mathbb{A}$, of attributes, where $a_{k|i}$ is a random variable that defines a distribution for the k^{th} attribute from $\mathbb{A}$ given label l_i. If a label l_i contains an attribute a_k, all of its sub–labels must contain a_k. However, the support or distribution of a_k in different labels may be different.

We also define a generative process for each label l_j. We denote a random variable g_j for each l_j, where

$$P(g_j = l_i) = \begin{cases} p_i & \text{if } l_i \leq l_j \\ 0 & \text{otherwise} \end{cases}$$

As $l_i \leq l_i$, this means that the generative process for a label can also have non–zero probability of generating itself as a label instead of any of its sub–labels. In other words, this process allows instances of internal labels of a hierarchy, as opposed to just the leaf labels. Since l_n is the top level label in the hierarchy, we can denote g_n as the random variable that samples any label from the hierarchy.

An instance $\mathcal{I} = (y, x_1, \ldots, x_k)$ represents a sample from the label generative process and subsequently from the attribute generative process, where $y \sim g_n,\ x_i \sim a_{i|y}$. This definition of a hierarchy is highly flexible and allows for a diverse representations (e.g., we do not enforce that $P(a_{k|j} = x) = \int_{l_i : l_i \leq l_j} P(a_{k|i} = x)\, P(l_i|l_j)$).

Hierarchies in PPLs

Consider the example hierarchy shown in Fig. 1 that shows a possible hierarchical organization of RSOs. The top label is RSO, which is divided into Geosynchronous Earth Orbit (GEO) and Low Earth Orbit (LEO). GEO is divided into two additional labels for communications and observation satellites. The generative process of the hierarchy is also shown.

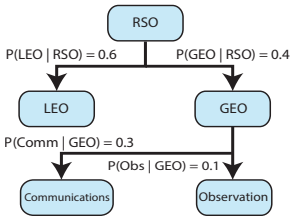

Figure 1: An example RSO classification hierarchy. The probabilities shown are conditioned upon the parent label.

Note that it is implied that a generic GEO label is generated with 0.6 probability (since the sub–labels of GEO do not account for all the probability mass).

It is easy to create a naive version of this hierarchy in a PPL. First, we encode the generative process of the labels as Figaro functions:

```
def rso() = Dist(0.6 -> leo(), 0.4 -> geo())
def geo() = Dist(0.6 -> Constant('geo),
  0.3 -> communications(), 0.1 -> observation())
def communications() = Constant('communications)
def observation() = Constant('observation)
def leo() = Constant('leo)
```

where `Dist` is a categorical distribution over elements. Note Scala uses type inference, so explicit declaration of types can usually be omitted.

Next, we create a process for the attributes of an RSO, in this case the attitude of the object:

```
def attitude(v: Element[Symbol]) = CPD(v,
  'leo            -> Constant('spinStable),
  'communications -> Constant('nadir),
  'observation    -> Constant('stable),
  'geo -> Select(0.25 -> 'uncontrolled, 0.75 -> 'stable))
```

`Select` is Figaro's notation for a categorical distribution. Reasoning about an instance from the hierarchy based on observed evidence is quite simple. We create an instance from this hierarchy by generating a label and an attribute distribution:

```
val myRSO: Element[Symbol] = rso()
val myAttitude: Element[Symbol] = attitude(myRSO)
```

where `myRSO` is an element representing generative process of the RSO label. At this point, we can apply evidence to the `myAttitude` element and perform inference using one of Figaro's built–in inference algorithms.

However, this representation is ill suited to represent more complex hierarchies. There is one large CPD with a case for each label. This results in a large and unwieldy CPD, especially with multiple non–independent attributes, where the dependencies between attributes in different labels must all be encoded in the same CPD. Furthermore, with each new label, the CPD must be rewritten, so a user cannot extend the hierarchy without modifying its core. In addition, the CPD of an attribute must define a distribution for every label, even if the label does not contain the attribute.

Clearly, an object-oriented representation with inheritance would be a more modular solution. We could define a set of classes, each of which contains its attributes. An attribute could be defined for the first time in a class, or it could inherit or override the distribution of its parent. For example,

```
abstract class RSO { val attitude: Element[Symbol] }
class LEO extends RSO {
  val attitude = Constant('spinStable) }
```

creates a class for `LEO` by extending the `RSO` class, and we can change the generative process accordingly (i.e., to generate classes instead of symbols). This encoding also poses challenges, however. The problem is that now, instead of a single random variable (RV) representing the attitude of an RSO, all instances of `RSO` contain attitude RVs, so there are four RVs in this model (one for each possible value of `myRSO`).

It then becomes problematic to try to refer to the attitude of `myRSO`. A naive approach would be to use `myRSO.attitude`, but this is not correct. `myRSO` is an `Element[RSO]`, i.e., a RV. It does not have an `attitude` attribute; only the *values* of `myRSO` do. In addition, consider if the `GEO` class also defines a longitude attribute, but the `LEO` class does not. A system using this hierarchy might observe an RSO at a specific longitude, but this information cannot be asserted on `myRSO` because the `RSO` class does not have a longitude attribute (only the sub–class `GEO` does).

Figaro's `Chain` construct provides a possible solution to both these problems. We could write `Chain(myRSO, (v: RSO) => v.attitude)` to refer to the attitude of `myRSO`. However, this solution has the obvious problem that the entire hierarchy has to be hardcoded, losing the modularity benefits we have worked to achieve. In addition, this solution is highly sub–optimal for inference, as the `Chain` couples together the RVs for all the different classes. This has negative implications for both factor and sampling–based inference algorithms.

References

To solve this problem, we use *references*. References are an abstraction to access, modify and reason about Figaro elements in a model. A reference can refer to different RVs depending on the situation, enabling them to overcome the previous challenges.

Element Collections and Names

To enable references, we enforce that every Figaro element must have a name, which is just a string handle for the element. String names do not need to be unique, and when none is provided at element creation, the empty string is used. We also add element collections (EC) to Figaro. ECs are simply a way to organize Figaro elements into coherent groups. Every Figaro element is part of an EC; if an element is not explicitly placed into an EC at creation time, it is placed into the `default` EC.

To further explain this concept, we modify the RSO hierarchy with names and ECs as shown below:

```
abstract class RSO extends ElementCollection {
  val attitude: Element[Symbol]
}
class LEO extends RSO {
```

```
 val attitude =
   Constant('spinStable)(``attitude'', this)
}
val myRSO = Constant(new LEO)(``myRSO'', default)
```

The RSO class extends the EC trait. Any elements (attributes) defined inside a RSO class are placed into an EC defined by the newly created class. When we define a LEO class, we also give its attributes a name. When a LEO is instantiated, it will create an element named "attitude" inside a LEO, and this means that the element belongs to the EC of the instantiated LEO. Finally, we create a specific instance of a RSO and give it a name.

Using References

A Figaro reference is a string that, when resolved, refers to the name of an element within some EC. A reference can be resolved into one of more elements using a recursive procedure. For instance, each EC defines a function called get that returns the referred element given the current values of all other elements. So, we can write

```
val rsoAttitude: Element[Symbol] =
  default.get(``myRSO.attitude'')
```

To resolve the reference and assign rsoAttitude, the element representing the attitude of the RSO, we first retrieve the element named "myRSO" from the default EC. There is only one value of this element (a LEO class), which is an instance of an EC, so we then recursively call get with the next part of the reference. On the second call, we retrieve the element named "attitude" in the LEO EC and return it.

In this example, myRSO is a constant, so any reference to "myRSO" has only one resolution path. If we use the generative process of RSOs as defined earlier, then myRSO could take on a several values, where each possible value is also an EC. Resolving the reference "myRSO.attitude" may refer to multiple elements, each defining their own random process. In this case, the reference itself defines a distribution of values. That is, $P(myRSO.attitude = x)$ is

$$\int_{y \in myRSO} P(y.attitude = x | myRSO = y) \cdot P(myRSO = y)$$

References in Hierarchies

References are a natural means to build, access and reason on probabilistic hierarchies because they address many of the issues touched upon in the previous section. First, using ECs and naming, we do not need to define abstract label attributes for all attributes where $l_i \leq l_j$. Meaning, in our example, the RSO class does not need to contain abstract elements that refer to all of the attributes of sub–class of RSO.

This feature is a powerful tool for building hierarchies; adding a new sub–class to the hierarchy need not modify any other class definitions or propagate any information "up" the class hierarchy. Yet accessing the attributes can still be accomplished in a top–down manner using references. That is, the reference to "myRSO.attitude" does not require that the RSO class have any encoded knowledge of the sub–classes where $l_i \leq RSO$. This allows anyone to extend the hierarchy without changing any other class definitions, greatly enhancing the modularity and reusability of hierarchies for multiple purposes.

Using references, it is also not required that every class representing a label contains a definition for every attribute. Each class need only to define the attributes that the label actually contains, regardless of the other attributes in the hierarchy. So, for example, we can add a longitude attribute for only GEO objects as

```
class GEO extends RSO {
 val attitude = ...
 val longitude = Uniform(-180, 180)(``longitude'', this)
}
```

Calling default.get(``myRSO.longitude'') using the declaration of myRSO from above will create a distribution of the longitude of RSOs using only the labels that have defined a longitude attribute (i.e., only GEOs). Note that we did not need to modify the LEO or RSO classes to use this reference.

Handling Evidence

Evidence and References

In Figaro, evidence can be applied to elements as constraints or conditions (hard constraints). These are functions from the value of an element to either a double value (constraints) or a Boolean value (conditions). Conditions can also be applied using observation notation, which just asserts that the value of the element must be the observed value.

Figaro allows evidence to be applied to a reference, and ultimately, to the elements resolved from the reference. Evidence can be applied using the assertEvidence function in the EC trait, where the user must define the type of evidence to apply to the resolved elements. For example, the statement:

```
default.assertEvidence(
 NamedEvidence(``myRSO.attitude'', Observation('stable))
)
```

asserts the observation that any element that is resolved by the reference "myRSO.attitude" should have a value of 'stable.

In this situation, however, it is unclear as to how the evidence should be applied to the resolved elements. A naive solution is to apply the evidence to the currently referred element, i.e., the element returned by get, but is incorrect. As the Figaro model represents a random process, the current state of the model may change and the element the reference refers to may change. For example, if the current value of myRSO is an instance of LEO class, we cannot apply the evidence only to the LEO class attributes, since the value of myRSO may change to an instance of GEO, thus changing the current resolution of the reference.

It seems then that the evidence should be applied to all possible elements that could be resolved by the reference. This too, is incorrect. While there are many possible values of myRSO, at any one time, the element's value is a single instance of RSO. Hence it is incorrect to always condition the attributes of other values of myRSO on the evidence. To see this, consider the observation above that the

RSO's attitude is `'stable`. `'stable` is not a possible value for the attitude of a `LEO`. So if we apply the evidence to `LEO.attitude`, the evidence will have probability zero, even if the current value of `myRSO` is an instance of `GEO`.

Contingent Evidence

To solve this problem, we extend Figaro with *c*ontingent evidence. The idea behind contingent evidence is that evidence should be applied to a resolvable element of a reference only when the reference is currently resolved to the element. In other words, contingent evident applies evidence to an element conditioned upon another set of elements taking on specific values.

Consider the example evidence from above where we observe that the attitude of an instance of `RSO` is `'stable`. In this case, we want to assert that `LEO.attitude` $==$ `'stable` only when `myRSO.isInstanceOf[LEO]` $==$ $true$. In this manner, we only assert the evidence on references contingent upon the values needed to resolve the reference.

When evidence is asserted using a reference, contingent evidence is automatically created and applied by Figaro, without the need for the user to manually specify how to create the contingent evidence (e.g., as above). Creating contingent evidence is a multistep process. First, we have to find all possible resolutions of the reference. Figaro will recursively expand the ECs in the reference and find all possible resolvable elements. Next, for each possible element, Figaro will create a list of contingent elements and the values they need to take to make the reference resolve to the element. The list is stored, along with the evidence, in the resolved element. Finally, for all elements along a resolvable path, Figaro will assert that they are only allowed to take on values that will result in a resolvable element. If no value of an element in the resolution path results in a valid resolution, then assert the element's value as null.

The last step is needed to ensure that elements only take on values that are consistent with the evidence. For example, if we observe that a RSO's longitude is 100, and only `GEO` classes have a longitude, then we must ensure that `myRSO` is never a `LEO`. We cannot apply the evidence of the longitude to a `LEO` (since they do not have one), but the semantics of the evidence state that the RSO cannot be a `LEO`, since the RSO we did observe has a longitude attribute.

During inference on the model, the value of an element is checked against any evidence and the list of contingent elements stored for that evidence. For a condition (Boolean evidence), a value of the element x satisfies the condition if

$$\neg(\bigwedge_{e,v \in Cont} e.value == v) \vee x.value == condition.value$$

where $Cont$ is a list of contingent elements and the values needed to resolve the reference to element x. These semantics enforce that whenever the values of the contingent elements are not equal to the stored values (needed to resolve the reference to this element), the condition on the element is always true. Because of this feature, we can avoid the problem as previously described: Evidence can only be dissatisfied when an element's value dissatisfies the evidence *and* the element is the current resolution of a reference.

Using contingent evidence, we can now reason about the label of a RSO given its observed attributes. For example, let us assume that an optical telescope detects a RSO and determines that the RSO's mass is within a certain range, and we wish to determine the label of the unknown RSO. Assuming the generative process and hierarchy previously described, we can accomplish this now:

```
val myRSO = rso("r1")
default.assertEvidence(NamedEvidence("r1.mass",
  Condition((d: Double) => d > 6700 && d < 7300)))
val alg = MetropolisHastings(50000, myRSO)
alg.start()
```

We simply create a new instance of an RSO, assert the condition that the RSO's mass is between 6700 and 7300 kg, and run the Metropolis–Hastings inference algorithm. Once the algorithm is complete, we can query the posterior distribution of the RSO instance.

Experiments

We implemented a large RSO hierarchy to test the utility of references in probabilistic programming. The Figaro code below has been modified for space, but the hierarchy and full code can be found in the supplement.

Hierarchy and Data Set

The RSO hierarchy is based on a recently proposed taxonomy for space objects (Wilkins et al. 2013). At the top level of the taxonomy is the RSO class, followed by classes that define the orbit of the object (GEO, LEO, Medium Earth Orbit and Ellipse). Subsequent to the orbit, RSOs are further broken down by the general mission type of the object: Observation, Communications, Navigation or Debris. Observation RSOs are further broken down into sub–missions (e.g, Meteorology). The top level RSO class is defined as

```
abstract class RSO extends ElementCollection
with Missions {
 val attitude: Element[Symbol] = getAttitude(this)
 val mass = FromRange(minMass, maxMass)("mass", this)
 val distance =
  FromRange(minDist, maxDist)("distance", this)
 val brightness: Element[Int]
 val brightnessRate: Element[Double] }
```

The attitude, mass and distance of an RSO are defined in the RSO class, but each sub–class of RSO is free to define their own distributions of these attributes. The attitude attribute is created from the `getAttitude` function, which is defined per the mission of the object (in the `Missions` trait). `FromRange` is Figaro notation for a discrete uniform distribution.

Unfortunately, these attributes are not directly observable. When observing an RSO via optical telescopes, one can only observe the brightness of the object and the rate at which the brightness is observed (e.g., rapid glints or no glints). These attributes are conditioned upon the attitude, mass and distance of the object. For instance, objects that are farther from earth have less intense brightness, and objects that are

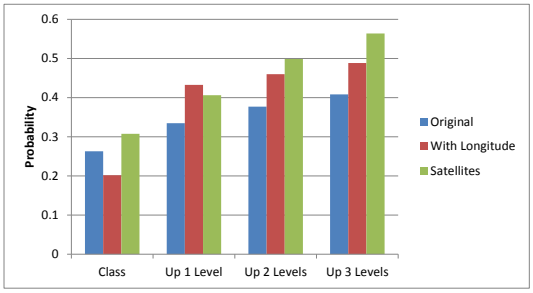

Figure 2: Testing results using the RSO taxonomy

spin stable have more glints than objects that are nadir. Since these attributes depend upon other attributes, they are not defined in the top level class (i.e., each sub–class of RSO must define them).

The RSO hierarchy just provides a taxonomical classification of RSOs. However, to highlight the benefits of using probabilistic programming for hierarchical reasoning, we also added class definitions for specific satellites in orbit. Using the database of 1047 known satellites compiled by the Union of Concerned Scientists (Grimwood 2011), we added class definitions for several satellites by extending existing hierarchy classes. For instance, the GOES13 satellite is defined as

```
class GOES13 extends GEO_Observation_Earth_Meteorology{
 override val mass = Constant(32)("mass", this)
 ...
 val longitude = Constant(-104.77)("longitude",this)
}
```

where the satellite definition overrides the parent class values for mass (and other attributes not shown). Note that this class also defines an attribute `longitude` that was not defined in the RSO class, as fixed longitudes are only properties of GEO satellites.

Testing

Currently, sensor resources are a significant limitation to accurate classification of RSOs. Given the large number of satellites in orbit, sensor resources must be used judiciously to ensure that observed data is valuable. For example, collecting low quality observations of a poorly lit but identified object instead of potentially high quality observations of a brightly lit but unknown object is a waste of resources. Hence, even low–specificity classifications using the RSO taxonomy can be extremely useful if the classification can prevent the misuse of further sensor resources. At present, no other RSO detection and classification scheme uses hierarchical reasoning as a means to triage sensor resources.

As a consequence, we tested the ability of the RSO hierarchy to detect and classify RSOs at different taxonomical classifications. We selected 50 random satellites from the UCS database, generated observation data from each satellite, and applied it as a constraint on the brightness and brightness rate of an unknown RSO. After applying evidence, posterior probabilities of the unknown class label were computed using the Metropolis–Hastings algorithm.

Fig. 2 shows the results of the random testing. First, we computed the expected posterior probability of the correct satellite label, shown in the first column. Then we computed the expected posterior probability of each parent label of the correct satellite label, up to three levels. As expected, the probability of each parent label increases at lower specificity classifications of an unknown RSO. It is clear, however, that this hierarchical reasoning can be combined with a decision rule to classify RSOs and possibly acquire more sensor data. For instance, if an RSO is an instance of an LEO with high probability, then additional sensor measurements can be taken to determine the sub–label of LEO more accurately; or if the probability of sub–labels are nearly equally likely, we can classify the RSO as the parent label instead of a more specific label.

To highlight the modularity and flexibility of using PPLs, we performed two additional tests, also shown in Fig. 2. First, we tested a situation where additional observation capabilities may be available, and if the longitude of an observed RSO can be determined, it is also applied as evidence to the model. Note that only GEO objects have a longitude, but application of this evidence to the appropriate objects is handled automatically. In this scenario, our posterior parent probabilities increased, reflecting the ability of the hierarchical model to reason with additional information when available, while requiring no changes in representation or inference. We also took observations from the 15 satellites encoded into the hierarchy and computed the posterior distributions for those observations. Not surprisingly, the distributions for this test were the highest. As adding new satellites to the hierarchy is extremely easy, this is an effective means to reason with an expanding catalog of satellites.

Conclusion

We presented a novel approach to representing and reasoning on probabilistic hierarchies in PPLs. Figaro references and contingent evidence enable flexible and modular probabilistic hierarchies. Hierarchies are easily expanded with new information, yet still accessed and manipulated without complete knowledge of the hierarchy. We also demonstrated the benefits of our approach for RSO classification, which has the potential to significantly impact the domain.

Due to the benefits of PPLs and references, creation of many powerful models is possible. One can easily create models with multiple RSOs that interact with each other, or create models with interacting hierarchies; the label of an RSO could also depend upon a hierarchy of body types, for example. In addition, the modularity and flexibility of this PPL representation enables easy creation of dynamic models, where we could use the hierarchy to track RSO observations over time. These types of models are targets for future research in probabilistic programming and RSO detection.

Acknowledgements

This work was supported by DARPA contract FA8750-14-C-0011.

References

Bakhtiari, A. S., and Bouguila, N. 2010. A hierarchical statistical model for object classification. In *IEEE International Workshop on Multimedia Signal Processing*, 493–498. IEEE.

Blei, D. M.; Ng, A. Y.; and Jordan, M. I. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research* 3:993–1022.

Burred, J. J., and Lerch, A. 2003. A hierarchical approach to automatic musical genre classification. In *Proceedings of the 6th International Conference on Digital Audio Effects*.

Fei-Fei, L., and Perona, P. 2005. A bayesian hierarchical model for learning natural scene categories. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, 524–531. IEEE.

Früh, C.; Jah, M.; Valdez, E.; Kervin, P.; and Kelecy, T. 2013. Taxonomy and classification scheme for artificial space objects. In *Advanced Maui Optical and Space Surveillance Technologies Conference*.

Goodman, N.; Mansinghka, V.; Roy, D.; Bonawitz, K.; and Tenenbaum, J. 2008. Church: a language for generative models with non-parametric memoization and approximate inference. In *Uncertainty in Artificial Intelligence*.

Goodman, N. D. 2013. The principles and practice of probabilistic programming. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 399–402. ACM.

Grimwood, T. 2011. Ucs satellite database. *Union of Concerned Scientists* 31.

Gupte, M.; Shankar, P.; Li, J.; Muthukrishnan, S.; and Iftode, L. 2011. Finding hierarchy in directed online social networks. In *Proceedings of the 20th international conference on world wide web*, 557–566. ACM.

Kuo, C.-L.; Buchman, D.; Katiyar, A.; and Poole, D. 2013. Probabilistic reasoning with undefined properties in ontologically-based belief networks. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (To appear)*.

Marszalek, M., and Schmid, C. 2007. Semantic hierarchies for visual object recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 1–7. IEEE.

McCallum, A.; Schultz, K.; and Singh, S. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems*, 1249–1257.

Microsoft Research. 2013. Infer.net api documentation. http://research.microsoft.com/en-us/um/cambridge/projects/infernet/.

Milch, B.; Marthi, B.; Russell, S.; Sontag, D.; Ong, D. L.; and Kolobov, A. 2007. Blog: Probabilistic models with unknown objects. In Getoor, L., and Taskar, B., eds., *Introduction to Statistical Relational Learning*. The MIT press. 373.

NASA. 2013. Space debris and human spacecraft.

Newton, J., and Greiner, R. 2004. Hierarchical probabilistic relational models for collaborative filtering. In *Proc. Workshop on Statistical Relational Learning, 21st International Conference on Machine Learning*.

Pfeffer, A. 2001. IBAL: A probabilistic rational programming language. In *International Joint Conference on Artificial Intelligence*.

Pfeffer, A. 2012. Creating and manipulating probabilistic programs with Figaro. In *2nd International Workshop on Statistical Relational AI*.

Sadovnik, A., and Chen, T. 2012. Hierarchical object groups for scene classification. In *IEEE International Conference on Image Processing (ICIP)*, 1881–1884. IEEE.

Sun, A., and Lim, E.-P. 2001. Hierarchical text classification and evaluation. In *IEEE International Conference on Data Mining*, 521–528. IEEE.

Wilkins, M. P.; Pfeffer, A.; Schumacher, P. W.; and Jah, M. K. 2013. Towards an artificial space object taxonomy. In *Advanced Maui Optical and Space Surveillance Technologies Conference*.