Tunable and Generic Problem Instance Generation for Multi-objective Reinforcement Learning

Deon Garrett
Icelandic Institute for Intelligent Machines
Reykjavík University, Iceland
Email: deong@ru.is

Jordi Bieger
Reykjavík University, Iceland
Email: jordi13@ru.is

Kristinn R. Thórisson
Icelandic Institute for Intelligent Machines
Reykjavík University, Iceland
Email: thorisson@ru.is

***Abstract*—A significant problem facing researchers in reinforcement learning, and particularly in multi-objective learning, is the dearth of good benchmarks. In this paper, we present a method and software tool enabling the creation of random problem instances, including multi-objective learning problems, with specific structural properties. This tool, called Merlin (for Multi-objective Environments for Reinforcement LearnINg), provides the ability to control these features in predictable ways, thus allowing researchers to begin to build a more detailed understanding about what features of a problem interact with a given learning algorithm to improve or degrade the algorithm's performance. We present this method and tool, and briefly discuss the controls provided by the generator, its supported options, and their implications on the generated benchmark instances.**

I. Introduction

The basic framework of Markov Decision Problems (MDPs) learned via interaction with the environment is a good fit for many difficult problems encountered in building life-long learning systems. However, current approaches are severely limited in how well they can cope with large numbers of disparate tasks, and relatively little is understood about how to build such systems. Solving these problems will require understanding exactly how the current approaches fail and characterizing the way that complex problem features affect new algorithms that may be developed.

Practitioners in the field of reinforcement learning often work with a small set of manually constructed benchmark problems. Classically, problems like the inverted pendulum or double-pendulum have been used to evaluate new learning algorithms. As algorithms evolved, new problem domains such as the mountain-car domain were created to pose additional challenges and exercise specific capabilities of targeted learning algorithms. However, in almost all cases, problem domains are manually constructed. The reasons for this are understandable; unlike in the world of optimization, the need for a coherent model of an underlying dynamical system to learn from makes human intervention one of the only feasible ways to generate new worlds. Naïve random generation of instances can result in problems that are too easily solved or are otherwise unrealistic.

This approach significantly limits the ability of researchers to begin to develop their own internal models of the interaction between features of a learning environment and features of a learning algorithm. Put simply, it can be difficult to tell how general a successful algorithm can be when testing of the algorithm is limited to only the few domains that humans have painstakingly developed and implemented.

Recently, researchers in the field of machine learning have also begun to consider multiple objectives in the context of reinforcement learning problems [5], [7]. However most prior work has focused on somewhat more narrowly construed goals or only a very small number of objectives, or both. These problems, involving domains such as robot navigation with a second task centered around power management (i.e., find the goal state, but also keep yourself charged), are interesting multi-objective problems, but the limited scope, size as measured by the number of tasks, and limited variability among tasks render them insufficient to provide a comprehensive test of multi-objective learning algorithms.

Here we should clarify a subtle distinction in terminology. Within the general framework of MDPs with multiple tasks, there are different interpretations of the precise nature of the problem, each with its own potentially unique approaches. These methods may be broken down into three categories: multi-objective learning, multi-task learning, and transfer learning [23]. In multi-objective learning, we assume there is a single MDP to be learned, but that the reward function is vector-valued. In this case, there is obviously a single set of dynamics involved, but the agent must learn a set of Pareto-optimal actions from each state in the environment. In multi-task and transfer learning, the agent is concerned with learning multiple MDPs, exploiting knowledge gained from prior learning to speed the acquisition of the new skill. In multi-task learning, it is assumed that all tasks are drawn from the same distribution.

The three formalisms share several similarities and can be applied in somewhat overlapping problem sets, but most naturally cover different scenarios in which reinforcement learning algorithms may be appropriate. For transfer and multi-task learning, the aim is largely to produce agents that may learn new skills over time – skills that were not needed and whose available feedback signals were not available during the agent's initial learning phases. Conversely, multi-objective learning more directly applies to systems that need to learn a more complex set of behaviors simultaneously. The different approaches may be combined as well, for example with a transfer learning system using information gained by one multi-objective learning algorithm to more efficiently acquire skills in a novel multi-objective environment. In this work, our focus is primarily on multi-objective reinforcment learning. The agent will be presented with a single defined environment involving multiple reward signals, possibly unrelated, and must learn a set of actions that are mutually non-dominated for each state in the environment.

It is our view that the current focus on manually constructed benchmark problems is only a part of the solution to the problem of developing new and better multi-objective learning algorithms. The advantages of real-world problem domains are many. They are much more direct measures of the ability of learning algorithms to solve the types of problems that occur in real situations, and they are often very large and challenging problems, providing ample opportunities to demonstrate improved algorithms. However, real-world problems are also very difficult to interpret. If a method performs well (or poorly), it is very difficult to generalize effectively from that result. Would other methods perform better or worse? Would the tested method perform well on similar types of problems? The lack of direct visibility into the structure in real-world problems hinders this sort of analysis greatly, while synthetic benchmark instances can be generated with specific structures, controls, etc. Thus we maintain that there is a need for both approaches in advancing the state of multi-objective learning.

So the situation is essentially that random problems, generated strategically, can help multi-objective learning researchers greatly expand the range of problems and algorithms they are able to study, but it can be very difficult to generate random problems that don't exhibit undesirable properties that can nullify this potential benefit. In this paper, we describe one approach to controlling some key properties of multi-objective learning problems.

II. Related Work

While no existing work has considered the problem of generating multi-objective reinforcement learning problems with the goal of providing controllable and tunable structure, there has been quite a lot of work done in the general field of multi-objective reinforcement learning. Modular Q-Learning [14] and similar approaches have formed the basis of much of this work. As we will discuss below, there is a significant problem awaiting techniques such as these as the number of simultaneous tasks required of the learner grows beyond the smallest example domains, and thus there is a need for benchmark problems that can be extended to better evaluate these methods under general domains. In addition, many studies of multiple objectives in reinforcement learning problems have focused on the problem of sequential objectives [10]. Rybicki et. al. [19] considered a multi-objective reinforcement learning problem using a combination of supervised and unsupervised learning that focused on the transfer of knowledge between tasks, but their work considered only a fairly simple test environment, and did not address the long term challenges of managing learning of many, often to some degree unrelated tasks.

Similarly, several researchers have focused on generating random MDPs for single-task learning domains. Early work on learning policies for MDPs often exclusively used randomly generated problems. Particularly in the case of finite MDPs, generating random problems is very simple. However, Archibald et. al. noted as early as 1995 the problem with significant numbers of papers in the field being focused on random problems with little or no control over critical properties of those problems [2]. In that work, a generator called PROCON was proposed that allowed for the control of several aspects of finite MDP structure, particularly the mixing time of the resulting Markov chains. More recently, Bhatnagar et. al. [3] developed a generator referred to as GARNET (Generated Average Reward Non-stationary Environment Test-bench) that remains a standard technique for researchers looking to apply learning methods to random problems today [6], [18], [20]. However, both the PROCON and GARNET problems are restricted to only single-task MDPs with a single global reward signal to be maximized, and neither includes support for real-valued state or action spaces.

In their work, Vamplew et. al. [24] proposed a set of small benchmark problems for multi-objective learning. Some of their problems were small enough to have the Pareto Fronts known via exhaustive search. Others adapted known problems from the single- task RL community by adding objectives. Only one problem exhibited more than two tasks – an extension of the mountain car problem. However, the tasks chosen to add there were to minimize the number of reversing actions and to minimize the number of accelerations. In both cases, given the dynamics of the domain, there are large correlations with the primary objective of escaping in as little time as possible. We know from the multi-objective optimization literature that the degree of correlation between tasks is one of the most important factors governing the performance of many multi-objective optimization algorithms [15], [16]. One contribution of this paper is to examine this in context of multi-objective reinforcement learning.

Recent work in multi-objective optimization (MOO) has shown that even for small numbers of tasks, naïve approaches to handling these trade-offs can fail catastrophically. Essentially, as the number of tasks increases, any process that tries to directly select Pareto optimal actions will necessarily decay to a random walk. To understand why, consider the definition of Pareto dominance. From a given state s, an action a_1 dominates an action a_2 if and only if taking a_1 results in an estimated long-term reward not worse than a_2 for all tasks in the domain, and strictly better at at least one task. Each time we add a task, we add a new dimension in which a_2 could be a better choice than a_1 when faced with state s. This exponential decrease in the probability that any one action will dominate another as the number of tasks increases beyond even very moderate levels means that direct search for Pareto-optimal actions must devolve into random search. Essentially, with five or more tasks, it becomes very likely that a random action will be Pareto-optimal or near-optimal – just picking any action at all is likely to result in moving toward a favorable outcome with respect to at least one task. As a result, MOO researchers view a qualitative difference between what they have called "multi-objective" and "many-objective" optimization, and to develop specific techniques for handling the latter (considered as problems with five or more objectives). Vamplew et. al. [24] advocated taking the lead from the multi-objective optimization community in attempting to better directly handle the problem of dealing with multiple objectives. Recent work from that community has demonstrated that it is critical to understand problem structure – most importantly the interaction between tasks – if one is to design algorithms that perform acceptably in higher dimensional task spaces.

It is also recognized within the RL community that testing on unknown problems is valuable. The 2013 Reinforcement Learning Competition (WRLCOMP) [25], held at ICML 2013, included real-world helicopter control [17], [1] and invasive species monitoring [12] tasks, but also included a third "polyathlon" task, in which learners were given a series of abstract and related problems to learn to solve. Such tasks are incredibly useful for estimating robustness of learning agents, but we aim to go a step further than this. In the WRLCOMP workshop, the polyathlon task was still created by humans with in-depth knowledge of how to specify the dynamics of the problem. This paper describes one approach to generating any number of problems automatically and provides methods by which the problems can be "tuned". That is, we aim to provide the ability to generate a problem domain with specific characteristics, and then to tweak these characteristics in a controlled way. For instance, transfer learning methods can be tested on a whole range of very similar problems, differing only in the degree to which the target tasks are related. Or a range of similar navigation problems can be generated, with control over the statistical relationships between features in the world (e.g., locations of goals and traps, etc.).

This is, of course, not to imply that existing problems should be abandoned. For better understanding the role that problem structure has on the performance of reinforcement learning algorithms however, there is a definite need for problem domains with tunable structure. This paper describes a preliminary approach to generating such a variety of single and multi-objective, continuous and discrete Markov Decision Problems in such a way that researchers can generate problems with specific characteristics to better understand the performance of their algorithms on a variety of search spaces.

III. Overview of the Merlin Generator

To address these issues, we have developed a generator called Merlin (Multi-objective Environments for Reinforcement LearnINg). Merlin is implemented in Python, and takes a modular approach to the problem, allowing the user to mix and match different components of an MDP to generate problems with the desired structure, including discrete and continuous problems, problems with varying numbers and types of goals, and problems including spatially structured dynamics.

MDPs, the underlying formalism behind reinforcement learning problems, define a particular framework for decision making in which an agent perceives the state of the environment, chooses an action from an available action set, executes the action, and observes the consequences of the action. In particular, in reinforcement learning, the results of executing an action are a pair: an immediate reward and a new observation of the state of the environment. The goal of the agent is to learn a *policy* – a mapping of states to actions which if followed many steps into the future, will result in maximizing the total reward received by the agent.

More formally, we can define an MDP by the following components:

- S – the state space consisting of all relevant information perceived by the agent.
- A – the action space consisting of all possible actions the agent may choose to take at a given time.
- $P_a(s, s')$ – the probability that executing action a when in observed state s will put the environment into state s'.
- $R_a(s, s')$ – the immediate reward received after taking action a in state s and transitioning to state s'.

Note that while some algorithms for learning policies on MDPs make further assumptions, often that S and A are finite for instance, the basic framework is extremely general. In Merlin's modular system, different options for these four components can be (mostly) independently selected to construct a given problem type.

The performance of any algorithm for solving these problems will depend on the specifics of the problem structure. As is known from the recent work in multi-objective optimization, one of the most important aspects of this problem structure is the number of distinct tasks and their degree of inter-correlations. As such, Merlin also allows an arbitrary number of tasks to be specified in the generated problem along with the correlation between each pair. In the remainder of the paper, we describe how Merlin works to combine selected features into a single defined Markov Decision Process with arbitrary numbers of tasks with (somewhat) arbitrarily complex interrelationships.

IV. Finite Discrete MDPs

It is easily seen that in the case of finite MDPs (i.e., those in which both the state and action sets S and A are finite), the dynamics can be captured by a non-deterministic finite state machine, in the context of finite MDPs, called a *transition graph*. If the state transition function P is deterministic, then the underlying state machine is also deterministic. The simplest form of random MDP considered here is a discrete model in which there are n states, m actions, and k distinct tasks. Each state-action pair thus maps to k rewards, one for each task. The state transition function is modeled by a directed random multi-graph with n nodes, each with exactly m outgoing edges (i.e., all actions are available from every state). Note that the in-degree of each node is *not* constant, but all nodes have positive in-degree. That is, all nodes are reachable in the graph from any starting position.

There is a large body of existing knowledge regarding the properties of different types of random graphs. Many of the most common types, such as the Erdõs-Rényi model [8], are not necessarily good candidates for representing the underlying transition dynamics in an MDP intended for testing reinforcement learning algorithms. In that specific case, the issue is that these graphs tend to have exhibit the so-called "small-world" structure in which the distance between any two nodes is likely to be small [4]. In the context of reinforcement learning, this means that the goal state is never very far away from any state in the graph, and the resulting problems may be too easy (this is related to the mixing time of the Markov chain, which the PROCON generator explicitly tries to ensure will be suitably large).

Other types of graphs, such as the random-caterpillar and random-lobster graph types [13], [11] allow more precise

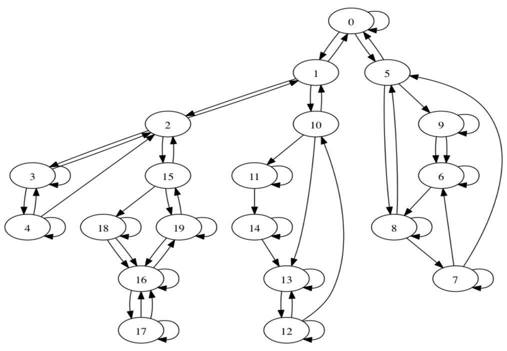

Fig. 1. Example of a small random fern with backbone length 5 and 5-node fronds.

control over the mixing time of the processes. Currently, all graph generators supported by the Python `networkx` package can be easily used as the basis for a generated instance in Merlin, as well as a few additional models the authors find useful. One particularly useful addition is what we're calling a *random fern* graph. A random fern is like a random lobster, but rather than simple 2-node chains hanging off the backbone, we attach larger and more complex graphs. Thus the overall structure is a backbone with many individually disjoint graphs attached as leaves, like the fronds on a fern. Figure 1 shows a small example.

No start or goal state is determined by the generator. Rather, there is a distribution of (vector-valued) rewards over every edge in the graph. The random graph can be viewed as the most basic transition graph, but Merlin supports several others which can be more useful in generating challenging instances.

Abstract graphs form an extremely flexible basis for specifying the transition dynamics of random MDPs, but can also be useful to look at more specific and semantically meaningful domains. Merlin provides some support for such problems. A common simple domain for benchmarking RL algorithms is the problem of learning navigation in spatial domains such as grid-world. Each state is connected to $2d$ other states in a fixed fashion dictated by spatial constraints (e.g., the agent can move left or right (1D); left, right, up, and down (2D); etc.). Mazes are a special case of the 2-dimensional grid-world domain in which there are walls present that block the agent from moving into certain states. We provide a specific generator for 2-dimensional mazes, along with code to visualize the mazes as well as the behavior of a given RL agent learning to navigate the maze. This allows Merlin to generate problems similar to the multi-objective deep-sea diver problem proposed by Vamplew [24].

Unlike most existing work in reinforcement learning, in the multi-objective model there exists no single reward signal to be maximized over time. Instead, there are k individual tasks to be learned, and each task contributes an immediate reward upon each state/action transition. Thus the multi-objective learning problem is one in which the reward signal is vector-valued. All other aspects of the learning problem remain the same.

A key difference between conventional reinforcement learning approaches and multi-objective methods lies in action selection. In the multi-objective realm, there is no well-defined "best" action to be selected in exploitative stages. Rather, there is a range of trade-offs represented by the available actions, each of which may be better or worse than others depending on which subset of tasks are viewed to be most valuable at the current time. Given an environment in which agents can learn to feed pets, vacuum the floors, and take out the garbage, we want a single agent to learn to perform all three tasks concurrently as it develops, but this implies that at any given time, it may find that it needs to go left to feed a hungry pet and go right to empty an overflowing garbage can. In this case, neither "left" nor "right" may be considered the optimal action; we can say only that they are (possibly) non-dominated. A key principle in multi-objective learning is that additional information is needed in order to perform the final action selection, often in the form of a human decision maker or a human-specified preference function. The goal of a multi-objective learner is to learn an appropriate set of options to pass on to this secondary decision-making process.

In Merlin, the rewards are assigned by sampling a multivariate normal distribution with a specified covariance matrix, given by the user in the form of a correlation matrix and standard deviations along each task. Formally, the user generates an instance by specifying the off-diagonal components of an upper-triangular matrix $\mathbf{C}^{\mathbf{R}}$, such that ${\mathbf{C}^{\mathbf{R}}}_{i,j} = r_{i,j}$, where $r_{i,j}$ denotes the correlation coefficient between tasks i and j. Along with the correlation matrix, a vector $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_k)$ is given with σ_i denoting the standard deviation of reward values for task i. A covariance matrix $\mathbf{C}$ is then computed using the identity $\mathbf{C} = \operatorname{diag}(\Sigma)\mathbf{C}^{\mathbf{R}}\operatorname{diag}(\Sigma)$.

Arbitrary correlation matrices can be specified by the user, subject only to physical constraints – specifically the derived covariance matrix $\mathbf{C}^{\mathbf{R}}$ must be positive definite. This restriction prohibits illogical inter-task relationships, such as the situation that task 1 is highly positively correlated with task 2; task 2 is highly positively correlated with task 3, and task 1 is highly anticorrelated with task 3. For any valid covariance matrix then, $\mathbf{C}^{\mathbf{R}}$ characterizes a multivariate normal distribution, which is then sampled as many times as needed to obtain a reward value for every state- action-task triplet in the generated instance. The resulting samples are then reshaped appropriately to yield k columns of mn (the number of states and actions respectively) values each such that the pairwise correlation coefficient between columns i and j is approximately $r_{i,j}$.

For spatial navigation problems, we take "correlation" between tasks to imply not necessarily that the value of each tasks rewards are high, but that the spatial location of the rewarding and penalizing states in the graph is correlated. That is, if there are two pots of gold in a grid-world problem with high correlation values, then they will be located near one another in the graph, but not necessarily with correlated values.

V. Continuous MDPs

For finite MDPs, the procedures described above are sufficient, but many of the most interesting applications of reinforcement learning are continuous problems. As such, it is vital that any proposed test suite or generator include the ability to deal with continuous models. Of course, it is easy to generate a continuous Markov Decision Problem, but the most obvious ways do not admit the possibility of controlling, or even predicting important aspects of the structure of the generated problems.

Most widely used reinforcement learning algorithms assume finite state and action sets. Continuous problems can be handled by discretizing the continuous values with a suitable granularity, but such knowledge may be difficult to obtain. In other cases, the granularity required to learn a suitable policy may be so fine that, while finite, the size of the discretized spaces is still too large to be effectively learned. Actor-critic methods [21], [3] are a commonly used technique for dealing with continuous problems and very large state and action spaces. In those methods, a continuous model tries to learn a more compact representation of the observed state-transition dynamics at run-time. The proposed methods here share many similarities with actor-critic algorithms, but rather than learning an approximation to a discrete model via observation at the time of learning, we do so as a way to directly represent continuous problems.

In finite MDPs, the states and actions may be completely arbitrary (e.g., "from state S, executing action A yields a new state S'"). No semantic information need be attached to the labels of states and actions. However, in problems with continuous state and/or action spaces, this is obviously not true. In the classic inverted pendulum problem, the states correspond to physical measurements of the system, and the transition dynamics are specific by numeric functions of the state-values and action-values. In order to generate random continuous MDPs then, Merlin must have some method by which states and actions may be assigned numeric values, and the dynamics should ideally have some degree of structure to them that depends on these values.

The approach proposed here is to split the problem into two separate parts. Initially, we generate a discrete problem, using the same graph generation algorithms described in the previous section. This allows us to ensure that, among other things, that the state space is reasonably well covered and that the underlying dynamics of the model are connected. Then the discrete model is annotated with state and action values and used to learn a generative (continuous) model which can serve as the true continuous MDP instance.

One option for producing these annotations is to generate state values uniformly at random. Merlin can support this method, but the resulting problems are likely not representative of the types of structure that we're most often interested in. In most real problems, the autocorrelation of state values along a random walk is usually somewhat positive. Applying a small action in a known state is more likely to result in a new state that is similar to the prior one. Ideally one would like the ability to control how "rugged" the walk through the state-space graph is for a given problem.

To approximate this type of structure, we take a series of random walks through the (un-annotated) graph without replacement of visited nodes, recording each node encountered along the way. When no unvisited nodes are reachable, we generate a time series of real values of length N (where N is the number of nodes visited on this walk) and assign each node in order of visitation the corresponding value in the time series. For multi-dimensional state vectors, we generate k independent time-series (one for each dimension of the state space) and use each as one dimension of the state vector. If there are still unlabeled nodes in the graph, new random walks are constructed until all nodes have been assigned a state value.

To provide the autocorrelation structure mentioned above, multiple methods of generating the time series are provided. In addition to uniform random generation, Merlin includes a Gaussian random walk in which each value is generated from the prior step by adding a zero-mean Gaussian random number with a specified variance, and the random midpoint displacement method popularized in fractal terrain generation [9]. Figure 2 shows examples of the two methods. As shown in the figure, each method has a parameter governing the ruggedness, but the fractal method allows a separation between micro-level ruggedness and macro-level ruggedness, while the Gaussian walk has a single scale parameter governing both.

Once all the nodes of the graph have been annotated with state values using the methods described above, each edge must be assigned a numeric action value. Unlike state values, the action values for a problem are typically bounded only by some range, e.g., you may apply a force of between -1 and $+1$. Randomly assigning action values to edges would be a reasonable approach, but as the state and action values are going to be used as training data for a regression model, it is useful if we can ensure that (a) we have some reasonable coverage of the range of possible actions from a given state in the training set, and (b) that we do not generate two nearly identical actions from a single state that map onto two very different successor states.

Here, a relatively simple solution is chosen. A user-specified parameter sets the range of possible action values, and this range is partitioned into E non-overlapping intervals, where E is the number of outgoing edges from the current node. Each edge is then labeled with an action value sampled uniformly at random from the interval corresponding to the edge.

This mapping of values to both states and actions then defines *part of* a continuous MDP. However, the vast majority of real-valued states are not represented in the finite set of annotated nodes, nor are the small set of actions representative of a problem with truly real-valued actions. To solve this problem, we use the annotated graph as training data for a supervised learner with the goal of learning the underlying dynamics of the discrete problem. If, in the graph, node `A` is connected to node `B` via edge `E`, we construct a training point from the assigned mapping as $(state\_value[A], action\_value[E]) \rightarrow state\_value[B]$. If the classifier used to learn these dynamics builds a continuous internal model for prediction, then we can use this internal model as a drop-in replacement for the discrete version of the problem.

In principle, any continuous model could be used, but the specific needs of the problem make some methods less suitable than others for the task. In particular, the variations in state-values as an agent moves from one part of the space to another may be important, but many learning models may be biased towards treating them as noise to be smoothed. To take a simplistic example, consider a "pot-of-gold" reward function that is zero everywhere except for one state. In this

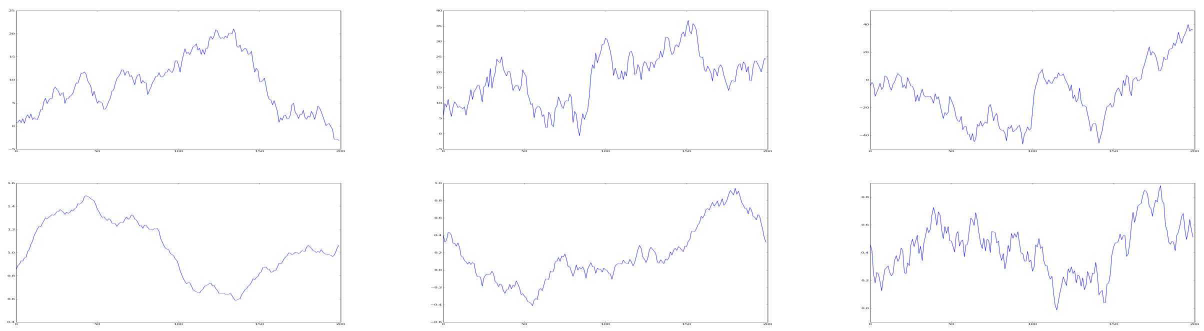
Fig. 2. Three sample random state-value progressions using Gaussian random walks (top, from left, $\sigma = 1.0, 2.5, 5.0$) and the Random Midpoint Displacement fractal method (bottom, from left, $R = 0.85, 0.6, 0.5$).

model, there is a single state/action pair which yields a positive reward; all others yield zero. If we approximate this function using, for example, linear regression, we lose the critical property of the reward function – that it is uniform throughout the space except for the one positive transition which is the very property that makes the problem interesting.

Because of this, Merlin attempts to approximate the dynamics of transition and reward functions using much less regularization that would be typical for conventional classification or regression tasks. In essence, we want to maintain as much of the local structure in the underlying dynamics as possible, while inferring only the real-values that were not present in the training set. This does open the door to overfitting becoming problematic. In practice, this aspect of Merlin is somewhat experimental. It often generates sensible continuous models, but some care must be taken with the generated instances to ensure that they do not exhibit undesirable behavior as they generalize away from the state-action pairs they were trained on. Work is ongoing to improve the reliability of the approximation schemes.

Because we do want some variation that would ordinarily be considered a sign of overfitting, Gaussian Processes can be a fairly natural approach to the problem. Unlike SVMs, for example, they are dense and guaranteed to perfectly reproduce the training data. Some care is needed to ensure that sufficient coverage of the state/action spaces is given during training to prevent the model from straying too far from the observed values, but if done properly, the resulting approximation model captures very well the structure present in the underlying graph, while still providing a true continuous model.

One drawback of Gaussian processes is that when used in this way, they can be very demanding of memory. For large graphs, the approximate model may be prohibitive to deal with. Merlin provides additional approximation methods as well for cases in which the Gaussian process model is a poor fit. Currently Merlin provides direct support feed-forward neural networks (trained via the standard backpropagation algorithm) and support vector regression to form the surrogate model. However, any other supervised regression method can be fairly easily swapped in if needed.

VI. Perturbation of Generated Instances

Once the model has been trained, it is written out in a form that can be read back in and used to provide a reinforcement learning agent with a model of the state transition and reward dynamics of the problem. Another benefit to this approach is that once learned, the model can be perturbed by a specified amount to yield many more problems that have similar, but not identical structure. This self-contained model makes integration with most code bases relatively simple, and work on integration with RL-Glue [22] is in progress.

Consider the case of a neural network approximation model of the state transition dynamics. The network has a set of parameters – the weights of the network trained via backpropagation. If we alter $X\%$ of those parameters by adding some noise, we obtain a new approximation model that will behave, to approximately a prescribed degree, similar to the existing one. This opens up the possibility of performing controlled experiments with learning algorithms. If a learner performs extremely well on one problem, we can not only generate many more random problem instances with the same hyper-parameters (here meaning the parameters passed to Merlin to generate the instance), but we can also begin to alter the instances to detect the point at which the algorithm's performance begins to change. While this method does not provide detailed information regarding exactly what features of the instance the algorithm was responding to, it does provide useful information regarding how robust the learner is to changes in the type of problems it has been tested with.

This provides one of the key benefits of our approach. Unlike manually constructed models, we can explore the space of problem domains around a given model. This allows a great deal of information to be obtained about how different learning algorithms react to different types of problem structure. Not only can you estimate robustness of the algorithm more effectively, but you can also begin to build an understanding of why a particular approach succeeds or fails by careful experimentation.

VII. Experimental Analysis

There are several important types of problem structure that are ultimately important to understanding algorithm performance. As the primary novelty in this contribution involves the generation of multi-task problems, we pay particular attention

TABLE I. AVERAGE PARETO FRONT SIZE ACROSS ALL STATES FOR (LEFT) BI-OBJECTIVE INSTANCES WITH THE SPECIFIED CORRELATION COEFFICIENTS BETWEEN TASKS, AND (RIGHT) MULTI-OBJECTIVE INSTANCES WITH 1000 STATES, 20 ACTIONS, AND k UNCORRELATED TASKS.

r	Size of Pareto Front
0.6	1.91 ± 0.85
0.2	2.39 ± 1.02
0.0	2.77 ± 1.03
-0.2	3.19 ± 1.10
-0.7	4.19 ± 1.20

k	Size of Pareto Front
2	3.60 ± 1.44
3	7.22 ± 2.33
4	10.98 ± 2.67
5	14.19 ± 2.51
10	19.68 ± 0.60

to aspects of the problems that directly impact the multi-objective aspects of the generated instances. Due to space constraints, we also focus on the discrete instance generation methods.

As mentioned previously, in the multi-objective optimization community, it has been found that a critical factor is the degree to which an algorithm can differentiate between the desirability of different candidate solutions. In the context of reinforcement learning, this refers to the ability to discriminate between good and bad actions in a given state. One critical factor affecting this structure is the number of tasks (and their inter-relationships). Merlin supports arbitrary numbers of tasks, and the task relationships may be specified using a correlation matrix as described in Section IV. We generated 50 instances with 500 states, 8 actions, and 3 tasks with a correlation matrix

$$R = \begin{pmatrix} 1.0 & 0.7 & -0.2 \\ 0.7 & 1.0 & -0.3 \\ -0.2 & -0.3 & 1.0 \end{pmatrix}.$$

Measuring the sample correlations between the rewards across the 50 instances yielded

$$\begin{pmatrix} 1.0 \pm 0.0 & 0.703 \pm 0.007 & -0.202 \pm 0.014 \\ 0.703 \pm 0.007 & 1.0 \pm 0.0 & -0.298 \pm 0.017 \\ -0.202 \pm 0.014 & -0.298 \pm 0.017 & 1.0 \pm 0.0 \end{pmatrix}.$$

The residuals are primarily a function of problem size, with larger problems yielding correlations closer to the target than smaller problems, but even with quite small problems, the generated instances tend to closely adhere to the desired task relationships. This has been verified over a larger set of sample instances, but these are elided from the paper for space.

We next measured the impact of both problem size (measured by number of tasks) and this correlation structure on the Pareto fronts across the state space. To do this, we generated another set of instances with varying parameters, and then approximated the Pareto action set at each state by running several long random walks and performing SARSA weight updates at each step. The algorithm is equivalent to SARSA with ϵ-greedy selection and $\epsilon = 1.0$ (i.e., all actions were chosen at random). We performed a 5000 of these random walks from different starting positions, with each walk consisting of 100,000 steps, and after all runs, calculated the nondominated actions from each state. Table I shows the results of these experiments.

From these tables, we see two obvious trends. Most strikingly, increasing the number of tasks dramatically increases the size of the Pareto-optimal set of actions. By the time we reach 10 objectives, essentially every possible action from any state is Pareto-optimal. This strongly highlights the need for

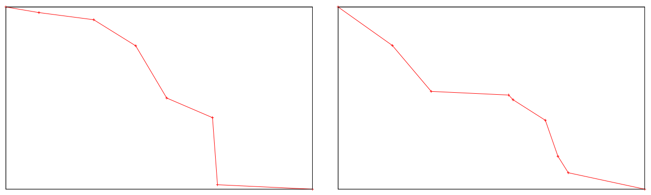
Fig. 3. Sample Pareto fronts obtained from randomly selected states on a problem with 1000 states, 20 actions, and two moderately anti-correlated objectives.

algorithms that do more than simply enumerate options. It says that all else being equal, multi-objective reinforcement learning algorithms likely mut include some form of external multi-criteria decision-making process to break ties during the learning process. We also see that even with smaller numbers of tasks, increased correlations between tasks reduces the number of Pareto-optimal actions available, and anti-correlation increases the dimensionality of the Pareto-optimal action set. As might be expected with random rewards, the obtained Pareto fronts are not necessarily convex. Figure 3 shows examples of the obtained fronts from one of the uniform random graph problems.

One final type of structure that is important for any MDP in the context of reinforcement learning, multi-objective or otherwise, is related to the hitting time of the chain. In a typical application of RL, there may be a small set of states that provide all the positive and negative reward to the agent. For instance, learning to play games generally involves some small uniform reward signal for every move until the game is won or lost, at which point a larger signal is generated for the agent to learn from. In order to model interesting problems of this type it is important for any generator to have the ability to create problem instances in which there are goal states that cannot be reached in a small number of moves from an arbitrary starting state.

Much of the relevant functionality in Merlin is orthogonal to the selection of the underlying graph type, so one could substitute an arbitrary transition graph in for the dynamics and then impose the desired number of tasks and the inter-task reward structures on top of this graph using the methods described earlier. One area of ongoing work is to incorporate more types of transition graphs from the literature into Merlin directly. However, we also include some rudimentary methods for having more control over the transition structure than the uniform random graphs allow.

The random fern graphs described in Section IV are one such approach. These more complex graphs serve to lead a reinforcement learner into a separate part of the state space from which it may be difficult to escape to other parts. This can be used to increase the expected hitting time of a random walk on the chain. Due to space constraints, we present only a small sample demonstrating this effect, as shown in Figure 4. There, we show how quickly a random walk can cover the graph using small and slightly larger instances of a uniform random graph and a random fern with the same number of nodes and edges. We see that when the "fronds" of the fern are relatively small, the walk covers the graph very quickly. However, by making the fronds deeper and more complex, we

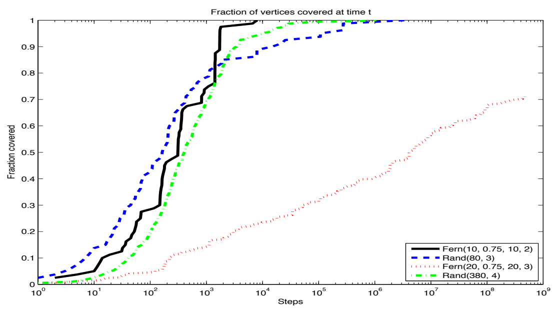

Fig. 4. Comparison of the estimated cover time for different classes of graphs, two random ferns and two uniform random graphs, in each pair a small and moderately larger instance.

can force the agent to spend large amounts of time exploring these somewhat isolated regions of the state space.

VIII. Summary

We have developed a tool for generating random MDP instances, including both discrete and continuous and single and multi-objective problems with support for tunable control over several aspects of problem structure. The primary benefit of the proposed method is to provide the ability to test reinforcement learning algorithms, particularly multi-objective algorithms, across a wide range of abstract problems.

The implementation of this method in the form of the Merlin tool is available from https://github.com/deong/merlin. The tool provides a flexible interface by which different aspects of a Markov Decision Process can be combined to generate large numbers of random problem instances sharing specified structural properties. These problems can be saved and restored allowing for integration into existing reinforcement learning code-bases.

Work is ongoing to integrate other existing work such as the GARNET and PROCON generators into the framework, which would allow multi-objective learning researchers to use direct extensions of these known generators. There are several other aspects in which Merlin could be improved as well. Support for continuous MDPs is included at present, but needs additional research to be a completely usable black-box generator – currently it requires some care to generate suitable continuous instances. Another such area of future work is in modeling non-deterministic MDPs and modeling additional types of real-world-inspired problems. Additionally, work is ongoing is to use the generator to perform a much greater review of the performance of existing multi-objective learning methods and extend the generator to more realistic problem classes.

IX. Acknowledgments

This work was supported by EU Marie Curie CIG grant number 304210 and by a Centres of Excellence grant from the Science and Technology Policy Council of Iceland.

References

[1] P. Abbeel, A. Coates, M. Quiqley, and A. Ng. An application of reinforcement learning to acrobatic helicopter flight. In *Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems (NIPS)*, 2006.

[2] T. W. Archibald, K. I. M. McKinnon, and L. C. Thomas. On the generation of markov decision processes. *Journal of the European Operational Research Society*, 46:354–361, March 1995.

[3] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee. Natural actor-critic algorithms. *Automatica*, 45(11):2471–2482, Nov. 2009.

[4] B. Bollobás. *Random Graphs (2nd Ed)*. Cambridge University Press, 2001.

[5] L. Bull and M. Studley. Consideration of multiple objectives in neural learning classifier systems. In *Parallel Problem Solving from Nature (PPSN VII)*, pages 549–557. Springer, 2002.

[6] D. D. Castro and R. Meir. A convergent online single time scale actor-critic algorithm. *Journal of Machine Learning Research*, 11:367–410, 2010.

[7] F. L. Crabbe. Multiple goal q-learning: Issues and functions. In *Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA)*, 2001.

[8] P. Erdős and A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[9] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25:371–384, 1992.

[10] Z. Gábor, Z. Kaár, and C. Szepesvári. Multi-criteria reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 197–205, 1998.

[11] J. Gallian. Dynamic survey of graph labeling. *Electronic Journal of Combinatorics*, 14(DS6), January 2007.

[12] K. Hall, M. A. Taleghan, and H. J. A. annd Thomas G. Dietterich. Managing invasive species in a river network. In *Proceedings of the Third International Conference on Computational Sustainability*, 2012.

[13] F. Harary and A. J. Schwenk. The number of caterpillars. *Discrete Mathematics*, 6(4):359âĂŞ365, 1973.

[14] J. Karlsson. *Learning to solve multiple goals*. Phd thesis, University of Rochester, Rochester, NY, USA, 1997.

[15] J. Knowles and D. Corne. Towards landscape analyses to inform the design of a hybrid local search for the multiobjective quadratic assignment problem. In *Soft Computing Systems: Design, Management and Applications*, pages 271–279. IOS Press, 2002.

[16] J. Knowles and D. Corne. Instance generators and test suites for the multiobjective quadratic assignment problem. In *Evolutionary Multi-Criterion Optimization (EMO 2003), Second International Conference*, pages 295–310, Faro, Portugal, April 2003.

[17] A. Ng, H. J. Kim, M. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In *Proceedings of the Eighteenth Annual Conference on Neural Information Systems Processing (NIPS)*, 2004.

[18] B. Piot, M. Geist, and O. Pietquin. Learning from demonstrations: Is it worth estimating a reward function? In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD 2013)*, pages 17–32. Springer, 2013.

[19] L. Rybicki, Y. Sugita, and J. Tani. Reinforcement learning of multiple tasks using parametric bias. In *Proceedings of the 2009 International Joint Conference on Neural Networks (IJCNN)*, pages 2013–2020, 2009.

[20] B. Scherrer. On the performance bounds of some policy search dynamic programming algorithms. Technical report, INRIA, 2013.

[21] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[22] B. Tanner and A. White. RL-Glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10:2133–2136, September 2009.

[23] M. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

[24] P. Vamplew, J. Yearwood, R. Dazeley, and A. Berry. On the limitations of scalarization for multi-objective reinforcement learning of pareto fronts. In *AI 2008: Advances in Artificial Intelligence*, pages 372–378, 2008.

[25] 2013 ICML workshop on the reinforcement learning competition, 2013. https://sites.google.com/site/rlcomp2013/icml_workshop.