Domain-Independent Planning Model Repair via Heuristic Search

Wiktor Piotrowski,[1] Roni Stern, [1,2] Yoni Sher, [1] Jacob Le,[1] Matthew Klenk, [3] Johan de Kleer[1] Shiwali Mohan,[1]

[1] Palo Alto Research Center, CA, USA
[2] Ben-Gurion University of the Negev, Beer-Sheva, Israel
[3] Toyota Research Institute, CA, USA
wiktorpi@parc.com, sternron@post.bgu.ac.il, yonsher@parc.com, jale@parc.com, klenk.matthew@tri.global, dekleer@parc.com, smohan@parc.com

Abstract

Automated planning agents are largely ill-equipped to act in dynamic environments where a novelty can affect their fundamental characteristics and behavior. Model-based agents are often characterized by static planning models and rigid unchanging assumptions about the dynamic environments in which they operate. Contingency strategies, such as immediate replanning upon execution failure, only address a small fraction of cases and fail when it is unknown how the novelty has changed the environment. We propose a heuristic search-based domain-independent approach to simultaneously identify the novelty and repair the agent's planning model to adapt to the unexpected changes in its environment. We report empirical evaluations on the well-known balancing Cartpole problem, a standard Reinforcement Learning (RL) benchmark and Angry Birds, an established IJCAI competition domain. Our results show that by exploiting the proposed model repair approach, our PDDL+-based agents are able to quickly and effectively detect and adapt to some classes of novelties.

Introduction

Automated Planning strives to tackle complex problems rooted in real-world applications. However, model-based planning agents are largely ill-equipped to act in dynamic environments. Realistic scenarios include shifting conditions and unforeseen changes affecting the core characteristics of the environments. **?** refers to such impactful changes as *novelty*. In this paper, we focus on scenarios where a novelty can change the underlying model suddenly and at any time, in unknown and unexpected ways *not according to the known system dynamics*.

The vast majority of planning models, and particularly ones described in PDDL (**?**) and its various levels, assume a static and deterministic world. They are rife with abstractions and approximations, and contain fixed values of important constants hand-selected by the designer. As a result, while planning models can capture the essence of a realistic scenario and offer explainability, they are often not sufficiently adaptable since the model can quickly fall out of alignment with the changing environment it is supposed to represent. Such inadequacies become apparent only after a plan has failed during its execution. Moreover, a failed plan execution does not identify the source of the issues, it merely notifies of an error or inaccuracy in the model. Unexpected changes in the latent part of the environment are notoriously difficult to detect and identify, and cannot be handled using conventional contingency acting strategies such as replanning (**?**).

Dynamic environments are often used as motivation for learning methods such as reinforcement learning (RL). RL makes minimal assumptions about the dynamics of the environment and uses feedback from the environment to learn action selection. While promising, RL methods have some significant shortcomings. **?** studied deep Q-learning agents and demonstrated that even changes that make the task easier can lead catastrophic failures. Further, RL agents require large amounts of training experience that may not be accessible in real-world settings. And, finally, knowledge acquired by RL agents is distributed in a q-network or a q-table and is challenging to explain.

Despite the aforementioned shortcomings of model-based planning, it is worthwhile to investigate approaches that adapt the model based on observations from the environment. Not only can planning models be robust to minor perturbations in the environment, adaptation to novelty usually requires changing only a small part of the model and can be done in a data-efficient manner.

The main contribution of this paper is an *domain-independent approach for adaptive repair of planning models*. Our approach can autonomously detect and characterize novelties introduced in the environment, as well as adapt the underlying planning model to the introduced novelty for robust behavior. The adaptation is performed via domain-independent heuristic search that generates modifications to the planning model that capture the effects of novelty on the environment.

Our approach is applicable to various levels of PDDL, though, we argue that representing meaningfully complex environments and novelties requires a highly expressive modeling language to capture their intricate structures and dynamics with sufficient accuracy. Our approach implemented in planning agents that reason over a PDDL+ model (**?**) of the environment to plan actions. We empirically evaluate the approach on a (RL) benchmark - OpenAI Gym's balancing Cartpole domain (**??**) and compare the results against deep-q network (DQN) RL agents. Our results show that model-based planning agents are *resilient* to novelties; i.e, the degradation in their performance is not as severe as the DQN agent. Second, we show that our proposed approach enables a planning agent to adapt *quickly*, requiring less than 20 episodes to return to optimal performance after novelty has been introduced, much faster than the DQN RL agent. Finally, our approach is *explainable* by design. The adaptations are represented in terms of changes to the elements of its PDDL+ domain model enabling inspection of the proposed changes.

In addition to being crucial for operations in dynamic environments with novelties, adaptive model repair is a versatile mechanism that can be exploited in various ways. It can be used improve the planning model accuracy by adjusting the values of the environment's parameters and variables based on observations made during plan execution. We report empirical results from AngryBirds, an IJCAI competition domain to demonstrate this capability.

Related Work

Our approach is best suited to numeric planning domains[1] (i.e., PDDL2.1 (**?**) and later versions). To capture the true nature of the physics-based simulation environments and showcase that our approach can handle realistic scenarios with complex system dynamics, we opted for PDDL+ (**?**) as our modeling language. PDDL+ is a feature-rich domain-independent planning modeling language that has been used to model a range of complex planning problems, including Chemical Batch Plant (**?**), Atmospheric Reentry (**?**), Urban Traffic Control (**?**), and Planetary Lander (**?**). Using PDDL+ allows our planning agent to be used in a broad range of applications, including domains that feature discrete and continuous state variables, exogenous activity, and non-linear system dynamics.

Model repair is a well studied field. **?** created a framework, called IRS, designated for model repair that uses both partial order planning with GDE-style (**?**) and model based diagnosis (**?**) to achieve an integrated approach to repair. Using the combination of the two approaches the authors address the cost difference of repairing different components, using the planner's output, and identification of the faulty component.

Chen et al. (**?**) and Yang et al. (**?**) studied the model repair problem in the context of repairing obsolete Markov Decision Problem (MDP) models. They proposed an approach based on using a model-checker to ensure that the repaired MDP satisfies some require constraints. Pathak et al. (**?**) explored the problem of repairing a discrete time Markov Chain.

Barriga et al. (**?**) proposed PARMOREL, a framework that considers the user's customization preferences in the repair process. The framework enables the user to choose the best sequence of actions for repairing a broken model by selecting the most suitable setting for the repair. Gragera et al. (**?**) created an algorithm to repair planning models where effects of some actions are incomplete. Their approach compiles, for each unsolvable task, a new extended task where actions are allowed to insert the missing effects.

Model repair for automated planning is largely unexplored. Molineaux et al.(**?**) used abductive reasoning about unexpected events to expand the knowledge base about the hidden part of the environment and improve their replanning process. In most works, however, PDDL domains are deemed correct by design and rarely are they tested by execution in a realistic simulation environment or the real world. Even then, any discrepancies during execution of the generated plans are usually attributed to partial observability or non-determinism of the target environment (which cannot be easily encoded in most PDDL-based domains). To date, to the best of our knowledge, there are only two commonly used approaches to handle cases where plan execution has failed.

Replanning (**??**) is considered to be an efficient and effective strategy with some evidence showing it to be the preferable option (**?**) for dealing with execution failure. It attempts to generate a new solution to the problem, either from the very beginning or from the point of failure of the plan, by using updated information from the environment. However, replanning works on the assumption that information on the parts of the environment that have unexpectedly changed and, thus, caused the plan execution failure are freely available and can be queried at any time. Therefore, if the target environment is not fully-observable (as is the case with most real-world scenarios) and novelty affects the latent space, replanning will continue to fail.

Plan repair (**???**) is another approach for handling plan execution failures. In this field of research a generated plan fails to achieve a requested goal and the algorithms are requested to adapt the plan to according to additional data so that it will then be able to achieve the desired goal. (**?**) show that plan repair can yield more robust solutions and perform more efficiently than replanning. Our proposed approach is novel in that it repairs the model that underlies plan generation in contrast with previous work that attempts to repair the plans that failed during execution.

Problem Definition

An *environment* in our context is a planning domain $D = \langle F, X, H, S_I, \mathcal{G} \rangle$ where F is a set of discrete state variables; X is a set of numeric state variables; H is a set of *happenings*, which can be either actions performed by the agents, events that were triggered, and durative processes that are may be active; $S_I \subseteq S$ is a set of possible initial states; and $\mathcal{G}$ is set of possible goals in the domain. A state is a complete assignment of values to the state variables $F \cup X$. A goal is a possibly partial assignment of values of the state variables $F \cup X$. A *problem* in a domain D is a pair $\langle s_I, G \rangle$ where $s_I \in S_I$ and $G \in \mathcal{G}$.

We consider a planning-based agent, i.e., an agent that acts in the world by first calling a planner to solve a problem $\langle s_I, G \rangle$ in a domain D, and then acting according to that plan π. The agent interacts with the environment in a sequence of **episodes**. Every episode represents an attempt the agent makes at solving a single problem. The result of an agent playing an episode is a *trajectory* τ, which comprises

[1]The approach is also applicable to classical domains by casting values of predicates as numeric inside the repair mechanism, and allowing the repair to change those values. When updating the PDDL, for some propositional variable $x = True$, if the internal repair value $x > 0$, and $x = False$ otherwise.

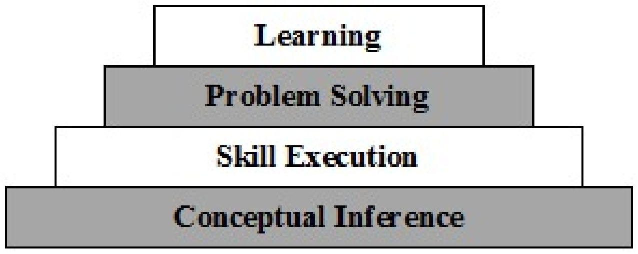

Figure 1: Diagram of the agent's architecture for model repair. Note that, when using an accurate planning model D, only the green components are engaged. Once novelty is detected, however, the remaining repair-focused components (shown in pink) begin modifying the PDDL+ model.

a sequence of tuples of the form $\langle s, a, s' \rangle$, representing that the agent performed action a in state s, and reached state s'.

The problem we consider in this work occurs when the domain the agents is using D is different from the real-world in which the agent is operating in. Let D^* denote the domain of this real world. We refer to D as agent's internal domain and to D^* as the real domain. Our objective is to modify D such that it allows the agent to solve problems in D^*. The input to the problem solver is the agent's current internal domain D, plan π, and the observations trajectory τ created by the agent playing an episode in D^*. The output is a modified internal domain D' that enables solving problems in D^*.

High-Level System Description

This section walks through the planning agent's typical reasoning process cycle complete with executing generated plans in the real environment. A graphical depiction of the overall architecture which supports the agent's acting and repair functionality is presented in figure 1. In the figure, steps 1-3 are represented by the outer loop and green components, steps 4-6 are carried out by the inner pink components. For clarity, we added gray guidance arrows on the outer edges of the graph for steps 1-3 to help the reader follow the initial stages of the process cycle.

1. **PDDL+ model generator.** At the beginning of an episode, the PDDL+ problem (s_I) is created by the PDDL+ model generator which takes as input a combination of the episode's initial state data from the environment and complementary assumptions about the environment. The problem is then paired with a PDDL+ domain file, hand-designed a priori, to compose the agent's internal PDDL+ model D.
2. **PDDL+ planner.** The agent attempts to solve the generated PDDL+ problem of the current episode, and stores the plan π.
3. **Plan execution.** The agent sends the generated plan for execution inside the real-world environment (D^*).
4. **Inconsistency estimator.** At the end of the episode, the agent computes the inconsistency score for the default model D by comparing the expected state trajectory (obtained by simulating plan π) and the observed execution trace (trajectory τ).
5. **Model repair.** If the inconsistency score exceeds a set threshold, the search-based model repair is engaged to adjust the agent's internal model D. The mechanism searches trough different candidate repairs to find one which sufficiently reduces inconsistency. The result is an updated model D' which aligns with the real-world model D^* such that the inconsistency score falls below the threshold once again.
6. The agent moves on to the next episode and uses the updated internal model $D = D'$ to solve the subsequent tasks. The whole process repeats again from the start (1.).

For clarity, note that the PDDL+ domain file describes general structure and system dynamics, and only needs to be defined once per environment. On the other hand, the PDDL+ problem is auto-generated per episode, combining the initial state data obtained from the environment and assumptions about the domain (i.e., values for variables that are required to accurately describe the environment and its dynamics, but which are not included in the data from the environment, e.g., gravity, velocities of moving entities).

Estimating Inconsistency

Inconsistency metric $C \in \mathbb{R}_{\geq 0}$ is a measure of how accurately the planning model D describes the actual environment and its dynamics D^*. It is computed using two aligned trajectories. One is obtained by simulating[2] the generated plan π and the PDDL+ model D, and the other trajectory τ is generated by the environment upon executing the plan π. Inconsistency is the sum of Euclidean distances between pairs of corresponding states in simulated and observed trajectories discounted proportionally by time.

Formally, let $S(\tau)$ be the sequence of states in the observed trajectory and $S(\pi, D)$ be the expected sequence of states obtained by simulating the generated plan π with respect to the agent's internal PDDL+ model D. Let $S(x)[i]$ denote the i^{th} state in the state sequence $S(x)$. Inconsistency score of domain D is computed as:

$$C(\pi, D, \tau) = \sum_i \gamma^i \cdot ||S(\tau)[i] - S(\pi, D)[i]|| \qquad (1)$$

where $0 < \gamma < 1$ is the discount factor. Errors can arise due to sensing noise, rounding errors, and other issues and can accumulate over time. Consequently, the Euclidean distance between corresponding states is likely to be higher later in the trajectories. The discount factor γ prevents such errors from dominating the inconsistency estimation.

In a non-novel environment, the expected evolution of the system predicted by the planner should perfectly match the observed behavior in simulation, i.e., the two resulting trajectories should align by default with the inconsistency score $C = 0$. In the real world, however, this is usually impossible to achieve due to rounding errors, perception inaccuracies, and similar common issues. To account for such noise, a domain-specific consistency threshold T is set. A model

[2]Plan simulation can be done using dedicated plan simulators such as VAL (**?**). In our work, we wrote our own Python-based general-purpose PDDL+ simulator used across various domains.

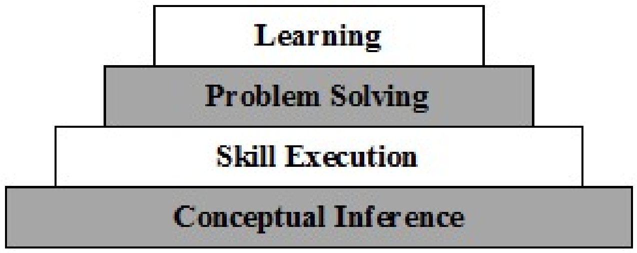

Figure 2: Graphical representation of inconsistency of the expected state trajectory obtained from the generated plan by using the agent's internal model D (purple nodes) with respect to the observations trajectory τ from the environment D^* (green nodes).

D is inconsistent with D^* if $C(\pi, D, \tau) \geq T$ and consistent otherwise. Setting T requires striking a fine balance between accurately estimating inconsistency and suppressing noise stemming from the execution environment.

A graphical representation of state trajectory-based inconsistency computation is shown in Figure 2 in which the expected state trajectory (purple nodes) is compared against the observed trajectory (green nodes). The gray dotted arrows denote the Euclidean distance between the observed state and the matching expected state generated by the planner using the agent's internal PDDL+ model D. The model D is deemed inconsistent if the inconsistency threshold is exceeded. Note that while the inconsistency threshold is fixed, in the figure the threshold line rises over time. This is equivalent to the discount being applied to individual inconsistency scores.

Search-Based Model Repair

The proposed search-based model repair algorithm works by searching for a *domain repair Φ*, which is a sequence of model modifications that, when applied to the agent's internal domain D, returns a domain D' that is consistent with the observed trajectories. To find such a domain repair, our algorithm accepts as input a set of all possible basic *Model Manipulation Operators* (MMOs), denoted $\{\varphi\} = \{\varphi_0, \varphi_1, ..., \varphi_n\}$. Each MMO $\varphi_i \in \{\varphi\}$ represents a possible change to the domain. Thus, a domain repair Φ is a sequence of one or more basic MMO $\varphi_i \in \{\varphi\}$. An example of an MMO is to add a fixed amount $\Delta \in \mathbb{R}$ to one of the numeric domain fluents. In general, one can define such an MMO for every domain fluent. In practice, however, not all domain fluents are equal in their importance. The proposed search-based model repair algorithm works by searching for a *domain repair Φ*, which is a sequence of model modifications that, when applied to the agent's internal domain D, returns a domain D' that is consistent with the observed trajectories. To find such a domain repair, our algorithm accepts as input a set of all possible basic *Model Manipulation Operators* (MMOs), denoted $\{\varphi\} = \{\varphi_0, \varphi_1, ..., \varphi_n\}$. Each MMO $\varphi_i \in \{\varphi\}$ represents a possible change to the domain. Thus, a domain repair Φ is a sequence of one or more basic MMO $\varphi_i \in \{\varphi\}$. An example of an MMO is to add a fixed amount $\Delta \in \mathbb{R}$ to one of the numeric domain fluents. In general, one can define such an MMO for every domain fluent. In practice, however, not all domain fluents are equal in their importance.

Algorithm 1: PDDL+ model repair algorithm.

```
Input  : $\{\varphi\}$, a set of basic MMOs
Input  : $D$, the original PDDL+ domain
Input  : $\pi$, plan generated using $D$
Input  : $\tau$, a trajectory
Input  : $T$, consistency threshold
Output: $\Phi_{best}$, a domain repair for $D$
1  OPEN$\leftarrow \{\emptyset\}$; $C_{best} \leftarrow \infty$; $\varphi_{best} \leftarrow \emptyset$
2  while $C_{best} \geq T$ do
3      $\Phi \leftarrow$ pop from OPEN
4      foreach $\varphi_i \in \{\varphi\}$ do
5          $\Phi' \leftarrow \Phi \cup \varphi_i$ ; /* Compose a domain repair */
6          DoRepair($\Phi'$, $D$)
7          $C_{\Phi'} \leftarrow$ InconsistencyEstimator($\pi$, $D$, $\tau$)
8          if $C_{\Phi} \leq C_{best}$ then
9              $C_{best} \leftarrow C_{\Phi'}$
10             $\Phi_{best} \leftarrow \Phi'$
11         Insert $\Phi'$ to OPEN with key $f(\Phi', C_{\Phi'})$
12         UndoRepair($\Phi'$, $D$)
13 return $\Phi_{best}$
```

Algorithm 1 lists the pseudo-code for our search-based model repair algorithm. Initially, the open list ($OPEN$) includes a single node representing the empty repair, and the best repair seen so far Φ_{best} is initialized to an empty sequence. This corresponds to not repairing the agent's internal domain at all. Then, in every iteration the best repair in OPEN is popped, and we compose new repairs by adding a single MMO to this repair, and add them to $OPEN$. For every such candidate repair Φ' we compute a consistency score $C_{\Phi'}$. This is done by modifying the agent's internal domain D with repair Φ', simulating the actions in plan π, and measuring the difference between the simulated outcome of these actions and the observed trajectory τ. The consistency score $C_{\Phi'}$ serves two purposes. First, we keep track of the best repair generated so far, and return it when the search halts. Second, we consider a repair's consistency score when choosing which repair to pop from $OPEN$ in each iteration. This is embodied in the function $f(\Phi', C_{\Phi'})$ in line 11 in Algorithm 1. In our implementation, f is a linear combination of the consistency score the size of the repair, i.e., the number of MMOs it is composed of. The latter consideration biases the search towards simpler repairs.

Figure visualizes an example search tree of the MMO-based model repair algorithm where MMOs are treated as actions and the state is composed of changes to the default model (0 indicates no change to the given fluent). Inconsistency is estimated for each generated repair, and the search terminates once a repair is found such that the updated domain D' is consistent with the true domain D^*. MMO repair is performed on a set of repairable fluents $\{x_1, x_2, ..., x_n\}$. Each MMO adjusts the value of a given fluent by a fixed amount Δ ($+\Delta$ or $-\Delta$), defined *a priori* per fluent (denoted

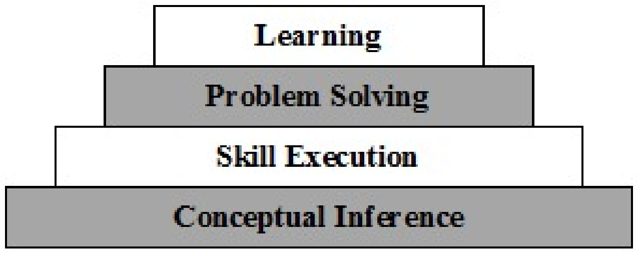

Figure 3: An example MMO search graph showing how a repair (i.e., a sequence of MMOs) is selected. The green node depicts a goal state in the MMO repair search, the minimal change to the model that causes the inconsistency to fall below the set threshold.

as delta in top left of Figure. 3). In this example scenario, the best repair is a sequence of three MMOs (each adjusting x_2 by $+0.2$) such that repair φ_{best} changes a single state variable $x_2 \in X$ by adding 0.6 to its value.

Experimental Evaluation

In this section, we evaluate the efficacy of model repair in evolving environments and demonstrate its capability to improve an inaccurate model. We report results from two domains: CartPole and Angry Birds. CartPole is a well-known classical control and RL benchmark. We used the standard CartPole implementation provided by the OpenAI Gym[3]. Angry Birds is a popular mobile game that has featured as a domain in IJCAI competitions[4] since 2013.

Experiment 1: CartPole

Our first set of experiments is designed to evaluate if the proposed model repair method can enable a planning agent to adapt to environment novelties. Recall that a novelty implies a sudden, unknown, and unexpected change in environment dynamics. In the CartPole domain, the agent's task is to balance the pole in the upright position for $n = 200$ steps by pushing the cart either left or right. The domain's system dynamics are defined by several parameters: mass of the cart, mass of the pole, length of the pole, gravity, angle limit, cart limit, push force. The CartPole environment only provides information on the velocities and positions of the cart and the pole (4-tuple). The agent has two actions, left and right, that apply a constant force. Every step returns a $+1$ reward unless the pole diverges from vertical by $\pm 12°$, at which point the episode ends in a loss.

We structured our experiments as a set of $t = 5$ trials consisting of $l = 200$ episodes each. Each trial begins with episodes in nominal conditions. Novelty is introduced after $k = 7$ canonical episodes where the environmental parameters are changed without informing the agent. Reward obtained by the agent in each episode is recorded.

Agents We compared model-free reinforcement learning (RL) and model-based planning agents in our novelty experiments. Model-free RL agents were built with a standard deep Q-network implementation with experience replay memory (**?**). The Q-network is built with an input layer (4×16), a hidden layer (16×16), and an output layer (16×2) and uses the Rectified Linear Unit (ReLU) activation function. The Q-network was trained to achieve perfect performance in the canonical setup prior to the experiments. The *DQN-static* agent applies the policy learned in the canonical setup in the novelty setup. This baseline was implemented to ascertain that the introduced novelty indeed impacts performance of the agent and motivates adaptation. The *DQN-adaptive* agent updated its network weights and learned a policy for the novelty setup.

The model-based planning agents were built with PDDL+. The expressiveness of PDDL+ enabled us to holistically encode CartPole's system dynamics as a planning domain (without using any external functions/solvers). Despite inherent complexity, efficient modeling allowed the use of off-the-shelf PDDL+ planners to solve the resulting problems. The planners were required to handle non-linear dynamics, large numbers of active happenings, and long temporal horizons. In our work on CartPole and other domains, we have successfully tested UPMurphi (**?**) and ENHSP (**?**), but we ultimately opted for a novel discretization-based PDDL+ planner that allows greater flexibility and customization (though, it relies on the same principles for solving PDDL+ problems as the other two planners). The planner solves full CartPole PDDL+ problems in approx. 0.1 seconds using GBFS algorithm with a domain specific heuristic prioritizing "safe states (pole upright, low velocities, close to the center). *Planning-static* agent uses the same PDDL+ model for both the canonical and novelty setup. *Planning-adaptive* agent monitors plan execution and automatically repairs the PDDL+ model inconsistency is detected. *Repairable fluents* are all parameters defining CartPole dynamics: mass of the cart, mass of the pole, length of the pole, gravity, angle limit, cart limit, push force.

Results

We report our observations on two novelties: length of the pole is updated to 1.1, gravity to 12; and length of the pole to 1.1 and mass of the cart is decreased to 0.9. These novelties were selected because they impact agent performance for all agents significantly. The performance of of all agents is summarized in Figure 4. The columns delineate the learning v/s planning methods and the rows, their static and adaptive versions. In each graph, x-axis captures the episodes in a trial and the y-axis shows the total reward collected by the agent per episode, represented as the proportion of its score with a perfect controller. The red line indicates the episode where the selected novelty (shown in the graph title) was introduced. The shaded area represents the 95% confidence interval computed over 5 trials.

As shown in Figure 4, all agents demonstrate perfect performance at the beginning of the trial and then experience a significant drop in performance as soon as the novelty is introduced (episode 8). This drop demonstrates that the changes in the environment dynamics impacts the performance of all agents. There is variability in how each agent

[3]https://gym.openai.com

[4]http://aibirds.org/

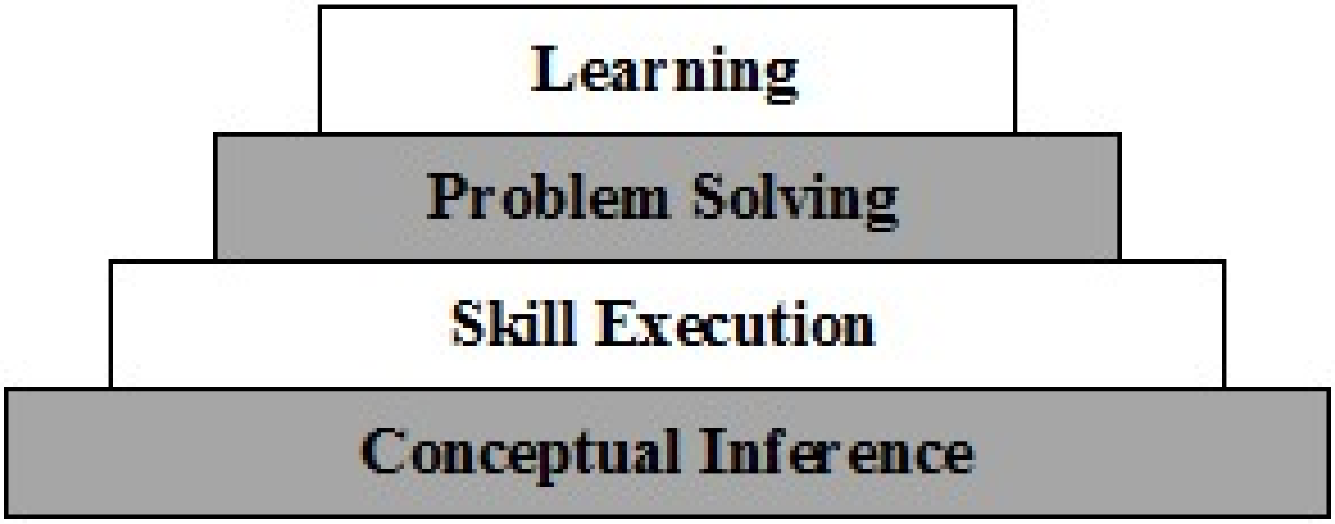

Figure 4: Graphs showing performance of DQN-static/adaptive and planning-static/repairing agents. Episodes in a trial are on the x-axis and reward earned is shown on the y-axis. The results are averaged over 5 trials. Red line indicates the episode where environment parameters were changed.

responds to the changes in the environment.

Resilience of model-based agents The novelty-induced performance drop is more significant in learning agents. For both types of novelties, the performance of the DQN agents drops below 30% of its value in the canonical setup. In contrast, the performance of the planning agents drop to $\approx 50\%$. This difference can be explained by the agents' design. The planning agents' PDDL+ model defines the system dynamics in a general manner. Thus, it can still be sufficiently accurate in some conditions. In contrast, the DQN agent's learned policy is not general and is only applicable for much reduced subset of cases after novelty.

Quick adaptation via model-space search As expected, after novelty is introduced, the static versions of the DQN and planning agents continue performing poorly, while the adaptive agents improve their performance over time. However, the time taken to improve differs greatly between the DQN-adaptive and planning-adaptive agents. Learning in DQN-adaptive is slow, requiring multiple interactions with the environment. In our experiments, DQN-adaptive took $\approx$ 75 episodes to reach 90% of optimal performance as shown in Appendix **??**, Fig. **??**. In contrast, the planning-adaptive agent recovers very quickly in $<$ 20 episodes for both novelties (Fig. 4). This observation supports our central thesis: model-space search enables quick adaptation in dynamic environments because it can localize changes in the explicit model. Repair-based adaptation is scalable and can handle very impactful novelty changes in system dynamics from the canonical setup (shown in Appendix **??**, Fig. **??**).

Explainable by design The model repair mechanism proposes concrete changes to the agent's explicit PDDL+ model. Thus, adaptation in the planning-adaptive agent is *explainable*. A model designer can inspect the proposed repair to understand why and how the novelty affected the agent's behavior. In contrast, learning in model-free systems such as DQN-adaptive cannot be interpreted directly.

The following text shows the distinct repairs that were found by the planning-repairing agent using the method proposed in this paper. Note that the reported values are changes from the nominal values used in the model.

```
Repair 1: mass_cart: 0, length_pole:
0.3, mass_pole: 0, force_mag: 0,
gravity: 0, angle_limit: 0, x_limit:
0
Repair 2: mass_cart: 0, length_pole:
0, mass_pole: 0, f The model-based
planning agents worce_mag: 0, gravity:
1.0, angle_limit: 0, x_limit: 0
Repair 3: mass_cart: 0, length_pole:
-0.1, mass_pole: -0.1, force_mag: -1.0,
gravity: 0, angle_limit: 0, x_limit: 0
```

Despite these repairs being different from each other, the agent's performance converged to optimal with each one. One potential reason is that there are equivalence classes in the set of environmental parameters and their values. The agent uses only its observations over one episode in the environment to guide its search. It is likely that the observations by themselves do not provide sufficient information to determine the parameter values exactly and only differentiate between equivalence classes.

Experiment 2

Next we evaluate if the proposed method can alleviate inaccuracies in domain modeling by using observed data from execution. This test was done in Angry Birds. The objective in Angry Birds is to destroy pigs by launching birds at them from a sling-shot (shown in Figure 5). The launched birds obey the laws of motion and gravity. It is a difficult problem

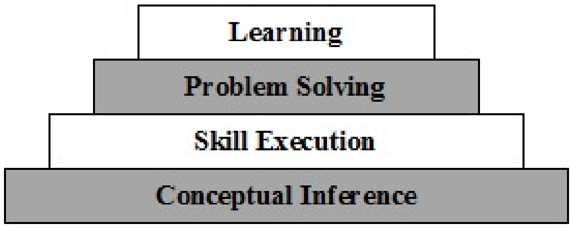

Figure 5: Angry Birds Level used in our experiment

as it requires predicting outcomes of physics-based actions without having complete knowledge of the world, and then selecting an appropriate action out of infinitely many possible options. An Angry Birds level is *passed* when all the pigs have been destroyed.

Our tests were conducted in a specific subset of Angry Birds levels containing a small number of birds, blocks, and pigs (a sample is shown in Figure 5). We implemented a competent *planning-static* agent using PDDL+. The PDDL+ model includes dynamics of launching a bird at maximum velocity at a certain angle, motion of birds and blocks, collisions between various objects, etc. The *planning-adaptive* agent engages repair of the PDDL+ model when its observations become inconsistent with its expectations. We conducted 11 trials of 25 levels each. To simulate inaccuracies in modeling that invariably creep in during model design phase, we deliberately encoded incorrect values for certain parameters in the agents. Specifically, we reduced the maximum bird velocity by 4 from its canonical value of $v_{bird} = 186$. We measured how frequently each agent passes levels in a trial or the *win rate*. We report the mean win rate for each agent and the 95% confidence interval.

The planning-static agent scored $0\%(0\%, 0\%)$ showing that the inaccuracy introduced in the model drastically degraded its performance. The planning-adaptive agent scored $52.36\%(28.72\%, 75.96\%)$ indicating that the proposed repair mechanism can alleviate inaccuracies in the agent's model and improve its performance significantly.

The results from CartPole and Angry Birds demonstrate that our proposed approach is applicable diverse scenarios. They highlight that not only the proposed approach can be used to develop robust planning agents that can handle sudden changes in the environment, it can also be used to improve accuracy of models of environment dynamics starting from rough approximations.

Conclusions and Future Work

Realistic dynamical environments require building planning models without perfect information or full observability of the target environment. Additionally, in such scenarios, unexpected novelty can be introduced, significantly impacting the environment's features and dynamics in unknown ways. Such novelties can render any existing planning models obsolete, resulting in plan execution failures.

In this paper, we presented a domain-independent approach to model repair using heuristic search which enables autonomous agents to reason with novelties and mitigate their impact on the agent's performance. The approach works to correct inaccuracies in the agent's internal PDDL model based by measuring inconsistencies between its own planned expectations and observations from the execution environment. A state-based search algorithm guided by an inconsistency-based heuristic searches through different combinations of model modifications to find a viable repair which accounts for the novelty interference. We demonstrated our approach on complex PDDL+ domains, proving its applicability to realistic applications. Additionally, the presented approach can be used to design more accurate PDDL models by helping to find exact values of environmental parameters starting from rough approximations.

To the best of our knowledge, this is the first attempt at model repair in state-based AI Planning, with previous works relating to plan execution failures choosing to focus largely on plan repair or replanning strategies.

Future work will concentrate on automatically defining the set of repairable fluents and corresponding deltas, and improving the accuracy of the inconsistency metric. In the next stage, we will extend model repair to include modifications to the structure of the PDDL domain by adding, removing, and modifying preconditions, effects, and entire happenings.

Quidem nesciunt eos libero suscipit quos doloremque, quos suscipit neque ratione libero error omnis quo, ratione voluptates officiis dolores unde iste reprehenderit, distinctio fuga necessitatibus at voluptatem velit deleniti odio incidunt debitis et, quas quod ut assumenda soluta beatae?Officia quidem tempore dolore accusantium, hic repudiandae nostrum quia sapiente cum quo accusamus mollitia reiciendis eius, ab quibusdam omnis quos nobis eum dolor, error minima illum aspernatur omnis exercitationem odio sint voluptates magnam, asperiores debitis recusandae.Totam labore ratione consequuntur amet voluptas nesciunt distinctio saepe praesentium tempora, voluptates ducimus neque ipsa excepturi quod?Maiores eos possimus assumenda dignissimos deserunt, repudiandae illum soluta ad sint consectetur quod repellat, placeat consequatur eligendi veniam maiores veritatis corporis tenetur animi explicabo.Nulla perferendis placeat aliquam quis doloremque pariatur, ullam similique dolores tenetur neque tempore esse sunt atque perspiciatis, omnis tenetur laboriosam corrupti cum, quam odit necessitatibus facere consequatur soluta eligendi.Ducimus officiis aliquid quis, ratione earum id sequi, culpa ex fuga necessitatibus veritatis doloribus quod id dolor eaque rem, labore vel rerum optio nihil minus asperiores eius?Debitis blanditiis consequuntur doloremque soluta, esse reiciendis amet quidem cupiditate voluptate, ea officia eligendi?Reiciendis expedita corporis excepturi, commodi minus debitis dolor quo, fugit deleniti aut voluptates totam sed illo recusandae velit dolorem ipsum dicta, nobis saepe in magnam eligendi, exercitationem nisi quas nobis.Architecto quae ab hic laudantium, eius quibusdam quaerat?