32nd International Conference on
Automated Planning and Scheduling
June 13 - 24, 2021, Singapore (virtual)

FinPlan 2022

Preprints of the 3rd Workshop on
Planning for
Financial Services (FinPlan)

Edited by:

Alberto Pozanco, Shirin Sohrabi, Parisa Zehtabi

Organization

Alberto Pozanco
J.P. Morgan AI Research, Spain

Shirin Sohrabi
IBM Research

Parisa Zehtabi
J.P. Morgan AI Research, UK

Program Committee

Daniel Borrajo (J.P. Morgan AI Research, Spain)
Giuseppe De Giacomo (Sapienza Università di Roma, Italy)
Mark Feblowitz (IBM, USA)
Fernando Fernández (Universidad Carlos III de Madrid, Spain)
Senka Krivic (University of Sarajevo, Bosnia and Herzegovina)
Alberto Pozanco (J.P. Morgan AI Research, Spain)
Rui Silva (J.P. Morgan AI Research, UK)
Shirin Sohrabi (IBM, USA)
Parisa Zehtabi (J.P. Morgan AI Research, UK)

Foreword

Planning is becoming a mature field in terms of base techniques and algorithms to solve goal- oriented tasks. It has been successfully applied to many domains including classical domains such as logistics or mars rovers, or more recently in oil and gas, as well as mining industry. However, very little work has been done in relation to financial institutions problems. Recently, some big financial corporations have started AI research labs and researchers at those teams have found there are plenty of open planning problems to be tackled by the planning community. For example, these include, trading markets, workflow learning, generation and execution, transactions flow understanding, risk management, fraud detection and customer journeys.

FinPlan'22 is the third workshop on Planning for Financial Services held in conjunction with ICAPS, whose aim is to bring together researchers and practitioners to discuss challenges for Planning in Financial Services, and the opportunities such challenges represent to the planning research community.

Alberto Pozanco, Shirin Sohrabi, Parisa Zehtabi, June 2022

Contents

Defending Against Adversarial Attacks on Policies Through Density Estimation

Alberto Villanueva[1], **Marcos Villacañas**[1], **Rubén Majadas**[1], **Javier García**[2], **Fernando Fernández**[1]

[1] Departamento de Informática, Universidad Carlos III de Madrid
[2] Departamento de Electrónica y Computación, Universidad de Santiago de Compostela
alvillan@inf.uc3m.es, mvillaca@pa.uc3m.es, rmajadas@pa.uc3m.es, franciscojavier.garcia.polo@usc.es, ffernand@inf.uc3m.es

Abstract

In recent years, reinforcement learning (RL) and, in particular, its "deep" variant, has been applied to tasks in the real world gradually. RL has shown unprecedented popularity, such as autonomous driving, robot control, solving complex video games. It was a matter of time before deep RL burst into finance and trading as well. Financial markets are simply too complex for non learning-based algorithms, as the state and action spaces are continuously expanding every second. With RL, however, we can learn autonomously complex trading strategies that can maximize profits in spite of the highly stochastic and non-stationary nature of financial markets. But if the field of finance and trading is benefiting from the power of deep RL algorithms, it has also inherited its vulnerabilities. Deep RL algorithms are well-known to be inherently vulnerable to manipulation by intentionally perturbed observations, rewards or actions, leading to unintended and potentially harmful results. This is particularly relevant in a financial context, where exploiting these vulnerabilities provides adversaries with the means to lead a company to millionaire losses. For this reason, in this paper we investigate a two-step defense mechanism not only able to detect these adversarial attacks, but also to recover from them. We show that our approach manages to achieve a nearly perfect defense in simple domains, and is proficient against several state-of-the-art attacking strategies.

Introduction

There has been an upward trend in recent years to use reinforcement learning (RL) in financial markets (Fischer 2018) and, as RL policies get applied to real world environments, a focus has to be placed on ensuring these policies are resilient against malign actors trying to interfere with the system. A perfect example of this is High Frequency Trading (HFT), a trading strategy which uses high speed algorithms in order to benefit from arbitrage opportunities. This way of operating constantly scans the Limit Order Book status, seeking to find mismatches of any size. As a result, hundreds of trades are closed every second placing value on, not only momentum, but also reliability. Recently, RL has gained popularity among policy training methods for HFT strategies (Briola et al. 2021). Thus, the motivation of this research is clear. Fighting adversarial attacks would help to prevent millionaire loses in this field. Recent advancements in adversarial machine learning have shown that agents based in neural networks are particularly sensitive to malignantly crafted small perturbations (Szegedy et al. 2014). These attacks have usually been applied to supervised learning tasks. However, they have also shown to be proficient when tackling RL policies (Kos and Song 2017).

In our work, we propose a two-step based defense system. First, we use experience tuple density estimation to identify when a new state has been perturbed. By using a non-supervised method, we avoid modeling specific attacking patterns, which aids in generalizing against any possible attack. Then, if a perturbation is detected in the state, the original state is recovered by using a k-nearest-neighbor approach on the experience tuple space. In contrast to previous defense methods, our approach avoids the use of neural networks, as defense systems based on neural networks have also shown to be sensitive to malignantly crafted perturbations (Carlini and Wagner 2017a). We show that our approach manages to achieve a nearly perfect defense in simple domains, and is proficient against several state-of-the-art attacking strategies.

Background

In this section, the concepts of RL and adversarial machine learning are reviewed.

Reinforcement Learning

RL environments are typically formalized by means of a Markov Decision Process (MDP) (Sutton and Barto 2018), which is a mathematical model of sequential decisions and a dynamic optimization method. It is represented by a tuple $< S, A, T, R >$ where *S* is the set of possible states, *A* is the set of possible actions available from each state, *T : S x A x S' → [0,1]* is the transition function where *T(s,a,s')* represents the probability of getting to state *s'* from state *s* when action *a* is performed, and $R : A \times S \rightarrow \Re$ is the reward function which maps a state *s* and an action *a* into a reward *R(s,a)*; *r* is used hereinafter to represent the stochastic reward result obtained from a distribution with mean *R(s,a)*.

In a Markov decision-making process, the transition probability and reward only depend on the current state and the action chosen. As a result, the objective is learning a policy π for every state to maximize the return of $J(\pi)$:

$$J(\pi) = \sum_{k=0}^{K} \gamma^k r_k \tag{1}$$

where γ affects how much future is considered from step *k* (discount factor, with $0 \leq \gamma \leq 1$) and r_k corresponds to reward received in step *k*. At this point, the value-function, which estimates the sum of rewards given a policy π, can be solved using Bellman's equation.

In Deep Q-Learning (Mnih, Kavukcuoglu, and Davi 2013), neural networks are used to estimate the action-value function, and the value over which the loss function takes place for the learning process, corresponds to the squared Bellman error, that is, the difference between the expected value and the predicted value.

$$\mathbb{E}_{s,a}[(r + \gamma \max_{a'} Q(s', a') - Q(s, a))^2] \tag{2}$$

Along to this, the learning takes place in a batch fashion, where instead of learning over one experience tuple at a time, the agent holds a memory of the last N experience tuples and, at every learning step, a sample is taken over the experience tuples in memory. These agents built using Neural Network take the name of Deep Q Networks (DQN) (Mnih, Kavukcuoglu, and Davi 2013).

Adversarial Machine Learning

In adversarial machine learning, adversarial examples are maliciously generated such that the machine learning model fails its task. Particularly, models based on neural networks have been shown to be specially vulnerable to such attacks.

These attacks can be classified as poisoning attacks, where adversarial examples are injected into the training phase, and exploratory attacks, where the adversarial examples are injected into the testing phase (Sethi and Kantardzic 2018). Furthermore, the previous classification can be enriched by taking into consideration a complementary perspective of the offensive strategy. Depending on whether the adversary has access to the model parameters, architecture or training data, or not, attacks are considered to be white-box or black-box, respectively. Even though having total knowledge to model's internal data for white-box attacks is game-changing capability, it is not common under real scenarios circumstances. From a realistic point of view, trying to exploit the model's vulnerabilities based on input/output pairs exclusively, as it is on black-box attacks, is closer to a down-to-earth approach. On adversarial machine learning tasks, the only thing that needs to be defined is how to create the adversarial example, however in adversarial RL, also it has to be decided when to create an adversarial example.

Adversarial example crafting Crafting an adversarial example x^δ for a model f from a given input x can be described as the optimization problem described in Equation 3, where l and l^δ are the labels of x and x^δ respectively and $\|\cdot\|$ denotes the p-norm distance. Here the perturbation can be defined as $\eta = x^\delta - x$ and the optimization problem can be described as finding the lowest perturbation η that manages to make the model f make a mistake when evaluating an instance x. In the area of classification, making a mistake would be equivalent to predicting the wrong label for x, and in the area of regression to predicting a value as far away as possible from the real value.

$$\begin{aligned} \min \quad & \left\|x^\delta - x\right\|_p \\ s.t. \quad & f(x^\delta) = l^\delta \\ & f(x) = l \\ & l \neq l^\delta \\ & x \in [0, 1] \end{aligned} \tag{3}$$

Adversarial attacks on policies Regarding RL in particular, with the intention of contemplating different approaches, the spotlight has been put on three recently proposed strategies which have been chosen to be tested against detection and recovery techniques presented hereunder.

- **Uniform attack**: this adversarial behavior consists on perturbing each state the agent observes, that is to say, attacking at every time step adding some noise (Huang et al. 2017). The perturbation injected can be created using FGSM (Szegedy et al. 2014) where the cost function needed to calculate the gradients $J(\theta, x, y)$ is the cross-entropy loss between y, which is the weighting over the possible actions, and the distribution that places all weight on the highest weighted action in y. (Huang et al. 2017). On the subject of our work, in order to approximate better to a real-life situation, it has been decided to not attack continually.
- **VF attack**: this attack injects a perturbation when the value function is greater than some threshold β. The reasoning is to perturb the agent on some crucial moments when it is close to reaching a reward. This can be seen as described in Equation 4. (Kos and Song 2017). Just like in uniform attack, to craft the adversarial examples, FGSM can be used (Szegedy et al. 2014).

$$\max_{a \in A} Q(s_t, a) > \beta \tag{4}$$

- **Strategically-Timed attack**: in this strategy, a perturbation is only incorporated to a normal state when there is a strong preference for an action over another one. Consequently, this attack inflicts more harm at the time the trained agent would strive to take a key action (Lin et al. 2019). The attack trigger is also set using a beta parameter, thus only when the difference between the highest and lowest Q values[1] exceeds beta, the attack will take place as can be seen in Equation 5. In addition, to avoid attacking excessively, a second parameter controls the maximum amount of times the agent can suffer an attack (Lin et al. 2019). To craft adversarial examples, it looks for an observation that can change the most preferred action to the least preferred one by using the Carlini & Wagner attack (Carlini and Wagner 2016).

$$\max_{a \in A} Q(s_t, a) - \min_{a \in A} Q(s_t, a) > \beta \tag{5}$$

[1] In the original work the difference between the preference to take an action is used. For policy gradient algorithms that is the probability to take an action and for value based methods it is the softmax function of the Q values

VF and Strategically-Timed attacks are white-box attacks, hence access to agent's Q values is required, while uniform attack is a black-box attack. However, it has been shown that, although less effective, an adversarial example can be transferred from one agent to another which could make white-box attacks work in a black-box environment (Huang et al. 2017).

Related work

To countermeasure these adversarial attacks, several defensive strategies have been proposed which can be divided into two distinct groups, proactive and reactive (Yuan et al. 2019). Proactive strategies protect the model from being affected from adversarial attacks before the model has been attacked. These include techniques such as adversarial training (Wu, Bamman, and Russell 2017; Dong et al. 2017; Goodfellow, Shlens, and Szegedy 2015; Tramèr et al. 2020; Huang et al. 2016), classifier robustifying (Bradshaw, de G. Matthews, and Ghahramani 2017; Abbasi and Gagné 2017), and network destilation (Papernot et al. 2016). Inside reactive strategies, two main types are discussed, adversarial detecting, in which the objective is to identify in test which of the given instances are adversarial, and input reconstruction, in which the objective is to reconstruct an adversarial example to the original example without the added noise so that the model can work normally with the clean example.

A plethora of different approaches have been taken to detect these adversarial attacks in the testing stage. A binary classifier sub-network can be trained with adversarial examples to distinguish them from the clean ones (Metzen et al. 2017; Gong, Wang, and Ku 2017) and another way is to add an additional output class to the networks output that corresponds to adversarial inputs (Grosse et al. 2017). Safety-net (Lu, Issaranon, and Forsyth 2017) uses an RDF-SVM that utilizes discrete codes computed from late stage ReLUs to detect adversarial examples, however similarly to the previous methods, it also requires to be trained with adversarial examples, which means that it has to model the attacker which is unlikely to generalize well to other processes to generate adversarial examples (Meng and Chen 2017).

Mag-Net (Meng and Chen 2017) uses an ensemble of 2 detectors, one is an autoencoder trained on the original clean dataset that predicts whether it is an adversarial example or not based on the reconstruction error, where an error higher than a threshold indicates an adversarial example. This performs well when the error is high enough but for smaller errors a second detector was added that measures the divergence between the logit of the input example and that of the autoencoded example, where a high divergence indicates an adversarial example, and finally, to ensemble them an adversarial attack is reported if any of the two detectors detect one.

Principal component analysis (PCA) whitening can be used on the input examples, and since adversarial ones have different coefficients for low-ranked principal components, a detector can be created from it (Hendrycks and Gimpel 2017). Furthermore, the authors say that a combination of detectors might be a better way to try to deal with adversarial detection.

A detector can also be built using a logistic regressor using as inputs the kernel density of the input example, calculated using the training set on the feature space of the last hidden layer, and the Bayesian uncertainty estimate of the network for said example (Feinman et al. 2017).

However, Carlini & Wagner (2017a) showed that most of these defenses are not as effective as previously analyzed, as they are still susceptible to their previous attack (Carlini and Wagner 2016) when the loss function is changed. Furthermore, they offer some guidelines on how to better approach detection defenses; such as using strong attacks for evaluation instead of single iteration ones, and that it should be resistant against white-box attacks too and not only gray or black box ones. They also showed the methods that performed the worst were the ones using another neural network for detection, because if an adversarial example can be made to fool one network, there can also be one that fools both (Carlini and Wagner 2017a,b).

Input reconstruction tries to transform the input data when it has been detected as having been attacked such that the output of the reconstructed input matches that of the unattacked one. Autoencoders can be used to perform input reconstruction (Meng and Chen 2017; Gu and Rigazio 2015) by learning from the training data. Another way is the approach taken by pixel defend (Song et al. 2017) where the training distribution probability of the novel example is estimated by using PixelCNN (van den Oord, Kalchbrenner, and Kavukcuoglu 2016) and then, a new value is generated for each pixel along each channel such that the probability distribution estimated of the new example is maximized and the difference between the original value and the new value is smaller than some value ϵ moving it closer towards the training dataset.

In the field of RL, on previous work on defensive mechanisms, the current state is predicted with the previous m states and m actions with which the result is then used to compare the states Q values with the received state Q values, where a high discrepancy indicates an adversarial attack has been produced, since the objective of the adversarial attack is to change the policy of the agent. Furthermore, an optimal action can be suggested based on the Q values of the predicted state and thus reconstruct the policy for the state (Lin et al. 2017).

Architecture - Proposal

The architecture for our defense system consists of two modules, the detection system, which detects if a state has been attacked, and the recovery system, which once a state has been detected as an attacked one, tries to recover the original values from the state. Both systems work by inferring knowledge over the experience tuple instead of just over the state to take into account the sequential nature of RL tasks, so after an action a is performed over the state s, the detection system analyzes the experience tuple $< s, a, s', r >$ to detect if the next state s' has been attacked or not.

To achieve this, both systems use experience tuples in the form of $\tau = < s_0, s_1, ..., s_n, \ a_0, a_1, ..., a_m, s'_0, s'_1, ..., s'_n, r >$, where s_i is the i-th feature of the state,

and a_i is the i-th feature of the action, where categorical actions or state features are one-hot encoded so that distance measures can be used, that are extracted by observing the agent's regular behavior on the testing phase and storing its experience tuples in the form of $\Gamma = \{\tau_0, \tau_1, ..., \tau_n\}$. To calculate similarity measures between any two experience tuples, ℓ_2 distance is used.

The overall process can be described as a number of steps where first the agent is trained (i), then Γ is extracted (ii) with which, a detection system (iii) and a recovery system (iv) are trained. During execution, the systems works as shown in Figure 1 where each new transition is given to the detector which predicts if it belongs to the distribution of Γ and, if it doesn't, it is given to the recovery system, which returns the reconstructed new state.

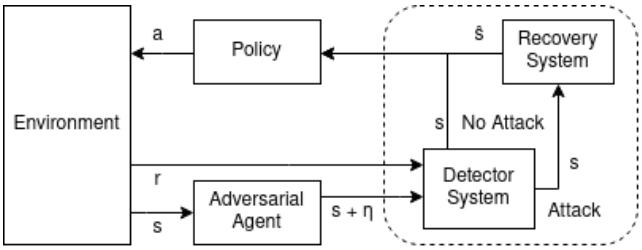

Figure 1: Defense Architecture Diagram

Detection System

When it comes to detecting an adversarial transition, our method compares the new perceived transition against the transitions in Γ, and if it deviates far enough from them, it is classified as an adversarial transition. This task can be seen as detecting outliers (adversarial examples) or estimating the underlying density function (DF) of Γ and labeling as adversarial transitions those with a low DF.

A number of different algorithms are used to estimate the DF, none of which use a Neural Network as the one used in pixel defense (Song et al. 2017) or in Visual Foresight (Lin et al. 2017), since Carlini and Wagner (2017a) found that defenses that used a Neural Network to defend from adversarial attacks happened to be the ones who performed the poorest as an attack could be designed to circumvent both neural networks, the predictor and the defender:

- **Kernel density** with a Gaussian kernel is used to estimate the underlying probability DF.
- **DBSCAN** (Ester et al. 1996) tries to identify the outliers in a set of data, however, in our case we already know a set of non-outlier data points, that being Γ, so instead we only calculate whether the new transition is an outlier or not in respect to Γ. Furthermore, instead of using a minimum number of samples needed inside the given radius, the number of samples inside the radius is used as the DF value.
- **KNN density estimation** is used to estimate the underlying DF, however, two different implementations of density estimation using KNN have been used:
  - The distance to the k nearest neighbors is taken, and a function is applied to them that returns a scalar such as the sum, the mean or maximum (which would simply be the distance to the k-th nearest neighbor).
  - The density is built according to the formula $\frac{n(x,a)}{NV(B(x,a))}$ where $B(x, a)$ is the hypersphere centered in x with radius a, $V(\cdot)$ represents a volume, N is the number of samples and $n(x, a)$ is the number of points within $B(x, a)$ (Zhao and Lai 2020). From now on, this one will be referred to as Hyper-KNN.
- **Gaussian Mixture** (Dempster, Laird, and Rubin 1977) estimates the underlying DF by fitting a mixture of Gaussian distributions to Γ and later using them to find the mixture probability.
- **K-Means** (Hartigan 1975) can be seen as a simplified version of Gaussian mixture, where the covariance matrix is fixed to being a scalar matrix, and the scalar controls the size of the cluster. The density metric used here is the inverse of the distance to the nearest centroid, as a higher number should correspond to a higher density. Furthermore, from this base DF two different approaches are taken:
  - No further changes are applied, and the DF depends solely on the distance to the nearest centroid. From now on this will be referred to as k-means global.
  - The DF is normalized with the distance to the furthest training sample assigned to the cluster the DF is based on. This makes it so that the DF takes into account contextual information from the cluster it is using, as it now depends on the size of the cluster.From now on this will be referred to as k-means local.
- **One class SVM** according to Schölkopf (Schölkopf et al. 2000) is used to establish a binary detection system. If the new transition falls between the origin and the calculated hyperplane, the sample is considered to be an adversarial transition.

Apart from one class SVM, every other method estimates a DF, but that is not enough to distinguish between adversarial and normal transitions, thus, a threshold has to be selected after which the transition is classified as adversarial. For that, the lowest DF value predicted from all transitions in Γ is used; that way, any new experience tuples with a lower DF than any in Γ is classified as an adversarial one. By selecting the decision threshold solely on normal experience tuples, it avoids the need to model any type of attacks, which ensures that it can not overfit the detection for that given attack.

Recovery System

The recovery system tries to increase the DF of the experience tuple that has been identified as an adversarial one, to do so, we once again step away from methods that use Neural Networks such as in Pixel Defense (Song et al. 2017) or visual foresight (Lin et al. 2017) for the same reasons as the ones described in the detection system, and so we propose a method that is independent of the detection system

and the learning agent, that repairs the next state by using the next state of the k nearest transitions in Γ, and weighting them by the distance to the original transition, where a k of 1 means that it takes the entire next state of the nearest neighbor. However, since the next state could be too corrupted due to adversarial injection, the recovery system calculated the distances in Γ', a subspace of Γ that does not include the next state. In order for the distances to have an actual meaning, the transition vector features have to be normalized, and discrete features have to be one-hot encoded if the set containing of all the possible values of the feature is not totally ordered.

$$\hat{s}' = \sum_{i}^{k} w_i s'_i \tag{6}$$

$$w_i = \frac{\|\tau' - \tau'_i\|_2}{\sum_{j}^{k} \left\|\tau' - \tau'_j\right\|_2}$$

Evaluation

This section evaluates the performance of the proposed defense in four different well-known domains from the OpenAI gym library: Acrobot, Cart pole, Mountain car and Taxi. But before presenting the results, the experimental setting is introduced.

Experimental setting

The proposed domains have been previously solved using tabular Q-Learning with an ϵ-greedy exploration-exploitation strategy. The agents are discretized along each dimension within the state limits in a number of bins[2], and are trained during H episodes with a limit of K steps per episode with the learning rate (α), and discount rate (γ) described in Table 2. Each of the training process will generate a policy π which will then be attacked and defended with the proposed system.

After the agents are trained, Γ is extracted (the algorithm Ball Tree (Dolatshah, Hadian, and Minaei-Bidgoli 2015) is used for efficient spacial indexing in the detector and recovery systems) from each agent by running them 500 episodes.

However, if a single adversarial example bypasses the defense system, it could throw off the trained policy off of the optimal path, which, if it is the only path the defense system has seen, would make it seem like every subsequent transition is adversarial (as it could be vastly different from what it has seen before). It is for this reason that instead of using the learned policy in a fully greedy way, an ϵ-greedy policy is used ($\epsilon = 0.1$) to prevent the transition sample from overfitting to the optimal path from the learned policy. That way, it accounts for slight deviations from the optimal path.

[2]These bin sizes are used to take into account the entire possible state space, however a lot of bins in Acrobot, Cart Pole and Mountain Car are empty as the states they describe are either unreachable, or so unlikely they are never visited. The number of bins where the learning actually takes place in is 14.3K, 13.7K and 7k for Acrobot, Cart Pole and Mountain Car respectively

For the adversarial perturbations, noise is generated at each dimension with a random value between 0 and $\frac{\delta}{\sqrt{d}}$, where d is the number of dimensions and δ is as shown in Table 1 for Acrobot, Cart Pole and Mountain Car, and then, the noise is added to the state normalized across the dimension limits such that $\| \eta \|_2 \leq \delta$. In the case of the Taxi environment, as the state space is discrete, the position of the agent is changed by 1 in the y or x coordinate. Only the position of the agent is attacked, as the other values, the passenger position and destination position, are given in the form of the index of a list that contain a set of possible positions, hence, a change of 1 in the index corresponds to a change of more than one in the actual positions, e.g. a change from 0 to 1 in the passenger position dimension (p) changes the position it is referring to, from (0,0) to (0, 4).

For the attacking strategies, three have been used; Uniform, Value Function and Strategically Timed attacks as described previously, whose parameters have been fine-tuned to achieve a compromise between the noise they inject and the damage they create, and are shown in Table 1.

Environment	δ	Uniform Frequency	VF β	ST β	max
Acrobot	0.10	0.5	-27	1.300	150
Cart Pole	0.10	0.5	40	0.800	50
Mountain Car	0.14	0.5	-15	1.300	50
Taxi	1.00	0.5	19	10.185	12

Table 1: Attack parameters

Results

Foremost, in Table 3 we showcase the average performance obtained by each agent across 200 episodes, how each attack affects the performance, and how much total perturbation (δ) it injects into an episode to achieve that loss in performance. The total perturbation injected is calculated by summing the ℓ_2 distance from the original state to the attacked state for each step in the episode. Here it can be seen how in the Cart Pole and Mountain Car environments, VF-attack and ST-attack achieve an equal or superior performance than Uniform attack while injecting much less perturbation. In the Taxi and Acrobot environments, ST-attack achieves a better performance with a similar or lower perturbation, but VF-attack achieves a higher performance at the cost of a lot more total perturbation. This is because with VF-attack, the states closer to the reward are attacked which prevents the agent from finishing the episode, and it enters a loop where it is constantly attacking the agent as the agent is always near the end of the episode but never really ending except if it performs the adequate action by pure chance, which explains the high variance.

Then Γ is normalized in one of two different ways, either using min max feature scaling normalization or z-score normalization. Afterwards, the detectors are trained using the parameters shown in Table 4. They are then evaluated by running the agent against each attack during 100 episodes

Domain	State	Bins	α	γ	ϵ	H	K
Acrobot	$\theta_1 \in [-\pi, \pi]$ $\theta_2 \in [-\pi, \pi]$ $\omega_{\theta_1} \in [-4\pi, 4\pi]$ $\omega_{\theta_2} \in [-9\pi, 9\pi]$	20 20 20 20	0.1	0.99	0.5	20000	500
Cart Pole	$p \in [-4.8, 4.8]$ $v \in [-4, 4]$ $\phi \in [-0.418, 0.418]$ $\kappa \in [-4, 4]$	10 50 50 50	0.1	0.99	0.3	20000	200
Mountain Car	$p \in [-1.2, 0.6]$ $v \in [-0.007, 0.007]$	100 100	0.1	0.99	0.3	300000	200
Taxi	$y \in \{0, 1, 2, 3, 4\}$ $x \in \{0, 1, 2, 3, 4\}$ $p \in \{0, 1, 2, 3, 4\}$ $d \in \{0, 1, 2, 3\}$	5 5 5 4	0.1	0.99	0.3	20000	200

Table 2: Train parameters for the victim policies

	No attack	Uniform	VF	ST
Acrobot	-140.96 $\pm$ 34.00	-171.13 $\pm$ 40.53 δ = 109.14 $\pm$ 28.07	-178.21 $\pm$ 52.59 δ = 131.28 $\pm$ 65.07	-180.20 $\pm$ 42.65 δ = 64.28 $\pm$ 18.43
Cart Pole	200.00 $\pm$ 0.00	116.74 $\pm$ 53.98 δ = 43.49 $\pm$ 19.13	50.77 $\pm$ 22.24 δ = 20.40 $\pm$ 11.79	52.80 $\pm$ 41.76 δ = 22.32 $\pm$ 9.47
Mountain Car	-98.98 $\pm$ 7.60	-149.87 $\pm$ 21.64 δ = 13.35 $\pm$ 2.40	-188.70 $\pm$ 23.42 δ = 9.27 $\pm$ 2.02	-154.43 $\pm$ 26.57 δ = 5.71 $\pm$ 2.39
Taxi	7.80 $\pm$ 2.56	-2.28 $\pm$ 9.92 δ = 9.95 $\pm$ 4.74	-51.40 $\pm$ 90.47 δ = 54.36 $\pm$ 82.50	-11.12 $\pm$ 14.88 δ = 9.80 $\pm$ 3.09

Table 3: Trained policy with and without attacks, and the total perturbation (δ) injected into an episode measured by ℓ_2 distance

where at each time step, if the state is detected as being an adversarial one, the original state is restored. This is done to decouple the results from the detector and the recovery systems and see how each of them behave independent of the other. The balanced accuracy is shown in Table 5 highlighting results of over 90% accuracy.

Detector	Parameters
Kernel Density	h: Scott's Rule
DBSCAN	ϵ: Scott's Rule
KNN-Hyper	a: Scott's Rule
KNN	k: 3, 5, 10, 15, 20, 30, 40, 50, 75, 100 f: sum, mean, max
Gaussian Mixture	k: 256, 512, 1024, 2048
K-means Global	k: 256, 512, 1024, 2048, 4096
K-means Local	k: 256, 512, 1024, 2048, 4096
One Class SVM	ν: 0.01, 0.05, 0.1, 0.5

Table 4: Detector parameters, where values separated by commas mean that all of those were tried and the best was selected

As can be seen in Table 5, the type of normalization has a strong impact on the performance of the detectors based on the environment; for every environment except for Cart Pole, z-score normalization has higher results for almost every detector, but in Cart Pole, min max normalization performs significantly better, having almost every detector a perfect detection score. The main reason for this can be that in a normal execution of Cart Pole with a perfect policy, every state it visits is really similar as it tries to keep the cart to the center of the screen and the pole as vertical as possible; this means that by doing min max normalization, if a single new state has a value lower than 0 or greater than 1 it is a strong indication of an adversarial attack. In contrast, in other environments, the states are a lot more varied, hence, a z-score normalization helps distinguish the most common states than the more uncommon. Furthermore, three detectors shine above others; DBSCAN, KNN-Hyper and Gaussian Mixture having a performance of above 90% in the best normalization method for every environment, however, this is while having a perfect recovery, so the performance also has to be analyzed with the recovery system.

To this end, the full defense system is now tested the same way as the detection system. Thereforere, now, when a new perceived transition is classified as an adversarial one, it is given to the recovery system which then returns the recovered new state. This recovery system is tried with multiple values for k; 1, 3, 5, 10, 15, 20, 40 and 50, and the best result is reported, although the method is not very sensitive to the different k values. Then the final reward achieved by the defense system (r_D) is measured against the reward obtained by the unaltered victim policy (r), using as a baseline the attacked reward (r_A), both of which can be seen in Table 1. In order to not just take into account the mean but also the

	Acrobot		Cart Pole		Mountain Car		Taxi	
normalization	**min max**	**z-score**	**min max**	**z-score**	**min max**	**z-score**	**min max**	**z-score**
UniformAttack								
Kernel Density	0.50	0.53	**1.00**	0.83	0.50	0.51	0.51	0.51
DBSCAN	0.65	**0.97**	**1.00**	0.63	0.63	**1.00**	**0.99**	**1.00**
KNN-Hyper	0.65	**0.98**	**1.00**	0.64	0.62	**1.00**	**0.99**	**1.00**
KNN	0.50	0.50	**1.00**	**0.98**	**1.00**	**1.00**	0.52	0.50
Gaussian Mixture	**1.00**	**1.00**	**1.00**	0.67	**1.00**	**1.00**	**1.00**	**1.00**
k-means global	0.50	0.60	**1.00**	**0.91**	**1.00**	**1.00**	-	**1.00**
k-means local	0.51	0.60	0.82	0.84	0.78	0.76	-	**0.93**
SVM	0.53	0.58	**1.00**	0.78	0.56	0.62	0.56	0.56
VF-Attack								
Kernel Density	0.50	0.75	0.50	0.50	0.51	0.53	0.61	0.53
DBSCAN	0.77	**0.98**	**1.00**	0.50	0.68	**1.00**	**0.99**	**1.00**
KNN-Hyper	0.77	**0.98**	**1.00**	0.50	0.68	**1.00**	**0.99**	**1.00**
KNN	0.50	0.52	**1.00**	0.71	**1.00**	**1.00**	0.64	0.50
Gaussian Mixture	**1.00**	**1.00**	**1.00**	0.52	**1.00**	**1.00**	**0.99**	**1.00**
k-means global	0.50	0.68	**1.00**	0.58	**1.00**	**1.00**	-	**1.00**
k-means local	0.50	0.52	0.61	0.58	0.72	0.72	-	**1.00**
SVM	**0.91**	0.89	**1.00**	0.50	0.77	0.78	0.72	0.69
ST-Attack								
Kernel Density	0.50	0.56	**1.00**	0.87	0.50	0.51	0.60	0.54
DBSCAN	0.63	**0.99**	**1.00**	0.67	0.66	**1.00**	**1.00**	**1.00**
KNN-Hyper	0.64	**0.98**	**1.00**	0.65	0.65	**1.00**	**1.00**	**1.00**
KNN	0.50	0.50	**1.00**	**0.97**	**1.00**	**1.00**	0.63	0.50
Gaussian Mixture	**1.00**	**1.00**	**1.00**	0.72	**1.00**	**1.00**	**1.00**	**1.00**
k-means global	0.50	0.59	**1.00**	**0.90**	**1.00**	**1.00**	-	**1.00**
k-means local	0.51	0.54	0.84	0.81	0.84	0.85	-	**0.93**
SVM	0.56	0.67	**0.99**	0.78	0.50	0.61	0.63	0.61

Table 5: Total balanced accuracy (using the transitions of 100 episodes) for the best parameter of each detector, using either min max feature scaling, or z-score normalization

variance of the rewards, Welch's t-test is used to compare both r_D and r_A to r, and finally this can be used to measure how the defense system affected the relative performance in a scale from 0 to 1 with the formula $\frac{t_D - t_A}{-t_A}$ where t_D is the t-score of r_d and r, and t_A is the t-score of r_A and r. The values can of course fall below 0 or raise above 1, where below 0 means the defense system creates a performance worse than the attack it is defending against, and above 1 a performance greater than the unaltered victim policy. These results can be seen in Table 6 where a performance over 90% is highlighted.

From these results, several conclusions can be drawn. Foremost, it can be seen that once again a distinction can be seen between different types of normalization depending on the environment, and said distinction corresponds to the one described before (min max normalization working better for Cart Pole and z-score normalization for the rest). Furthermore, it can also be seen how Gaussian mixture's performance drops significantly (specially in Cart Pole) compared to its performance in the detection tests, as with a perfect recovery it was achieving some of the highest results across all environments. This indicates that it was overfitting to the unaltered policy transitions, and as soon as the recovery introduced a slight error, the model did not recognize it as being valid. This could be because of a high number of clusters compared to the number of significantly different transitions, and lowering said number could lead to a better generalization.

In the Acrobot environment, a big difference can be seen from the different attacks, as with VF-attack results even better than the ones with the unaltered policy are obtained, however the detection accuracy was on par with the other attacks. The reason for the big difference in the reward, is because in Acrobot when the arm is near then top of the screen (which is when VF performs attacks) the arm already has velocity and a perfect recovery is not needed to finish the episode. On the other attacks, DBSCAN achieves the best results with a 62% and 74% recovered relative reward in uniform and ST attacks respectively.

In the Cart Pole environment, the performance is fully recovered in uniform and ST attacks, however, unlike in Acrobot, the defense does not manage to recover the performance against VF-attack. The reason for this is because, since VF-attack attacks performs multiple attacks in a row, the recovered state error starts accumulating until the detector starts failing. This was not that big of an issue in Acrobot because these repeated attacks happen towards the end of the episode, however, in Cart Pole the states at the beginning of the episode and at the end can have an equally high VF value, and so this carried error can start very early in the

	Acrobot		Cart Pole		Mountain Car		Taxi	
normalization	**min max**	**z-score**	**min max**	**z-score**	**min max**	**z-score**	**min max**	**z-score**
UniformAttack								
Kernel Density	-1.33	-0.90	**1.00**	-1.05	0.03	0.16	-0.13	0.15
DBSCAN	-1.08	0.62	**0.96**	-2.86	0.14	**1.01**	-0.60	0.64
KNN-Hyper	-1.47	0.47	**0.96**	-2.59	0.12	**0.99**	-0.30	0.64
KNN	-1.11	-0.41	**1.00**	-0.41	**1.01**	**1.00**	0.13	0.20
Gaussian Mixture	-0.43	0.34	-1.71	-2.35	**1.05**	**1.01**	-0.25	0.74
k-means global	-1.25	-0.54	**1.00**	-0.66	**1.04**	**1.03**	-	0.71
k-means local	-0.95	-0.11	0.15	-0.89	0.52	0.52	-	0.87
SVM	-1.02	-0.65	**1.00**	-1.90	0.07	0.16	0.39	0.17
VF-Attack								
Kernel Density	-1.38	**1.37**	-0.04	-0.15	0.02	0.18	0.16	0.75
DBSCAN	-0.69	**1.04**	-0.11	-0.16	0.70	**0.97**	0.38	0.88
KNN-Hyper	-0.71	**1.01**	-0.12	-0.21	0.71	**0.95**	0.44	0.74
KNN	-1.00	**1.48**	-0.02	-0.16	**1.02**	**1.01**	0.78	0.39
Gaussian Mixture	-0.24	**1.35**	-0.07	-0.07	**1.04**	**1.01**	0.35	**1.04**
k-means global	-1.22	**1.51**	-0.02	-0.17	**1.02**	**1.01**	-	0.82
k-means local	-1.20	**1.47**	0.12	0.09	0.58	0.75	-	**1.04**
SVM	-0.22	**1.35**	-0.06	-0.22	0.38	0.82	**0.97**	**0.96**
ST-Attack								
Kernel Density	-0.81	0.04	**1.00**	-0.11	0.14	0.08	0.12	0.22
DBSCAN	-0.76	0.74	**1.00**	-0.68	0.50	**1.00**	0.19	**0.93**
KNN-Hyper	-0.78	0.53	**0.99**	-0.72	0.47	**1.00**	0.17	0.88
KNN	-0.71	0.35	**1.00**	-0.09	**1.03**	**1.04**	0.17	0.22
Gaussian Mixture	-0.13	0.58	-0.29	-0.49	**1.03**	**1.04**	0.23	**0.95**
k-means global	-0.58	0.27	**1.00**	-0.05	**1.03**	**1.02**	-	0.88
k-means local	-0.73	0.37	0.62	0.07	0.63	0.63	-	**1.01**
SVM	-0.47	0.17	**1.00**	-0.24	0.48	0.47	0.60	0.59

Table 6: Relative recovered performance obtained by the defense using the best parameters

episode.

Finally, in the Mountain Car and Taxi environments multiple detectors achieve a performance comparable with that of the unaltered policy, but DBSCAN and Gaussian Mixture particularly show better results than other defenses across both domains.

Conclusion

In this work, a two-step defense system against adversarial attacks in RL is created; (i) a density estimation based approach to detect adversarial examples and (ii) a KNN approach to recover the original states. Different methods to estimate the density are used to show the viability of using density estimation on experience tuples to detect adversarial examples. None of the methods used relies on the use of neural networks, which also helps against attacks that could target the detecting network alongside the victim policy. It is also shown that choosing the detection threshold solely on the observed experience tuples of the victim agent is enough to successfully detect adversaries, avoiding the need to model any particular attack which prevents overfitting into any particular attack and helps to generalize. Furthermore, it is also shown how the normalization method impacts the system and how the nature of an environment affects which normalization is best to use.

The defense system is benchmarked against three state-of-the-art attacks across four well-known environments, managing to recover most of the lost performance for most attacks and environments, albeit it suffers from the carried recovery error in consecutive attacks that start early on an episode. A recovery system with a lower error would fix these problems and achieve a better performance.

Acknowledgments

This research was funded in part by JPMorgan Chase & Co. Any views or opinions expressed herein are solely those of the authors listed, and may differ from the views and opinions expressed by JPMorgan Chase & Co. or its affiliates. This material is not a product of the Research Department of J.P. Morgan Securities LLC. This material should not be construed as an individual recommendation for any particular client and is not intended as a recommendation of particular securities, financial instruments or strategies for a particular client. This material does not constitute a solicitation or offer in any jurisdiction.

References

Abbasi, M.; and Gagné, C. 2017. Robustness to Adversarial Examples through an Ensemble of Specialists. arXiv:1702.06856.

Bradshaw, J.; de G. Matthews, A. G.; and Ghahramani, Z. 2017. Adversarial Examples, Uncertainty, and Transfer Testing Robustness in Gaussian Process Hybrid Deep Networks. arXiv:1707.02476.

Briola, A.; Turiel, J.; Marcaccioli, R.; and Aste, T. 2021. Deep Reinforcement Learning for Active High Frequency Trading.

Carlini, N.; and Wagner, D. A. 2016. Towards Evaluating the Robustness of Neural Networks. *CoRR*, abs/1608.04644.

Carlini, N.; and Wagner, D. A. 2017a. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. *CoRR*, abs/1705.07263.

Carlini, N.; and Wagner, D. A. 2017b. MagNet and "Efficient Defenses Against Adversarial Attacks" are Not Robust to Adversarial Examples. *CoRR*, abs/1711.08478.

Dempster, A. P.; Laird, N. M.; and Rubin, D. B. 1977. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1): 1–38.

Dolatshah, M.; Hadian, A.; and Minaei-Bidgoli, B. 2015. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. *CoRR*, abs/1511.00628.

Dong, Y.; Su, H.; Zhu, J.; and Bao, F. 2017. Towards Interpretable Deep Neural Networks by Leveraging Adversarial Examples. *CoRR*, abs/1708.05493.

Ester, M.; Kriegel, H.-P.; Sander, J.; Xu, X.; et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, 226–231.

Feinman, R.; Curtin, R. R.; Shintre, S.; and Gardner, A. B. 2017. Detecting Adversarial Samples from Artifacts. arXiv:1703.00410.

Fischer, T. G. 2018. Reinforcement learning in financial markets - a survey. Technical report.

Gong, Z.; Wang, W.; and Ku, W.-S. 2017. Adversarial and Clean Data Are Not Twins. arXiv:1704.04960.

Goodfellow, I. J.; Shlens, J.; and Szegedy, C. 2015. Explaining and Harnessing Adversarial Examples. arXiv:1412.6572.

Grosse, K.; Manoharan, P.; Papernot, N.; Backes, M.; and McDaniel, P. 2017. On the (Statistical) Detection of Adversarial Examples. arXiv:1702.06280.

Gu, S.; and Rigazio, L. 2015. Towards Deep Neural Network Architectures Robust to Adversarial Examples. arXiv:1412.5068.

Hartigan, J. A. 1975. *Clustering algorithms*. John Wiley & Sons, Inc.

Hendrycks, D.; and Gimpel, K. 2017. Early Methods for Detecting Adversarial Images. arXiv:1608.00530.

Huang, R.; Xu, B.; Schuurmans, D.; and Szepesvari, C. 2016. Learning with a Strong Adversary. arXiv:1511.03034.

Huang, S.; Papernot, N.; Goodfellow, I.; Duan, Y.; and Abbeel, P. 2017. Adversarial Attacks on Neural Network Policies. arXiv:1702.02284.

Kos, J.; and Song, D. 2017. Delving into adversarial attacks on deep policies. arXiv:1705.06452.

Lin, Y.-C.; Hong, Z.-W.; Liao, Y.-H.; Shih, M.-L.; Liu, M.-Y.; and Sun, M. 2019. Tactics of Adversarial Attack on Deep Reinforcement Learning Agents. arXiv:1703.06748.

Lin, Y.-C.; Liu, M.-Y.; Sun, M.; and Huang, J.-B. 2017. Detecting Adversarial Attacks on Neural Network Policies with Visual Foresight. arXiv:1710.00814.

Lu, J.; Issaranon, T.; and Forsyth, D. 2017. SafetyNet: Detecting and Rejecting Adversarial Examples Robustly. arXiv:1704.00103.

Meng, D.; and Chen, H. 2017. MagNet: A Two-Pronged Defense against Adversarial Examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 135–147. New York, NY, USA: Association for Computing Machinery. ISBN 9781450349468.

Metzen, J. H.; Genewein, T.; Fischer, V.; and Bischoff, B. 2017. On Detecting Adversarial Perturbations. arXiv:1702.04267.

Mnih, V.; Kavukcuoglu, K.; and Davi. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602.

Papernot, N.; McDaniel, P.; Wu, X.; Jha, S.; and Swami, A. 2016. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, 582–597.

Schölkopf, B.; Williamson, R. C.; Smola, A.; Shawe-Taylor, J.; and Platt, J. 2000. Support Vector Method for Novelty Detection. In Solla, S.; Leen, T.; and Müller, K., eds., *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

Sethi, T. S.; and Kantardzic, M. 2018. Data driven exploratory attacks on black box classifiers in adversarial domains. *Neurocomputing*, 289: 129–143.

Song, Y.; Kim, T.; Nowozin, S.; Ermon, S.; and Kushman, N. 2017. PixelDefend: Leveraging Generative Models to Understand and Defend against Adversarial Examples. *CoRR*, abs/1710.10766.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I.; and Fergus, R. 2014. Intriguing properties of neural networks. arXiv:1312.6199.

Tramèr, F.; Kurakin, A.; Papernot, N.; Goodfellow, I.; Boneh, D.; and McDaniel, P. 2020. Ensemble Adversarial Training: Attacks and Defenses. arXiv:1705.07204.

van den Oord, A.; Kalchbrenner, N.; and Kavukcuoglu, K. 2016. Pixel Recurrent Neural Networks. *CoRR*, abs/1601.06759.

Wu, Y.; Bamman, D.; and Russell, S. 2017. Adversarial training for relation extraction. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 1778–1783.

Yuan, X.; He, P.; Zhu, Q.; and Li, X. 2019. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9): 2805–2824.

Zhao, P.; and Lai, L. 2020. Analysis of KNN Density Estimation. arXiv:2010.00438.

PFPT: a Personal Finance Planning Tool by means of Heuristic Search and Automated Planning

Alberto Pozanco, Kassiani Papasotiriou, Daniel Borrajo*
J.P. Morgan AI Research
{alberto.pozancolancho, kassiani.papasotiriou, daniel.borrajo}@jpmorgan.com

Abstract

A crucial component to an individual's financial well-being is staying proactive in terms of the personal finances. Seeking such advice helps individuals or households to plan, save, and spend monetary resources over time, while taking into account various financial risks and future life events. Receiving such advice at the individual level usually happens by consulting a personal finance advisor which can be very expensive. In this paper we present PFPT, a Personal Finance Planning Tool that can use different search approaches to propose actionable plans to end users in order to achieve their financial goals. We evaluate PFPT in different problems using two different approaches: domain-independent automated planning and domain-dependent heuristic search. Results show that while automated planning struggle to generate good plans in this domain, our suggested heuristics are able to scale on generating realistic financial plans.

Introduction

Setting financial goals and planning ahead plays a significant role in ensuring financial health for an individual or a household. Personal finance planning activities include managing monetary resources through expenditure, investments, and savings, while considering various life events, risks and goals. The benefits of financial planning have been studied and quantified using economic well-being indicators in both empirical (Peng et al. 2007; Farinella, Bland, and Franco 2017; Warschauer and Sciglimpaglia 2012) and theoretical settings (Hanna and Lindamood 2010).

The most common way of seeking financial advice is by consulting a personal finance professional who can help clients make decisions about investments, budgeting or other courses of action to achieve their goals. Such services are often very expensive and thus inaccessible to a lot of people. Alternatives to speaking to an advisor include personal finance assessment tools and questionnaires which offer semi-personalized advice to users based on their input. However, these tools fail to recommend actionable points of advice on a more personal and detailed level.

We present in this paper PFPT, a Personal Finance Planning Tool, which offers financial advice at the individual level. It allows users to define both long-term and short-term financial goals and recommends actions to successfully achieve them based on their financial habits. We model this problem from a search perspective by defining states, actions and goals and apply domain-independent automated planning and domain-dependent heuristic search to recommend plans that maximize the likelihood of being executed based on the individual financial habits. To the best of our knowledge this is the first financial tool that applies planning and search techniques for personal finance management.

Previous technical methods of financial planning include expert systems which try to mimic the knowledge and experience of a human experts. The systems collect detailed user information regarding an individual's financial state and consists of a rules base to produce possible solutions to a goal (Kindle et al. 1989; Phillips, Nielson, and Brown 1992). More recent approaches used rule-based approaches based on different metrics and definitions on financial well-being (Althnian 2021). The main weakness of these approaches is that they do not provide flexible and detailed solutions and do not take into account the feasibility of the recommended plans. Other methods use Deep Reinforcement Learning techniques that often address a subset of financial goals, such as portfolio management (Irlam 2018; 2020).

The rest of the paper is organized as follows. In the next section we provide some background on numerical planning. Then, we define the problem solved by PFPT: finding a plan to go from an initial finance state to a goal finance state by maximizing the likelihood of the employed actions. After that, we introduce two different approaches to solve the PFPT problem: domain-independent automated planning and domain-dependent heuristic search, where we define a set of heuristics to guide the search. Later, we evaluate both approaches, focusing on analyzing the behavior of the different heuristics. Finally, we draw our main conclusions and outline future work.

Background

We use the standard classical STRIPS definition of a planning task, augmented with numeric variables (Fox and Long 2003). Formally:

*On leave from Universidad Carlos III de Madrid

Definition 1. *A **numeric planning task** is a tuple* $\Pi = \langle F, A, I, G \rangle$*, where* F *is a set of boolean and numeric variables,* A *is a set of actions,* $I \subseteq F$ *is the initial state and* $G \subseteq F$ *is a set of goals.*

We denote with S the set of all states of the planning task Π. A (full) state $s \in S$ is a valuation of all the variables in F; a boolean value for all the boolean variables and a numeric value for the numeric ones.

Each action $a \in A$ is defined in terms of its preconditions (pre(a)) and effects (eff(a)). Effects can set to true the value of a boolean variable (add effects, add(a)), set to false the value of a boolean variable (del effects, del(a)), and change the value of a numeric variable (numeric effects, num(a)). Action execution is defined as a function $\gamma : S, A \rightarrow S'$; that is, it defines the state that results of applying an action in a given state. It is usually defined as $\gamma(s, a) = (s \backslash \text{del}(a)) \cup \text{add}(a)$ if $\text{pre}(a) \subseteq s$ when only boolean variables are considered. When using numeric variables, γ should also change the values of the numeric variables (if any) in num(a), according to what the action specifies; increasing or decreasing the value of a numeric variable or assigning a new value to a numeric variable. If the preconditions do not hold in s, the state does not change.

The solution of a planning task is called a plan, and it is a sequence of instantiated actions that allows the system to transit from the initial state I to a state s where goals are true. Therefore, a plan $\pi = \langle a_1, a_2, \dots a_n \rangle$ solves a planning task Π (valid plan) iff $\forall a_i \in \pi, a_i \in A$, and $G \subseteq \gamma(\dots \gamma(\gamma(I, a_1), a_2) \dots), a_n)$. In case the cost is relevant, each action can have an associated cost, $c(a_i), \forall a_i \in A$ and the cost of the plan is defined as the sum of the costs of its actions: $c(\pi) = \sum_i c(a_i), \forall a_i \in \pi$. A plan with minimal cost is called optimal.

PFPT Problem Definition

We aim to find realistic financial plans that allow users to go from their current financial state to their goal financial state. We define a financial state as follows:

Definition 2. *A **financial state** is a tuple* $s = \langle t, Inc, DExp, FExp, B \rangle$*, where:*

- $t \in \mathbb{N}$ *is a time step*
- $Inc \in \mathbb{R}$ *is the income per time step*
- $DExp \in \mathbb{R}$ *are the discretionary expenses per time step*
- $FExp \in \mathbb{R}$ *are the fixed expenses per time step*
- $B = (\hat{B} + Inc - DExp - FExp)$ *is the account balance, where* $\hat{B}$ *is the account balance of* s*'s parent at* $t - 1$.

The initial financial state is fully specified, while the goal financial state is usually partially specified. For example, the goal state could specify that the balance at a given time step should be higher than a given quantity.

At each time step, some actions can be applied in order to change the financial state into another one. We define two types of **actions**: income increases and discretionary expenses decreases. We assume the fixed expenses cannot be changed, or will be very unlikely changed. Actions might produce changes in the financial state. For example, an income increase of 20% in state s_t will result in a new state at the next time step s_{t+1} with income $\text{Inc}(s_{t+1}) = 1.2 \times \text{Inc}(s_t)$. Besides these effects over the financial state, each action has associated a *likelihood* score, which is a real number between 0 and 1 that reflects how feasible or probable is that a user executes the action. This likelihood score can be given by users or inferred from their financial habits. For example, increasing the income by 0% will have a higher likelihood score than increasing the income by 20%, since the former is an easier or more feasible action than the latter. Table 1 summarizes a potential set of actions along with their effects to the financial state and their likelihood score. We will use this set of actions as an example in the rest of the paper.

Income increase and discretionary expenses decrease actions can be combined to generate **joint actions**. Assuming the actions described in Table 1, we would have 9 possible joint actions that can be applied at each time step, i.e., [Increase Inc 10%, Decrease DExp 0%], [Increase Inc 20%, Decrease DExp 10%], etc. A plan π solves this problem optimally if it achieves the financial goal state by maximizing the likelihood product of its actions. Formally:

$$\max \prod_{a \in \pi} \text{likelihood}(a) \tag{1}$$

We face two obstacles when trying to use search algorithms to (optimally) solve the problem as defined in Expression 1: (1) plan optimality is defined as a product, while search algorithms ordering functions are typically additive; and (2) plan optimality is defined as a maximization task (maximize likelihood), while most search algorithms aims to minimize a given function. To overcome the first problem, we compute the logarithm of each action's likelihood score so we can sum them. To overcome the second problem, we transform the maximization task into a minimization task by subtracting the logarithm of the likelihood score from one. By performing these two transformations, now we have the following additive cost function that search algorithms can minimize:

$$c(a) = 1 - log(\text{likelihood}(a)) \tag{2}$$

Given a plan π and its cost $c(\pi)$, we can compute back its likelihood score by applying the following operation:

$$\text{likelihood}(\pi) = \exp(-(c(\pi) - |\pi|)) \tag{3}$$

In the next sections we describe two different search-based approaches to solve this problem: domain-independent automated planning and domain-dependent heuristic search.

Automated Planning Approach

The first approach uses automated planning models and planners to generate solutions. The domain is composed of actions that model the actions described in Table 1. As an example, the action that increases the income, would be modeled as shown in Figure 1.

At each time step, we only allow the execution of one action of each kind: modify income or modify expenses. This

Action	Effect	Likelihood
Increase Income by 0%	-	1.0
Increase Income by 10%	$\text{Inc}(s_{t+1}) = 1.1 \times \text{Inc}(s_t)$	0.8
Increase Income by 20%	$\text{Inc}(s_{t+1}) = 1.2 \times \text{Inc}(s_t)$	0.6
Decrease Discretionary Expenses by 0%	-	1
Decrease Discretionary Expenses by 10%	$\text{DExp}(s_{t+1}) = 0.9 \times \text{DExp}(s_t)$	0.9
Decrease Discretionary Expenses by 20%	$\text{DExp}(s_{t+1}) = 0.8 \times \text{DExp}(s_t)$	0.8

Table 1: Actions' summary.

```
(:action increase-income
    :parameters (?p - percentage
                 ?t - time-step)
    :preconditions
      (and (current-time ?t)
           (not (done-income ?t)))
    :effects
      (and (increase (total-cost)
                     (likelihood ?p))
           (increase (income ?t)
                     (* (income ?t)
                        (percentage ?p)))
           (increase (balance ?t)
                     (income ?t))
           (done-income ?t)))
```

Figure 1: Model of the action that increases the income by 20%.

action is the only one defined to modify income and summarizes all possible increase operations over income. The parameters of the action are a time step and a percentage of increase. Since we do not need complex temporal reasoning, we consider discrete temporal problems, so we model time explicitly as a sequence of time steps. Percentages are also represented as discrete amounts and are defined in the problem description. In the case of the percentages defined in Table 1, we would define three objects of type percentage in the problem for the income (0, 10 and 20) and another three for modifying the expenses (also 0, 10 and 20). The reason to separate income percentages from expenses percentages is that we need also to define their corresponding likelihoods (predicate likelihood) which have different values. For instance, the likelihood of increasing income in a 20% is 0.6, while the likelihood of decreasing the expenses in a 20% is 0.8.

The preconditions of the actions are that we are at a given time step and that we did not update yet the income at that time step. The expected effects are that the income will increase based on the percentage, the balance is increased with the new income, and the total cost is updated. We use as cost the one defined in Equation 2. Apart from the increase-income and decrease-expenses, the domain also includes a move-time action that progresses time.

The problem description contains objects related to the set of time steps and the percentages. If the user sets as goal to have a balance x at time step T, the problem will be automatically generated with all the time steps between 0 and T. The initial state defines the initial income, balance, and expenses, as well as the likelihoods of each percentage, the initial total cost of 0, the initial time step of 0 and the needed next predicates to connect in sequence all the time steps. The goal description is compiled from the user goals, as for instance, the balance being greater than a given value at a given time step.

The plans are sequences of actions that achieve the goals from the initial state. They are comprised of a joint action (income, expenses) at each time step, plus a move-time action to progress to the next time step. As an example, a plan would be:

```
(increase-income t0 p-inc-0)
(decrease-expenses t0 p-exp-20)
(move-time t0 t1)
(increase-income t1 p-inc-10)
(decrease-expenses t1 p-exp-0)
```

that would not increase income and decrease expenses in a 20% in the first time step, and increase income in a 10% and not modify expenses in the second time step.

The main drawback of using planning for solving this task is that it is a numerical planning task. First, there are very few planners that can handle this kind of domain complexity. Second, to the best of our knowledge, there is no planner that can perform optimal numerical planning. Thus, we have defined a search-based solution that allows us to compute optimal solutions for this task which is presented in the next section.

Heuristic Search Approach

We use the most popular algorithm for optimal search, A*(Hart, Nilsson, and Raphael 1968), to solve this problem in a domain-dependent fashion. A* uses a function $f(s) = g(s) + h(s)$ to order the nodes in the open list. The solutions returned by A* are guaranteed to be optimal if the heuristic h is admissible, i.e., it does not over-estimate the cost of reaching the goal from any state.

The cost of reaching a state s, $g(s)$, is computed using Equation 2. In order to estimate the cost of reaching the goal from s, $h(s)$, we propose the following domain-dependent heuristics.

Minimum Cost Action

The first heuristic, which we called Min, consists on choosing the cost of the cheapest joint action $c(a)_{min}$ and multiply it by the number of remaining time steps: $h(s) = c(a)_{min} \times (t(G) - t(s))$. In our case, the cheapest joint action

is to do nothing, i.e., increase the salary by 0% and decrease the discretionary expenses by 0%.

Lemma 1. *Min is admissible.*

Proof. By construction, at each time step, there is no cheaper joint action than $c(a)_{min}$. The result of multiplying the minimum cost by the $(t(G) - t(s))$ will necessarily be less than or equal h^*. Therefore, Min is admissible. □

Greedy

The next heuristic we propose consists on solving a relaxation of the problem where only the same action can be applied at every time step. In other words, the number of potential plans is limited to the number of joint actions considered. The procedure that computes the heuristic is outlined in Algorithm 1. The algorithm receives as input the current (s)

Algorithm 1 Greedy Heuristic

Require: $s, G, A,$ Admissible
Ensure: GH
1: GH $\leftarrow \infty$
2: remainingTimeSteps $\leftarrow t(G) - t(s)$
3: sortedActions $\leftarrow$ SORTBYCOST(A)
4: **for** $a \in$ sortedActions **do**
5: $s' \leftarrow$ EXECUTE(remainingTimeSteps, a, s)
6: **if** $G \subseteq s'$ **then**
7: **if** Admissible = True **then**
8: GH $\leftarrow c(a)$
9: **else**
10: GH $\leftarrow c(a) \times$ remainingTimeSteps
11: **return** GH
12: **return** GH

and goal (G) state, the available actions (A), and a parameter that indicates whether we are interested in the heuristic to be admissible or not. The algorithm first computes the number of remaining time steps from s (line 2). If the goal state does not specify any time step, this is set to a high number. Then, the actions in A are sorted according to their cost as per Equation 1. Next, the algorithm iterates over the sorted list of actions, executing the given action a from s for the number of remaining steps, yielding a state s'. If the goal is satisfied in s', the algorithm finishes and returns the heuristic estimate. This heuristic value will depend on the admissibility parameter. If we are interested in an admissible heuristic (GH_a), Algorithm 1 will return the cost of executing that action, $c(a)$.

Lemma 2. *GH_a is admissible.*

Proof. Suppose GH_a returns $c(a)$ and $c(a) > h^*$. It means that there is a solution that only uses actions with a cost less than $c(a)$. If it would be using actions whose cost would be greater than $c(a)$, then $c(a)$ would be less than h^*, so the assumption would be false. And, if there would be a solution using only a subset of less costly actions, it would had been found before a, since they are studied from less costly to more costly. Thus, $c(a)$ is less than h^*, and it is admissible. □

If we want a more informative but inadmissible heuristic (GH_i), Algorithm 1 returns the cost of executing that action multiplied by the number of remaining steps.

Lemma 3. *GH_i is inadmissible.*

Proof. The heuristic value returned by GH_i considers executing the cheapest possible action a that reaches the goal (ensured by the actions' sorting in line 3 and the loop in line 4) in all the remaining time steps. However, reaching the goal state could only require executing a in a subset of the remaining time steps together with some lower cost actions in the other steps. Thus, GH_i could return greater values than h^* for some state/goal combinations, so it is inadmissible. □

If after iterating over all the possible actions the goal cannot be achieved, Algorithm 1 will return ∞, meaning that the goal is not reachable from a.

Heuristics Behavior Example

Let us exemplify how the heuristics work and their accuracy by computing them at the initial state ($h(I)$) of the following PFPT problem:

$$I = \langle t = 0, \text{Inc} = 5, \text{DExp} = 2, \text{FExp} = 2, B = 10 \rangle$$
$$G = \langle t = 4, B = 17 \rangle$$

The optimal solution to this problem has a cost of 8.43 ($h^*(I)$). The minimum cost action heuristic Min returns the cost of the cheapest action multiplied by the number of remaining time steps. The cheapest joint action is to Increase Inc 0% and Decrease DExp 20%, and has an associated cost of 2. After multiplying it by the 4 remaining time steps, Min will return a cost of 8, which is a lower bound on $h^*(I)$. The greedy algorithm returns that [Increase Inc 0%, Decrease DExp 20%] is the cheapest joint action that can be subsequently executed from I in the remaining time steps to achieve the goal. This joint action has an associated cost of 2.22. Therefore, GH_a will return that cost, which is a lower bound on $h^*(I)$, while GH_i will return $2.22 \times 4 = 8.88$, which is an upper bound on $h^*(I)$.

Evaluation

We randomly generate PFPT problems of increasing difficulty by increasing the time horizon at which the goal balance has to be achieved. To generate hard problems, we (1) set the goal balance to be twice the initial balance; and (2) make the expenses per time step (sum of DExp and FExp) to be a random ratio between 0.9 and 1 of the income per time step, thus rendering problems where little savings are generated if the initial financial state remains unchanged. We solve these problems with the previously described heuristics: Min, GH_a and GH_i, plus Blind, which we will use as a baseline to compare heuristics' search performance. All the heuristics have been implemented in Python 3.6, as well as the search algorithm, which is a vanilla implementation of A*. Heuristic search experiments were run in Intel(R) Xeon(R) CPU E3-1585L v5 @ 3.00GHz machines with 64GB of RAM. We also tried to solve the PDDL version

of these problems using LPG (Gerevini, Saetti, and Serina 2004), a stochastic planner for numerical planning. Since the planner is stochastic, we ran the planner five times on each problem. Automated planning experiments were run in an Apple M1 Pro machine with 16GB of RAM.[1]

Automated Planning vs Heuristic Search

For the first set of experiments, we generated 10 random problems with the time horizon t set to 4. LPG is only able to solve 4 out of the 10 problems, reporting that no solution could be found for the other 6. The reason LPG failed to solve the problems was not due to time nor memory constraints. The execution time in the solved problems is always below 2 seconds, but the solutions returned are suboptimal. Given that the machine where LPG was run is different than the machine where the rest of code was run, the time cannot be compared directly. Instance #9 is the problem where LPG gets the closest to the optimal, returning a plan with cost 8.80, while the optimal cost is 8.32. On the other hand, instance #8 is where LPG obtain the worst results, returning a plan with cost 10.19 in one of the executions, while the optimal cost is again 8.32. Cost differences might look small, but they translate into large likelihood score differences. In instance #8, the optimal plan has a likelihood score of 0.73, while the plan returned by LPG has a likelihood score of 0.11, meaning it would be a very unrealistic plan to propose to an end user.

Heuristic	Plan Cost	Expanded	Generated	Search Time (s)
Blind	**8.80**	1698.8	14350.0	56.5
Min	**8.80**	5320.3	10310.1	60.1
GH_a	**8.80**	344.7	2931.2	1.6
GH_i	8.81	**11.3**	**91.2**	**0.0**

Table 2: Heuristics comparison in random problems with $t = 4$. Numbers represent average across 10 problems. Best values are shown in bold.

Table 2 summarizes the results of the different heuristics in the same set of problems. In this case, all the heuristics are able to solve all the problems. As expected, the three admissible heuristics return the optimal plan in all the problems (average plan cost of 8.80), while GH_i returns a slightly suboptimal solution in one of the problems, thus increasing the average plan cost to 8.81. In terms of search efficiency, Min is not able to outperform Blind in this problem setting. The distribution of f values of both searches is shown in Figure 2. As we can see, Min generates search spaces where many states have the same f value of 8, generating a large plateau at the beginning of the open list. This occurs because many states have the same f even if they are still far from reaching the goal due to the heuristic value being a constant function of the remaining time steps, which translates into more expanded nodes. On the other hand, Blind generates more diverse f values, with most of the nodes having f values between 11 and 12, therefore being able to better discriminate between the states that are closer to the goal time

[1]As the following results show, the fact that we are using different machines is not relevant in this comparison.

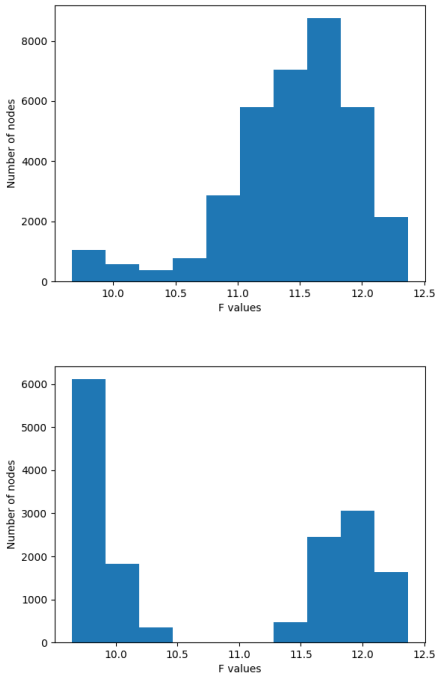

Figure 2: Open list f values distribution at the end of the search when using Blind (upper histogram) and Min (lower histogram) heuristics.

horizon. This behavior could not be substantially improved even when considering different tie-breaking rules.

GH_a reduces the search effort by an order of magnitude, returning optimal solutions in less than two seconds on average. The inadmissible greedy heuristic GH_i achieve the best results in terms of search efficiency, expanding only around 11 states to reach the goal. This represents a 0.6% of the states that a Blind search needs to expand, meaning this heuristic is really informative. GH_i also yields faster searches that reach the goal in less than a second.

Heuristics Scalability and Optimality

As we have seen, we cannot rely on LPG to solve this kind of problems. The behavior of the Blind, Min, and GH_a heuristics also quickly deteriorates as we increase t. For example, Blind and Min are not able to find a solution for any of the 10 problems with $t \geq 5$ in less than $1800s$, while GH_a cannot find solutions within that time limit in most problems with $t \geq 6$. Hence, we evaluated the scalability and performance of GH_i in harder problems with longer time horizons. The scalability of GH_i is shown in Figure 3, where we use violinplots to show the search time distribution when solving problems with $t = 4$, 6, 8, and 10. As we can see, solving time grows exponentially with the time horizon. However, GH_i is able to generate fast searches that can find the solution in less than a second in many of the problems. We are not solving problems with larger time horizons because our vanilla implementation of A^* is not able to scale in some of

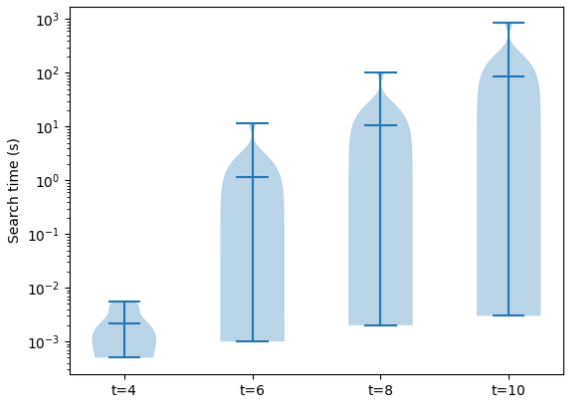

Figure 3: Search time (log scale) needed to find a solution by GH_i in problems of increasing difficulty, i.e., longer time horizons.

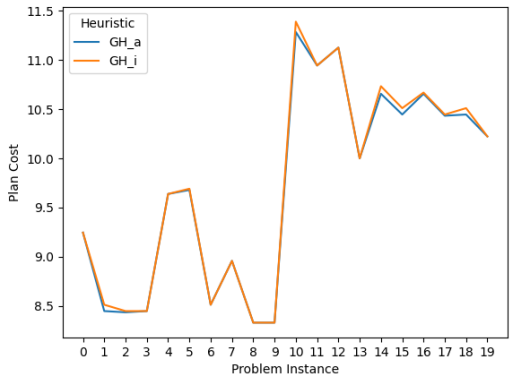

Figure 4: Plan cost as returned by GH_a and GH_i in problems of increasing difficulty, i.e., longer time horizon.

the bigger instances. However, our suggested greedy heuristics would be likely able to scale provided a better implementation of A^*, which we leave as future work.

Finally, we also wanted to better understand GH_i's optimality loss in relation to the optimal solution. Figures 4 and 5 show the plan cost and likelihood respectively as returned by GH_a and GH_i in problems with $t = 4$ and $t = 5$, where GH_a can compute the optimal plan within the time bound. As we can see in Figure 4, the optimality gap is really small (less than 1% on average), meaning that both heuristics achieve plans with very similar costs. GH_i is able to compute the optimal plan in 11 out of the 20 problems. We see the biggest optimality gap in a problem with $t = 5$ (problem instance $\#10$), where there is a 0.93% optimality gap. When we translate the costs of this problem back into likelihood scores (see Figure 5), we get that the optimal plan likelihood is 0.27, while the likelihood of the plan returned by GH_i is 0.24. On average, GH_a returned plans with an

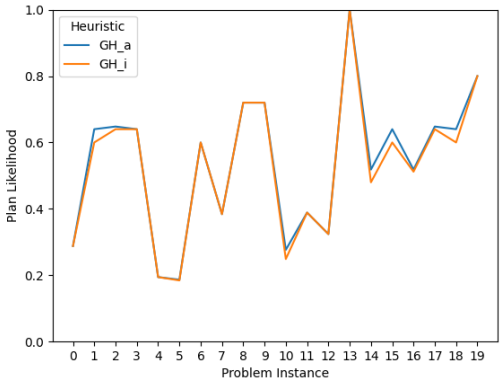

Figure 5: Plan likelihood as returned by GH_a and GH_i in problems of increasing difficulty, i.e., longer time horizon.

optimal likelihood score of 0.54, while GH_i returned plans with a likelihood score of 0.52.

Conclusions and Future Work

We have proposed PFPT, a personal finance planning tool that offers financial advice at the individual level. The suggested financial plans achieve users' financial goals by maximizing the likelihood of being executed based on their financial habits. We model this problem from a search perspective and propose two different approaches to solve it: domain-independent automated planning and domain-dependent heuristic search. We evaluated both approaches in a set of financial problems with increasing complexity. Results showed that, as expected, while automated planning struggles to generate good plans in this domain, our suggested heuristics are able to scale on generating realistic financial plans.

Currently, our set of actions is limited to income increases and discretionary expenses decreases. We are exploring how to enrich the action space to include actions such as invest in different financial products that might yield different interest rates. We would also like to have the ability to impose arbitrary constraints to the generated plans. For example, users might want to see plans that do not suggest any income increase. Finally, we are currently assuming constant likelihood scores. In future work we would like to consider conditional likelihood scores, where the likelihood of executing one action depends on the previous actions.

Acknowledgements

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co. and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or

sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

Althnian, A. 2021. Design of a rule-based personal finance management system based on financial well-being. *International Journal of Advanced Computer Science and Applications* 12(1).

Farinella, J.; Bland, J.; and Franco, J. 2017. The impact of financial education on financial literacy and spending habits. *International Journal of Business, Accounting, & Finance* 11(1).

Fox, M., and Long, D. 2003. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research* 20:61–124.

Gerevini, A.; Saetti, A.; and Serina, I. 2004. Planning with numerical expressions in lpg. In *Proceedings of the 16th European Conference on Artificial Intelligence*, 667–671.

Hanna, S. D., and Lindamood, S. 2010. Quantifying the economic benefits of personal financial planning. *Financial Services Review* 19(2).

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Irlam, G. 2018. Financial planning via deep reinforcement learning ai. *Available at SSRN 3201703*.

Irlam, G. 2020. Multi scenario financial planning via deep reinforcement learning ai. *Available at SSRN 3516480*.

Kindle, K. W.; Cann, R. S.; Craig, M. R.; and Martin, T. J. 1989. Pfps-personal financial planning system. In *IAAI*.

Peng, T.-C. M.; Bartholomae, S.; Fox, J. J.; and Cravener, G. 2007. The impact of personal finance education delivered in high school and college courses. *Journal of family and economic issues* 28(2):265–284.

Phillips, M. E.; Nielson, N. L.; and Brown, C. E. 1992. An evaluation of expert systems. *Journal of Financial Counseling and Planning* 3(1).

Warschauer, T., and Sciglimpaglia, D. 2012. The economic benefits of personal financial planning: An emperical analysis. *Financial Services Review* 21(3).

Filtering Top-k Relevant Plans

Mauricio Salerno, Miguel Tabernero, Raquel Fuentetaja, Alberto Pozanco

Department of Computer Science and Engineering
Universidad Carlos III de Madrid, 28911 Leganés, Madrid, Spain
msalerno@pa.uc3m.es, mtaberne@pa.uc3m.es, rfuentet@inf.uc3m.es, alberto.pozanco@gmail.com

Abstract

The automatic generation of a set of plans rather than just one is a relevant problem in Automated Planning, with a wide range of applications, including applications to finance and banking. Such sets can be computed through top-k planning, which aims to find the best k plans that solve a planning task. Existing approaches to solve the top-k planning problem might generate plans that are not *relevant* for some practical applications. In particular, plans might contain actions that can be removed from the plan while maintaining its validity. These unnecessary actions not only increase the cost of plans, but might particularly reduce the utility of top-k planning. In this work we propose an Automated Planning approach for identifying and eliminating redundant actions from plans, and show how to incorporate this method into top-k planning to guarantee that the generated plans do not contain redundant actions. We perform an empirical analysis to study the existence of redundant actions in plans in several benchmarks, and analyze how top-k planning methods are affected when forced to find plans without redundant actions.

Introduction

There exist many planning applications where it is necessary to compute a set of plans rather than only one. This is the case of tools where planning is supporting human decision makers, who are typically keen on exploring different alternatives and scenarios. Having a diverse set of plans (Srivastava et al. 2007; Roberts, Howe, and Ray 2014) at hand when making these decisions is crucial in many finance applications, that include but are not limited to: goal and plan recognition (Ramírez and Geffner 2009; Sohrabi, Riabov, and Udrea 2016) to predict customer goals in order to provide adequate services to them (Borrajo, Gopalakrishnan, and Potluru 2020; Borrajo and Veloso 2020; Borrajo, Veloso, and Shah 2020); planning approaches to predict stock market movements (Mund, Vallati, and McCluskey 2020); or cybersecurity (Boddy et al. 2005; Pozanco et al. 2021), where experts are interested in understanding a set of possible attacks to communication networks and protocols.

Top-k planning (Riabov, Sohrabi, and Udrea 2014; Katz et al. 2018; Speck, Mattmüller, and Nebel 2020) is one of the existing approaches to compute sets of plans, which aims to find the best k plans solving a planning task. In top-k planning, as in other Automated Planning problems, the quality of plans is measured using a cost function. However, there are additional notions of quality that can be considered. Quality can be also defined from the point of view of *plan relevance* or *justified plans* (Fink and Yang 1992). From a domain independent perspective, relevant plans can be understood as those that do not contain loops and or irrelevant/unnecessary actions (Fink and Yang 1992; Nebel, Dimopoulos, and Koehler 1997). From a domain specific point of view more subtle notions of relevancy could be considered.

Current planners guided by powerful heuristics will probably not include (many) irrelevant actions in the output plan, but this is not enough for top-k planning. Iterative approaches to top-k have shown promising results (Katz et al. 2018). However, resulting plans include more irrelevant information as more iterations are carried out, to the point that truly alternative plans can not be easily distinguished from those that were extended with unnecessary operators. This may have an important negative impact in applications. For instance, in goal and plan recognition, agents are usually assumed to follow optimal or at *least* sub-optimal plans to achieve their goals (Ramírez and Geffner 2010), but redundant actions do not contribute to achieve the goals to such task. Also, it is not useful to consider these plans in explainable AI, since they are of little interest for users.

As an illustrative example of the notion of plan relevance consider the planning task depicted in Figure 1, where a

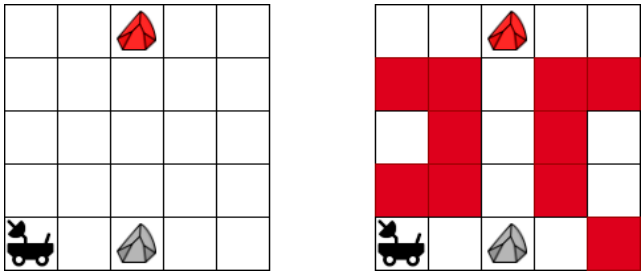

Figure 1: Two planning tasks of a navigation domain where a robot has to sample a rock. The robot can move to adjacent white tiles.

robot has to take a sample of the red rock situated on the top. Here, the plans that represent the different routes the agent can use to sample the red rock can be considered relevant of justified. In contrast, any plan where the agent passes through the same cell more than once would not be justified, since the actions that create the loop can be removed from the plan and it still achieves the goal. Another plan that would not be justified is one where the agent reaches the goal state (samples the rock) and then executes additional actions: these subsequent actions can be subtracted from the plan, and generate equally valid plans. The same occurs in plans where the robot samples also the grey rock, which is not necessary to sample. Figure 1 shows two planning problems. For the one on the left there are many relevant plans, while for the one on the right there is only one relevant plan. It is not difficult to find additional examples in large-scale real-world applications where the domain can include many operators, static facts and objects, which might be irrelevant for specific goals and initial situations. The number of possible non-relevant plans can be quite large, and these plans are somehow *artificial*, since they do not represent actual different alternatives to achieve the goals.

In this work, we focus on two ideas: (1) filtering unnecessary actions in a plan post-optimization step, in the same line as some previous works (Nakhost and Müller 2010; Chrpa, McCluskey, and Osborne 2012b,a; Balyo, Chrpa, and Kilani 2014); specifically, we contribute with a compilation that encodes the problem as an Automated Planning task, so that in can be solved using an off-the-shelf automated planner; and (2) incorporating the notion of plan justification in the context of top-k planning; specifically, we present an approach to find relevant/justified plans when using iterative approaches to top-k planning. This work expands on top-k and is closely related with the ideas of top-quality (Katz, Sohrabi, and Udrea 2020) and diverse planning (Katz and Sohrabi 2020), since these approaches also find plans that should meet certain conditions.

The rest of the paper is organized as follows. Next section introduces basic notions of classical planning, plan justification and top-k planning. Then, we define a planning compilation for plan justification. After that, we describe how to integrate it with top-k planning. Next, we include an empirical evaluation on how the incorporation of plan justification affects top-k planning. Finally, we discuss related work and draw some conclusions.

Background

Classical Planning

A classical planning task (Fikes and Nilsson 1971) is defined as a tuple $\Pi = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, where $\mathcal{F}$ is a set of propositions; $\mathcal{A}$ is a set of actions; $\mathcal{I} \subseteq \mathcal{F}$ is the initial situation, encoding what propositions are true initially; and $\mathcal{G} \subseteq \mathcal{F}$ is a set of goal propositions. Every $a \in \mathcal{A}$ has preconditions, denoted as $pre(a) \subseteq \mathcal{F}$, added effects $add(a) \subseteq \mathcal{F}$ and negative effects $del(a) \subseteq \mathcal{F}$.

A planning task Π defines a state model which states $s \in S$ are subsets of $\mathcal{F}$ and are represented by the fluents that are true in the corresponding state. In this model, the initial state is $s_i = \mathcal{I}$, and the goal states are those s_g that include the goals $\mathcal{G} \subseteq s_g$. The actions $a \in \mathcal{A}$ that are applicable in a state s, denoted as $A(s)$, are those for which $pre(a) \subseteq s$. The transition function is γ, where $\gamma(s, a) = (s \setminus del(a)) \cup add(a)$ represents the state s' that results from the application of the action a in state s.

A solution or valid plan for Π is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that induces a state sequence $\mathcal{S}_\pi = \langle s_0, \ldots, s_n \rangle$ such that $s_0 = \mathcal{I}$ and, for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and $s_i = \gamma(s_{i-1}, a_i$. A plan π solves Π if and only if $\mathcal{G} \subseteq s_n$. We denote the set of all the plans that solve a planning task Π as P_Π. Each action $a \in \mathcal{A}$ is assumed to have a non-negative cost $c(a)$, so that the cost of a plan is $c(\pi) = \sum c(a_i)$. A plan is optimal if it has minimum cost.

Top-k planning

The objective of a top-k planning problem (Riabov, Sohrabi, and Udrea 2014; Sohrabi, Riabov, and Udrea 2016) is to find the k plans of lowest cost for a planning task.

Definition 1. *A top-k planning problem is a tuple* (Π, k)*, where* $\Pi = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ *and* $k \in \mathbb{N}$*. The goal is to find a set of plans* $P_k \subseteq P_\Pi$ *such that:*

- *For each* $\pi \in P_k$*, if there exists a plan* π' *with* $c(\pi') < c(\pi)$ *then* $\pi' \in P$*.*
- $|P_k| \leq |P_\Pi|$*, where* $|P_k| < k$ *implies* $P_k = P_\Pi$*.*

Note that the plans in P_k are not required to meet any condition other than being those with the *k* lowest costs. This means that algorithms that solve the top-k planning problem might find plans with actions or set of actions that are not *relevant/justified* under certain settings. The conditions a plan must meet to be *relevant* might be domain dependent and depend on the semantics of the planning task. Nevertheless, plan justification has been widely studied, with special interest in finding redundant actions that can be removed from the plan without affecting its validity. We want to introduce this notion into top-k planning.

Katz et al. (2018) proposed an iterative approach to solve the top-k planning problem. The main idea of their algorithm is to find additional solutions to a planning task by reformulating the original task. The reformulations *forbid* already found plans, so that previous plans will not be valid solutions, thus forcing the planner to find an alternate solution. The best *k* plans are found by iteratively solving and reformulating planning tasks. We are interested in this approach because it is straightforward to extend, including additional conditions the found plans must meet. When a solution is found, it can then be checked to verify if it meets a certain condition. If it does, this solution is counted as a *valid* plan, and the iterative procedure continues as normal. If it does not meet the condition, this solution is forbidden with the reformulation as usual, but the number of found plans so far is not increased.

Plan Justification

The notion of plan justification can be traced back to the early 1990s (Fink and Yang 1992). In that work, Fink and

Yang define different types of plan justifications: backward justification, well justification and perfect justification.

Given $\Pi = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, a plan $\pi = (a_1, ..., a_n)$ and the set of causal links between them, an action $a_i \in \pi$ is *backward justified* if $\exists p \in add(a_i)$ such that $p \in \mathcal{G}$ or $\langle a_i, a_j, p\rangle$ is a causal link and a_j is backward justified[1]. The triple $\langle a_i, a_j, p\rangle$ forms a causal link if a_i adds p, p is a precondition of a_j, and p is neither added nor deleted by any action between a_i and a_j (Celorrio et al. 2013). Then, a_i is backward justified if it is causally related to the goals. A plan π is backward justified if all of its actions are backward justified. *Well justified* actions are those that can not be removed from the plan without affecting the applicability of other actions. *Perfectly-justified* plans are those for which no subset of actions can be removed from the plan without invalidating plan. In this paper we also apply this idea.

AP compilations for Action Elimination

This section introduces some formal definitions used in the rest of the paper, and explains the proposed AP compilations to eliminate unnecessary actions from plans.

Formal Definitions

We use the notion of *perfectly-justified* introduced by Fink and Yang. Thus, a plan is perfectly-justified if no actions can be *skipped* or eliminated while maintaining the plans' validity. We introduce it using the following definitions.

Definition 2 (Reduced Plan). *Given a plan $\pi = \langle a_1, \ldots, a_n\rangle$ for Π and a strict subset of its actions $A_\pi \subset \pi$, $A_\pi \neq \emptyset$, the reduced plan $\pi_{\setminus A_\pi}$ is the action sequence resulting from eliminating the actions $a_i \in A_\pi$ from π.*

Definition 3 (Well-justified action set). *A subset of plan actions $A_\pi \subseteq \pi$, $A_\pi \neq \emptyset$, is well justified if the corresponding reduced plan $\pi_{\setminus A_\pi}$ is not a valid plan for Π.*

The previous definition just extends the notion of well-justified actions to well-justified subsets of actions. [2]

Now, we consider a plan π to be perfectly-justified if all of its subsets of actions are well justified, i.e. all the plans reduced by those subsets are invalid, so that there is no way of reducing the plan while maintaining its validity.

Definition 4 (Perfectly-justified plan). *A plan π is perfectly-justified iff all non-empty strict subsets of its actions, $A_\pi \subset \pi$, are well-justified.*

Then, if there is at least a subset of actions which is not well-justified, the plan is not perfectly-justified. In that case will say the actions in that subset are **unnecessary**.

Given Π and a plan π, the task of finding the smallest perfectly-justified plan by eliminating actions from π is called Minimal Length Reduction (MLR) (Balyo, Chrpa, and Kilani 2014). Balyo, Chrpa, and Kilani also define the Minimal Reduction (MR) task. The aim of this task is to find a plan with the smallest possible cost by eliminating actions from π. In this paper we are particularly interested in MLR. Both tasks have been shown to be NP-complete (Fink and Yang 1992; Nakhost and Müller 2010). In the following section we propose a compilation of the MLR problem into an AP one so that it can be solved with an off-the-shelf planner.

[1] Causal links were called initially *establishments* (Fink and Yang 1992).

[2] This notion is also defined by Balyo, Chrpa, and Kilani [2014] as plan reductions.

Perfect Justification as Planning

The idea is to define a classical planning task that, given a planning task and a solution plan, can identify and eliminate sets of unnecessary actions from plans. The compilation consists of encoding the planning task in a way that allows to include any action occurring in the original plan after the last action that was previously included. We achieve this by creating an order relation between the actions in π. More formally, given the planning task Π and a solution plan π we define $\Pi^{order} = (\mathcal{F}', \mathcal{A}', \mathcal{I}', \mathcal{G})$ as follows:

- $\mathcal{F}' = \mathcal{F} \cup \mathcal{F}_{last} \cup \mathcal{F}_{order} \cup \mathcal{F}_{planact}$, where:
  - $\mathcal{F}_{last} = \{last_i \,|\, 0 \leq i \leq n\}$ facts represent the last position considered. There is a position (order in the sequence) for every action in the original plan plus an additional zero position,
  - $\mathcal{F}_{order} = \{order_{i,j} \,|\, 0 \leq i \leq n, i < j \leq n\}$ are static facts to encode that position i is before position j, and
  - $\mathcal{F}_{planact} = \{planact\_a_i \,|\, 1 \leq i \leq n\}$ are static facts to represent the action a appears in the plan π at position i.
- $\mathcal{A}' = \{a_{i,j} \,|\, a \in \mathcal{A}, 0 \leq i \leq n, i < j \leq n\}$, where there is an $a_{i,j}$ action for every action a in the original task and combination of positions i, j, defined as follows:

$$\begin{aligned} pre(a_{ij}) &= pre(a) \cup \\ &\quad \{last_i, order_{i,j}, planact\_a_j\} \\ add(a_{ij}) &= add(a) \cup \{last_j\} \\ del(a_{ij}) &= del(a) \cup \{last_i\} \end{aligned}$$

- $\mathcal{I}' = \mathcal{I} \cup \{last_0\} \cup \{order_{i,j} \,|\, 0 \leq i, j < n, i < j\} \cup \{planact\_a_i \,|\, a_i \in \pi\}$.

Actions $a_{i,j}$ in $\mathcal{A}'$ will only be applicable if there is an occurrence of action $a \in \mathcal{A}$ in the original plan π at position j. For that, facts of type $planact\_a_i$ are included in $\mathcal{I}'$, representing the plan π. There is one of such facts per plan action, indicating π contains an occurrence action $a \in \mathcal{A}$ at position i. The $last_i$ fact represent the position of the last included action. Thus, the preconditions of $a_{i,j}$ actions check that the action being included at position j, occurs in the original plan before the last included action with position i. The new initial state $\mathcal{I}'$ sets the relation order between the actions in π, where there is a fact $order_{0,j}$ for every position j in the plan, which allows the application of any action of the plan.

When Π^{order} is solved, the resulting plan will only contain the actions in the original plan that are necessary without altering the order. The plan that solves Π^{order} can be easily compiled-back to be a plan of the original task, just by removing the action parameters representing orders. The correspondence between the actions of both plans is one-to-one.

Definition 5. *Let π' be a valid plan of Π^{order}. Then, the compiled-back plan for Π is $\pi'' = \{a \mid a \in \pi \wedge a_{ij} \in \pi'\}$.*

This means that every a_{ij} action is replaced by its original action a.

Proposition 1. *The plan π'' obtained from any valid plan π' for Π^{order} is a valid plan for Π.*

Proof sketch. A plan π that induces the state sequence $\mathcal{S}_\pi = \langle s_0, \ldots, s_n \rangle$ is valid if all of its actions are applicable in the state they are applied and $\mathcal{G} \subseteq s_n$. Since the goals of both Π and Π^{order} are the same, if π' is valid for Π^{order}, then it will also achieve all the goals in Π by definition. Since $pre(a) \subset pre(a_{i,j})$ for every $a_{i,j} \in \mathcal{A}'$, if $a_{i,j}$ is applicable in a state, then the corresponding action $a \in \mathcal{A}$ is also applicable in that state. We know that π' is valid, so, starting from $\mathcal{I}$ all of its actions are applicable. Since $\mathcal{I} \subset \mathcal{I}'$, all actions in π'' are in turn applicable starting from $\mathcal{I}$. The goals are met and all actions are applicable, so π'' is a valid plan for Π.

Theoretical Properties

This section shows the type of plans that are obtained from solving the compiled planning tasks. In particular, we show that the set of valid plans for the compilations is the set of valid reduced plans of the original task.

Let P_Π be the set of valid plans for a planning task Π. Two planning tasks Π, Π' are *equivalent* if they have the same sets of valid plans, $P_\Pi = P_{\Pi'}$. For two plans π^-, π we say that π^- is a subset of π, $\pi^- \subset \pi$ if π^- can be generated from eliminating actions from π. Since the compilation **only** allows for the execution of actions in the original plan or for their (implicit or explicit) elimination, the set of valid plans for Π^{order} contain exactly all the valid plans that can be generated from eliminating subsets of actions from the original task if their corresponding transformations as defined in 5 are considered. More formally:

Proposition 2. *Let Π be a planning task and $\pi = \langle a_1, ..., a_n \rangle$ a valid plan for Π. Let P_{order} be the set of compiled-back valid plans for Π^{order}. Then $P_{order} = \{\pi^- \subseteq \pi \mid \pi^- \in P_\Pi\}$.*

Proof sketch. We have to show that (i) any plan in P_{order} is in $\{\pi^- \subseteq \pi \mid \pi^- \in P_\Pi\}$, and that (ii) any plan in $\{\pi^- \subseteq \pi \mid \pi^- \in P_\Pi\}$ is in P_{order}:
(i) Let $\pi' = \langle b_1, ..., b_m \rangle \in P_{order}$ be any compiled-back valid plan for Π^{order}. Since all a_{ij} belonging to Π^{order} have in their precondition $planact\_a_j$, only actions in π can be in π'. It is trivial to show that only one $last_i$ proposition is true in each state. In $\mathcal{I}$ only $last_0$ is true. Because of Π^{order} encoding, any a_i for which $pre(a_i) \subseteq \mathcal{I}$ is applicable initially. Any of these actions delete $last_0$ and add $last_i$. An action a_j is now applicable only if $i < j$. Continuing this process until $last_n$ is true it is easy to see that π' can only have actions in π respecting their order, where some of them might be skipped. By Proposition 1, $\pi' \in P_\Pi$. Therefore, it is proven that $(\forall\, \pi', \pi' \in P_{order} \implies \pi' \subseteq \pi \wedge \pi' \in P_\Pi)$.
(ii) Let $\pi = \langle b_1, ..., b_m \rangle \in \{\pi^- \subset \pi \mid \pi^- \in P_\Pi\}$. Following similar reasoning we can prove that $(\forall \pi, \pi \in \{\pi^- \subseteq \pi \mid \pi^- \in P_\Pi\} \implies \pi \in P_{order})$. This is omitted for space reasons. Therefore $P_{order} = \{\pi^- \subseteq \pi \mid \pi^- \in P_\Pi\}$. □

The branching factor for solving Π^{order} can be as high as $|\pi| = n$, and $(\sum_{i=1}^{n-1} i)$ *order* propositions must be created. This might become an issue when the length of plans is particularly long. We have another compilation introducing additional *skip* actions that allow to omit single actions, in which the order relation is defined only for consecutive actions. In this way the branching factor is reduced to exactly 2 in each step: either the current action or the skip action can be applied. In this paper we only consider Π^{order}. But, we have observed that the impact of using skip actions instead is not significant in our specific experiments.

Top-k Relevant Plans

This section introduces the top-k relevant planning problem. The idea is simple: given a characteristic/condition we are interested on, finding the best *k* plans that meet that condition. We denote the set of all plans that solve a planning task and meet a condition ω as ${P_\Pi}^\omega \subseteq P_\Pi$. More formally:

Definition 6 (Top-k relevant planning problem). *A top-k relevant planning problem is a tuple $\langle \Pi, k, \omega \rangle$, where $\Pi = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ is a planning task, $k \in \mathbb{N}$ and $\omega : P_\Pi \mapsto \{true, false\}$. The goal is to find a set of plans $P_k \subseteq {P_\Pi}^\omega$ such that:*

- *For each $\pi \in P_k$, if there exists a plan π' such that $c(\pi') < c(\pi) \wedge \omega(\pi')$ then $\pi' \in P_k$.*
- *$|P_k| \leq |{P_\Pi}^\omega|$, where $|P_k| < k$ implies $P_k = {P_\Pi}^\omega$.*

This means that we want to find the k plans of least cost that meet a condition ω. In this work we are particularly interested in plans that are perfectly justified. To do so, we extend Katz et al. (2018) iterative top-k planning approach as the Algorithm 1 shows. Initially P_k (solution set) is empty. Then, as long as k relevant plans have not been found, we repeat the following procedure. Get the next plan using an iterative top-k planning approach. If the problem is unsolvable, we return the found plans so far. Otherwise, we check if the plan π meets the condition ω. If it does, it is added to P_k. Finally, a new planning task is defined following the iterative top-k planning procedure to forbid plans. We leave out the details of the reformulation of the planning task since we are using exactly the same approach proposed by Katz et al. (2018).

An open identified issue with this approach is determining when to stop. In many instances, the number of plans that meet a condition might be lower than the number of desired plans k. When the plans must be perfectly justified, an example of this is easy to imagine. For the planning problem shown on the right of Figure 1 there is only one perfectly justified plan. Any other plan, including plans with loops, contains sets of actions that can be eliminated from the plan to get equally valid plans. Since there is an infinite number of plans (with loops) that solve the problem, the described approach would continue trying to find plans to reach k indefinitely. After finding the first relevant plan, a new (loopy or with unnecessary actions) plan will be found on each iteration. It will not be perfectly justified, and therefore it will not be added to the solution set. Then, if $k > 1$, the process

will continue indefinitely, unless it is explicitly stopped using time or memory limits or memory is exhausted. A possible solution would be to incorporate the relevance check into the search process. Specifically, it can be performed when checking the goal condition so that the search can continue when the plan is not relevant. However, it is not trivial to do this without losing completeness (i.e. guarantee that all existing relevant plans can be found), because the fact that a plan is relevant not only depends on the state but also on the path. Then, determining whether or not there are more relevant plans remains a subject of future work. In this work we consider a time bound. But, if the top-k planning system could provide guarantees on loop-less paths completeness would be ensured.

Algorithm 1: IterativeRelevantTopK($\langle \Pi, k, \omega \rangle$)

$P_k \leftarrow \emptyset, \pi \leftarrow \emptyset$
while $|P_k| < k \wedge \neg$*timeout* **do**
  $\pi \leftarrow$ *get_next_plan*(Π)
  if $\pi = \emptyset$ **then**
    return P_k
  end if
  if $\omega(\pi)$ **then**
    $P_k \leftarrow P_k \cup \{\pi\}$
  end if
  $\Pi \leftarrow$ *forbid*(Π, π)
end while
return P_k

Evaluation

Evaluation Setting

For this particular work, we wish to analyse the number of perfectly justified plans found when solving the top-k and top-k relevant planning problems. We perform this analysis over 300 planning tasks from 15 different domains that are widely used in plan and goal recognition research (Ramírez and Geffner 2009; Sohrabi, Riabov, and Udrea 2016; Pereira, Oren, and Meneguzzi 2020). We selected this benchmark[3] and not the standard International Planning Competition (IPC) benchmark for two main reasons. Firstly, IPC tasks are usually difficult to solve optimally, and thus computing large sets of plans using an iterative approach might be too time consuming. Secondly, these are domains and problems were used also in other settings as diverse planning (Roberts, Howe, and Ray 2014).

We modify Katz et al. (2018) software to find relevant plans in an iterative manner, following Algorithm 1. Since we are interested in perfectly justified plans, the condition to meet, ω, will be perfect justification for all the experiments reported. We use our implementation of the Π^{order} compilation to check if found plans are perfectly justified. To do this, we set it to solve the MLR (Minimal Length Reduction) problem. This implementation takes as input a planning task

[3]The set of planning tasks we used will be made publicly available.

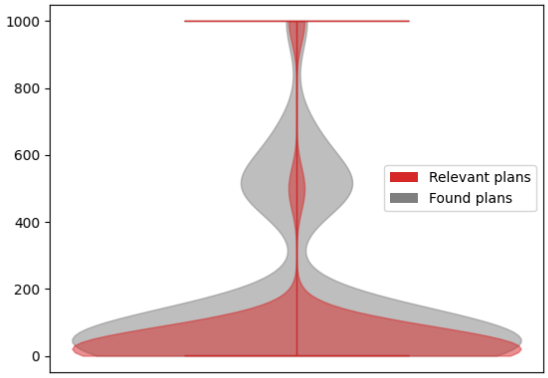

Figure 2: Violin plot of the distribution of the number of plans and relevant plans found in the kitchen domain.

(in PDDL) and a plan, and generates the Π^{order} task (also in PDDL). If the number of actions in the solution of this task is the same as in the original plan, then the plan did not have unnecessary actions. We use the Fast-Downward (FD) planning system (Helmert 2006) to solve the Π^{order} tasks, configured to find an optimal solution using A* as a search algorithm and the h_{max} heuristic (Bonet and Geffner 2001).

The experiments were run on Intel(R) Xeon(R) CPU X3470 @ 2.93GHz machines. We used a time limit of 15 minutes and a memory limit of 14GB to solve each top-k and top-k relevant problem. For each instance of each domain, we solve both the top-k and top-k relevant planning problems for $k \in [1, 5, 10, 50, 100, 500, 1000]$. We did not conduct experiments with larger values of k since most tasks run out of memory or time.

Results

Table 1 shows a summary of the results for the top-k relevant planning problem. For each domain and value of k we report two values: the mean number and standard deviation of plans found before reaching k or the time limit (column Plans), and the average number and standard deviation of those plans that where relevant (column R-Plans). Remember that in this particular setting we consider a plan to be relevant if it is perfectly justified.

When the number of R-Plans is smaller than k, this indicates that on average we were not able to find the desired amount of relevant plans. This happens often in some domains as blocks-world, campus, depots, grid and sokoban. In general, the ratio of Plans to R-Plans decreases as k increases. Currently we can not know if there are enough additional relevant plans to reach k. As explained previously, this is a subject of further research. But we can make some analysis considering specific domain characteristics. For instance, in blocks-world, many actions involving blocks that are not in the goals are not relevant and in general there are few relevant plans. This is reflected on the number of Plans vs. R-Plans for blocks-world.

	$k=5$		$k=10$		$k=50$		$k=100$		$k=500$		$k=1000$	
Domain	Plans	Rel-Plans	Plans	Rel-Plans	Plans	Rel-Plans	Plans	Rel-Plans	Plans	Rel-Plans	Plans	Rel-Plans
blocks	44.1 ± 65.09	4.95 ± 0.22	158.9 ± 125.41	8.55 ± 2.14	255.45 ± 82.27	22.2 ± 14.97	261.55 ± 69.82	22.9 ± 16.57	261.55 ± 69.82	22.9 ± 16.57	261.55 ± 69.82	22.9 ± 16.57
campus	8.2 ± 6.57	5.0 ± 0.0	13.2 ± 6.57	10.0 ± 0.0	120.5 ± 56.7	50.0 ± 0.0	261.15 ± 88.26	97.35 ± 8.16	445.9 ± 52.34	220.85 ± 72.27	445.9 ± 52.34	220.85 ± 72.27
depots	9.05 ± 18.11	5.0 ± 0.0	25.3 ± 52.57	9.9 ± 0.45	83.05 ± 71.18	44.5 ± 13.52	125.55 ± 53.73	87.0 ± 31.79	457.0 ± 94.55	417.25 ± 179.42	839.35 ± 296.94	797.25 ± 379.85
driverlog	5.0 ± 0.0	5.0 ± 0.0	10.9 ± 2.47	10.0 ± 0.0	100.65 ± 100.78	45.95 ± 9.78	149.95 ± 87.45	85.15 ± 29.1	434.2 ± 109.3	343.05 ± 201.86	696.25 ± 332.56	600.1 ± 434.46
dwr	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	67.6 ± 54.17	46.8 ± 9.85	112.6 ± 38.78	91.8 ± 25.24	440.35 ± 99.52	370.4 ± 182.27	721.05 ± 320.51	651.1 ± 409.12
ferry	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	55.0 ± 11.23	50.0 ± 0.0	117.8 ± 29.94	99.0 ± 3.64	420.6 ± 155.36	341.6 ± 165.52	565.8 ± 310.21	471.8 ± 342.68
grid	47.9 ± 63.38	4.55 ± 0.83	69.45 ± 64.73	7.7 ± 2.94	104.15 ± 53.6	14.45 ± 13.78	107.3 ± 51.12	15.95 ± 18.22	107.3 ± 51.12	15.95 ± 18.22	107.3 ± 51.12	15.95 ± 18.22
intruder	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	500.0 ± 0.0	500.0 ± 0.0	1000.0 ± 0.0	1000.0 ± 0.0
kitchen	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	406.15 ± 269.05	38.8 ± 8.79	423.65 ± 244.68	56.3 ± 32.99	563.65 ± 52.84	196.3 ± 228.66	738.65 ± 198.02	371.3 ± 473.33
logistics	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	500.0 ± 0.0	500.0 ± 0.0	1000.0 ± 0.0	1000.0 ± 0.0
miconic	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	395.1 ± 139.83	383.2 ± 124.99	558.5 ± 370.72	518.8 ± 308.3
rover	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	112.25 ± 54.78	99.7 ± 1.34	499.05 ± 47.3	475.2 ± 91.95	914.15 ± 174.57	859.6 ± 266.66
satellite	5.4 ± 0.82	5.0 ± 0.0	11.2 ± 4.02	10.0 ± 0.0	63.5 ± 35.14	50.0 ± 0.0	127.0 ± 59.89	99.3 ± 3.13	524.05 ± 169.89	327.95 ± 115.18	558.85 ± 202.5	361.55 ± 197.39
sokoban	17.8 ± 20.11	5.0 ± 0.0	47.95 ± 43.0	9.45 ± 1.23	92.55 ± 45.05	23.4 ± 15.3	95.45 ± 42.66	26.3 ± 21.15	95.45 ± 42.66	26.3 ± 21.15	95.45 ± 42.66	26.3 ± 21.15
zenotravel	5.0 ± 0.0	5.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	109.3 ± 28.77	99.0 ± 4.47	444.85 ± 130.84	396.55 ± 132.22	716.85 ± 477.12	602.75 ± 342.55

Table 1: Results for top-k relevant plans. ω is perfect justification. Plans column shows the mean number of plans found. R-Plans shows the mean number of relevant plans found.

	$k=5$		$k=10$		$k=50$		$k=100$		$k=500$		$k=1000$	
Domain	TOP-K	TOP-K-R	TOP-K	TOP-K-R	TOP-K	TOP-K-R	TOP-K	TOP-K-R	TOP-K	TOP-K-R	TOP-K	TOP-K-R
blocks	0.61 ± 0.21	61.79 ± 197.31	1.28 ± 0.45	403.32 ± 407.96	13.01 ± 6.0	848.78 ± 194.22	71.33 ± 47.74	892.13 ± 2.83	892.67 ± 4.58	892.13 ± 2.83	892.67 ± 4.58	892.13 ± 2.83
campus	0.6 ± 0.1	3.42 ± 3.01	1.06 ± 0.17	5.32 ± 2.82	7.03 ± 3.05	73.14 ± 47.69	23.96 ± 7.16	305.19 ± 248.23	856.96 ± 77.69	885.98 ± 2.37	893.99 ± 1.86	885.98 ± 2.37
depots	4.46 ± 4.77	11.42 ± 24.09	4.54 ± 4.7	57.11 ± 197.6	7.04 ± 6.08	156.38 ± 318.39	19.93 ± 29.91	167.9 ± 313.52	183.9 ± 363.04	290.95 ± 307.78	227.08 ± 393.18	429.71 ± 274.82
driverlog	0.32 ± 0.22	1.46 ± 0.28	0.43 ± 0.29	3.14 ± 1.58	3.96 ± 7.23	188.14 ± 356.62	17.23 ± 35.84	239.66 ± 381.36	352.7 ± 431.0	453.29 ± 398.78	446.25 ± 457.37	543.48 ± 348.29
dwr	1.65 ± 1.74	4.46 ± 3.56	1.66 ± 1.74	6.83 ± 5.06	2.7 ± 3.2	113.46 ± 265.99	10.0 ± 25.22	135.88 ± 259.73	358.41 ± 447.01	547.2 ± 304.83	358.45 ± 446.98	718.61 ± 237.36
ferry	0.25 ± 0.04	1.6 ± 0.18	0.26 ± 0.04	2.72 ± 0.31	4.11 ± 9.19	26.6 ± 45.9	39.61 ± 79.46	165.11 ± 296.76	499.92 ± 399.58	609.16 ± 344.1	691.51 ± 355.47	738.12 ± 247.45
grid	2.17 ± 2.98	277.25 ± 412.79	7.07 ± 11.28	459.18 ± 438.24	203.9 ± 247.39	815.92 ± 200.78	609.08 ± 324.62	880.73 ± 23.99	883.33 ± 11.98	880.73 ± 23.99	883.33 ± 11.98	880.73 ± 23.99
intruder	0.2 ± 0.02	1.47 ± 0.06	0.19 ± 0.02	2.52 ± 0.11	0.2 ± 0.02	10.92 ± 0.48	0.21 ± 0.02	21.44 ± 0.96	0.25 ± 0.01	105.21 ± 4.82	0.28 ± 0.01	209.76 ± 9.63
kitchen	0.22 ± 0.04	1.44 ± 0.25	0.31 ± 0.12	2.78 ± 0.36	1.47 ± 1.02	579.35 ± 426.1	5.01 ± 3.58	583.86 ± 419.8	291.94 ± 228.38	619.15 ± 370.45	579.58 ± 436.1	663.62 ± 308.28
logistics	0.22 ± 0.02	1.55 ± 0.04	0.21 ± 0.02	2.63 ± 0.06	0.22 ± 0.02	11.36 ± 0.24	0.23 ± 0.02	22.3 ± 0.47	0.29 ± 0.03	109.28 ± 2.28	0.33 ± 0.03	218.01 ± 4.63
miconic	0.27 ± 0.06	1.47 ± 0.1	0.42 ± 0.23	2.85 ± 0.49	4.32 ± 4.62	16.91 ± 6.6	23.14 ± 28.47	48.37 ± 32.12	579.04 ± 391.23	642.95 ± 330.82	700.86 ± 334.11	771.58 ± 228.86
rover	0.42 ± 0.48	1.74 ± 0.53	0.42 ± 0.48	2.86 ± 0.57	0.43 ± 0.48	11.76 ± 1.05	0.58 ± 0.81	66.12 ± 192.94	48.66 ± 199.57	190.02 ± 236.65	223.99 ± 397.08	391.2 ± 288.52
satellite	0.28 ± 0.14	1.47 ± 0.31	0.38 ± 0.21	2.83 ± 1.49	3.18 ± 2.55	22.86 ± 30.06	12.89 ± 16.98	84.58 ± 188.72	571.24 ± 328.02	817.98 ± 204.97	846.09 ± 198.99	848.75 ± 153.6
sokoban	6.99 ± 10.57	55.75 ± 115.06	19.87 ± 29.48	297.85 ± 374.79	310.81 ± 312.7	811.89 ± 177.28	635.6 ± 297.76	883.9 ± 10.72	882.85 ± 13.61	883.9 ± 10.72	882.85 ± 13.61	883.9 ± 10.72
zenotravel	0.61 ± 0.58	1.76 ± 0.67	0.73 ± 0.78	2.86 ± 1.02	4.2 ± 6.19	15.09 ± 7.91	15.18 ± 24.28	80.94 ± 192.4	379.63 ± 384.06	505.01 ± 378.75	614.63 ± 412.74	687.66 ± 308.04

Table 2: Time results for the top-k and top-k relevant problems.

	$k=5$		$k=10$		$k=50$		$k=100$		$k=500$		$k=1000$	
Domain	K	K-R	K	K-R	K	K-R	K	K-R	K	K-R	K	K-R
blocks	0	1	0	7	0	19	0	20	20	20	20	20
campus	0	0	0	0	0	0	0	2	13	20	20	20
depots	0	0	0	1	0	3	0	3	4	4	5	5
driverlog	0	0	0	0	0	4	0	5	7	9	10	10
dwr	0	0	0	0	0	2	0	2	8	8	8	12
ferry	0	0	0	0	0	0	0	2	9	12	15	16
grid	0	6	0	10	2	18	10	20	20	20	20	20
intruder	0	0	0	0	0	0	0	0	0	0	0	0
kitchen	0	0	0	0	0	13	0	13	0	13	13	13
logistics	0	0	0	0	0	0	0	0	0	0	0	0
miconic	0	0	0	0	0	0	0	0	11	12	15	16
rover	0	0	0	0	0	0	0	1	1	2	5	5
satellite	0	0	0	0	0	0	0	1	9	18	19	19
sokoban	0	0	0	5	3	17	10	20	20	20	20	20
zenotravel	0	0	0	0	0	0	0	1	6	9	13	14

Table 3: Number of timeouts recorded for the top-k and top-k relevant planning problems

There are several domains for which a really large amount of plans must be found to find a small amount of relevant plans, and in most cases the desired number k is never achieved. In contrast, there are domains like logistics where all found plans are relevant. This occurs because a single plan for logistics can be used to derive additional plans by reducing it to a partial order plan. Each plan derived from the first (optimal) plan is perfectly justified because this is an unit-cost domain. Even more, in all our instances the all the desired (k) plans are derived from the first plan found. Figure 2 shows the distribution of the number of plans and relevant plans in the kitchen domain. This is one of the domains where the number of non-relevant plans is especially high. The width of the shadows shows that more data points share that value. The upper and lower horizontal lines represent the maximum an minimum values of the results respectively. The vertical lines correspond to the medians. The median of the first is 50 while for the second is 10. It is straightforward to verify that the presence of relevant plans considerably decreases as larger k values are required. As expected, irrelevant information in plans grows as more iterations are carried out.

Table 2 shows the required time (in seconds) to solve the top-k (TOP-K) and top-k relevant (TOP-K-R) problems for different values of k. As expected, the time spent on verifying that the found plans are relevant is high, given that guaranteeing a plan does not contain redundant actions is NP-complete (Fink and Yang 1992; Nakhost and Müller 2010). There are domains, as blocks-world, grid and kitchen, where most of the found plans are not relevant, and therefore this time can be better justified than for other domains, like logistic, where all found plans are relevant. This can motivate further investigation on how to determine if the extra time needed to solve the top-k relevant problem is worth it depending on domain/problem characteristics.

Table 3 shows the number of timeouts for each domain and for top-k (K) and top-k relevant (K-R). As expected, the number of timeouts for the relevant variant greatly exceeds the one for the regular top-k. Note that with the current approach if there are not k relevant plans our approach always stops due to the time limit.

Related Work

There exists several approaches to top-k planning (Riabov, Sohrabi, and Udrea 2014; Katz et al. 2018; Speck, Mattmüller, and Nebel 2020), but so far all of them are concerned with the problem of finding the best k plans for a planning task, and not much has been studied the semantics and utility of these plans. Closer works to the idea intro-

duced in this paper are those about diverse planning (Srivastava et al. 2007; Roberts, Howe, and Ray 2014; Katz and Sohrabi 2020), where resulting plans are required to be different and similarity metrics are defined between plans. We consider the notion of plan relevance that applies only to single plans and can be considered as complementary to plan diversity. In applications of top-k planning one might want to generate plans that are at the same time diverse and relevant.

In this work, the condition for plan relevance is based on perfectly justified plans. Specifically we filter those plans that are not perfectly justified in a post-optimization step. There are some other works on plan post-optimization. Fink and Yang [1992] formalized different notions of plan justifications and provided complexity results for them. Specifically, they defined greedily justified actions as those that make the plan invalid when they are removed from it, and perfectly-justified plans as those with no redundant actions. Nakhost and Müller [2010] proposed Action Elimination, an algorithm based on greedy justification, and an additional technique based on plan neighborhood graph search. There are methods based on identifying redundant actions and non-optimal sub-plans by analyzing action dependencies, independencies (Chrpa, McCluskey, and Osborne 2012b), by checking pairs of inverse actions (Chrpa, McCluskey, and Osborne 2012a), and SAT-based approaches (Balyo, Chrpa, and Kilani 2014; Muise, Beck, and McIlraith 2016). The work in this paper is closely related to all these works with the difference that we approach the problem using Automated Planning. However, our method could be replaced by any other.[4]

There are also techniques that remove irrelevant information at preprocessing. For instance, Nebel, Dimopoulos, and Koehler 1997 proposed heuristics for selecting relevant information based on minimizing the number of initial facts by computing a fact generation tree going backwards from the goals; and a recent approach (Silver et al. 2020) learns convolutional graph neural networks to predict subsets of objects that are sufficient for solving the planning task. Approaching the problem at preprocessing has the additional advantage that it can make easier the planning process. This is specially interesting when the number of objects is very large. In this case, most modern heuristic planners that ground the actions over objects during preprocessing scale poorly. This is also one of the motivations for recent research on lifted planning (Corrêa et al. 2020), abstractions that simplify the problem (Fuentetaja and de la Rosa 2016) and some approaches based on generalized planning, as the aforementioned work of Silver et al.. We believe that studying techniques that can be applied in preprocessing or even during search in the context of top-k relevant planning would be an interesting research direction.

[4]We have some results comparing it to the Max-SAT approach of Balyo, Chrpa, and Kilani (2014), showing both very similar performance times.

Conclusions and Future Work

In this work we proposed the idea of top-k relevant plans as plans that meet some extra condition in the context of top-k planning. Specifically, we consider plans that are perfectly justified (i.e. they do not contain subsets of actions that can be removed while maintaining plan validity). We have incorporated this notion into a top-k planner as a filtering step which is applied every time a new plan is found. The filtering process is posed as an Automated Planning task. In particular, given a plan we show how to create planning tasks to solve Minimal Length Reduction (MLR) problem. We have performed experiments in a variety of domains. Many problems have few relevant plans or the number of plans found in order to find the desired relevant plans is high.

Regarding the proposed approach, determining whether there exist additional relevant plans to those already found is an interesting line of future work. We also plan to consider our work in conjunction to additional techniques that can be applied to filter irrelevant information in a preprocessing step or during search. Additionally, we wish to consider different definitions of relevance. Finally, we want to study the impact of plan relevance in applications of top-k planning as goal recognition.

Acknowledgments

This work has been partially funded by FEDER/Ministerio de Ciencia, Innovación y Universidades - Agencia Estatal de Investigación/TIN2017-88476-C2-2-R, RTC-2016-5407-4, and the Madrid Government (Comunidad de Madrid-Spain) under the Multiannual Agreement with UC3M in the line of Excellence of University Professors (EPUC3M17), and in the context of the V PRICIT (Regional Programme of Research and Technological Innovation)."

References

Balyo, T.; Chrpa, L.; and Kilani, A. 2014. On different strategies for eliminating redundant actions from plans. In *Seventh Annual Symposium on Combinatorial Search*.

Boddy, M. S.; Gohde, J.; Haigh, T.; and Harp, S. A. 2005. Course of Action Generation for Cyber Security Using Classical Planning. In *ICAPS 2005*, 12–21.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*, 129(1): 5–33.

Borrajo, D.; Gopalakrishnan, S.; and Potluru, V. K. 2020. Goal Recognition via Model-based and Model-free Techniques. In *ICAPS Workshop on Planning for Financial Services (FinPlan)*.

Borrajo, D.; and Veloso, M. 2020. Domain-independent Generation and Classification of Behavior Traces. In *ICAPS Workshop on Planning for Financial Services (FinPlan)*.

Borrajo, D.; Veloso, M.; and Shah, S. 2020. Simulating and Classifying Behavior in Adversarial Environments Based on Action-State Traces: An Application to Money Laundering. In *Proceedings of the 2020 ACM International Conference on AI in Finance*. New York (EEUU).

Celorrio, S. J.; Haslum, P.; Thiebaux, S.; et al. 2013. Pruning bad quality causal links in sequential satisfying planning.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012a. Determining redundant actions in sequential plans. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, volume 1, 484–491. IEEE.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012b. Optimizing plans through analysis of action dependencies and independencies. In *Twenty-Second International Conference on Automated Planning and Scheduling*.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Frances, G. 2020. Lifted successor generation using query optimization techniques. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 80–89.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

Fink, E.; and Yang, Q. 1992. Formalizing plan justifications.

Fuentetaja, R.; and de la Rosa, T. 2016. Compiling irrelevant objects to counters. special case of creation planning. *AI Communications*, 29(3): 435–467.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Katz, M.; and Sohrabi, S. 2020. Reshaping diverse planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9892–9899.

Katz, M.; Sohrabi, S.; and Udrea, O. 2020. Top-Quality Planning: Finding Practically Useful Sets of Best Plans. In *AAAI 2020*, 9900–9907.

Katz, M.; Sohrabi, S.; Udrea, O.; and Winterer, D. 2018. A Novel Iterative Approach to Top-k Planning. In *ICAPS 2018*, 132–140.

Muise, C.; Beck, J. C.; and McIlraith, S. A. 2016. Optimal partial-order plan relaxation via MaxSAT. *Journal of Artificial Intelligence Research*, 57: 113–149.

Mund, S.; Vallati, M.; and McCluskey, T. L. 2020. An Exploration of the Use of AI Planning for Predicting Stock Market Movement. In *ICAPS 2020 Workshop on AI Planning for Financial Services (FinPlan)*.

Nakhost, H.; and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, 121–128. AAAI.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *European Conference on Planning*, 338–350. Springer.

Pereira, R. F.; Oren, N.; and Meneguzzi, F. 2020. Landmark-based approaches for goal recognition as planning. *Artificial Intelligence*, 279: 103217.

Pozanco, A.; Polychroniadou, A.; Magazzeni, D.; and Borrajo, D. 2021. Proving Security of Cryptographic Protocols using Automated Planning. In *ICAPS Workshop on Planning for Financial Services (FinPlan)*.

Ramírez, M.; and Geffner, H. 2009. Plan recognition as planning. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Ramírez, M.; and Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Riabov, A.; Sohrabi, S.; and Udrea, O. 2014. New algorithms for the top-k planning problem. In *Proceedings of the scheduling and planning applications workshop (spark) at the 24th international conference on automated planning and scheduling (icaps)*, 10–16.

Roberts, M.; Howe, A. E.; and Ray, I. 2014. Evaluating diversity in classical planning. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*.

Silver, T.; Chitnis, R.; Curtis, A.; Tenenbaum, J.; Lozano-Perez, T.; and Kaelbling, L. P. 2020. Planning with learned object importance in large problem instances using graph neural networks. *arXiv preprint arXiv:2009.05613*.

Sohrabi, S.; Riabov, A.; and Udrea, O. 2016. Plan Recognition as Planning Revisited. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic Top-k Planning. In *AAAI 2020*, 9967–9974.

Srivastava, B.; Nguyen, T. A.; Gerevini, A.; Kambhampati, S.; Do, M. B.; and Serina, I. 2007. Domain Independent Approaches for Finding Diverse Plans. In *IJCAI*, 2016–2022.