Artificial Intelligence 173 (2009) 722–747

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

The factored policy-gradient planner[☆]

Olivier Buffet[a,*], Douglas Aberdeen[b]

[a] LORIA-INRIA, Nancy University, Nancy, France
[b] Google Inc., Zurich, Switzerland

ARTICLE INFO

Article history:
Received 8 October 2007
Received in revised form 31 October 2008
Accepted 9 November 2008
Available online 27 November 2008

Keywords:
Concurrent probabilistic temporal planning
Reinforcement learning
Policy-gradient
AI planning

ABSTRACT

We present an any-time concurrent probabilistic temporal planner (CPTP) that includes continuous and discrete uncertainties and metric functions. Rather than relying on dynamic programming our approach builds on methods from stochastic local policy search. That is, we optimise a parameterised policy using gradient ascent. The flexibility of this policy-gradient approach, combined with its low memory use, the use of function approximation methods and factorisation of the policy, allow us to tackle complex domains. This factored policy gradient (FPG) planner can optimise steps to goal, the probability of success, or attempt a combination of both. We compare the FPG planner to other planners on CPTP domains, and on simpler but better studied non-concurrent non-temporal probabilistic planning (PP) domains. We present FPG-IPC, the PP version of the planner which has been successful in the probabilistic track of the fifth international planning competition.

1. Introduction

Only a few planners have attempted to handle concurrent probabilistic temporal planning (CPTP) domains in their most general form. These tools have been able to produce good or optimal policies for relatively small problems. We designed the factored policy gradient (FPG) planner with the goal of creating tools that produce good policies in real-world domains with complex features. Such features may include metric functions (resources for example), concurrent actions, uncertainty in the outcomes of actions *and* uncertainty in the duration of actions.

In a single paragraph, our approach is to: 1) use gradient ascent for local policy search; 2) factor the policy into simple approximate policies for starting each action; 3) base policies on important elements of state only (implicitly aggregating similar states); 4) estimate gradients using Monte-Carlo style algorithms that allow arbitrary distributions; and 5) optionally parallelising the planner.

The AI planning community is familiar with the value-estimation class of reinforcement learning (RL) algorithms, such as RTDP [1], and arguably AO* [2]. These algorithms represent probabilistic planning problems as a state space and estimate the long-term value, utility, or cost of choosing each action from each state [3,4]. The fundamental disadvantage of such algorithms is the need to estimate the values of a huge number of state-action pairs. Even algorithms that prune most states still fail to scale due to the exponential increase of important states as the domains grow. There is a wealth of literature on the use of function approximation for estimating state-action values (e.g., [5,6]), however this has been little adopted (see [7] for an example) in the planning community, perhaps due to the difficulty of interpreting such approximated policies.

☆ The majority of this work was performed while the authors were employed by National ICT Australia.
* Corresponding author.
E-mail addresses: olivier.buffet@loria.fr (O. Buffet), doug.aberdeen@google.com (D. Aberdeen).
URLs: http://www.loria.fr/~buffet/ (O. Buffet), http://sml.nicta.com.au/~daa (D. Aberdeen).

0004-3702/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2008.11.008

On the other hand, the FPG planner borrows from policy-gradient (PG) reinforcement learning [8–11]. This class of algorithms does not estimate state-action values, and thus memory use is not directly related to the size of the state space. Instead, policy-gradient RL algorithms estimate the gradient of the long-term average reward of the process. In the context of stochastic shortest path problems, which covers most probabilistic planning problems, we can view this as estimating the gradient of the long-term value of only the initial state. Gradients are computed with respect to a set of real-valued parameters governing the choice of actions at each decision point. These parameters summarise the policy, or plan,[1] of the system. As distinct from value-based approaches, even those using function approximation, the parameters do not encode the absolute value of actions or plans. Instead they encode only *relative* merit of each action. Hence we hope to achieve a compact policy representation. Stepping the parameters in the direction of the gradient increases the long-term average reward, improving the policy. Also, PG algorithms are guaranteed to converge to at least a local maximum when using approximate policy representations, which is necessitated when the state space is continuous, otherwise infinite, or simply very large. Our setting permits an infinite state space when action durations are modelled by continuous distributions.

The policy takes the form of a function that accepts an observation of the planning state as input, and returns a probability distribution over currently legal actions. The policy parameters modify this function. In other words, the policy parameters tune the shape of the probability distributions over actions, given the current planning state observation. In our temporal planning setting, an *action* is defined as a single grounded durative action (in the planning domain definition language (PDDL) 2.1 sense [12]). A *command* is defined as a decision to start 0 or more actions concurrently. The command set is therefore at most the power set of actions that could be started at the current decision-point state.

From this definition it is clear that the size of the policy, even without learning values, can grow exponentially with the number of actions. We combat this command space explosion by factoring the parameterised policy into a simple policy for each action. This is essentially the same scheme explored in the multi-agent policy-gradient RL setting [13,14]. Each action has an independent agent/policy that implicitly learns to coordinate with other action policies via global rewards for achieving goals. By doing this, the number of policy parameters—and thus the total memory use—grows only linearly with the length of the grounded input description.

An advantage of using function approximators is the ability to generalise learned knowledge. If the starting state changes, or an exogenous event occurs, or we modify the current plan by direct interaction, FPG can return a new suggested action effectively instantly. The quality of the suggested action will depend on how different the current state is to one that it has frequently encountered in training. This is an important quality for mixed-initiative planning [15].

At the same time, this policy-gradient has the advantage of not only ignoring irrelevant states—because of an implicit reachability analysis—but also of focusing its effort on states which are the most relevant to the best policy, which are naturally encountered more frequently. This feature is reminiscent of real-time dynamic programming [1].

Our first parameterised action policy is a simple linear function approximator that takes the truth value of the predicates at the current planning state, and outputs the probability of starting the command. A criticism of policy-gradient RL methods compared to search-based planners—or even to value-based RL methods—is the difficulty of translating vectors of parameters into a human readable plan. Thus, the second parameterised policy we explore is a readable decision tree of high-level planning strategies. Our non-concurrent, non-temporal version of FPG for the International Planning Competition (IPC), could be considered a third form of policy that ensures that only one action out of many eligible actions are chosen. Finally, we will describe how elements of the Relational Online Policy Gradient (ROPG) planner [16], can be viewed as an FPG style parameterised policy.

We believe that the contribution of this paper is an exploration of how existing Monte-Carlo local optimisation methods can feed into planning under uncertainty. In summary, the local optimisation and factored policies framework allow good policies to be found for very rich domains. Sometimes this is at the cost of long optimisation times or local minima. However, we demonstrate that the FPG approach can find optimal policies where other state-of-the-art planning methods, e.g., replanning, can fail. There are many alternate local optimisation methods and parameterisations that could yield interesting results, those presented here should be considered as useful examples.

The paper starts with a background section introducing Markov decision processes (MDPs), policy-gradient algorithms for MDPs and related work. Section 3 describes FPG, including example function approximators and some implementation details. Experimental results from Section 4 show the overall quality of this approach, pinpointing its main strengths and weaknesses.

2. Background

We describe some relevant background in planning, Markov decision processes, policy-gradient algorithms and previous probabilistic planning approaches.

[1] We will generally prefer the term *policy* over *plan* to mean the final output of the planning phase. In a probabilistic setting plans change, policies do not.

```
<probabilistic>
  <outcome label="HeavyTraffic" probability="0.5">
    <effect>
      <delay lambda="7.07e-04" type="exponential"/>
      <functionEffect type="increase">
        <function name="LapsDone"/><number>30</number>
      </functionEffect>
      <predicate name="Racing" negated="true"/>
      <functionEffect type="decrease">
        <function name="Fuel"/><number>30</number>
      </functionEffect>
    </effect>
  </outcome>
```

Fig. 1. A snippet in our XML format of a racing car domain, showing a probabilistic effect with a discrete probability outcome and continuous probability delay.

2.1. Concurrent probabilistic temporal planning (CPTP)

FPG's input language is the temporal STRIPS fragment of PDDL 2.1 but extended with probabilistic outcomes and uncertain durations, as in PPDDL [17,18]. In particular, we support continuous uncertain durations, functions, at-start, at-end, over-all conditions, and finite probabilistic action outcomes. In addition, we allow effects (probabilistic or otherwise) to occur at any time within an action's duration. FPG's input syntax is actually XML with a schema designed to map almost directly to PPDDL (see Fig. 1). Our PPDDL to XML translator grounds actions and flattens nested probabilistic statements to a discrete distribution of action outcomes with delayed effects.

Grounded actions are the basic planning unit. An action is *eligible* to begin when its preconditions are satisfied. It is possible that certain combinations of eligible actions may be mutually exclusive. We will return to this possibility later. Action execution may begin with at start effects. Execution then proceeds to the next probabilistic event, an outcome is sampled, and the outcome effects are queued for the appropriate times. We use a sampling process rather than enumerating outcomes because it will transpire that we only need to *simulate* executions of the plan in order to estimate the necessary gradients. A benefit of this approach is that we can sample from both continuous and discrete distributions, whereas enumerating continuous distributions is not possible.[2]

With N eligible actions there are up to 2^N possible commands. Current planners explore this command space systematically, attempting to prune commands via search or heuristically. When combined with probabilistic outcomes the state space explosion cripples existing planners with just a few tens of actions. We deal with this explosion by factorising the overall policy into independent policies for each action. Each policy learns whether to start its associated action given the current predicate values, independent of the decisions made by the other action policies. This idea alone does not simplify the problem. Indeed, if the action policy approximations were sufficiently rich, and all receive the same state observation, they could learn to predict the decision of the other actions and still act optimally. The significant reduction in complexity arises from using approximate policies, which implicitly assumes similar states will have similar policies.

The FPG planner aims to produce good plans in very rich and large domains. It is not, however, complete in the sense that it will always return a solution if one exists. In particular, FPG shares the completeness problems found in other temporal planners [19,20] that arise when decisions to start new actions are restricted to times where an event is already queued to occur, that is, *at happenings*. Since such examples are not frequent, and existing solutions are either computationally prohibitive or require significant restrictions on domains (e.g., Temporal Graphplan (TGP) style actions only [20]), we choose not to attempt guaranteed completeness for FPG.

2.2. Probabilistic planning

Although FPG was initially developed with the objective of handling concurrent probabilistic temporal planning (CPTP) problems, a simplified version called FPG-IPC participated in the probabilistic track of the fifth International Planning Competition (IPC-5) in 2006. We detail the simplifications in Section 3.2.3.

This *probabilistic planning* (PP) setting can be seen as a restriction of CPTP, the objective being to maximise the probability of reaching the goal. Candidate planners had to process PPDDL specifications using `:adl` requirements [17,21].

2.3. Previous work

Previous probabilistic temporal planners include DUR [22], Prottle [4], and a Military Operations (MO) planner [23]. All these algorithms use some optimised form of dynamic programming (either RTDP [1] or AO* [2]) to associate values with each state-action pair. However, this requires that values be stored for each encountered state. Even though these algorithms

[2] We sample integer times, and there is a maximum permitted makespan, so these distributions are in reality still finite, but extremely large.

prune off most of the state space, their ability to scale is still limited by memory size. Tempastic [24] uses the generate, debug, and repair planning paradigm. It overcomes the state space problem by generating decision tree policies from *sample* trajectories that follow good deterministic policies, and repairing the tree to cope with uncertainty. This method may suffer in highly non-deterministic domains, but is a rare example of an approach that also permits modelling continuous distributions for durations. Prottle, DUR, and Tempastic minimise either plan duration or failure probability. The FPG planner allows for simple trade-offs of these metrics.

The 2004 and 2006 probabilistic tracks of the International Planning Competition (IPC) represent a cross section of recent approaches to *non-temporal* probabilistic planning. Along with the FPG planner, other entrants included FOALP, Paragraph and sfDP. FOALP [25] solves a first order logic representation of the underlying *domain* MDP, prior to producing plans for specific problems drawn from that domain. Paragraph [26] is based on Graphplan extended to a probabilistic framework. sfDP [27] uses a symbolic form of dynamic programming based on Algebraic Decision Diagrams (ADDs). A surprisingly successful approach to the competition domains was FF-rePlan [28], winning the 2004 competition. A subsequent version could have also achieved first place at the 2006 competition. FF-rePlan uses the FF heuristic [29] to quickly find a potential short path to the goal. It does so by creating a deterministic version of the domain. Thus, it does not attempt to directly optimise either the probability of reaching the goal or the cost-to-go. In practice though, for many domains the ability to reach the goal at all leads to good performance. However, any replanning approach can perform poorly when the cost of failure must be taken into account [30].

Policy-gradient RL for multiple-agents MDPs is described by [13,14], providing a precedent for factoring policy-gradient RL policies into independent “agents” for each action. This paper also builds on earlier work presented by [31,32].

The challenge of learning to generalise across problems within a particular domain is another use of function approximation for generalisation. The classical view of learning for planning is to acquire knowledge about a given domain by (1) planning in small problem instances; and (2) reusing this knowledge—such as heuristic rules—to plan in larger domains [33]. In this direction there have been attempts at using Relational Reinforcement Learning (RRL) to find generic policies for planning problems expressed in first-order logic [7,16,34]. But this remains a very challenging task: the search space is much larger and abstract notions may be required, such as `above(A,B)` and `numberofblockson(X,N)` in the Blocksworld. The Approximate Policy Iteration algorithm used in Classy is similar to FPG in the sense that it avoids computing a value function and heavily relies on Monte-Carlo simulations, but does not attempt to factorise the policy. The Relational Online Policy-Gradient (ROPG) [16] uses exactly the same policy-gradient algorithm as FPG. ROPG learns which higher-order control strategy to follow in a given state. While the contribution of ROPG is to develop possible control strategies, the policy-gradient component is needed to learn which strategies work and when.

2.4. Markov decision processes and policy-gradient algorithms

We describe our Markov decision process (MDP) framework and then give an overview of gradient ascent and policy-gradient algorithms.

2.4.1. Markov decision processes

A finite partially observable Markov decision process consists of: a (possibly infinite) set of states $s \in \mathcal{S}$; a finite set of actions $\mathbf{c} \in \mathcal{C}$, that correspond to our *command* concept; probabilities $\mathbb{P}[s' \mid s, \mathbf{c}]$ of making state transition $s \to s'$ under command $\mathbf{c}$; a reward for each state $r(s) : \mathcal{S} \to \mathbb{R}$;[3] and a finite set of observation basis vectors $\mathbf{o} \in \mathcal{O}$ used by action policies in lieu of complete state descriptions.

We can trade off the complexity of the action policies with the amount of state information provided as an observation. As we provide more and more state information, the policies can become richer and richer. At the extreme end of this spectrum we provide a unique command for every state (essentially a state $\to$ command lookup table). At the other end, the policy knows nothing about the current state and can only generalise across all states by estimating the best stationary policy.

However, in the case of FPG with a linear approximator, we construct policy observation vectors from the state as follows. Each predicate value—and only the predicate values—becomes an observation bit. We set the bit to 1 for asserted predicate, and 0 otherwise. A constant 1 observation bit is also provided as a bias element to assist the linear approximator.

Goal states occur when the predicates and functions match a PPDDL goal state specification. From *failure states* it is impossible to reach a goal state, usually because time or resources have run out, but it may also be due to an at-end or over-all condition being invalid. These two classes of end state form the set of *terminal* states, ending plan simulation.

Policies are stochastic, mapping the observation vector $\mathbf{o}$, generated from the current planning state, to a probability distribution over commands. Fundamentally, this is necessary to enable the exploration of the command space. Over the course of optimisation, we hope that the policy distribution becomes increasingly peaked over equally optimal commands.

Let N be the number of grounded actions available to the planner. For FPG a command $\mathbf{c}$ is a binary vector of length N. An entry of 1 at index $n \in \{1, \dots, N\}$ means ‘Yes’ begin action n, and a 0 entry means ‘No’ do not start action n. The probability of a command is $\mathbb{P}[\mathbf{c} \mid \mathbf{o}; \boldsymbol{\theta}]$, where conditioning on $\boldsymbol{\theta}$ reflects the fact that the policy is tuned by a set of real

[3] This work remains valid when the reward depends on a complete transition $r(s, \mathbf{c}, s')$. We consider a simpler setting for readability reasons.

```
1: Initialisation:
2: $t \leftarrow 0$
3: repeat
4:     $t \leftarrow t+1$
5:     $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} + \alpha \nabla R(\boldsymbol{\theta}_{t-1})$
6: until stoppingCriterion($\epsilon, \ldots$)
7: Return $\boldsymbol{\theta}_t$
```

Algorithm 1. Generic **gradientAscent**$(R, \boldsymbol{\theta}_0, \alpha, \epsilon)$.

valued parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. Commands with non-eligible actions are guaranteed to have probability 0. We assume that all stochastic policies (i.e., any values for $\boldsymbol{\theta}$) reach terminal states in finite time when executed from s_0. This is enforced by limiting the maximum makespan of a plan.

FPG's optimisation criteria is very general: it maximises the long-term average reward

$$R(\boldsymbol{\theta}) := \lim_{T\to\infty} \frac{1}{T} \mathbb{E}_{\boldsymbol{\theta}} \left[\sum_{t=0}^{T-1} r(s_t) \right], \tag{1}$$

where the expectation $\mathbb{E}_{\boldsymbol{\theta}}$ is over the distribution of state trajectories $\{s_0, s_1, \ldots\}$ induced by the current joint policy. In the context of planning, the instantaneous reward provides the action policies with a measure of progress toward the goal. A simple reward scheme is to set $r(s) = 1000$ for all states s that represent the goal state, and 0 for all other states. To maximise $R(\boldsymbol{\theta})$, *goal* states must be reached as frequently as possible. This has the desired property of simultaneously minimising steps to goal and maximising the probability of reaching the goal because failure states achieve no reward.

There are some pitfalls to avoid when describing a reward scheme. For example, if we have a large negative reward for failure, the optimisation may choose to extend the plan execution as long as possible to reduce the frequency of negative rewards.

We also provide intermediate rewards for progress toward the goal. These additional *shaping* rewards provide an immediate reward of 1 for achieving a goal predicate, and -1 for every goal predicate that becomes unset. Shaping rewards are provably "admissible" in the sense that they do not change the optimal policy [35]. The shaping assists convergence for domains where long chains of actions are necessary to reach the goal and proved important in achieving good results in IPC domains. The reward shaping helps reinforcements occur as soon as an action moves the system closer to (or more distant from) the goal, not just when the goal is reached. This helps solve the reward assignment problem.

2.4.2. Gradient algorithms

We want to maximise $R(\boldsymbol{\theta})$ by *gradient ascent*. That is, repeatedly computing gradients $\nabla R(\boldsymbol{\theta})$ and stepping the parameters in that direction. In our setting, the gradient is a vector operator mapping any differentiable function $R : \mathbb{R}^p \to \mathbb{R}$ to $\nabla R : \mathbb{R}^p \to \mathbb{R}^p$ defined as

$$\nabla R(\theta_1, \ldots, \theta_p) = \begin{bmatrix} \frac{\partial R}{\partial \theta_1}(\theta_1, \ldots, \theta_p) \\ \vdots \\ \frac{\partial R}{\partial \theta_n}(\theta_1, \ldots, \theta_p) \end{bmatrix}. \tag{2}$$

A gradient ascent is an iterative algorithm used to find a local maximum of $R(\boldsymbol{\theta})$. The principle is to compute a sequence $\{\boldsymbol{\theta}_t\}_{t\in\mathbb{T}}$ by following, at each parameter values point $\boldsymbol{\theta}_t$, the direction of the gradient at this point. Algorithm 1 gives a very generic overview of the process, which requires: a differentiable function R, a starting point $\boldsymbol{\theta}_0$, a step-size $\alpha > 0$ and, often, a threshold $\epsilon > 0$ used by some stopping criterion. The stopping criterion will typically be a function of the parameters, the gradients, time, and/or the number of steps.

The details of how Algorithm 1 can be implemented constitute an entire complex field. FPG is actually an *online* gradient ascent, where the function R has additional external inputs which effect the gradient from step to step. In FPG's case the additional inputs are the observation of the planning state $\mathbf{o}_t$. However, as it shall transpire, the Markov nature of the process means a weighted average of these direct "policy" gradients with specific observations, converges to an approximate estimate of the gradient of $R(\boldsymbol{\theta})$, and thus can be safely used in Algorithm 1.

The gradients are stochastic because the planning domains contain uncertainty and we are sampling from distributions over actions. The combination of being online and stochastic means that many of the more advanced gradient ascent algorithms such as line searches, BFGS, and other conjugate methods cannot be immediately applied [36]. However, we will comment on the use of some approximate second-order gradient ascent methods in the discussion.

Gradient optimisation methods perform *local* search. That is, they greedily step in the direction of the gradient until reaching a maximum which may not be the global maximum. We accept this possibility because gradient ascent is far more tractable than global optimisation methods, such as tabula rasa dynamic programming variants. In practice, gradient ascent may achieve good results on a planning domain which a global optimiser cannot return any policy for, typically because memory runs out.

2.4.3. Introduction to policy-gradient algorithms

Computing the gradient of the long-term average reward $R(\boldsymbol{\theta})$ in a closed-loop Markov decision process is not entirely trivial. However, several papers have solved this problem with practical algorithms [8,9,11,37]. The benefits of this approach compared to other reinforcement learning methods are: 1) *local* convergence under function approximation and partial observability; 2) memory usage that is linear in the number of parameters in the policy function; 3) a convenient (although not optimal) implicit approach to the exploration versus exploitation trade-off.

We will use Baxter and Bartlett's approach to policy-gradient algorithms [37]. For historical reasons we begin with an introduction based on Williams' REINFORCE algorithm [8], which is credited with being the first policy-gradient method in a reinforcement learning context.

Horizon 1 gradient estimate Let us first consider a planning problem with maximum horizon length 1. This means starting in a random state s, choosing a command $\mathbf{c}$, executing it and transitioning to an end state s'. The transition generates an instant reward r. The criterion to optimise is:

$$R(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}}\big[r(s,\mathbf{c},s')\big] = \sum_{s,\mathbf{o},\mathbf{c},s'} \underbrace{\mathbb{P}(s)\,\mathbb{P}(\mathbf{o}\mid s)\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]\,\mathbb{P}(s'\mid s,\mathbf{c})}_{\mathbb{P}(s,\mathbf{o},\mathbf{c},s')} r(s,\mathbf{c},s'),$$

which can be estimated with N iid samples of r as $\tilde{R}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{k=1}^{N} r_k$.

Similarly, taking the gradient of $R(\boldsymbol{\theta})$ gives:

$$\begin{aligned}
\nabla R(\boldsymbol{\theta}) &= \nabla\Bigg[\sum_{s,\mathbf{o},\mathbf{c},s'} \mathbb{P}(s)\,\mathbb{P}(\mathbf{o}\mid s)\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]\,\mathbb{P}(s'\mid s,\mathbf{c})r(s,\mathbf{c},s')\Bigg]\\
&= \sum_{s,\mathbf{o},\mathbf{c},s'} \mathbb{P}(s)\,\mathbb{P}(\mathbf{o}\mid s)\big[\nabla\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]\big]\,\mathbb{P}(s'\mid s,\mathbf{c})r(s,\mathbf{c},s')\\
&= \sum_{s,\mathbf{o},\mathbf{c},s'} \mathbb{P}(s)\,\mathbb{P}(\mathbf{o}\mid s)\left[\frac{\nabla\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]}{\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]}\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]\right]\mathbb{P}(s'\mid s,\mathbf{c})r(s,\mathbf{c},s')\\
&= \sum_{s,\mathbf{o},\mathbf{c},s'} \left[\frac{\nabla\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]}{\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]} r(s,\mathbf{c},s')\right]\underbrace{\mathbb{P}(s)\,\mathbb{P}(\mathbf{o}\mid s)\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]\,\mathbb{P}(s'\mid s,\mathbf{c})}_{\mathbb{P}(s,\mathbf{o},\mathbf{c},s')}\\
&= \mathbb{E}_{\boldsymbol{\theta}}\left[\frac{\nabla\,\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]}{\mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]} r(s,\mathbf{c},s')\right],
\end{aligned}$$

which provides the following estimate (with N samples):

$$\tilde{\nabla} R(\boldsymbol{\theta}) = \frac{1}{N}\sum_{k=1}^{N} \nabla \ln \mathbb{P}[\mathbf{c}_k\mid\mathbf{o}_k;\boldsymbol{\theta}]r_k$$

(noting that $\frac{\nabla\mathbb{P}[\mathbf{c}_k\mid\mathbf{o}_k;\boldsymbol{\theta}]}{\mathbb{P}[\mathbf{c}_k\mid\mathbf{o}_k;\boldsymbol{\theta}]} = \nabla\ln\mathbb{P}[\mathbf{c}_k\mid\mathbf{o}_k;\boldsymbol{\theta}]$).

Horizon 1 REINFORCE algorithm In a horizon 1 online gradient ascent, parameter θ_i is incremented after each iteration with an instant estimate of the gradient ($N=1$), i.e., using:

$$\Delta\theta_i = \alpha r e_i,$$

where $\alpha > 0$ is a *step size* (or *learning rate factor*) and $e_i = \partial \ln \mathbb{P}[\mathbf{c}\mid\mathbf{o};\boldsymbol{\theta}]/\partial\theta_i$ is called the *characteristic eligibility* of θ_i.

Under the same conditions, let us consider the following increment:

$$\Delta\theta_i = \alpha_i(r - b_i)e_i,$$

where α_i is a specific learning rate for parameter θ_i and b_i is a baseline. If b_i is conditionally independent of $\mathbf{o}$ and α_i depends at most on $\boldsymbol{\theta}$ and k, then such a learning algorithm is called a *REINFORCE* algorithm. Such an algorithm has the interesting property [8, Theorem 1] that $E[\Delta\boldsymbol{\theta}\mid\boldsymbol{\theta}]$ is always an ascent direction. Its inner product with $\nabla_{\boldsymbol{\theta}} E[r\mid\boldsymbol{\theta}]$ is non-negative, and is zero only when $\nabla_{\boldsymbol{\theta}} E[r\mid\boldsymbol{\theta}] = \mathbf{0}$. REINFORCE has two parameters. Let us observe that:

- an appropriately decreasing step size α_i, chosen to satisfy the standard stochastic function approximation conditions [38], ensures that θ_i converges to some limit value without missing a local optimum;
- the choice of the reinforcement baseline makes it possible to improve the convergence behavior as well: for example, $b_i = \tilde{r}$ (an empirical value of $E[r]$) leads to less instabilities than $b_i = 0$ because it compares the instant reward with what is usually received, equivalent to a variance reduction in gradient estimates [39].

To show how the characteristic eligibility e_i is computed, let us assume an example where:

- a single action a is available, the two possible commands being `do not execute` a ($c=0$) and `execute` a ($c=1$);
- the parameterised policy takes the following form:

$$\mathbb{P}[c \mid \mathbf{o}; \boldsymbol{\theta}] = \begin{cases} f(\mathbf{o}, \boldsymbol{\theta}) & \text{if } c=0, \\ 1 - f(\mathbf{o}, \boldsymbol{\theta}) & \text{if } c=1, \end{cases}$$

  where f is differentiable with respect to parameters $\theta_1, \ldots, \theta_p$. One such f might be a simple logistic regression $f(\mathbf{o}, \boldsymbol{\theta}) = \frac{1}{1-\exp(\mathbf{o}^\mathsf{T}\boldsymbol{\theta})}$.

Then, the characteristic eligibility for parameter θ_i is given by

$$\begin{aligned} \frac{\partial \ln \mathbb{P}[\mathbf{c} \mid \mathbf{o}; \boldsymbol{\theta}]}{\partial \theta_i} &= \frac{1}{\mathbb{P}[\mathbf{c} \mid \mathbf{o}; \boldsymbol{\theta}]} \frac{\partial\, \mathbb{P}[\mathbf{c} \mid \mathbf{o}; \boldsymbol{\theta}]}{\partial \theta_i} \\ &= \begin{cases} \frac{1}{f(\mathbf{o},\boldsymbol{\theta})} \frac{\partial f(\mathbf{o},\boldsymbol{\theta})}{\partial \theta_i} & \text{if } c=0, \\ \frac{1}{1-f(\mathbf{o},\boldsymbol{\theta})} \frac{-\partial f(\mathbf{o},\boldsymbol{\theta})}{\partial \theta_i} & \text{if } c=1. \end{cases} \end{aligned}$$

Indefinite horizon REINFORCE algorithm For a true policy-gradient approach we need to extend this algorithm to problems with an indefinite horizon. Each trial should have a finite, but possibly unknown, duration. A first approach is, for a given trial of length N, to accumulate the reward in r_{trial} and then compute the update direction for parameter θ_i as

$$\Delta\theta_i = \alpha_i (r_{trial} - b_i) \sum_{k=1}^{N} e_i(k),$$

where $e_i(k)$ is the characteristic eligibility evaluated at time k. Another option is to make an update at each time step k using

$$\Delta\theta_i = \alpha_i \big(r(k) - b_i\big) \sum_{t=1}^{k} e_i(t).$$

The latter approach has the advantage of providing an online algorithm that makes use of instant rewards as soon as they are obtained.

REINFORCE algorithms are not exactly gradient ascent algorithms. They follow an estimate of an ascent direction (within 90°) but not an estimate of the gradient direction itself (due to the step sizes being different for each parameter). Another issue is that REINFORCE's updates give the same weight to old and recent decisions, although in some settings old decisions should be assigned little credit for the current reward. This is why we prefer using a policy-gradient algorithms designed for infinite horizon problems. In the next section we present the reinforcement learning algorithms used by FPG, which are based on a direct estimate of the gradient and are meant for infinite horizon POMDPs.

2.4.4. Baxter and Bartlett's policy-gradient algorithms

We follow the presentation of Baxter and Bartlett [10,37], and note that while other derivations [9,11] differ substantially, the resulting algorithms vary only a little. We begin our overview of this policy-gradient approach with an expression for the exact gradient $\nabla R(\boldsymbol{\theta})$.

Theorem 1 *(Exact policy-gradients [37]). Suppose there are S possible planning states.*[4] *Let $P(\boldsymbol{\theta})$ be an $S \times S$ ergodic stochastic transition matrix that gives the probability of a transition from state s to state s′. Note that $P(\boldsymbol{\theta})$ describes the Markov chain resulting from setting parameter vector $\boldsymbol{\theta}$. Let $\boldsymbol{\pi}(\boldsymbol{\theta})$ be a $1 \times S$ column vector that gives the stationary probability of being in any particular state, derived from the unique solution of $\boldsymbol{\pi} P(\boldsymbol{\theta}) = \boldsymbol{\pi}$. Let $\mathbf{r}$ be the $1 \times S$ vector with the reward for each state. The identity matrix is given by I, and $\mathbf{e}$ is a column vector of 1's. With these definitions*

$$\nabla R(\boldsymbol{\theta}) = \pi(\boldsymbol{\theta})^\mathsf{T} \big(\nabla P(\boldsymbol{\theta})\big) \big(I - P(\boldsymbol{\theta}) + \mathbf{e}\pi(\boldsymbol{\theta})^\mathsf{T}\big)^{-1} \mathbf{r}. \tag{3}$$

The point we wish to make with this theorem is that policy-gradients could be used to do true model-based planning [40]. The planning domain and problem specification contain all the information necessary to calculate $\pi(\boldsymbol{\theta})$ and $P(\boldsymbol{\theta})$ for a given initial policy described by $\boldsymbol{\theta}$ (such as uniformly randomly starting eligible actions). We can construct a reward vector $\mathbf{r}$ as described in Section 2.4.1. Also, $\nabla P(\boldsymbol{\theta})$ can be computed and the matrix inverse in (3) exists. Thus if the state

[4] See [37] for the details required to extend this theorem to continuous state spaces.

space is finite (that is, no continuous durations and a makespan limit) we could perform an exact model-based policy-gradient optimisation by repeatedly solving (3) in the Algorithm 1 loop. This seems preferable to our suggested approach of using the model simply to create a plan execution simulator for use with reinforcement learning. However, any sizeable planning problems involve millions of possible states, making the $O(S^3)$ matrix inversion in (3) intractable. It is worth noting that $P(\boldsymbol{\theta})$ is typically very sparse. This fact, along with a truncated iterative solution to (3), was used to perform model-based policy-gradient on systems with tens of thousands of states [40].

As an aside, we attempted to exploit a structured ADD (algebraic decision diagram) representation of (3). ADDs can compactly represent factored matrices and the standard matrix operations can be redefined in terms of ADD manipulations. This fact has previously been used in planning [41]. Our idea was to create a parameterised ADD that allowed an analytic solution to (3), which could then be efficiently evaluated for any parameter values. However, the initially compact ADD representations of $P(\boldsymbol{\theta})$ and $\pi(\boldsymbol{\theta})$ explode in size during the solution of the matrix inverse [42].

Because an exact computation of the gradient is intractable we use Monte-Carlo gradient estimates generated from repeatedly simulating plan executions as if they were one long Markov process.

Theorem 2 *(Estimated approximate policy-gradients [37]). Let r_t be the scalar reward received at time, i.e., $r_t = \mathbf{r}(s_t)$. Let $\beta \in [0, 1)$ be a* discount factor. *Then*

$$\nabla R(\boldsymbol{\theta}) = \lim_{\beta \to 1} \lim_{T \to \infty} \frac{1}{T} \sum_{t=1}^{T} \frac{\nabla_{\boldsymbol{\theta}} \mathbb{P}[\mathbf{c}_t \mid \mathbf{o}_t; \boldsymbol{\theta}]}{\mathbb{P}[\mathbf{c}_t \mid \mathbf{o}_t; \boldsymbol{\theta}]} \sum_{\tau=t+1}^{T} \beta^{\tau-t-1} r_\tau. \tag{4}$$

This quantity is an estimate that becomes exact as $T \to \infty$, of an approximation that becomes exact as $\beta \to 1$. The detailed derivation is out of the scope of this paper, but we do wish to provide some insight into how this expression relates to (3). In particular, the role of the discount factor β. The theorem is derived from (3) by first establishing that

$$\left(I - P(\boldsymbol{\theta}) + \mathbf{e}\pi(\boldsymbol{\theta})^{\mathsf{T}}\right)^{-1}\mathbf{r} = \lim_{\beta \to 1} \mathbf{v}_\beta(\boldsymbol{\theta}),$$

where $\mathbf{v}_\beta(\boldsymbol{\theta})$ is a $1 \times S$ vector of discounted state values under the current policy. For state s this is defined as

$$v_\beta(\boldsymbol{\theta}, s) := \mathbb{E}_{\boldsymbol{\theta}}\left[\sum_{t=0}^{\infty} \beta^t r(X_t) \mid X_0 = s\right],$$

where X_t is the random variable denoting the state at time t steps into the future. Note that this is the usual definition of the discounted value in reinforcement learning [6]. Combining this with (3) gives

$$\nabla R(\boldsymbol{\theta}) = \lim_{\beta \to 1} \pi(\boldsymbol{\theta})^{\mathsf{T}} \left(\nabla P(\boldsymbol{\theta})\right) \mathbf{v}_\beta(\boldsymbol{\theta}).$$

The Ergodic Theorem can then be applied to turn the summations (implicit in the matrix operations) over state, next state, actions and observations into a Monte-Carlo estimate over a single infinite trajectory of states, next states, actions and observations. So why was β introduced? You can observe from the infinite summation that without $\beta < 1$ (4) may be unbounded. That is, infinite variance arises during the Monte-Carlo estimate. This can be loosely thought of as trying to assign the credit for a reward to a possibly infinite number of actions into the past. So β creates a decaying artificial horizon on how long ago actions could occur and still be assigned some credit for achieving the current reward.

However, $\beta < 1$ introduces a bias [37]. Thus we try and keep β as close to 1 as possible, while still achieving reasonable estimates of the gradient. Alternatively, if we can observe a point where the state is reset (such as at the end of a plan execution), we can impose this true horizon on the impact of actions on rewards [8], and leave $\beta = 1$ to achieve an unbiased estimate. In practice we have found the variance reduction due to $\beta < 1$ can be useful, even for finite-horizon planning.

We have not yet defined a form for the policy $\mathbb{P}[\mathbf{c}_t \mid \mathbf{o}_t; \boldsymbol{\theta}_t]$. We discuss this in Section 3.2, and for now only note that it is a parameterised policy function that we construct to produce a correct probability distribution. Its log derivative must exist and be bounded to satisfy the assumptions described in [37].

Note that (4) requires looking forward in time to observe rewards, so we reverse the summations

$$\nabla R(\boldsymbol{\theta}) = \lim_{\beta \to 1} \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T-1} r_t \sum_{\tau=0}^{t} \beta^{t-\tau} \frac{\nabla \mathbb{P}(\mathbf{c}_\tau \mid \mathbf{o}_\tau; \boldsymbol{\theta})}{\mathbb{P}(\mathbf{c}_\tau \mid \mathbf{o}_\tau; \boldsymbol{\theta})}. \tag{5}$$

This now becomes easy to implement by using an *eligibility trace* $\mathbf{e}_t$ in place of the second summation, as shown in Algorithm 2.

The eligibility trace $\mathbf{e}_t$ contains the discounted sum of normalised policy gradients for recent commands (equivalent to log-policy gradients). This can provide an intuition for how the algorithm works: stepping the parameters in the direction of the eligibility trace will increase the probability of choosing recent commands under similar observations, with recency weighting determined by β. But it is the relative value of *rewards* that indicate if we should increase or decrease the probability of recent command sequences. So the instant gradient at each time step is $r_t\mathbf{e}_t$.

```
1: $t = 0$
2: repeat
3:     $\mathbf{o}_t = $ sim.getObservation
4:     sample $\mathbf{c}_t$ from $\mathbb{P}(\cdot \mid \mathbf{o}_t; \boldsymbol{\theta}_t)$
5:     $\mathbf{e}_t = \beta \mathbf{e}_{t-1} + \frac{\nabla_{\boldsymbol{\theta}} \mathbb{P}(\mathbf{c}_t \mid \mathbf{o}_t; \boldsymbol{\theta}_t)}{\mathbb{P}(\mathbf{c}_t \mid \mathbf{o}_t; \boldsymbol{\theta}_t)}$
6:     sim.doAction($\mathbf{c}_t$)
7:     $r_t = $ sim.getReward()
8:     if OLPOMDP then
9:         $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha r_t \mathbf{e}_t$
10:    else
11:        $\mathbf{g}_t = \frac{1}{t+1}(t\mathbf{g}_{t-1} + r_t \mathbf{e}_t)$     // batch estimate
12:    $t \leftarrow t + 1$
13: until stoppingCriterion($\epsilon, \ldots$)
14: if not OLPOMDP then
15:     return $\mathbf{g}_t$
```

Algorithm 2. policyGrad(θ_0, Simulator sim, α).

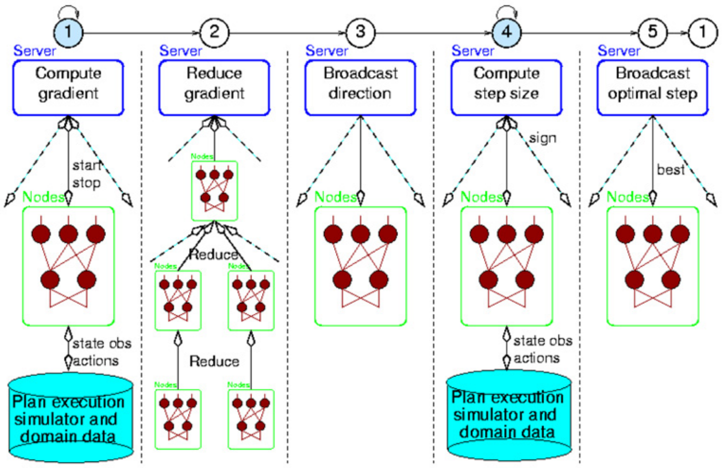


Fig. 2. A high level illustration of how to parallelise FPG. Each node contains a separate process implementing Algorithm 2.

There are two offered optimisation methods using the instant gradients [10]. These correspond to the OLPOMDP and batch cases distinguished in Algorithm 2. OLPOMDP is the simple online stochastic gradient ascent $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha r_t \mathbf{e}_t$ with scalar gain α. Alternatively, CONJPOMDP averages $r_t \mathbf{e}_t$ over T steps to compute the batch gradient $\mathbf{g}_t$, approximating (4), followed by a line search for the best step size α in the search direction.[5] OLPOMDP can be considerably faster than CONJPOMDP in highly stochastic environments because it is tolerant of stochastic gradients and adjusts the policy at every step. We prefer it for all our single processor experiments.

However, the batch approach is used for parallelising FPG as shown in Fig. 2. Each processor runs independent simulations of the current policy with the same fixed parameters. Instant gradients are averaged over many simulations to obtain a per processor estimate of the gradient (4). A master process averages the gradients from each processor and broadcasts the resulting search direction. All processors then take part in evaluating points along the search direction to establish the best α. Once found, the master process then broadcasts the final step size. The process is repeated until the aggregated gradient drops below a threshold.

3. FPG

We now describe how policy-gradients can be used to optimise CPTP policies. We begin by describing the construction of a plan simulator.

3.1. State space simulator

FPG's planning state is:

- the makespan so far (the plan starts at time 0), i.e., absolute time;

[5] We do not apply the conjugation of gradients described in [10], which gives the algorithm its name, because conjugate directions collapse under noise.

- the truth value of each predicate;
- the function values;
- a dynamic length queue of (at least) future events including:
  - outcome sampling events,
  - outcome end events,
  - effect implementation events,
  - exogenous events.

In a particular state, only the *eligible* actions have satisfied all at-start preconditions for execution. Recall that a command is the decision to start a set of eligible actions. While actions might be individually eligible, starting them concurrently may require too many resources. Or starting could cause a precondition of an eligible action to be invalidated by the deterministic start-effects of another. Both of these are examples of actions which we consider mutually exclusive (mutex). We do not deal with any other type of conflict when determining mutexes for the purpose of deciding to start actions. For example, we do not consider mutexed outcomes. This is because probabilistic planning means such mutexes may, or may not, occur. If they do occur the plan execution enters a failure state, moving the optimisation away from this policy.

The planner handles the execution of actions using a time-ordered event queue. When starting an action, at-start effects are processed, adding `effect` events to the queue if there are any delayed at-start effects. Additionally, a `sample-outcome` event is scheduled for some point during the execution of the action (this duration possibly being sampled from a continuous distribution). The `sample-outcome` event indicates the point when chance decides which particular discrete outcome is triggered for a given action. This results in adding the corresponding `effect` events for this outcome, and any other at-end effects, to the event queue (again possibly with an additional sampled delay). An action ends when all possible effects due to an action have occurred, although this is an arbitrary definition given our execution model. These definitions allow predicates to change at any time during an action.

The only element of state that is presented to the policy is the truth value of each predicate. We could trivially present additional features except for the dynamically sized event queue. However, empirically we found that the state of the predicates alone was a good summary of the overall planning state. This is intuitive for many domains because their human designers describe important state with predicates. Note that it is particularly dangerous to supply a representation of the future event queue to the policy. Doing this could allow decisions to be based on a particular probabilistic outcome that has not occurred yet, and which would not be foreseen in real life.

Exogenous events, if any, are handled by inserting these events into the event queue as they occur. They can include manually (de)scheduling an action, effect, or probabilistic outcome. Note that this permits a form of mixed initiative planning, and is very useful to see how a policy would adjust to unexpected events.

To estimate policy gradients we need a plan execution simulator to generate a trajectory through the planning state space. It takes commands from the factored policy, checks for mutex constraints, implements at-start effects, and queues `sample-outcome` events. The state update then proceeds to process `sample-outcome` and `effect` events from the queue until a new decision point is met. Decision points equate to *happenings*, which occur when: (1) time has increased since the last decision point; and (2) there are no more events for this time step. Under these conditions a new action can be chosen, possibly a no-op if the best action is to simply proceed to the next event. The process of simulating the plan execution from one command choice to another is described by `simulateTillHappening()` (Algorithm 3).

When processing events, the algorithm also ensures no running actions have violated *over-all* conditions. If this happens, the plan execution ends in a failure state. Note that making decisions at happenings results in FPG being incomplete in domains with some combinations of effects and at-end conditions [19,22]. A simple fix is to set a maximum delay d between two consecutive decision points. In the discrete time case, $d = 1$ guarantees completeness, but a small d introduces too many feasible plans. Another fix is to learn how long to wait until the next decision point.[6] We have not yet pursued either approach.

Only the current parameters, the eligibility trace, initial and current states, and the current observation are kept in memory at any point in time. The number of parameters depends on the choice of policy function approximation but is typically $O(N \times |\mathbf{o}|)$, where N is the number of actions and $|\mathbf{o}|$ is the dimensionality of the state observations. This is in stark contrast with dynamic programming based planners that expand in memory all states relevant to planning, and the number of states is generally exponential in the number of state-variables. We emphasise that low memory use is a key advantage of FPG's approach. The planning problem can be orders of magnitude larger than other CPTP planners attempt. However, this is a time/space trade off. FPG may require more time than other approaches to compensate for not retaining detailed state information.

3.2. Choice of the function approximator

A benefit of the FPG approach, arising from the use of policy-gradients, is flexibility in the choice of function approximator. It is possible to trade off the richness of a policy representation with its compactness and ability to be trained quickly.

[6] This is feasible in the continuous time case because policy-gradient algorithms can optimise controllers with continuous action spaces.

```
1: for each $a_{tn}$ = 'Yes' in $\mathbf{c}_t$ do
2:     if $a_{tn}$.isMutex() then return MUTEX
3:     $s$.addEvent($a_{tn}$, sample-outcome, $s$.time + sample($a_{tn}$.duration-distribution))
4:     for each $f \in$ atStartEffects($a_{tn}$) do
5:         $s$.processEffect($f$)
6:     for each $f \in$ delayedEffects($a_{tn}$) do
7:         $s$.addEvent($f$, effect, $s$.time + sample($f$.delay-distribution))
8: repeat
9:     if $s$.time > maximum makespan then
10:        s.failure = true
11:        return
12:    if $s$.operationGoalsMet() then
13:        s.goal = true
14:        return
15:    if $\neg s$.anyEligibleActions() & $s$.noEvent() then
16:        s.failure = true
17:        return
18:    event = $s$.nextEvent()
19:    $s$.time = event.time
20:    if type(event) = effect then
21:        $s$.processEffect(event.effect)
22:    else if type(event) = sample-outcome then
23:        sample outcome out from event
24:        for each $f \in$ immediateEffects(out) do
25:            $s$.processEffect($f$)
26:        for each $f \in$ delayedEffects(out) do
27:            $s$.addEvent($f$, effect, $s$.time + sample($f$.delay-distribution))
28: until $s$.isDecisionPoint()
```

Algorithm 3. simulateTillHappening(State s, Command $\mathbf{c}_t$).

On the other hand, it can be difficult to choose a representation that achieves the best balance between these considerations. This section provides the generic approximator requirements, and then discusses four specific approximators.

The command $\mathbf{c}_t = \{a_{t1}, a_{t2}, \ldots, a_{tN}\}$ at time t is a combination of independent 'Yes' or 'No' choices made by each of the grounded action policies. Each policy has an independent set of parameters that make up $\boldsymbol{\theta} \in \mathbb{R}^p$: $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_N$. With the independence of parameters the command policy factors into

$$\begin{aligned}\mathbb{P}[\mathbf{c}_t \mid \mathbf{o}_t, \boldsymbol{\theta}] &= \mathbb{P}[a_{t1}, \ldots, a_{tN} \mid \mathbf{o}_t; \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N] \\ &= \mathbb{P}[a_{t1} \mid \mathbf{o}_t; \boldsymbol{\theta}_1] \times \cdots \times \mathbb{P}[a_{tN} \mid \mathbf{o}_t; \boldsymbol{\theta}_N]. \end{aligned} \tag{6}$$

The computation of the log-policy gradients also factorises trivially. It is not necessary that all action policies receive the same observation, and it may be advantageous to have different observations for different actions, leading to a *decentralised* paradigm.[7] Similar factored policy-gradient approaches are adopted by [13] and [14]. The main requirement for each action policy is that $\log \mathbb{P}[a_{tn} \mid \mathbf{o}_t; \boldsymbol{\theta}_n]$ be differentiable and bounded with respect to the parameters for each choice of action start a_{tn} = 'Yes' or 'No' [37]. The gradient must also be bounded.

In the situation where all action policies have access to the same state information, and rewards, and their policy approximations are sufficiently rich, the action policies will be able to coordinate optimally given the state information available. This coordination is a natural consequence of the learning process: individual actions taking part in rewarding joint actions are reinforced so that the joint actions are preferred [43]. We note that difficulties do arise if there are many equivalent optimal policies, for example if all policies that pick any *single* action out of all the eligible actions are equally optimal. In this case a gradient based planner can become stuck in saddle regions. The usual, somewhat unsatisfactory, resolution is to rely on random initialisation to give slightly higher probability to one such equivalent policy from the outset. If each action policy sees different state information, optimal coordination becomes NExp-hard [44]. Methods such as coordinated RL [45] exist to efficiently solve such problems by performing belief propagation between agents, or policies. The policies become conditionally independent, where the structure of this conditional independence is exploited to make the process efficient. In this context, the policies would become conditionally dependent on the mutex relationships between actions. However, we have so far not experimented with this approach. We ensure that all action policies see the same observation.

3.2.1. Linear function approximators

One simple and effective action policy is a linear approximator mapped to probabilities using a logistic regression function[8]

[7] Although we note that the use of a single shared simulator for optimisation means this cannot be made a truly decentralised planning algorithm.

[8] In the RL this function is more commonly expressed as the soft-max, or Gibbs, or Boltzmann distributions.

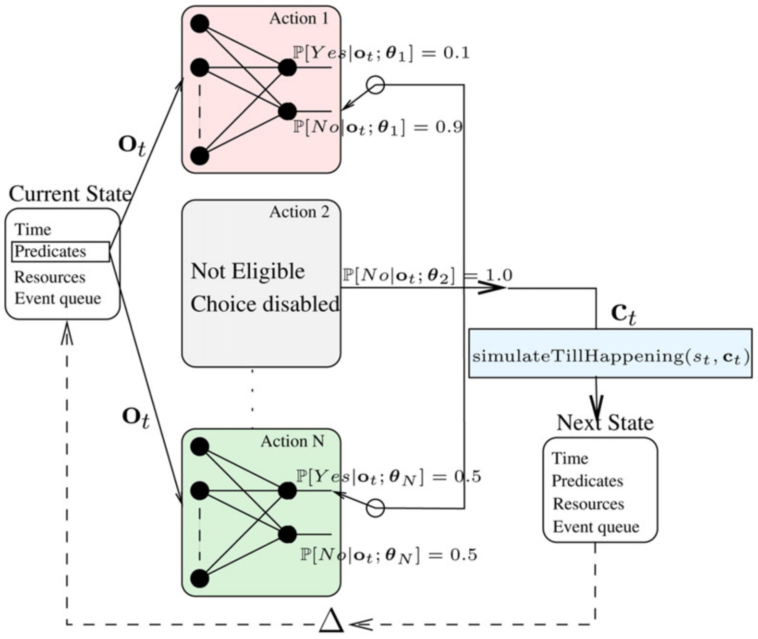

Action 1
$\mathbb{P}[Yes|\mathbf{o}_t;\boldsymbol{\theta}_1] = 0.1$
$\mathbb{P}[No|\mathbf{o}_t;\boldsymbol{\theta}_1] = 0.9$
$\mathbf{o}_t$
Current State
Time
Predicates
Resources
Event queue
Action 2
Not Eligible
Choice disabled
$\mathbb{P}[No|\mathbf{o}_t;\boldsymbol{\theta}_2] = 1.0$
$\mathbf{c}_t$
simulateTillHappening$(s_t, \mathbf{c}_t)$
$\mathbf{o}_t$
Action N
$\mathbb{P}[Yes|\mathbf{o}_t;\boldsymbol{\theta}_N] = 0.5$
$\mathbb{P}[No|\mathbf{o}_t;\boldsymbol{\theta}_N] = 0.5$
Next State
Time
Predicates
Resources
Event queue
Δ

Fig. 3. High level overview of FPG's decision making loop, showing linear function approximators.

$$\begin{aligned}
&\mathbb{P}[a_{tn} = Yes \mid \mathbf{o}_t; \boldsymbol{\theta}_n] = \frac{1}{\exp(\mathbf{o}_t^\top \boldsymbol{\theta}_n) + 1}, \\
&\mathbb{P}[a_{tn} = No \mid \mathbf{o}_t; \boldsymbol{\theta}_n] = 1 - \mathbb{P}[a_{tn} = Yes \mid \mathbf{o}_t; \boldsymbol{\theta}_n].
\end{aligned} \tag{7}$$

Recall that the observation vector **o** is a vector representing the current predicate truth values plus a constant bias. True predicates get a 1 entry, and false predicates get 0. The bias entry is a constant 1 that allows the linear approximator to represent a decision boundary that does not pass through the origin. If the dimension of the observation vector is $|\mathbf{o}|$ then each set of parameters $\boldsymbol{\theta}_n$ can be thought of as an $|\mathbf{o}|$ vector that represents the approximator weights for action n. The required log-policy instant gradients over each parameter $\theta \in \boldsymbol{\theta}_n$ are

$$\begin{aligned}
&\frac{\nabla_{\boldsymbol{\theta}_n} \mathbb{P}[a_{tn} = Yes \mid \mathbf{o}_t; \boldsymbol{\theta}_n]}{\mathbb{P}[a_{tn} = Yes \mid \mathbf{o}_t; \boldsymbol{\theta}_n]} = -\mathbf{o}_t \exp\bigl(\mathbf{o}_t^\top \boldsymbol{\theta}_n\bigr) \mathbb{P}[a_{tn} = Yes \mid \mathbf{o}_t; \boldsymbol{\theta}_n], \\
&\frac{\nabla_{\boldsymbol{\theta}_n} \mathbb{P}[a_{tn} = No \mid \mathbf{o}_t; \boldsymbol{\theta}_n]}{\mathbb{P}[a_{tn} = No \mid \mathbf{o}_t; \boldsymbol{\theta}_n]} = \mathbf{o}_t \mathbb{P}[a_{tn} = Yes \mid \mathbf{o}_t; \boldsymbol{\theta}_n].
\end{aligned} \tag{8}$$

These log-policy gradients are added to the eligibility trace (Algorithm 2, line 5) based on the yes/no decisions for each action. Looping this calculation over all eligible actions computes the normalised gradient of the probability of the *joint* command (6). Fig. 3 illustrates this scheme.

Initially, the parameters are usually set to 0 giving a uniformly random policy, encouraging exploration of the command space. To increase the long-term average reward the policy parameters must gradually adjust to prefer some commands over others. Typically we see the policy start to converge to a function that gives high probability to commands that have been established to be good. If FPG maintains that two or more commands have similar probabilities given a particular observation, then this is indicative of FPG being unable to determine which command is better. This might be because there is insufficient state information encoded in **o**, the parameterisation is too simple, or because the two commands really are equivalent in the long run.

3.2.2. Trees of experts

To demonstrate the flexibility of the FPG approach we develop an alternative parameterised policy function. This policy has the task of switching between a collection of known expert policy strategies. As described below, this is achieved using a partly-defined decision tree whose decision nodes are tuned by stochastic gradient ascent.

Rather than start with a uniform policy we may be given a selection of heuristic policies that work well across a range of domains. For example, in a probabilistic setting we may have access to a replanner, an optimal non-concurrent planner, and a naïve planner that attempts to run all eligible commands. Indeed, the best planner to invoke may depend on the current state as well as the overall domain. The decision tree policies described here are a simple mechanism to allow FPG to switch between such high level expert policies. We assume a user declares an initial tree of all available policies. The leaves represent a policy to follow, and the branch nodes represent decision rules for which policy to follow. We show how to learn

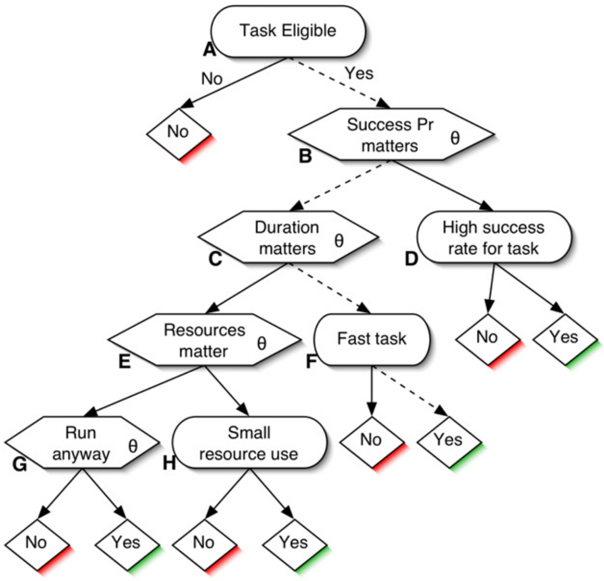


Fig. 4. Decision tree action policy.

these rules. In the factored setting, each action has its own decision tree policy. All actions start with the same template tree but adapt them independently. Whether to start an action is decided by starting at the root node and following a path down the tree, visiting a set of decision nodes $\mathcal{D}$. At each node we either apply a hard-coded branch selection rule, or sample a stochastic branch rule from a parameterised policy for that node. Assuming the conditional independence of decisions at each node, the probability of reaching an action leaf l equals the product of branch probabilities at each node

$$\mathbb{P}[a = l \mid \mathbf{o}; \boldsymbol{\theta}_n] = \prod_{d \in \mathcal{D}} \mathbb{P}[d' \mid \mathbf{o}; \boldsymbol{\theta}_{n,d}], \tag{9}$$

where d represents the current decision node, and d' represents the next node visited in the tree. The probability of a branch followed as a result of a hard-coded rule is 1. The individual $\mathbb{P}[d' \mid \mathbf{o}, \boldsymbol{\theta}_{n,d}]$ functions can be any differentiable function of the parameters, such as the linear approximator. Parameter adjustments have the simple effect of pruning parts of the tree that represent poor policies in that region of the state space.

For example, nodes A, D, F, H (Fig. 4) represent hard-coded rules that switch deterministically between the Yes and No branches based on the truth of the statement in the node, for the current state. Nodes B, C, E, G are parameterised so that they select branches as a result of learning. In our example, the probability of choosing the left or right branches is a single parameter logistic function that is independent of the observations. For action n, and decision node C "action duration matters?" we have

$$\mathbb{P}[Yes \mid \mathbf{o}; \theta_{n,C}] = \mathbb{P}[Yes; \theta_{n,C}] = \frac{1}{\exp(\theta_{n,C}) + 1}.$$

In general the policy pruning could also be a function of the current state. The log derivatives of the 'Yes' and 'No' decisions are given by (8), noting that in this case $\mathbf{o}$ is a scalar constant 1. The normalised action probability gradient for each node is added to the eligibility trace independently. We can do this because the product terms in (9) cancel when taking the gradient of the *log* of the same expression.

If the parameters converge in such a way that prunes Fig. 4 to just the dashed branches we would have the policy: *if the action IS eligible, and probability of this action success does NOT matter, and the duration of this action DOES matter, and this action IS fast, then start, otherwise do not start.* Thus we can encode highly expressive policies with only a few parameters (four parameters in the case of Fig. 4). This approach allows extensive use of control knowledge, using FPG simply to switch between experts. Even though it is ignored in the example above, state can be taken into account if that is useful.

3.2.3. *Approximators for non-temporal probabilistic planning*

We discus the particular case of non-concurrent non-temporal probabilistic planning (PP) domains. The probabilistic track of the international planning competition (IPC) was based on this class of domains. In PP, we choose *one* action from all currently eligible actions. This is in contrast to choosing a *set* of actions to contribute to a command. Even though this is somewhat different problem to CPTP, a similar factorisation can be applied.

The factored policy still uses a parameterised policy function per action. But, instead of turning the output into a {Yes, No} probability distribution per action, logistic regression is used to compute a single probability distribution over all eligible actions. The probability of choosing n at time t is

$$\mathbb{P}[a_t = n \mid \mathbf{o}_t; \boldsymbol{\theta}] = \frac{\exp(\mathbf{o}_t^\top \boldsymbol{\theta}_n)}{\sum_{k \in E(\mathbf{o}_t)} \exp(\mathbf{o}_t^\top \boldsymbol{\theta}_k)}, \tag{10}$$

where $E(\mathbf{o}_t)$ is the set of eligible actions. The normalised derivative is

$$\frac{\nabla \mathbb{P}[a_t = n \mid \mathbf{o}_t; \boldsymbol{\theta}]}{\mathbb{P}[a_t = n \mid \mathbf{o}_t; \boldsymbol{\theta}]} = \mathbf{o}_t \big(U(n) - \mathbb{P}[\cdot \mid \mathbf{o}_t; \boldsymbol{\theta}]\big)^\mathsf{T},$$

where $U(n)$ is a unit vector with a 1 in element n, and $\mathbb{P}[\cdot \mid \mathbf{o}_t; \boldsymbol{\theta}]$ is treated as an N dimensional probability vector.

Also, the `simulateTillHappening()` function becomes substantially simpler. We implement the effects of action a_t, sample an outcome, and implement its effects as well. Time and the event queue are no longer elements of the state.

3.2.4. The relational online policy gradient

The relational online policy gradient (ROPG) is described in [16]. It is not part of the FPG planner, but we show how the optimisation component of ROPG can be cast as an example of an FPG policy parameterisation.

ROPG is a relational reinforcement learning (RRL) approach to solving probabilistic non-Markovian decision processes. RRL computes policies, expressed in a relational language, that work across a range of problems drawn from a domain. The state and observation spaces may vary across problem instances, but higher-order representations can capture the commonalities in a concise way. The aim is to learn policies from a small number of problems, and then generalise to other problems from the same domain. An initial reasoning phase discovers candidate relational control strategies. In the second phase policy-gradient learning is used to discover when to use each strategy.

ROPG uses exactly Algorithm 2 in order to optimise the choice of control strategy for the current observation. ROPG chooses one action for each step, and in that sense it resembles FPGs IPC policy from Section 3.2.3. However, the output of the parameterised policy is the choice of strategy to evaluate, which is then resolved deterministically to produce a grounded action. In that sense, ROPG more closely resembles a single layer of the tree of experts policy from Section 3.2.2. Assume there are N control strategies to choose from that may or may not be available at the current time step. The observations $\mathbf{o}_t$ provided by ROPG can be thought of as binary vectors giving the eligibility of each strategy according to the relational representation of the current state. In fact, due to the non-Markovian nature of the domains, the observations, control rules and rewards are over histories instead of only the current state. The ROPG parameterised policy is expressed as

$$\mathbb{P}[a_t = n \mid \mathbf{o}_t; \boldsymbol{\theta}] = \kappa(n, \mathbf{o}_t) \frac{\exp(\boldsymbol{\theta}_n)}{\sum_{k=1}^{N} \kappa(k, \mathbf{o}_t) \exp(\boldsymbol{\theta}_k)},$$

where $\kappa(n, \mathbf{o}_t)$ is 1 if element n of $\mathbf{o}_t$ is non-zero, and 0 otherwise. The use of κ restricts the policy to choose from eligible strategies only, and is analogous the use of the function $E(\mathbf{o}_t)$ in (10).

The results shown in [16] demonstrate that this is an effective strategy for automatically evaluating generated control strategies.

3.3. The policy-gradient algorithm

Algorithm 4 completes our description of FPG by showing how to implement the gradient estimate (4) for planning with factored action policies (assuming the more complex CPTP case). The algorithm works by repeatedly simulating plan executions:

1. the initial state represents makespan 0 in the plan (not to be confused with the step number t in the algorithm);
2. the policies all receive the observation $\mathbf{o}_t$ of the current state s_t;
3. each policy representing an eligible action emits a probability of starting;
4. each action policy samples ‘Yes’ or ‘No’ and these are issued as a joint command;
5. the plan state transition is sampled (see Section 3.1);
6. the planner receives the global reward for the new state;
7. for OlPomdp all parameters are immediately updated by $\alpha r_t \mathbf{e}_t$, or for parallel planning $r_t \mathbf{e}_t$ is averaged over T steps before being passed to an additional line search phase controlled by a master process.

Note the link to the planning simulator on line 9. If the simulator indicates that the action is impossible due to a mutex constraint, the planner successively disables one action in the command (according to an arbitrary lexical ordering) until the command is eligible. This mutex elimination process can be considered part of the system’s dynamics. Some actions are automatically cancelled if they are mutexed with other actions so that it does not influence the policy-gradient algorithm itself. While this seems like a dangerously ad hoc scheme, if the wrong decision is made the long-term average reward will suffer and the choice of actions that led to the mutex will be discouraged in the future. More studied approaches to dealing with mutexes may permit faster learning. Examples include coordinated RL [45] that provide efficient methods to include dependencies between action policies.

```
1: Set $s_0$ to initial state, $t = 0$, $\mathbf{e}_t = [0]$, init $\boldsymbol{\theta}_0$ randomly
2: repeat
3:     $\mathbf{e}_{t+1} = \beta \mathbf{e}_t$
4:     $\mathbf{o}_t = $ sim.getObservation
5:     for each eligible action $a_n$ do
6:         Evaluate action policy $n$ $\mathbb{P}[a_{tn} = \{\text{Yes}, \text{No}\} \mid \mathbf{o}; \boldsymbol{\theta}_{tn}]$
7:         Sample $a_{tn} = $ Yes or $a_{tn} = $ No
8:         $\mathbf{e}_{t+1} = \mathbf{e}_{t+1} + \frac{\nabla_{\boldsymbol{\theta}} \mathbb{P}[a_{tn} \mid \mathbf{o}; \boldsymbol{\theta}_{tn}]}{\mathbb{P}[a_{tn} \mid \mathbf{o}; \boldsymbol{\theta}_{tn}]}$
9:     while ($s_{t+1} = $ simulateTillHappening($s_t, \mathbf{c}_t$)) == MUTEX do
10:        arbitrarily disable action in $\mathbf{c}_t$ due to mutex
11:    $r_t = $ sim.getReward()
12:    $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha r_t \mathbf{e}_{t+1}$
13:    if $s_{t+1}$.isTerminalState then $s_{t+1} = s_0$
14:    $t \leftarrow t + 1$
15: until stoppingCriterion($\epsilon, \ldots$)
```

Algorithm 4. FPG based on OLPOMDP.

Line 8 computes the normalised gradient of the sampled action probability and adds the gradient for the nth action's parameters into the eligibility trace. Because planning is inherently episodic we could alternatively set $\beta = 1$ and reset $\mathbf{e}_t$ every time a terminal state is encountered. However, empirically, setting $\beta = 0.95$ performed better than resetting $\mathbf{e}_t$, probably due to the variance reduction associated with a lower β.

The gradient for parameters not relating to action n is 0. We do not compute $\mathbb{P}[a_{tn} \mid \mathbf{o}_t; \boldsymbol{\theta}_n]$ or gradients for actions with unsatisfied preconditions. If no actions are chosen to begin, we issue a no-op action. If the event queue is not empty, we process up to the next happening, otherwise time is simply incremented by 1 to ensure all possible policies will eventually reach a maximum makespan and hence a reset state.

3.3.1. The Q-learning variant

FPG's approach raises the question of whether a value-based reinforcement learning algorithm with a factored approximator could work just as well. FPG's factorisation is easy to extend in the non-temporal non-concurrent case. The result is the *Factored Q-learning* planner (FQL), which is identical to FPG except that:

- the policy-gradient algorithm is replaced by standard $Q(\lambda)$-learning algorithm;
- each linear approximator has to learn the Q-value associated with its action.

Contrary to FPG, there is no simple way of trading off exploration and exploitation. Our choice is, while learning, to use an ϵ-greedy policy with ϵ following a sigmoid function going from 0.95 down to 0.05 during the allocated learning period. We also explored various settings for λ and the learning rate. FQL also has no convergence guarantees. Even with $\lambda = 0$, Q-learning with linear function approximation may diverge under some exploration strategies [46].

3.4. Implementation details

This section is devoted to some of the engineering details needed to achieve good performance with FPG. Unless specified these details are for FPG for CPTP and FPG-IPC for PP.

3.4.1. Grounding actions and variables

Because PPDDL uses first-order constructs, a preliminary step when creating the function approximators is to ground the domain. That is, we build:

- a set of relevant actions A^r, i.e., actions which are reachable from the initial state, and
- a set of relevant predicates P^r, i.e., predicates whose value can change because of relevant actions.

Exact reachability analysis to determine precise sets A^r and P^r is expensive. Instead, a simple iterative process is used to compute supersets A and P by following a relaxed reachability analysis.

The relaxation is similar to that employed by many planners [47]. It relies on the fact that PDDL is based on the notion of atoms rather than variables (predicates). States are described by a list of currently true/positive atoms. Two sets of grounded actions are used but initially empty: the set of new eligible actions A_n and the set of processed actions A_p. The reachability analysis starts by putting all initially true atoms in a set P, then, as shown in Algorithm 5, it alternates between:

- adding new eligible actions to A_n based on atoms in P, using positive preconditions only to determine if an action is eligible; and
- searching for new atoms to add to P based on actions in A_n (which are moved to A_p once processed), using only positive effects.

```
1: Initialisation:
2: $A_n \leftarrow \emptyset$ /* {new eligible actions} */
3: $A_p \leftarrow \emptyset$ /* {processed eligible actions} */
4: $P \leftarrow$ atoms in $s_0$
5: repeat
6:     $A_n \leftarrow A_n \bigcup \text{eligibleActions}(P) \setminus A_p$
7:     Pick $a \in A_n$
8:     $A_n \leftarrow A_n \setminus \{a\}$
9:     $P \leftarrow P \bigcup \text{atomsInEffects}(a)$
10:    $A_p \leftarrow A_p \bigcup \{a\}$
11: until $A_n = \emptyset$
```

Algorithm 5. ground().

The process stops when A_n is empty.

3.4.2. Progress estimator

In FPG-IPC, the reward function is shaped with a simple progress estimator. The progress estimator counts how many of the goal's facts have been added or deleted during the last transition. This possibly negative number f is multiplied by a coefficient ρ to compute the additional reward $r_{progress}$. Provided the net progress reward per episode is 0, ensured by adjusting the final goal state reward, this shaping does not alter the overall objective function [35].

This approach works well because goals are often specified as a conjunction of facts to satisfy. It could be improved to tackle more complex goals involving metric comparisons, disjunctions, and so on.

3.4.3. Saving computation time when rewards are rare

Domains of the probabilistic track of IPC-5 only reward reaching the goal. So most of the time $r_t = 0$, particularly if no progress estimator is used. This makes it possible to avoid or delay various computations in OLPOMDP as follows:

- update the parameter vector $\boldsymbol{\theta}$ only when $r_t \neq 0$;
- while an action a remains ineligible and $r = 0$, do not discount (with β) the corresponding part of the eligibility vector $\mathbf{e}_a$ but increment a counter $\#_a$;
- when an action a is eligible or $r = 0$, discount $\mathbf{e}_a$ by $\beta^{\#_a}$ before performing the normal update, and reset $\#_a$ to zero.

This process is not compatible with the use of a baseline reward—i.e., subtracting a running average $\bar{r}$ of the reward to the instant reward r_t—commonly used in RL to reduce the variance of gradient estimates. However, we gain more from the additional simulated plan executions that can be generated with the saved computation time.

3.4.4. Saving computation time when few actions are used

Section 3.4.1 covers creating a superset of the relevant actions. Let us denote this set as $A = A^r \bigcup A^i$, where r denotes the relevant subset, and i the irrelevant subset. It is useless to perform computations for actions in A^i: they will never be eligible and their eligibility traces will remain null. But we do not know in advance which actions belong to A^i.

In the same vein, one can observe that an action's eligibility vector $\mathbf{e}_a$ remains null during an episode as long as it has not been eligible. During an episode, many actions will remain ineligible for a long time. So many computations can be avoided by just storing the actions that have been eligible at least once. Irrelevant actions will never become eligible. The result is:

- when restarting from the initial state, set *hasBeenEligible*(a) to `false` for all $a \in A$;
- when action a is eligible, set *hasBeenEligible*(a) to `true`;
- until a becomes eligible once, do not update the corresponding part of the eligibility vector ($\mathbf{e}_a$), and parameter vector ($\boldsymbol{\theta}_a$): both remain 0.

We have yet to exploit the same strategy to reduce memory usage. If one suspects that there are many irrelevant actions in A, we could allocate memory for an action's parameter vector and eligibility vector only when the action is encountered for the first time.

3.4.5. Software

The reinforcement learning routines were provided by LibPG, a C++ library written by the authors, and used in multiple research projects. It makes use of the Boost C++ libraries for matrix operations, and optionally the ATLAS basic linear algebra routines library for faster mathematics.

In the CPTP case Algorithm 3 was implemented from scratch in C++ for the class of domains described in 2.1. As discussed, XML was used to describe the domains and problems, writing PPDDL to XML translations where necessary. This decision was made because of the ease of parsing and transforming XML across a range of programming languages. In the PP case we tied LibPG to the MDPSim package, which is the official simulator for the IPC.

Table 1
Main parameter settings of FPG-IPC as used during IPC-5 and of FQL. Other TD discount factors have been experimented ($\lambda = 0.8$ and $\lambda = 1$) but with near-identical results.

	step size	discount factor	λ	success reward	progress reward
FPG	$\alpha = 0.00005$	$\beta = 0.85$	N/A	$r_s = 1000$	$r_{progress} = 100$
FQL	$\alpha = 0.001$	discount = 0.85	$\lambda = 0$	$r_s = 1000$	$r_{progress} = 100$

Table 2
Summary of non-temporal domain results on IPC-5 benchmarks. Values are % of plan simulations that reach the goal (out of 30 runs). A dash indicates the planner was not run, or failed to produce results.

Domain	FOALP	sfDP	FPG	Paragraph	FF-replan	FQL
BW	100	29	63	–	86	0
Exploding BW	24	31	43	31	52	7
Tire	82	–	75	91	82	11
Zeno	–	7	27	7	100	7
Drive	–	–	63	9	71	11
Elevator	100	–	76	–	93	1
Pitchcatch	–	–	23	–	54	0
Schedule	–	–	54	1	51	54
Random	–	–	65	5	100	10

4. Experiments

All the domains and source code for the following experiments are available from http://fpg.loria.fr/.

4.1. Probabilistic planning

Because the probabilistic planning setting of the International Planning Competition is simpler and widely known, we first present experiments performed with FPG-IPC. In this section, we refer to FPG-IPC simply as FPG, since no confusion is possible.

4.1.1. IPC

One benefit of the IPC is to offer a variety of benchmarks recognised by the AI planning community. This was the main reason for developing FPG-IPC. Many of the approximations made in FPG were designed to cope with a combinatorial action space, thus there was little reason to believe FPG would be competitive in non-temporal domains.

In the competition there was a time limit of 40 minutes for each problem instance including optimisation and evaluation. Table 2 shows the overall summary of results from the IPC-5, by domain and planner. These results include FF-replan and FQL which were run after the competition but following similar rules. FQL had a slight advantage because it was run on a more powerful machine that the competition server (Core2 DUO 3.2 GHz), and we only limited the optimisation time (10 minutes), not the evaluation time. FPG-IPC's and FQL's parameters were manually tuned to give the best results across various domains from IPC-4 and from preliminary benchmarks of IPC-5, however they were not further tuned to the competition problems. These parameters are given in Table 1.

The results are based on 9 PPDDL specified domains, averaged over 15 instances from each domain, and tested on 30 simulations of plans for each instance. This is too few simulations to reliably measure the performance of a planner, but competition time constraints required a small number. Many of these domains, such as Blocksworld (BW), are classical deterministic domains with noise added to the effects.

FPG and FF-replan prove to be much more robust to a variety of domains than the other planners. They have been able to run on all problem instances and return results on most of them. Software maturity explains some of this performance: FPG and FF-replan are both largely based on stable software (FPG$\longleftarrow$MDPSim + LibPG, FF-replan$\longleftarrow$FF), whereas two of the three other planners were not bug-free at the time of the competition. On some domains Paragraph and FOALP—which are optimal planners, unlike FPG and FF-replan—outperformed other planners on most problem instances. We defer to [48] for details.

FQL turns out to be competitive only on some problems of the Drive, Random, Schedule and Tireworld domains.[9] We surmise that FPG is more successful than FQL because FQL aims at approximating the Q-value of each state-action pairs while it is generally sufficient for FPG to simply return a good action in each state (policies being often deterministic). This is consistent with experiments with the Tetris game which show that learning a ranking function is more efficient than learning a value function [49]. We note that this is empirical evidence only. There are situations in which the additional information encoded in values might be advantageous to learning, allowing faster convergence than PG methods that are

[9] Problems Ex-blocksworld(p03) and Zenoworld(p01) are trivial since the initial state is a goal state.

Table 3
FPG's (in)stability: mean and standard deviation of the success rate over $N = 30$ runs on various problems ($\alpha = 0.0001$).

Domain	Mean ± StdDev	Min	Max	Note
Blocksworld p10	67.0 ± 41.9	0	100	6/10 policies found goal > 90%
Zenoworld p05	39.7 ± 15.0	26	70	
Schedule p09	44.9 ± 4.46	43	51	

known for their high variance. We have not yet come across such a domain in the planning context. The Schedule domain is one where FQL and FPG perform equally well on each problem instance.

FQL is more difficult to tune than FPG because the exploration policy has to be set explicitly and a method found to decay exploration at the correct rate, which is tied to the step size used and the value of λ. For FQL we tried $\lambda = \{0, 0.8, 1\}$. The variance in the final results was not significant. We quote the results for $\lambda = 0$, the standard version of Q-learning. It is probable that further tuning would result in significant improvements to FQL's results. FPG also requires tuning a step size and discount factor. However, this process is simpler than FQL's, amounting to tuning the discount factor for a small step size, and then increasing the step size until convergence becomes unstable on some domains.

4.1.2. Instabilities

Because it is a randomised algorithm, FPG will find different results each time it is run with a different random seed. They may for instance:

- fall in different local optima;
- not reach a locally optimal policy because the learning had insufficient time to converge; or
- diverge due to the use of an inappropriately large step size.

In some domains of the IPC (Blocksworld, Zenotravel, Elevators, and to a lesser extent, Random and Exploding-Blocksworld) there is a clear divide between "easy" and "hard" problem instances. FPG achieves a 100% success rate on easy ones and near 0% on hard ones. Table 3 presents experimental results on some problem instances identified as leading to FPG being unstable. It gives, for each problem, the average success rate (±standard deviation), and the minimal and maximal success rates obtained on 10 repeats of the optimisation and evaluation cycle.

We could implement automatic random restarts to get more robust results on such problems. Yet, this solution was avoided due to the time constraints of the competition. On easy problems restarts find very similar solutions, and for very hard problems all restarts fail. It was a more efficient strategy to make a single complete gradient ascent for each problem and move to the next problem in the case of failure.

4.1.3. Validating FPG's speed up tricks

Section 3.4.3 presented two ideas for avoiding useless computations. To illustrate their benefit FPG was run several times on the same problem. The same random seed was used each time so that the learning process was identical. Optimisation was limited to 15 minutes but with different combinations of speed-up tricks. Fig. 5 shows in each case the computation time as a function of the number of simulation steps since the start of optimisation.

For an example of how many grounded actions might become eligible, we calculated some statistics for IPC-5 Blocksworld p07:

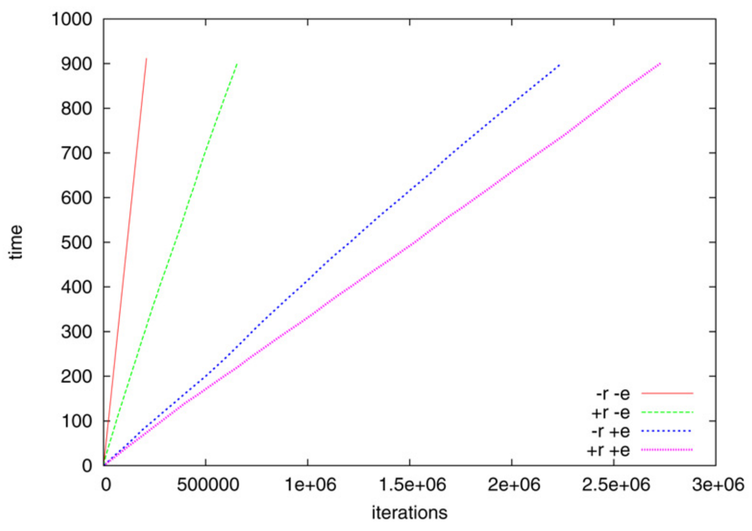
- between 120 and 200 actions are eligible at any single point in time during an episode (with a max episode length of 750); and
- 1770 out of 2310 actions turn out to be eligible at some point during the episode.

4.1.4. Benefits of the progress estimator

The efficiency of the progress estimator is illustrated with IPC-5 Blocksworld problem p07. Fig. 6 shows learning curves of FPG with four different weights ρ for the progress estimator. While the reward in case of success is always set to $r_s = 1000$, the progress reward is the number of goal facts added or subtracted multiplied by $\rho = \{0, 1, 10, 100\}$. Fig. 6(a) shows the average number of goals reached per simulation step. Fig. 6(b) shows the average reward per simulation step, that is, $R(\boldsymbol{\theta})$. All curves are averages over 10 runs, with error bars showing the standard deviation.

This problem is hard enough that a progress estimator is necessary in order for FPG to initially find the goal, and then bootstrap into a good policy. The learning time is limited to 15 minutes. Because uniformly scaling rewards is equivalent to scaling the step-size bigger r_p values lead to faster learning. But the resulting policy after 900 seconds is better with $r_p = 10$ than $r_p = 100$ because the latter favours short-term rewards linked to progresses rather than long-term and non-guaranteed rewards linked to goal states. Plus, the error bars show a higher standard deviation when $r_p = 100$, which is due to FPG learning too fast.

We present figures with different x-axes on purpose to show that the number of simulation steps performed in 900 seconds depends on the setting. The fastest setting is with $r_p = 0$ because never receiving a reward is a good way to avoid excessive computations.



$$\frac{\texttt{-r -e}}{\texttt{-r -e}} = 1.0 \quad \frac{\texttt{+r -e}}{\texttt{-r -e}} = 3.2 \quad \frac{\texttt{-r +e}}{\texttt{-r -e}} = 10.8 \quad \frac{\texttt{+r +e}}{\texttt{-r -e}} = 13.2$$

Fig. 5. Speed-up gained from avoiding useless parameter updates. `+r` and `-r` indicate whether the speed-up based on observing the **r**eward was used or not. `+e` and `-e` indicate whether the speed-up relying on whether an action has been **e**ligible or not.

4.1.5. When FF-replan fails

A second set of domains (one instance each) demonstrates domains introduced by [30] that are more challenging for replanners. They are characterised by short plans having a high failure probability. Deterministic planners like FF often look for the shortest path to the goal, without the ability to avoid dangerous paths. Replanners may also fail to optimise the cost-to-go, which was not measured in IPC-5. Results are shown in Table 4. The optimal Paragraph planner does very well until the problem size becomes too large. Triangle-tire-4 in particular shows a threshold for this domain where an approximate probabilistic planning approach is required in order to find a good policy.

To summarise the probabilistic planning results, FPG appears to be a good compromise between the scalability of re-planning approaches, and the capacity of optimal probabilistic planner to perform reasoning about uncertainty. Note that we shall describe FPG's limitations after describing its performance on CPTP domains.

4.2. CPTP

These experiments compare FPG to two earlier probabilistic temporal planners: Prottle [4], and a Military Operations (MO) planner [23]. The MO planner uses LRTDP, and Prottle uses a hybrid of AO* and LRTDP. They both require storage of state values but attempt to prune off large branches of the state space. The Prottle planner has the advantage of using good heuristics to prune the state space. The MO planner did not use heuristics.

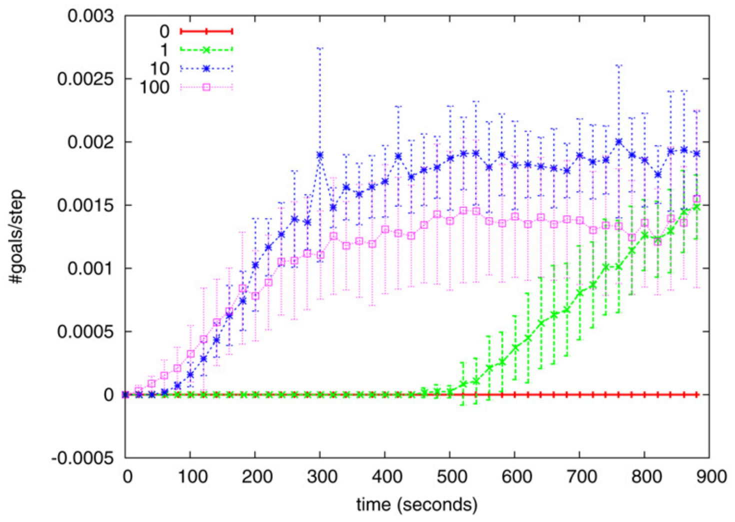
We present results along three criteria: the probability of reaching a goal state, the average makespan (including executions that end in failure), and the long-term average reward (FPG only). However, each planner uses subtly different optimisation criteria:

- *FPG* — maximises the average reward per step $R = 1000\frac{(1-Pr(fail))}{steps}$, where steps is the average number of decision points in a plan execution, which is related to the makespan;
- *Prottle* — minimises the probability of failure;
- *MO* — minimises the cost-per-trial, here based on a weighted combination of $\mathbb{P}(failure)$, makespan, and resource consumption.

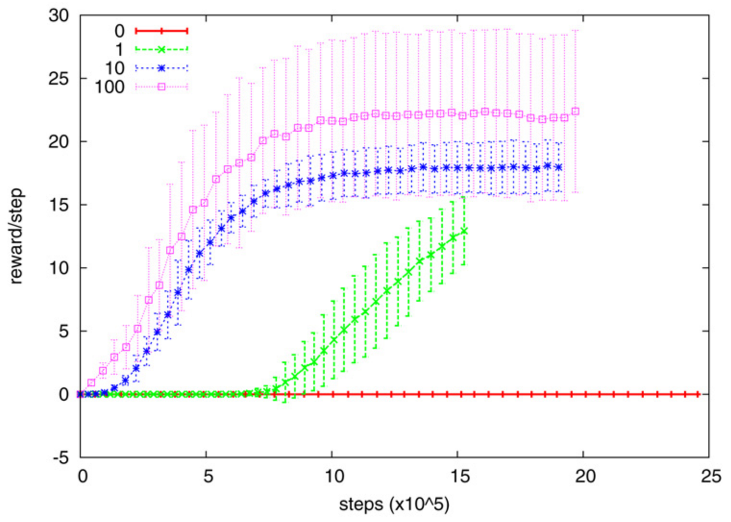
It may not be clear why FPG's minimisation of steps also helps to minimise makespan. Steps occur at decision points. The way to minimise decision points is to start as many actions as possible in a single command, maximising concurrency. It is also very easy to adapt the reward function to emphasise other criteria such as resource consumption.

The first three domains are:

Probabilistic Machine Shop (MS) [3]: Multiple sub-actions such as shape, paint, and polish need to be performed on different objects using different machines, possibly in parallel. Not all machines are capable of every action and they cannot work on the same object concurrently. Objects need to be transported from one machine to another for different sub-actions. The version we used was based on Mach6, the largest variant used in [3], but was subsequently modified exactly



(a) Number of successes per simulation step (as a function of wall clock time).



(b) Average reward per simulation step (as a function of simulation time).

Fig. 6. FPG's learning curve on Blocksworld problem p07 for several settings of the progress estimator (the success reward is always $r_s = 1000$).

Table 4
Summary of results on non-replanner-friendly domains. Values are % of plan simulations that reach the goal (minimum 30 runs). A dash indicates the planner was run, but failed to produce results, typically due to memory constraints. A starred result indicates a theoretical upper limit for FF-replan that it failed to reach in these experiments.

Domain	Paragraph	FPG	FF-replan	Prottle
Climber	100	100	62	100
Bus fare	100	22	1	10
Tri-tire 1	100	100	50	–
Tri-tire 2	100	92	13*	–
Tri-tire 3	100	91	3*	–
Tri-tire 4	3	68	0.8*	–

as in the original Prottle experiments.[10] It has 9 durative actions and 13 predicates that expand to 38 grounded actions and 28 grounded predicates. The maximum makespan is 20 (Prottle used 15).

Maze (MZ) [4]: Maze is based on the idea of moving between connected rooms and finding the keys necessary to unlock closed doors. There are doors and keys of different colours, and it is possible to try unlocking several doors at the same time

[10] Some actions have nested probabilistic events and outcome-dependant durations.

Table 5
Results on 3 benchmark domains. The experiments for MO and FPG were repeated 100 times. *Success%* = percentage of successful executions, *MS* = makespan, *R* is the final long-term average reward, and *Time* is the optimisation time in seconds.
FPG-L = FPG + linear network. FPG-T = FPG + tree of experts.

Problem	Algorithm	*Success%*	*MS*	*R*	*Time*
MS	$\overline{\text{FPG-L}}$	98.6	6.6	118	532
MS	FPG-L	99.9	5.5	166	600
MS	$\overline{\text{FPG-T}}$	30.0	13	20.9	439
MS	FPG-T	35.0	13	21.4	600
MS	Prottle	97.1			272
MS	MO		Out of memory		
MS	random	0.7	18	0.1	
MS	naïve	0.0	20	0.0	
MZ	$\overline{\text{FPG-L}}$	19.1	5.5	134	371
MZ	FPG-L	85.3	6.9	130	440
MZ	$\overline{\text{FPG-T}}$	80.3	5.5	136	29
MZ	FPG-T	84.7	5.7	115	17
MZ	Prottle	82.2			10
MZ	$\overline{\text{MO}}$	92.1	8.0		71
MZ	MO	92.8	8.2		72
MZ	random	23.5	13	16.4	
MZ	naïve	9.2	16	8.6	
TP	$\overline{\text{FPG-L}}$	34.4	18	298	340
TP	FPG-L	66.7	18	305	600
TP	$\overline{\text{FPG-T}}$	65.6	18	302	258
TP	FPG-T	66.7	18	301	181
TP	Prottle	20.2			442
TP	MO		Out of memory		
TP	random	0.4	15	1.0	
TP	naïve	0.0	19	0.0	
PitStop	$\overline{\text{FPG-L}}$	100.0	20180	142	41
PitStop	random	71.0	12649	41.0	
PitStop	naïve	0.0	66776	0.0	
500	FPG-T	97.5	158	1.56	3345
500	random	23.4	765	0.231	
500	naïve	30.5	736	0.100	

(or grab keys). Actions all have a duration of 1 or 2, and many have nested probabilistic outcomes. There are 6 durative actions and 7 predicates that expand to 165 grounded actions and 207 grounded predicates. Plans fail if the makespan reaches 20 units (Prottle used 10). Despite the large number of actions, this problem is easy to solve compared to Machine Shop.

Teleport (TP) [4]: Objects can teleport between locations, if their objects have been successfully 'linked' to their destination. There are 'fast' and 'slow' forms of teleporting, but 'slow' has a higher probability of success. There is a fixed number of links, each with a fixed source location. One can change the destination of multiple links at the same time, and also try to teleport oneself along a stable link. The problem has actions with outcome-dependent durations. It has 3 durative actions and 3 predicates that expand to 63 grounded actions and 24 grounded predicates. Plans fail if the makespan reaches 25 units (Prottle used 20).

We additionally introduce two novel domains to illustrate particular strengths of FPG.

PitStop: A proof-of-concept continuous duration uncertainty domain representing alternative pit stop strategies in a car race, a 2-stop strategy versus a 3-stop. For each strategy a pit-stop and a racing action are defined. The 3-stop strategy has shorter racing and pitting time, but the pit stop only injects 20 laps worth of fuel. The 2-stop strategy has longer pit times, but injects 30 laps worth of fuel. The goal is to complete 80 laps. The pit-stop actions are modelled with Gaussian durations. The racing actions take a fixed minimum time but there are two discrete outcomes (with probability 0.5 each): a clear track adds an exponentially distributed delay, or encountering backmarkers adds a normally distributed delay. Thus this domain includes *continuous* durations, discrete outcomes, and metric functions (fuel counter and lap counters).

500: To provide a demonstration of scalability and parallelisation we generated a 500 grounded actions, 250 predicates domain as follows: the goal state required 18 predicates to be made true. Each action has two outcomes, with up to 6 effects and a 10% chance of each effect being negative. Two independent sequences of actions are generated that potentially lead to the goal state with makespan of less than 1000 (but there may be many more than two possible routes the goal). There are 40 types of resource, with 200 units each. Each action requires a maximum of 10 units from 5 types, potentially consuming half of the occupied resources permanently. Resources limit how many actions can start.

Our experiments used a combination of: (1) FPG with the linear network (FPG-L) action policies; (2) FPG with the tree (FPG-T) action policy shown in Fig. 4; (3) the MO planner; (4) Prottle; (5) a random policy that starts eligible actions with a coin toss; (6) a naïve policy that attempts to start *all* eligible actions in each command.

```
MOVE (P1 START L3): Return No;
MOVE (P1 L1 START): if (eligible)
                         if (fast action) return Yes;
                         else return 54% Yes, 46% No;
                    else return No;
MOVE (P1 L3 START): if (eligible) return Yes;
                    else return No;
```

Fig. 7. Three decision-tree action sub-policies extracted the final Maze policy. Returning 'Yes' means start the action.

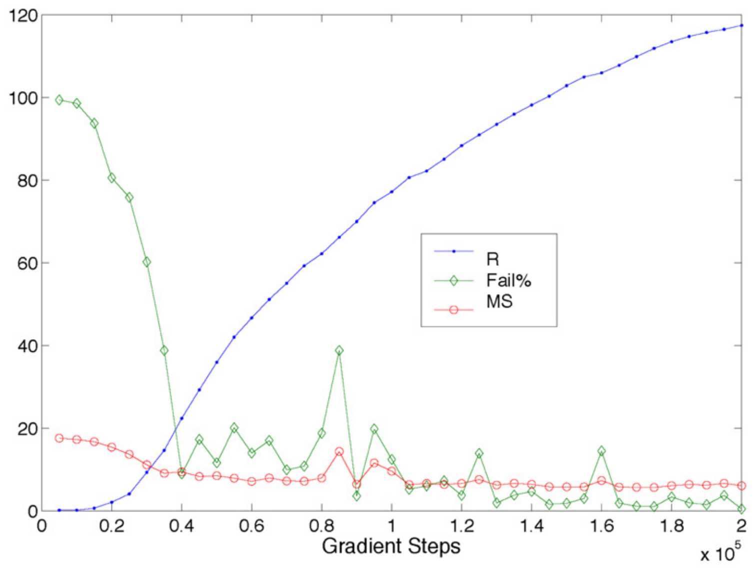


Fig. 8. Relative convergence of long-term average reward R, failure probability, and makespan over a single linear network FPG optimisation of Machine Shop. The y-axis has a common scale for all three units.

All of these experiments were limited to 10 minutes. Other parameters are described in Table 7. In particular, the single gradient step size α was selected as the single highest value that ensured reliable convergence over 100 runs over all domains. Experiments in this section were conducted on a dedicated 2.4 GHz Pentium IV processor with 1 GB of ram. The results are summarised in Table 5. Reported success percentage and makespan was estimated from 10,000 simulated executions of the optimised plan. Prottle results were taken directly from [4], quoting the highest probability of success result. FPG and MO optimisations/evaluations were repeated 100 times to investigate the impact of local minima. The $\overline{\text{FPG}}$ and $\overline{\text{MO}}$ results show the mean result over 100 runs, and the unbarred results show the single best run out of the 100, measured by probability of success. The small differences between the mean and best results indicate that local minima were not severe.

The random and naïve experiments are designed to demonstrate that optimisation is necessary to get good results. In general, Table 5 shows that FPG is at least competitive with Prottle and the MO planner. FPG achieves the best performance in Machine Shop. The poor performance of Prottle in the Teleport domain—20.2% success compared to FPG's 65.6%—is due to Prottle's short maximum permitted makespan of 20 time units. At least 25 units are required to achieve a higher success probability. We also observe that FPG's linear action policy generally performs slightly better than the tree, but takes longer to optimise. This is expected given that the linear action-policy can represent a much richer class of policies at the expense of more parameters. In fact, it is surprising that the decision tree does so well on all domains except Machine Shop, where it only reduces the success rate to 30% compared to the 99% the linear policy achieves.

We explored the types of policy that the decision tree structure in Fig. 4 produces. The pruned decision tree policy for three grounded Maze actions is shown in Fig. 7. The first action is pruned such that it *never* runs. The third action always runs when it is eligible to do so. The second action is more interesting: FPG has decided this is a useful action if there is some predicate it can set faster than any other eligible action. Over the 165 Maze actions, the majority had been optimised to never start, or always start, or had not been optimised at all. The latter case indicates that the actions were never part of the active plan for long. The good performance of the simplistic tree-policy indicates that for our test domains—and possibly many others—a large part of the planning effort is deciding which grounded actions are useful in the final plan. Machine shop exhibited a significant difference between the linear and decision tree action policy, indicating that it is a more complex domain. FPG with a simple decision tree like this could be used to generate control knowledge for reasoning based planners. For example, a pre-processing stage FPG could determine which actions can be immediately discarded.

Table 6
FPG's results when optimisation is terminated at 75% of the mean R achieved in Table 5.

Problem	Algorithm	*Success%*	*MS*	*R*	*Time*
MS	$\overline{\text{FPG-L}}$	95.3	7.1	89	37
MS	FPG-L	99.9	6.5	89	32
MS	$\overline{\text{FPG-T}}$	29.6	13	16	10
MS	FPG-T	34.1	14	16	14
MZ	$\overline{\text{FPG-L}}$	80.7	5.5	100	13
MZ	FPG-L	85.3	5.6	100	22
MZ	$\overline{\text{FPG-T}}$	80.3	5.5	102	2
MZ	FPG-T	84.2	7.0	102	2
TP	$\overline{\text{FPG-L}}$	65.4	18	224	3.5
TP	FPG-L	67.0	19	224	4
TP	$\overline{\text{FPG-T}}$	65.3	18	226	2.3
TP	FPG-T	66.9	18	226	1

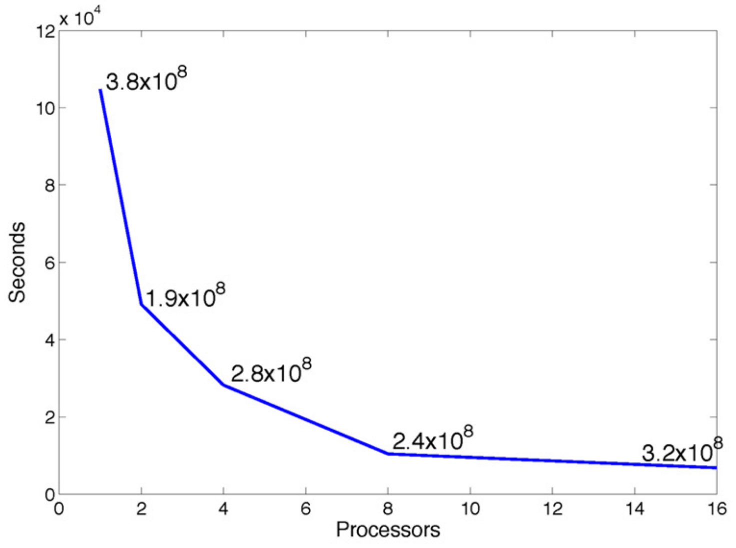


Fig. 9. Convergence times for the 500 action experiment run in parallel mode on a varying number of processors. The numbers on the curve show total aggregated simulation steps taken.

Table 5 shows that Prottle achieves good results *faster* on Maze and Machine Shop. The apparently faster Prottle optimisation is due to the asymptotic convergence of FPG using the criterion *optimise until the long-term average reward fails to increase for 5* $R(\boldsymbol{\theta})$ *estimations of* 10,000 *steps each*. In reality, good policies are achieved long before convergence to this criterion. To demonstrate this we plotted the progression of a single optimisation run of FPG-L on the Machine Shop domain in Fig. 8. The failure probability and makespan settle near their final values at a reward of approximately $R = 85$, however, the mean long-term average reward obtainable for this domain is $R = 118$. In other words, the tail end of FPG optimisation is removing unnecessary no-ops. To further demonstrate this, and the any-time nature of FPG, optimisation was stopped at 75% (chosen arbitrarily) of the average reward obtained with the stopping criterion used for Table 5. The new results in Table 6 show a reduction in optimisation times by orders of magnitude, with very little drop in the performance of the final policies.

The experimental results for the continuous time PitStop domain show FPG's ability to optimise under mixtures of discrete and continuous uncertainties.

Results for the 500 action domain are shown for running the parallel version of FPG algorithm with 16 processors. No other CPTP planner we know of is capable of running domains on this scale. As expected, we observed that optimisation times dropped inversely proportional to the number of processors for up to 16 processors. This is shown in Fig. 9. However, on a single processor the parallel version requires double the time of OLPOMDP. This is due to the less efficient use of step-by-step gradients, and somewhat due to the communication overheads of parallelisation. As the number of processors increases, we would observe these overheads grow for a fixed problem size, until the point where adding processors would decrease performance.

Table 7
Parameter settings not discussed in the text.

Param	Val.	Opt.	Notes
$\boldsymbol{\theta}$	0	All FPG	Initial $\boldsymbol{\theta}$
α	1×10^{-5}	FPG-L	
α	5×10^{-5}	FPG-T	
β	0.95	FPG	Both L&T
ϵ	1	MO	LRTDP Param
ϵ	0.0 to 0.6	Prottle	Prottle Param
T	6×10^{6}	Parallel FPG-T	For search dir.
T	1×10^{6}	Parallel FPG-T	For line search

Table 8
Success probability on the XOR problem.

Domain	FPG-linear	FPG-MLP	FF-replan
XOR	$\langle 74 \pm 5\%, 1.0\rangle$	$\langle 75 \pm 5\%, 0.72\rangle$ $\langle 100\%, 0.28\rangle$	$\langle 100\%, 1.0\rangle$

4.3. *When FPG fails*

As with any reinforcement learning algorithm relying on function approximation, FPG may fail if the function approximator is not capable of representing a good policy. This is easy to demonstrate with FPG-IPC using the *XOR* problem defined by:

- x and y are randomly initialised boolean predicates;
- action a leads to plan success if and only if $x \neq y$ (otherwise the plan fails); and
- action b leads to plan success if and only if $\neg(x \neq y)$ (otherwise the plan fails).

Using linear function approximators, the best policies always opt for one action in 1 out of 4 of the possible states, and the other action in the remaining states. This results in a 75% probability of success. But if we make the function approximation richer by adding a hidden layer—a standard multi-layer perceptron (MLP)—with two squashed hidden units then we can represent a policy that achieves 100% success.

Experiments have been conducted for FPG-linear, FPG-MLP and FF-replan. We observed in this problem that FPG converges to various local optima. When hidden layers are used the second layer parameters must be initialised randomly. This is why FPG-linear and FPG-MLP have been run 100 times each on this problem, and the resulting scores have been automatically clustered to identify local optima (the number of clusters was initialised to 2).

Table 8 lists identified clusters for each algorithm. A cluster is described by a tuple $\langle$mean $\pm$ standard deviation, weight$\rangle$. As expected, FF-replan always finds the goal. FPG-linear happens to give stable results confirming the theoretical value of 75% success. This 75% value policy is also, unsurprisingly, a local optimum for FPG-MLP, which reaches 100% success probability after only 28% of the optimisations.

This toy problem not only illustrates the fact that a linear network may not be able to represent an optimal solution, but also exhibits a situation where the gradient ascent tends to fall into local optima. Greater tuning of the number of hidden units and step size would probably overcome this problem, but such tuning would also likely be highly domain specific, which is undesirable for automated planning. Another solution is enriching the observation space, perhaps with the classical strategy used when such complex preconditions or conditional-effects appear in a planning problem: duplicate each action so that each copy has its own simple preconditions [50].

On the other hand we conducted many experiments with MLPs on the IPC problems, and *never* achieved better performance with them. We suspect that with appropriate encoding of the observation, a linear approximator will produce good policies in most domains.

There are also domains for which, even if FPG could easily represent the optimal policy, it is too difficult to learn. As we studied in Section 4.1.4, the progress estimator is crucial for achieving good results in domains such as Blocksworld. However, the larger blocks world domains still proved too difficult for FPG. This hints at domains that are generally challenging for FPG. They are characterised by requiring long chains of correct actions to obtain rewards. A random policy essentially walks around the state space for a long time without achieving a reward. In these cases the failure mode of FPG is that it will optimise forever without achieving a gain in long-term reward. Essentially it stays perpetually in a flat region of gradient space. However, such domains are challenging for many planners, and are a great motivation for reasoning about the structure that exists in domains such as Blocksworld.

5. Discussion and future work

FPG diverges from traditional planning approaches in two key ways: we search for plans directly, using local optimisation procedures; and we approximate the policy representation with a factored collection of parameterised policies. Apart from reducing the representation complexity, this approach allows policies to generalise to states not encountered during training. This is an important feature of FPG's "learning" approach. We obtain a similar effect by restricting the input of the policies to only part of the state. That is, we concentrate on the predicate values. If an action a is useful given observation $\mathbf{o}$, it may be generally useful for similar observations and therefore similar states.

FPG scales in the sense that its memory use and *per step* computation times grow linearly with the domain. However, the total number of gradient steps needed for convergence is a function of the *mixing time* of the underlying POMDP, which can grow exponentially with the number of state variables. How to compute the mixing time of an arbitrary MDP is an open problem, which in turn hints at the difficulty of assessing in advance the hardness of an arbitrary planning problem.

In recent years, policy-gradient algorithms have been developed that make a better use of the samples they get by employing approximate second order gradient ascent, and/or the use of critic value estimates [51,52]. They are of particular interest when obtaining samples has a high cost. However, these algorithms have a time complexity cost of at least $O(k^2)$ per gradient step, where k is the number of parameters plus observation dimensions. In our setting, samples are obtained using a fast simulator. Using the available time to generate more samples was better than performing $O(1000^2)$ matrix operations for every step taken by the simulator.

It is strange to use the planning model information only to build a plan execution simulator. We did this to avoid intractable computations on large models. However, we believe that a hybrid of dynamic programming/RL algorithms is highly desirable to achieve the best of both approaches. This might be achieved through a Rao-Blackwellisation of the gradient estimates, reducing their variance by using the known per-step transition probabilities.

We also believe that fast and efficient methods from non-probabilistic planning can be used to bootstrap probabilistic planners. This can be done in several ways, including replanning. One method we have experimented with uses FF-replan to suggest actions to help FPG reach the goal in domains where random actions tend not to reach the goal [53]. Over time, FPG takes over from FF-replan in the choice of actions, optimising for the probabilistic structure.

6. Conclusion

To conclude, FPG is a demonstration that Monte-Carlo local optimisation methods can supplement AI planning methods. This is particularly true in domains that involve large or infinite state spaces, uncertainty, and continuous quantities such as time. Ultimately, we believe that hybrid planning/learning approaches will become the state-of-the art for complex domains.

Acknowledgements

This work was supported by National ICT Australia, funded by the Australian Government's Backing Australia's Ability program and the Centre of Excellence program. This project was also funded by the Australian Defence Science and Technology Organisation. Thank you to Sylvie Thiébaux and Iain Little for many helpful insights. We also wish to thank the organisers of the IPC-5 probabilistic track for opportunity to test FPG and for the subsequent feedback.

References

[1] A. Barto, S. Bradtke, S. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1995) 81–138.
[2] E. Hansen, S. Zilberstein, LAO*: A heuristic search algorithm that finds solutions with loops, Artificial Intelligence 129 (2001) 35–62.
[3] Mausam, D.S. Weld, Concurrent probabilistic temporal planning, in: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS'05), Monterey, CA, 2005.
[4] I. Little, D. Aberdeen, S. Thiébaux, Prottle: A probabilistic temporal planner, in: Proceedings of the Twentieth American National Conference on Artificial Intelligence (AAAI'05), 2005.
[5] D.P. Bertsekas, J.N. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, 1996.
[6] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge MA, ISBN 0-262-19398-1, 1998.
[7] A. Fern, S. Yoon, R. Givan, Approximate policy iteration with a policy language bias: Solving relational Markov decision processes, Journal of Artificial Intelligence Research 25 (2006) 85–118.
[8] R.J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine Learning 8 (1992) 229–256.
[9] R.S. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: Advances in Neural Information Processing Systems (NIPS'99), vol. 12, MIT Press, 2000, pp. 1057–1063.
[10] J. Baxter, P. Bartlett, L. Weaver, Experiments with infinite-horizon, policy-gradient estimation, Journal of Artificial Intelligence Research 15 (2001) 351–381.
[11] H. Kimura, K. Miyazaki, S. Kobayashi, Reinforcement learning in POMDPs with function approximation, in: Proceedings of the Fourteenth International Conference on Machine Learning (ICML'97), Morgan Kaufmann, 1997, pp. 152–160.
[12] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains, Journal of Artificial Intelligence Research 20 (2003) 61–124.
[13] L. Peshkin, K.-E. Kim, N. Meuleau, L.P. Kaelbling, Learning to cooperate via policy search, in: Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI'00), 2000.
[14] N. Tao, J. Baxter, L. Weaver, A multi-agent, policy-gradient approach to network routing, in: Proceedings of the Eighteenth International Conference on Machine Learning (ICML'01), Morgan Kaufmann, 2001.
[15] M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J.C. jung Hsu, A. Jonsson, B. Kanefsky, P. Morris, K. Rajan, J. Yglesias, B. Chafin, W. Dias, P.F. Maldague, MAPGEN: Mixed-initiative planning and scheduling for the Mars exploration rover mission, IEEE Intelligent Systems 19 (1) (2004) 8–12.

[16] C. Gretton, Gradient-based relational reinforcement-learning of temporally extended policies, in: Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS'07), 2007.
[17] H.L.S. Younes, M.L. Littman, PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects, Tech. Rep. CMU-CS-04-167, Carnegie Mellon University, October 2004.
[18] H.L.S. Younes, Extending PDDL to model stochastic decision processes, in: Proceedings of the ICAPS'03 Workshop on PDDL, 2003.
[19] W. Cushing, S. Kambhampati, Mausam, D. Weld, When is temporal planning really temporal? in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'07), Hyderabad, India, 2007.
[20] Mausam, P. Bertoli, D. Weld, Planning with durative actions in stochastic domains, Journal of Artificial Intelligence Research.
[21] B. Bonet, B. Givan, Results of probabilistic track in the 5th international planning competition, in: Not in the Proceedings of the Fifth International Planning Competition (IPC-5), 2006.
[22] Mausam, D.S. Weld, Probabilistic temporal planning with uncertain durations, in: Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS'06), 2006.
[23] D. Aberdeen, S. Thiébaux, L. Zhang, Decision-theoretic military operations planning, in: Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS'04), 2004, pp. 402–411.
[24] H.L.S. Younes, R.G. Simmons, Policy generation for continuous-time stochastic domains with concurrency, in: Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS'04), 2004.
[25] S. Sanner, C. Boutilier, Practical linear value-approximation techniques for first-order MDPs, in: Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI'06), 2006.
[26] I. Little, S. Thiébaux, Concurrent probabilistic planning in the graphplan framework, in: Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS'06), 2006.
[27] P. Fabiani, F. Teichteil-Königsbuch, Symbolic focused dynamic programming for planning under uncertainty, in: Proceedings of the IJCAI'05 Workshop on Reasoning with Uncertainty in Robotics (RUR'05), 2005.
[28] S. Yoon, A. Fern, B. Givan, FF-Replan: a baseline for probabilistic planning, in: Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS'07), 2007.
[29] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.
[30] I. Little, S. Thiébaux, Probabilistic planning vs replanning, in: Proceedings of the ICAPS'07 Workshop on the International Planning Competition: Past, Present and Future, 2007.
[31] D. Aberdeen, Policy-gradient methods for planning, in: Advances in Neural Information Processing Systems (NIPS'05), vol. 18, MIT Press, 2006.
[32] D. Aberdeen, O. Buffet, Concurrent probabilistic temporal planning with policy-gradients, in: Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS'07), Providence, USA, 2007.
[33] Y. Xu, A. Fern, S. Yoon, Discriminative learning of beam-search heuristics for planning, in: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), 2007.
[34] S. Dzeroski, L.D. Raedt, K. Driessens, Relational reinforcement learning, Machine Learning 43 (2001) 7–52.
[35] A. Ng, D. Harada, S. Russell, Policy invariance under reward transformations: Theory and application to reward shaping, in: Proceedings of the Sixteenth International Conference on Machine Learning (ICML'99), 1999.
[36] T.G. Nicol, N. Schraudolph, Conjugate directions for stochastic gradient descent, in: Proceedings of the International Conference on Artificial Neural Networks (ICANN'02), in: Lecture Notes in Computer Science, vol. 2415, Springer-Verlag, 2002, pp. 1351–1356.
[37] J. Baxter, P.L. Bartlett, Infinite-horizon policy-gradient estimation, Journal of Artificial Intelligence Research 15 (2001) 319–350.
[38] A. Benveniste, M. Metivier, P. Priouret, Adaptive Algorithms and Stochastic Approximation, Springer-Verlag, 1990.
[39] E. Greensmith, P. Bartlett, J. Baxter, Variance reduction techniques for gradient estimates in reinforcement learning, Journal of Machine Learning Research 5 (2004) 1471–1530.
[40] D. Aberdeen, J. Baxter, Scaling internal-state policy-gradient methods for POMDPs, in: Proceedings of the Nineteenth International Conference on Machine Learning (ICML'02), Morgan Kaufmann, Sydney, Australia, 2002.
[41] J. Hoey, R. St-Aubin, A. Hu, C. Boutilier, SPUDD: Stochastic planning using decision diagrams, in: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI'99), 1999, pp. 279–288.
[42] J. Nicholls, Algebraic decision diagrams for reinforcement learning, Tech. rep., Australian National University, honours thesis, September 2005.
[43] C. Boutilier, Sequential optimality and coordination in multiagent systems, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99), 1999.
[44] D. Bernstein, S. Zilberstein, N. Immerman, The complexity of decentralized control of Markov decision processes, in: Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI'00), 2000.
[45] C. Guestrin, M.G. Lagoudakis, R. Parr, Coordinated reinforcement learning, in: Proceedings of the Nineteenth International Conference on Machine Learning (ICML'02), Morgan Kaufmann Publishers Inc., 2002, pp. 227–234.
[46] G.J. Gordon, Reinforcement learning with function approximation converges to a region, in: Advances in Neural Information Processing Systems (NIPS'00), vol. 13, 2001, pp. 1040–1046.
[47] A. Blum, M. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1997) 281–300.
[48] O. Buffet, D. Aberdeen, The factored policy gradient planner, in: Proceedings of the Fifth International Planning Competition (IPC-5), 2006, see http://www.ldc.usb.ve/~bonet/ipc5 for all results and proceedings.
[49] B. Scherrer, A. Boumaza, C. Thiery, Personal communication, 2008.
[50] C. Anderson, D. Smith, D. Weld, Conditional effects in graphplan, in: Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS'98), 1998.
[51] J. Peters, S. Vijayakumar, S. Schaal, Natural actor-critic, in: Proceedings of the Sixteenth European Conference on Machine Learning (ECML'05), in: Lecture Notes in Computer Science, vol. 3720, Springer-Verlag, 2005.
[52] S. Kakade, A natural policy gradient, in: Advances in Neural Information Processing Systems (NIPS'03), vol. 14, 2003.
[53] O. Buffet, D. Aberdeen, FF+FPG: Guiding a policy-gradient planner, in: Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS'07), Providence, USA, 2007.