Concurrent Probabilistic Temporal Planning with Policy-Gradients

Douglas Aberdeen
National ICT australia & The Australian National University
Canberra, Australia
firstname.lastname@anu.edu.au

Olivier Buffet
LAAS-CNRS
University of Toulouse, France
firstname.lastname@laas.fr

Abstract

We present an any-time concurrent probabilistic temporal planner that includes continuous and discrete uncertainties and metric functions. Our approach is a direct policy search that attempts to optimise a parameterised policy using gradient ascent. Low memory use, plus the use of function approximation methods, plus factorisation of the policy, allow us to scale to challenging domains. This Factored Policy Gradient (FPG) Planner also attempts to optimise *both* steps to goal and the probability of success. We compare the FPG planner to other planners on CPTP domains, and on simpler but better studied probabilistic non-temporal domains.

Introduction

To date, only a few planners have attempted to handle general concurrent probabilistic temporal planning (CPTP) domains. These tools have only 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. These domains may include metric functions (e.g., resources), concurrent durative actions, uncertainty in action outcomes *and* uncertainty in action durations. We achieve this by: 1) using gradient ascent for policy search; 2) factoring the policy into simple approximate policies for starting each action; 3) basing policies on only important elements of state (implicitly aggregating similar states); 4) using Monte-Carlo style algorithms that permit sampling continuous distributions and that have memory requirements that are independent of the state space; and 5) parallelising the planner.

The AI planning community is familiar with the value-estimation class of reinforcement learning (RL) algorithms, such as RTDP (Barto, Bradtke, & Singh 1995), and arguably AO*. 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 (Mausam & Weld 2005; Little, Aberdeen, & Thiébaux 2005). 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.

On the other hand, the FPG planner borrows from Policy-Gradient (PG) reinforcement learning (Williams 1992; Sutton *et al.* 2000; Baxter, Bartlett, & Weaver 2001). This class of algorithms does not estimate state-action values, and thus has memory use that is independent of 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 shortest stochastic path problems, such as probabilistic planning, 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, of the system. Stepping the parameters in the direction given by the gradient increases the long-term average reward, improving the policy. Also, PG algorithms are still guaranteed to converge when using approximate policy representations, which is necessitated when the state space is continuous. Our setting has such a 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 our temporal planning setting, an *action* is defined as a single grounded durative action (in the PDDL 2.1 sense). 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 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 (Peshkin *et al.* 2000; Tao, Baxter, & Weaver 2001). 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. Our first parameterised action policy is a simple linear function approxima-

```
<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>
```

Figure 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.

tor that takes the truth value of the predicates at the current planning state, and outputs a probability distribution over whether to start the action. 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 understandable plan. Thus, the second parameterised policy we explore is a decision tree of high-level planning strategies.

Background

Concurrent Probabilistic Temporal Planning

FPG's input language is the temporal STRIPS fragment of PDDL2.1 but extended with probabilistic outcomes and uncertain durations, as in Younes & Littman (2004) and Younes (2003). In particular, we support continuous uncertain durations, functions, at-start, at-end, over-all conditions, and finite probabilistic action outcomes. We additionally allow effects (probabilistic or otherwise) to occur at any time within an action's duration. FPG's input syntax is actually XML. 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. 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 we only need to *simulate* executions of the plan to estimate gradients. A benefit of this approach is that we can trivially sample from both continuous and discrete distributions, whereas enumerating continuous distributions is impossible.

With N eligible actions there are up to 2^N possible commands. Current planners explore this action space systematically, attempting to prune actions 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 policies 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 state will have similar policies.

Previous Work

Previous probabilistic temporal planners include DUR (Mausam & Weld 2006), Prottle (Little, Aberdeen, & Thiébaux 2005), and a Military Operations (MO) planner (Aberdeen, Thiébaux, & Zhang 2004). All these algorithms use some optimised form of dynamic programming (either RTDP (Barto, Bradtke, & Singh 1995) or AO*) to associate values with each state/action pair. However, this requires that values be stored for each encountered state. Even though these algorithms prune off most of the state space, their ability to scale is still limited by memory size. Tempastic (Younes & Simmons 2004) 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. CPTP, Prottle, and Tempastic minimise either plan duration or failures. The FPG planner trades-off these metrics via a natural objective function.

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 successful entrants included FOALP and Paragraph. FOALP (Sanner & Boutilier 2006) solves a first order logic representation of the underlying *domain* MDP, prior to producing plans for specific problems drawn from that domain. Paragraph (Little & Thiébaux 2006) is based on Graphplan extended to a probabilistic framework. A surprisingly successful approach to the competition domains was FF-rePlan (Yoon, Fern, & Givan 2007), winning the 2004 competition and a subsequent version could have achieved 1st place at the 2006 competition. FF-rePlan uses the FF heuristic (Hoffmann & Nebel 2001) to quickly find a potential short path to the goal. It does so by creating a deterministic version of the domain, thus does not directly attempt to optimise the probability of reaching the goal.

Policy gradient RL for multiple-agents MDPs is described by Peshkin *et al.* (2000), providing a precedent for factoring policy-gradient RL policies into "agents" for each action. This paper also builds on earlier work presented by Aberdeen (2006).

POMDP Formulation of Planning

A finite partially observable Markov decision process consists of: a (possibly infinite) set of states $s \in \mathcal{S}$; a finite set of actions (that correspond to our *command* concept) $\mathbf{c} \in \mathcal{C}$; probabilities $\mathbb{P}[s'|s, \mathbf{c}]$ of making state transition $s \to s'$ under command $\mathbf{c}$; a reward for each state $r(s) : \mathcal{S} \to \mathbb{R}$; and a finite set of observation basis vectors $\mathbf{o} \in \mathcal{O}$ used by action policies in lieu of complete state descriptions. Observation vectors are constructed from the current state by producing a binary element for each predicate value, i.e., 1 for true and 0 for false. A constant 1 bias element is also provided. *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 state form the set of *terminal* states, ending plan simulation.

Policies are stochastic, mapping the observation vector $\mathbf{o}$ to a probability distribution over 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 means 'Yes' begin action n, and a 0 entry means 'No' do not start action n. The probability of a command is $\mathbb{P}[\mathbf{c}|\mathbf{o};\boldsymbol{\theta}]$, where conditioning on $\boldsymbol{\theta}$ reflects the fact that the policy is tuned by a set of real valued parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. 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 planning maximises the long-term average reward

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

where the expectation $\mathbb{E}_{\boldsymbol{\theta}}$ is over the distribution of state trajectories $\{s_0, s_1, \dots\}$ 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 (failure states achieve no reward). We also provide intermediate rewards for progress toward the goal. This additional *shaping* reward provides 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. 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.

Planning State Space

The state includes absolute time, a queue of past and future events, the truth value of each predicate, and function values. In a particular state, only the *eligible* actions have satisfied all at-start preconditions for execution. A command is the decision to start a set of non-mutex eligible actions. While actions might be individually eligible, starting them concurrently may require too many resources, or cause a precondition of an eligible action to be invalidated by another, which is when we consider them mutex. We do not deal with any other type of conflict when determining mutexes for the purpose of deciding to start actions, particularly 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 the end of the action (the duration of the action 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).

To estimate policy gradients we need a plan execution simulator to generate a trajectory through the planning state space. It takes actions 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 process the next event. 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 (Mausam & Weld 2006). One fix makes every time step a decision point at the cost of introducing infeasibly many feasible plans. Finally, only the current parameters, initial and current state, and current observation are kept in memory.

Policy Gradients

The idea is to perform gradient ascent on the long-term reward by repeatedly computing gradients $\frac{\partial}{\partial \theta_i} R(\boldsymbol{\theta})$ and stepping the parameters in that direction. Because an exact computation of the gradient is very expensive (but possible) we use Monte-Carlo gradient estimates (Williams 1992; Sutton *et al.* 2000; Baxter, Bartlett, & Weaver 2001) generated from repeated simulated plan executions

$$\frac{\partial R(\boldsymbol{\theta})}{\partial \theta_i} = \lim_{\substack{T\to\infty \\ \beta \to 1}} \frac{1}{T} \sum_{t=1}^{T} \frac{\nabla_{\boldsymbol{\theta}}\,\mathbb{P}[\mathbf{c}_t|\mathbf{o}_t;\boldsymbol{\theta}_t]}{\mathbb{P}[\mathbf{c}_t|\mathbf{o}_t;\boldsymbol{\theta}_t]} \sum_{\tau=t+1}^{T} \beta^{\tau-t-1} r_\tau . \qquad (2)$$

However, (2) requires looking forward in time to observe rewards. In practice we reverse the summations, using an *eligibility trace* to store previous gradient terms:

$$\mathbf{e}_t = \beta \mathbf{e}_{t-1} + \frac{\nabla_{\boldsymbol{\theta}}\,\mathbb{P}(\mathbf{c}_t|\mathbf{o}_t;\boldsymbol{\theta}_t)}{\mathbb{P}(\mathbf{c}_t|\mathbf{o}_t;\boldsymbol{\theta}_t)} . \qquad (3)$$

Thus, the eligibility trace $\mathbf{e}_t$ contains the discounted sum of normalised policy gradients for recent commands. 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 $\mathbf{g}_t = r(s_t)\mathbf{e}_t$. The discount factor $\beta \in [0, 1)$ decays the effect of old commands on the eligibility trace, effectively giving exponentially more credit for rewards to recent actions. Additionally, β may be 1.0 for finite-horizon problems such as planning (Williams 1992). Baxter, Bartlett, & Weaver (2001) gives two optimisation methods using the instantaneous gradients $\mathbf{g}_t$. OLPOMDP is the simple online gradient ascent just described, setting $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\mathbf{g}_t$ with scalar gain α. Alternatively, CONJPOMDP averages $\mathbf{g}_t = r_t\mathbf{e}_t$ over T steps to compute the batch gradient (2), followed by a line search for the best step size α in the search direction. OLPOMDP can be considerably faster than CONJPOMDP because it is tolerant of noisy gradients and adjusts the policy at every step. We use OLPOMDP for most of our experiments. However, the CONJPOMDP batch approach is used for parallelising FPG as follows. 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 (2). 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 gradient drops below some threshold.

Policy Gradient Optimisation for Planning

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

$$\begin{aligned} \mathbb{P}[\mathbf{c}_t|\mathbf{o}_t, \boldsymbol{\theta}] &= \mathbb{P}[a_{t1}, \ldots, a_{tN}|\mathbf{o}_t; \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N] \\ &= \mathbb{P}[a_{t1}|\mathbf{o}_t; \boldsymbol{\theta}_1] \times \cdots \times \mathbb{P}[a_{tN}|\mathbf{o}_t; \boldsymbol{\theta}_N]\,. \end{aligned} \tag{4}$$

The computation of the 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* planning algorithm. Similar approaches are adopted by Peshkin *et al.* (2000) and Tao, Baxter, & Weaver (2001). The main requirement for each action policy is that $\log \mathbb{P}[a_{tn}|\mathbf{o}_t, \boldsymbol{\theta}_n]$ be differentiable with respect to the parameters for each choice of action start $a_{tn} =$'Yes' or 'No'. We describe two such parameterised classes of action policy.

Linear Approximator Policies One representation of action policies is a linear network mapped to probabilities using a logistic regression function

$$\begin{aligned} \mathbb{P}[a_{tn} = Yes|\mathbf{o}_t, \boldsymbol{\theta}_n] &= \frac{1}{\exp(\mathbf{o}_t^\top \boldsymbol{\theta}_n) + 1} \\ \mathbb{P}[a_{tn} = No|\mathbf{o}_t, \boldsymbol{\theta}_n] &= 1 - \mathbb{P}[a_{tn} = Yes|\mathbf{o}_t, \boldsymbol{\theta}_n]\,. \end{aligned} \tag{5}$$

Recall that the observation vector is a vector representing the current predicate truth values plus a constant bias. If the dimension of the observation vector is $|\mathbf{o}|$ then each set of

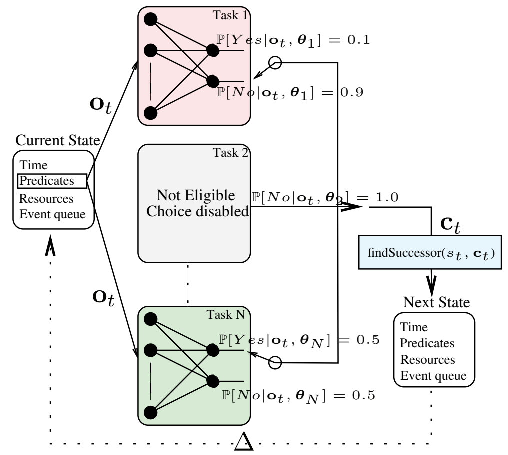

Figure 2: Action policies make independent decisions.

parameters $\boldsymbol{\theta}_n$ can be thought of as an $|\mathbf{o}|$ vector that represents the approximator weights for action n. The required normalised gradients over each parameter $\theta \in \boldsymbol{\theta}_n$ is

$$\begin{aligned} \frac{\nabla_{\boldsymbol{\theta}_n} \mathbb{P}[a_{tn} = Yes|\mathbf{o}_t, \boldsymbol{\theta}_n]}{\mathbb{P}[a_{tn} = Yes|\mathbf{o}_t, \boldsymbol{\theta}_n]} &= \\ &-\mathbf{o}_t \exp(\mathbf{o}_t^\top \boldsymbol{\theta}_n)\, \mathbb{P}[a_{tn} = Yes|\mathbf{o}_t, \boldsymbol{\theta}_n] \\ \frac{\nabla_{\boldsymbol{\theta}_n} \mathbb{P}[a_{tn} = No|\mathbf{o}_t, \boldsymbol{\theta}_n]}{\mathbb{P}[a_{tn} = No|\mathbf{o}_t, \boldsymbol{\theta}_n]} &= \mathbf{o}_t\, \mathbb{P}[a_{tn} = Yes|\mathbf{o}_t, \boldsymbol{\theta}_n]\,. \end{aligned} \tag{6}$$

These normalised policy derivatives are added to the eligibility trace (3) 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 (4). Fig. 2 illustrates this scheme. Initially, the parameters are set to 0 giving a uniformly random policy, encouraging exploration of the action space. Over time, the parameters typically, but not necessarily, move closer to a deterministic policy.

Decision Tree Policies 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 naive 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 policies. We assume a user declares an initial tree of 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 these rules. In the factored setting, each action has its own decision tree. We assume 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 human-coded branch selection rule, or sample a stochastic branch rule from the current stochastic policy for that node. Assuming the conditional independence of

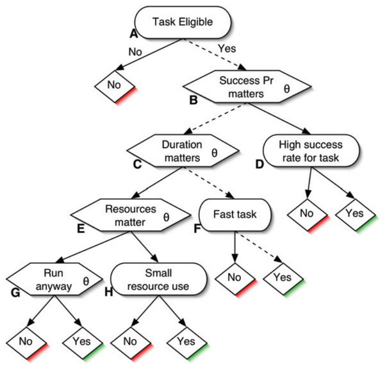

Figure 3: Decision tree action policy.

decisions at each node, the probability of reaching an action leaf l equals the product of branch probabilities at each node

$$\mathbb{P}[a = l|\mathbf{o}, \boldsymbol{\theta}] = \prod_{d \in \mathcal{D}} \mathbb{P}[d'|\mathbf{o}, \boldsymbol{\theta}_d], \tag{7}$$

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 $\mathbb{P}[d'|\mathbf{o}, \boldsymbol{\theta}_d]$ 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 for that action and in that region of state space.

For example, nodes A, D, F, H (Fig. 3) represent hard-coded rules that switch with probability one 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 stochastically. For this paper the probability of choosing the left or right branches is a single parameter logistic function that is independent of the observations. E.g, for action n, and decision node C "action duration matters?", we have

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

In general the policy pruning could be a function of the current state. In Fig. 3 the high level strategy switched by the parameter is written in the node label. For example for action policy n, and decision node C "action duration matters?", we have

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

The log derivatives of the 'Yes' and 'No' decisions are given by (6), noting that in this case $\mathbf{o}$ is a scalar 1. The normalised action probability gradient for each node is added to the eligibility trace independently.

If the parameters converge in such a way that prunes Fig. 3 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. This approach allows extensive use of control knowledge, using FPG to fill in the gaps.

The FPG Algorithm

Alg. 1 completes our description of FPG by showing how to implement (3) for planning with independent action policies. The algorithm works by repeatedly simulating plan executions: 1) the initial state represents time 0 in the plan (not 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 plan command; 5) the plan state transition is sampled (see the Planning State Space section); 6) the planner receives the global reward for the new state and calculates $\mathbf{g}_t = r_t \mathbf{e}_t$; 7) for OLPOMDP all parameters are immediately updated by $\alpha \mathbf{g}_t$, or for parallelised planning $\mathbf{g}_t$ is batched over T steps.

Algorithm 1 OLPOMDP FPG Gradient Estimator

1: Set s_0 to initial state, $t = 0, \mathbf{e}_t = [0]$, init $\boldsymbol{\theta}_0$ randomly
2: **while** R not converged **do**
3: $\quad \mathbf{e}_{t+1} = \beta \mathbf{e}_t$
4: $\quad$ Generate observation $\mathbf{o}_t$ of s_t
5: $\quad$ **for** each eligible action a_n **do**
6: $\quad\quad$ Evaluate action policy n $\mathbb{P}[a_{tn} = \{\text{Yes}, \text{No}\}|\mathbf{o}, \boldsymbol{\theta}_{tn}]$
7: $\quad\quad$ Sample a_{tn} =Yes or a_{tn} =No
8: $\quad\quad \mathbf{e}_{t+1} = \mathbf{e}_{t+1} + \frac{\nabla_{\boldsymbol{\theta}} \mathbb{P}[a_{tn}|\mathbf{o}, \boldsymbol{\theta}_{tn}]}{\mathbb{P}[a_{tn}|\mathbf{o}, \boldsymbol{\theta}_{tn}]}$
9: $\quad$ **end for**
10: $\quad$ **while** (s_{t+1} = findSuccessor($s_t, \mathbf{c}_t$)) == MUTEX **do**
11: $\quad\quad$ arbitrarily disable action in $\mathbf{c}_t$ due to mutex
12: $\quad$ **end while**
13: $\quad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha r_t \mathbf{e}_{t+1}$
14: $\quad$ **if** s_{t+1}.isTerminalState **then** $s_{t+1} = s_0$
15: $\quad t \leftarrow t + 1$
16: **end while**

Note the link to the planning simulator on line 10. 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 ordering) until the command is eligible. Line 8 computes the normalised gradient of the sampled action probability and adds the gradient for the n'th action's parameters into the eligibility trace (3). 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 $\mathbf{e}_t$. The gradient for parameters not relating to action n is 0. We do not compute $\mathbb{P}[a_{tn}|\mathbf{o}_t, \boldsymbol{\theta}_n]$ or gradients for actions with unsatisfied preconditions. If no actions are chosen to begin, we issue a no-op action and increment time to the next decision point.

Experiments

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

Non-Temporal Probabilistic Domains

Due to the lack of CPTP benchmark problems, we ran FPG on a range of non-temporal domains, including competing in the probabilistic track of the 2006 IPC. To do this, we removed the temporal features of FPG by: 1) changing the lo-

Table 1: Summary of non-temporal domain results. Values are % of plan simulations that reach the goal (minimum 30 runs). Blank results indicate the planner was not run on that domain. 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 in practice it failed to reach.

Domain	Para.	sfDP	FPG	FOALP	FF-r.	Prot.
Zeno	–	7	27	7	100	
Elevator	100	–	76	–	93	
Schedule	–	–	54	1	51	
Tire	82	–	75	91	82	
Random	–	–	65	5	100	
BW	100	29	63	–	86	
Ex. BW	24	31	43	31	52	
Drive	–	–	63	9	71	
Pitchcatch	–	–	23	–	54	
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^*	–

gistic regression for each eligible action (5) to be a *soft-max* probability distribution over *which single action* to choose; 2) removing the event queue and simply processing the current action to completion. We used the linear approximation scheme rather than the decision tree. 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. Table 1 shows the overall summary of results, by domain and planner. The IPC Results were based on 9 PPDDL specified domains, averaged over 15 instances from each domain, and tested on 30 simulations of plans for each instance. Many of these domains, such as Blocksworld (BW), are classical deterministic domains with noise added to the effects. We defer to Bonet & Givan (2006) for details.

The second set of domains (one instance each) demonstrates results on domains introduced by Little & Thiébaux () that are more challenging for replanners. The optimal Paragraph planner does very well until the problem size becomes too large. Triangle-tire-4 in particular shows a threshold for problems where an approximate probabilistic planning approach is required in order to find a good policy. To summarise, in non-temporal domains FPG appears to be a good compromise between the scalability of replanning approaches, and the capacity of optimal probabilistic planner to perform reasoning about uncertainty.

Temporal Probabilistic Domains

These experiments compare FPG to two earlier probabilistic temporal planners: Prottle (Little, Aberdeen, & Thiébaux 2005), and a Military Operations (MO) planner (Aberdeen, Thiébaux, & Zhang 2004). 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 modified 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). We note, however, that 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.

The first three domains are *Probabilistic Machine Shop (MS)* (Mausam & Weld 2005), *Maze (MZ)*, and *Teleport (TP)* (Little, Aberdeen, & Thiébaux 2005). In all cases we use the versions defined in Little, Aberdeen, & Thiébaux (2005), and defer descriptions to that paper. Additionally we introduce two new domains.

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 has longer times, but injects 30 laps worth. 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 task, 250 predicate domain as follows: the goal state required 18 predicates to be made true. Each task has two outcomes, with up to 6 effects and a 10% chance of each effect being negative. Two independent sequences of tasks are generated that potentially lead to the goal state with makespan of less than 1000. There are 40 types of resource, with 200 units each. Each task requires a maximum of 10 units from 5 types, potentially consuming half of the occupied resources permanently. Resources limit how many tasks 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. 3; (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 run *all* eligible actions. All experiments we performed were limited to 600 seconds. Other parameters are described in Table 4. In particular, the single gradient step size α was selected as the highest value that ensured reliable convergence over 100 runs over all domains. Experiments in this section were conducted on a dedicated 2.4GHz Pentium IV processor with 1GB of ram. The results are summarised in Table 2. Reported Fail% and makespan was estimated from

Table 2: Results on 3 benchmark domains. The experiments for MO and FPG were repeated 100 times. The *Opt.* column is the optimisation engine used. *Fail%*=percent of failed executions, *MS*=makespan, *R* is the final long-term average reward, and *Time* is the optimisation time in seconds.

Prob.	*Opt.*	*Fail%*	*MS*	*R*	*Time*
MS	$\overline{\text{FPG-L}}$	1.33	6.6	118	532
MS	FPG-L	0.02	5.5	166	600
MS	$\overline{\text{FPG-T}}$	70.0	13	20.9	439
MS	FPG-T	65.0	13	21.4	600
MS	Prottle	2.9			272
MS	MO		Out of memory		
MS	random	99.3	18	0.1	
MS	naïve	100	20	0.0	
MZ	$\overline{\text{FPG-L}}$	19.1	5.5	134	371
MZ	FPG-L	14.7	6.9	130	440
MZ	$\overline{\text{FPG-T}}$	19.7	5.5	136	29
MZ	FPG-T	15.3	5.7	115	17
MZ	Prottle	17.8			10
MZ	$\overline{\text{MO}}$	7.92	8.0		71
MZ	MO	7.15	8.2		72
MZ	random	76.5	13	16.4	
MZ	naïve	90.8	16	8.6	
TP	$\overline{\text{FPG-L}}$	34.4	18	298	340
TP	FPG-L	33.3	18	305	600
TP	$\overline{\text{FPG-T}}$	34.4	18	302	258
TP	FPG-T	33.3	18	301	181
TP	Prottle	79.8			442
TP	MO		Out of memory		
TP	random	99.6	15	1.0	
TP	naïve	100	19	0.0	
PitStop	$\overline{\text{FPG-L}}$	0.0	20180	142	41
PitStop	random	29.0	12649	41.0	
PitStop	naïve	100	66776	0.0	
500	FPG-T	2.5	158	1.56	3345
500	random	76.6	765	0.231	
500	naïve	69.5	736	0.100	

10,000 simulated executions of the optimised plan, except for Prottle. Prottle results were taken directly from Little, Aberdeen, & Thiébaux (2005), quoting the smallest probability of failure result. FPG and MO experiments were repeated 100 times due to the stochastic nature of the optimisation. FPG tepeat experiments are important to measure the effect 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 failure. 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 2 shows that FPG is at least competitive with Prottle and the MO planner, while winning in the hardest of the existing benchmarks: Machine Shop. The poor performance of Prottle in the Teleport domain — 79.8% failure compared to FPG's 34.4% — is due to Prottle's short maximum permitted makespan of 20 time units. At least 25 units are required to achieve a higher success

```
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;
```

Figure 4: Three decision-tree action policies extracted the final Maze policy. Returning 'Yes' means start the action.

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 many 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 failure rate to 70% compared to the 1% the linear policy achieves. We explored the types of policy that the decision tree structure in Fig. 3 produces. The pruned decision tree policy for three grounded Maze actions is shown in Fig. 4.

Table 2 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 estimations over 10,000 steps*. 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. 5. 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 arbitrarily stopped at 75% of the average reward obtained with the stopping criterion used for Table 2. The new results in Table 3 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 FPGs ability to optimise under mixtures of discrete and continuous uncertainties. We have yet to try scaling to larger domains with these characteristics. Results for the 500 domain are shown for running the parallel version of FPG algorithm with 16 processors. As expected, we observed that optimisation times dropped inversely proportional to the number of CPUs for up to 16 processors. However, on a single processor the parallel version requires double the time of OLPOMDP. No other CPTP planner we know of is capable of running domains of this scale.

Discussion and Conclusion

FPG diverges from traditional planning approaches in two key ways: we search for plans directly, using a local optimisation procedure; and we approximate the policy representation by factoring into a policy for each action. Apart from reducing the representation complexity, this approach

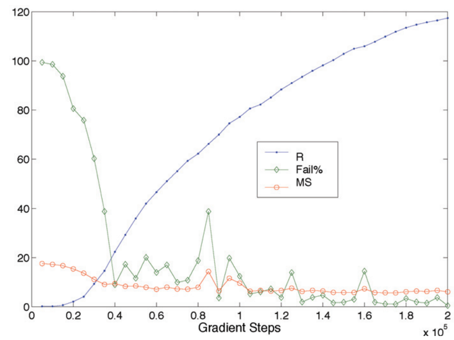

Figure 5: Relative convergence of long-term average reward R, failure probability, and makespan over a single linear network FPG optimisation of Machine Shop. The y-axis has a common scale for all three units.

Table 3: FPG's results when optimisation is terminated at 75% of the mean R achieved in Table 2.

Prob.	*Opt.*	*Fail%*	*MS*	*R*	*Time*
MS	$\overline{\text{FPG-L}}$	4.68	7.1	89	37
MS	FPG-L	0.11	6.5	89	32
MS	$\overline{\text{FPG-T}}$	70.4	13	16	10
MS	FPG-T	65.9	14	16	14
MZ	$\overline{\text{FPG-L}}$	19.3	5.5	100	13
MZ	FPG-L	14.7	5.6	100	22
MZ	$\overline{\text{FPG-T}}$	19.7	5.5	102	2
MZ	FPG-T	15.8	7.0	102	2
TP	$\overline{\text{FPG-L}}$	34.6	18	224	3.5
TP	FPG-L	33.0	19	224	4
TP	$\overline{\text{FPG-T}}$	34.7	18	226	2.3
TP	FPG-T	33.1	18	226	1

allows policies to generalise to states not encountered during training; an important feature of FPG's "learning" approach. We conducted many experiments that space precludes us from reporting. These include: multi-layer perceptrons (improvements in pathologically constructed XOR-based domains only); observations that encode more than just predicate values (insignificant improvements); and using the FF heuristic to bias the initial action sampling of FPG (which can sometimes assist FPG to initially find the goal) (Buffet & Aberdeen 2007). To conclude, FPG's contribution is a demonstration that Monte-Carlo machine learning methods can contribute to the state of the art in planning, particularly to deal with large or infinite state spaces.

Table 4: 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

Acknowledgments

Thank you to Iain Little for his assistance with domains and comparisons. National ICT Australia is funded by the Australian Government's Backing Australia's Ability program.

References

Aberdeen, D.; Thiébaux, S.; and Zhang, L. 2004. Decision-theoretic military operations planning. In *Proc. ICAPS*.

Aberdeen, D. 2006. Policy-gradient methods for planning. In *Proc. NIPS'05*.

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72.

Baxter, J.; Bartlett, P.; and Weaver, L. 2001. Experiments with infinite-horizon, policy-gradient estimation. *JAIR* 15.

Bonet, B., and Givan, R. 2006. Proc. of the 5th int. planning competition (IPC-5). *See* `http://www.ldc.usb.ve/~bonet/ipc5` *for all results and proceedings*.

Buffet, O., and Aberdeen, D. 2007. FF+FPG: Guiding a policy-gradient planner. In *Proc. ICAPS*.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Little, I., and Thiébaux, S. Probabilistic planning vs replanning. Submitted for Publication.

Little, I., and Thiébaux, S. 2006. Concurrent probabilistic planning in the graphplan framework. In *Proc. ICAPS*.

Little, I.; Aberdeen, D.; and Thiébaux, S. 2005. Prottle: A probabilistic temporal planner. In *Proc. AAAI*.

Mausam, and Weld, D. S. 2005. Concurrent probabilistic temporal planning. In *Proc. ICAPS*.

Mausam, and Weld, D. S. 2006. Probabilistic temporal planning with uncertains durations. In *Proc. AAAI'06*.

Peshkin, L.; Kim, K.-E.; Meuleau, N.; and Kaelbling, L. P. 2000. Learning to cooperate via policy search. In *Proc. UAI*.

Sanner, S., and Boutilier, C. 2006. Practical linear value-approximation techniques for first-order MDPs. In *Proc. UAI*.

Sutton, R. S.; McAllester, D.; Singh, S.; and Mansour, Y. 2000. Policy gradient methods for reinforcement learning with function approximation. *Proc. NIPS*.

Tao, N.; Baxter, J.; and Weaver, L. 2001. A multi-agent, policy-gradient approach to network routing. In *Proc. ICML*.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8:229–256.

Yoon, S.; Fern, A.; and Givan, R. 2007. FF-replan, a baseline for probabilistic planning. In *Proc. ICAPS'07*.

Younes, H. L. S., and Littman, M. L. 2004. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical Report CMU-CS-04-167.

Younes, H. L. S., and Simmons, R. G. 2004. Policy generation for continuous-time stochastic domains with concurrency. In *Proc. ICAPS*.

Younes, H. L. S. 2003. Extending PDDL to model stochastic decision processes. In *Proc. ICAPS Workshop on PDDL*.