Policy-Gradient for Robust Planning

Olivier Buffet and **Douglas Aberdeen**[1]

Abstract. Real-world Decision-Theoretic Planning (DTP) is a very challenging research field. A common approach is to model such problems as Markov Decision Problems (MDP) and use dynamic programming techniques. Yet, two major difficulties arise: 1- dynamic programming does not scale with the number of tasks, and 2- the probabilistic model may be uncertain, leading to the choice of unsafe policies. We build here on *Policy Gradient* algorithms to address the first difficulty and on robust decision-making to address the second one through algorithms that train competing learning agents. The first agent learns the plan while the second learns the model most likely to upset the plan. It is known from gradient-based game theory that at least one player may not converge, so we focus on convergence of the robust plan only, using non-symmetric algorithms.

1 INTRODUCTION

Robust Decision-Theoretic Planning (DTP) looks at problems where the stochastic model is uncertain, and searches for the plan achieving the best performance whatever the true model. Since DTP problems are often dealt with as Markov Decision Problems (MDPs), recent robust algorithms for uncertain MDPs [4, 14, 8] seem to be promising approaches based on dynamic programming. Yet, they all rely on a specific assumption (Assumption-1 described in Section 2.3) that does not hold in many planning settings. Thus, a different approach to robust planning is required. We propose to avoid the MDP formulation of a DTP problem, and to make use of policy-gradient algorithms.

Without Assumption-1, the problem is a generic two-player zero-sum Markov game with partial observability (as explained in Section 2.3), where one player has to choose a plan and the other has to choose a model. Our first idea was to simultaneously learn both plan and model. Yet, there is no algorithm that guarantees convergence towards an equilibrium of the game in our partially observable setting. Here are some important related results:

- In [15], Singh et al. study the infinitesimal gradient ascent applied to two-player two-action games. For zero-sum games, it leads to an oscillatory behavior around the Nash equilibrium.
- In [7], Bowling and Veloso present the "Win or Learn Fast" principle, making it possible to get convergence in these games. But results do not extend to more than two actions, as in the game rock-paper-scissors (Roshambo).[3]
- In [10], Chang and Kaelbling explain how a gradient-ascent learner can be "tricked" by its opponent, using sudden changes its policy.
- In [17], Zinkevich proposes to use no-regret algorithms to avoid being tricked in such a way.

Unfortunately, ensuring no-regret does not necessarily lead to convergence of the strategies in self-play. In Roshambo, it is common to see both players cycling together through Rock-vs-Rock $\rightarrow$ Paper-vs-Paper $\rightarrow$ Scissors-vs-Scissors. The value is always the value of the Nash equilibrium (0), and there is no regret since each player is not doing worse than the random strategy.

In this paper, we propose two algorithms to find the best plan against the worst model, but not trying to find the model at the same time. After introducing the problem in some detail in Section 2, Section 3 presents both algorithms. Then, experiments illustrate their respective behaviors before a discussion and conclusion.[2]

2 BACKGROUND

2.1 Partially Observable Markov Decision Problems

A Partially Observable Markov Decision Problem (POMDP) [9] is defined here as a tuple $\langle S, A, T, r, \Omega, O\rangle$. It describes a control problem where S is the finite set of **states** of the system considered and A is the finite set of possible **actions** a. Actions control transitions from one state s to another state s' according to the system's probabilistic dynamics, described by the **transition function** T defined as $T(s,a,s') = Pr(s_{t+1} = s'|s_t = s, a_t = a)$. The aim is to optimise a performance measure based on the **reward function** $r : S \times A \times S \rightarrow \mathbb{R}$.[3] Only a partial observation o ($\in \Omega$: finite set of possible observations) of the state is accessible, sampled according to the probability distribution $O(o,s) = Pr(o|s)$.
Note: in this paper, we always take $O(o,s) = 0$ or 1, but this assumption could be removed with no consequences.

We prefer not using short-term memory when making a decision. Thus, the optimisation algorithm has to find a **policy** mapping observations to probability *distributions* over actions $\pi : \Omega \rightarrow \Pi(A)$ so as to optimise the chosen performance measure, here the average *reward* R gained during a transition.

Policy-Search — In this work, we prefer using policy-gradient algorithms rather than classical dynamic programming. Policies are stochastic and depend on a vector of parameters $\vec{\theta} \in \Re^n$. The probability of choosing action a given state s is written as $Pr(a|\mathbf{o};\vec{\theta})$. For a given $\vec{\theta}$, the probability of transition $s \rightarrow s'$ is therefore:

$$Pr(s'|s;\vec{\theta}) = \sum_{a\in A} Pr(a|\mathbf{o};\vec{\theta})T(s,a,s').$$

---

[1] National ICT Australia & The Australian National University, email: firstname.lastname@nicta.com.au

[2] More details appear in an extended version [3].

[3] As the model is not sufficiently known, we do not make the usual assumption $r(s,a) = \mathbb{E}_{s'}[r(s,a,s')]$. The expectation depends on the (unknown) true model.

If the resulting Markov chain is ergodic, it will evolve to a unique stationary distribution over states P_S. The average reward per time step is then given by:

$$R(\vec{\theta}) \quad = \quad \sum_{s\in S} P_S(s;\vec{\theta}) \sum_{a\in A} Pr(a|\mathbf{o};\vec{\theta}) \sum_{s'\in S} r(s,a,s')T(s,a,s').$$

We mainly use simulation-based policy-gradient algorithms as described by Baxter et al. [5, 6]. As gradient-ascent algorithms, they compute an estimate of the gradient of the average reward $\nabla_{\vec{\theta}} R$ and follow its direction to update the vector $\vec{\theta}$. To that end, at each step of the simulation they update an eligibility trace $\mathbf{e}$ estimating the gradient of the log action probability $\log Pr(a|\mathbf{o};\vec{\theta})$.

Here, we mainly consider two algorithms:

- GPOMDP, which estimates the gradient using:

$$\nabla R_{t+1} = \nabla R_t + \frac{1}{t+1}[r_t \mathbf{e}_{t+1} - \nabla R_t], \tag{1}$$

  and has to be alternated with a gradient-following step to modify the parameters; and
- OL-POMDP, which is continuously re-estimating an eligibility trace while simultaneously reinforcing the parameters with:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t r_t \mathbf{e}_{t+1}, \tag{2}$$

  where α_t is a learning rate.

2.2 Decision-Theoretic Planning

Our planning domains are described by: a set of state variables $\mathcal{V} = \{v_1 \in V_1, \ldots, v_{|\mathcal{V}|} \in V_{|\mathcal{V}|}\}$, and a set of tasks $\mathcal{T} = \{T_1, \ldots, T_{|\mathcal{T}|}\}$. The current state of the system is described by the values of the variables, by current time step and by currently active tasks.

In a given state, a task T can be triggered (is eligible) if certain conditions hold regarding the state variables (if some propositions are true or some resources are sufficient). After the task's duration, one of several possible outcomes $out_T(1), out_T(2)\ldots$ arise, depending on a probability distribution $Pr(Out_T)$. This outcome changes the value of some state variables.

Tasks can run simultaneously and can be repeated if required, and a reward is associated to each of a task's outcomes.

A planning problem is specified by an initial state s_0, the reward function and a set of goal states. These goal states can be specified by conditions over state variables or with a maximal plan length (so that a goal state is necessarily reached). The aim is then to reach a goal state while maximising the average reward, which sets us in the framework of Decision-Theoretic Planning. Similar problems have been addressed with planners as Tempastic [16], a military operations' planner [2], CPTP [13], Prottle [12], and FPG [1].

Note: for readability reasons, the above description is a simplified version of the model used.

DTP with MDPs — A simple way of turning such a decision-theoretic planning problem into an MDP is, as in [11], to define:

- States as instants when some tasks end, resulting in a new situation where a decision can be made. A state is then again defined by: current state variables and currently running tasks.
- Actions as the decision of triggering a subset of the eligible tasks. Triggering no task is valid as it amounts to waiting for the next decision point.
- Transition probabilities depend on each task's probability distribution over its possible outcomes.
- Rewards depend on rewards/costs associated with transitions and goal states.

Figure 1 shows a running system represented as a planning problem and as an equivalent MDP where s_1 corresponds to the state at time t, action a_2 triggers tasks T_2 and T_3 and leads to s_5 when outcomes $out_{T_1}(2)$ and $out_{T_3}(1)$ occur at $t + 18$.

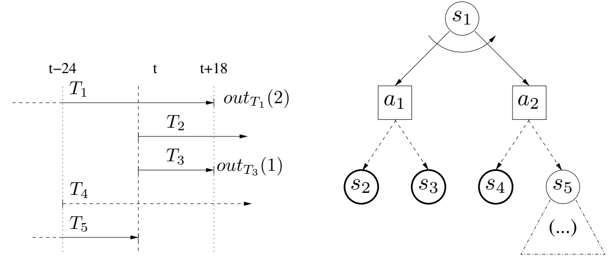

Figure 1. A running planning problem and a simple MDP

2.3 Robustness

Because they are often obtained using expert knowledge or statistics, tasks' models are prone to uncertainty. This section explains how we model uncertainty and how it is taken into account in this planning problem.

Modeling Uncertainty — Our model of uncertainty uses only knowledge of whether a model is possible or not, without considering how probable a model is. For our initial planning problem, we choose to model uncertain probability distributions using interval-based uncertain probabilities:

$$Pr(out_T(k)) \in [Pr^{\min}(out_T(k)), Pr^{\max}(out_T(k))]$$

knowing that they must sum to one: $\sum_k Pr(out_T(k)) = 1$. This choice is originally motivated by the simplicity of this representation, and the fact that expert knowledge or statistics easily lead to such intervals. For a given state-action pair, the upper and lower bounds on the probability of each outcome give constraints on possible models.

Fig. 2 illustrates these constraints for one task with three possible outcomes. On this figure, the triangle is a probability simplex representing all possible probability distributions with three different outcomes ($Pr(out_i) = 1$ at the out_i vertex). The horizontal trapezium gives the interval constraint for out_1. Possible models are defined by the intersection of the three interval constraints.

An interesting characteristic of the resulting set of possible distributions is its convexity (because of the conjunction of constraints), which is helpful when searching for optimal solutions. The opponent responsible for this task's model has to find a worst point in this set.

Similarly, uncertain (PO)MDPs are often modelled by specifying intervals for transition probabilities:

$$T(s,a,s') \in [Pr^{\min}(s'|s,a), Pr^{\max}(s'|s,a)].$$

Note that there is no direct equivalence between uncertain MDP and DTP models. Indeed, since a given task from the planning problem can be triggered in various situations, changing the probability

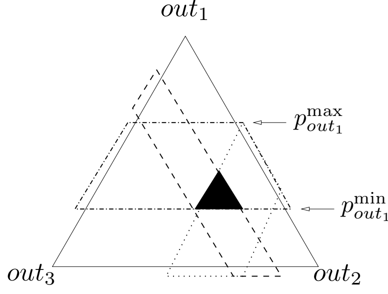

Figure 2. The black triangle represents all possible probability distributions for some task's outcomes.

distribution over its outcomes $Pr(Out_k)$ will effect several probability distributions $T(s, a, \cdot)$. For an uncertain MDP to properly represent an uncertain DTP problem, one needs to add constraints linking $T(s, a, \cdot)$ distributions.

Problem — Finding a *robust* plan means finding a plan that achieves the best possible performance whatever the true model. So, to get a robust plan, we need to find the best plan against the worst possible models. Denoting the set of possible models (either DTP or MDP) as M, this requires solving:

$$\arg\max_{\pi \in \Pi} \min_{m \in M} R(\pi, m). \tag{3}$$

Uncertain MDPs make it possible to efficiently use dynamic programming to solve this equation as long as the following assumption holds:

Assumption-1: *Probability distributions $T(s, a, \cdot)$ are independent from one state-action pair to another.*

Yet, we have just seen that a good translation of an uncertain planning problem as an uncertain MDP requires adding constraints which precisely break this assumption. This hinders the interest for using MDP models, and suggests looking for other optimisation techniques with the specific difficulty that we are dealing with a game-theoretic problem.

A good way to see that dynamic programming does not work is to try it on the problem presented in Section 4.3.

Game Being Played — This game is a two-player zero-sum game, with one player (the "planner") looking for a policy maximising the average reward and the second player (the "opponent") looking for a model minising the same average reward. The opponent has the following constraints:

- If the outcomes of two different tasks are decided at the same time, they should be sampled independently. So, we can factor this single opponent in a team of opponents, one for each task.
- The probability distribution over a task's outcomes should not depend on current state. Each of these opponents should be blind (no observation), which justifies the partial observability in our problem.

3 ALGORITHMS

As explained in Sec. 1 and 2, both the game-theoretic setting and partial observability suggest using policy-search algorithms rather than dynamic programming. This is also the direction taken by FPG [1] to design a scalable planning algorithm, but with the difference that factorisation happens on the side of the opponent in our approach, not on the planner side.

Here, experience is gathered by simulating the domain. To that end, we have a generic simulation loop for use by our robust policy gradient algorithms. As shown in Alg. 1, it takes as parameters the functions that the planner agent and its opponents should use for learning: OL-POMDP's `reinforcement` (Eq. (2)), GPOMDP's `gradientUpdate` (Eq. (1)), or `nothing`.

Algorithm 1 SimulationLoop($algoPlan$, $algoOpp$)

1: s =initial state
2: **while** s.goal $\neq$ true **do**
3: o = getObservation(s)
4: a = plan.getAction(o)
5: s' = findSuccessor(s, a)
6: r = getReward(s, a, s')
7: plan.$algoPlan(r)$
8: **for all** $a' \in A$ **do**
9: opp(a').$algoOpp(r)$
10: $s \leftarrow s'$

The domain simulation mainly occurs in function `findSuccessor()` (see Algorithm 2), based on the same code as FPG. Given a state s and action a, it samples the next state using the current model of the domain. To make the problem a bit simpler, only one action can be triggered at a time (but there may be several actions running concurrently). In this algorithm, the step where an event's outcome is sampled is where the corresponding "opponent" is making a decision (and modifies its model).

Algorithm 2 findSuccessor(s:state, a:action)

1: s.addEvent(a, `sample-outcome`, s.time+a.duration)
2: **for all** $f \in instantEffects(a)$ **do**
3: s.processEffect(f)
4: **for all** $f \in delayedEffects(a)$ **do**
5: s.addEvent(f, `task-effect`, s.time+f.delay)
6: **repeat**
7: **if** s.time $>$ maximum makespan **then**
8: s.failure=true
9: return
10: **if** s.operationGoalsMet() **then**
11: s.goal=true
12: return
13: **if** $\neg s$.anyEligibleTasks() & s.noEvent() **then**
14: s.failure=true
15: return
16: $event$ = s.nextEvent()
17: s.time = $event$.time
18: **if** type($event$) = `task-effect` **then**
19: s.processEffect($event$.effect)
20: **else if** type($event$) = `sample-outcome` **then**
21: out=sampleOutcome($event$)
22: **for all** $f \in instantEffects(out)$ **do**
23: s.processEffect(f)
24: **for all** $f \in delayedEffects(out)$ **do**
25: s.addEvent(f, `task-effect`, s.time+f.delay)
26: **until** s.isDecisionPoint()

We now see how the simulation loop can be used by two robust policy-gradient algorithms. The key idea is that the opponent's unstability will make the planner's policy stable.

3.1 Alternate learners

The first algorithm, *Sequential Robust Policy-Gradient* (seqRPG), is a direct translation of Equation 3 into an algorithm based on policy gradient algorithms. Its outter loop is performing a gradient ascent to optimise the policy π (i.e. to maximise $\min_{m \in M} R(\pi, m)$) through its vector of parameters $\vec{\theta}$. At each iteration of this loop:

- line 2-3: a first inner loop finds the worst model for current policy (that's where $\min_{m \in M} R(\pi, m)$ is computed),
- line 4-6: a second inner loop estimates the gradient of the expected reward with respect to $\vec{\theta}$, and
- line 7: a single gradient-following step is accomplished to update the policy's parameters.

Algorithm 3 Sequential Robust Policy-Gradient

```
1: while θ not converged do
2:    while model not converged do
3:       SimulationLoop( nothing(), reinforcement())
4:    ∇_θ R = 0
5:    while ∇_θ R not converged do
6:       SimulationLoop( gradientUpdate(), nothing())
7:    plan.followGradient()
```

It could be useful not to re-initialise the gradient's estimate in early stages of the algorithm, since it will probably evolve rather smoothly. Yet, after some time, vector $\vec{\theta}$ will tend to oscillate around an equilibrium, the gradient direction becoming very unstable.

An apparent drawback of this algorithm is that it relies on re-optimising the model at each iteration, which suggests a huge increase in computation time.

3.2 Simultaneous learners

The second algorithm, *Simultaneous RPG* (simRPG), is an attempt to reduce time complexity. As presented in Alg. 4, it consists of all agents simultaneously learning on-line. It mimicks Alg. 3 in that the opponents try to always maintain the worst model for the current policy. This is done by having the opponents learn much faster than the planner.

Algorithm 4 Simultaneous Robust Policy-Gradient

```
1: while θ_pla and θ_opp not converged do
2:    SimulationLoop( reinforcement(), reinforcement())
```

The opponents learning faster than the planner ensures that the planner is stabilised around an equilibrium point. As soon as the planner moves away from such a position, the opponents quickly adapt the model, inciting the planner to move back. With a fixed ratio $\alpha_{pla}(t)/\alpha_{opp}(t)$, this leads to periodic oscillations (see IGA with two-action games in [15] or with Roshambo in [3]), even with decreasing learning rates.

A decreasing ratio $\alpha_{pla}(t)/\alpha_{opp}(t)$ is required to progressively attenuate oscillations on the planner's side. So, a necessary condition to get the plan to converge is that the planner's learning rate α_{pla} should decrease faster than the opponents' learning rate α_{opp}: $\alpha_{pla}(t)/\alpha_{opp}(t) \rightarrow 0$. An important problem is to check whether this condition is also sufficient.

A solution to have these two learning rates verifying the usual hypothesis $\sum_{t=0}^{\infty} \alpha(t) > \infty$ and $\sum_{t=0}^{\infty} \alpha(t)^2 < \infty$ is to define them as $\alpha_{pla}(t) = t^{-\beta}$ and $\alpha_{opp}(t) = t^{-\beta'}$ with $0 < \beta' < \beta < 1$.

4 EXPERIMENTS

These experiments, conducted on a 2.8 GHz Pentium IV, are mainly intended to illustrate the behaviors of seqRPG and simRPG, as well as some prototypical robust planning problems. The learner's policy is a linear network mapped to probabilities using a softmax, parameters being its weights [1]. The opponents' policies are look-up tables with one parameter per probability.[4] As FPG, the algorithm does not enumerate the state-space, which leads to a low memory usage.

In seqRPG, the inner loops' stopping criterion is a fixed deadline of 1000 simulation loops (to learn the model *and* to estimate the gradient). In each experiment, measures are gathered every T iterations of the outer loop (of seqRPG or simRPG), and the algorithm runs for a fixed time t (given in seconds). Finally, the learning rates α_{pla} and α_{opp} are constant. These four values are given below each plot.

4.1 Matching pennies & queueing problem

Matching pennies — We start with a first matrix game translated into a planning problem. This is a non-symmetric version of matching pennies using the following payoff matrix for one player (and the opposite matrix for the other player):[5]

$$\begin{bmatrix} -1 & +\frac{1}{2} \\ +1 & -\frac{1}{2} \end{bmatrix},$$

where the planner is choosing the column. The game is chosen to be non-symmetric in order to check that we get the non-symmetric optimal strategy: play 2/3 head and 1/3 tail (1st and 2nd columns).

Figures 3 and 4 show the evolution of: the planner's probability of playing head or tail (`planner1` and `planner2`), the model's probability of playing head (`opp`) and the average reward (`R`).

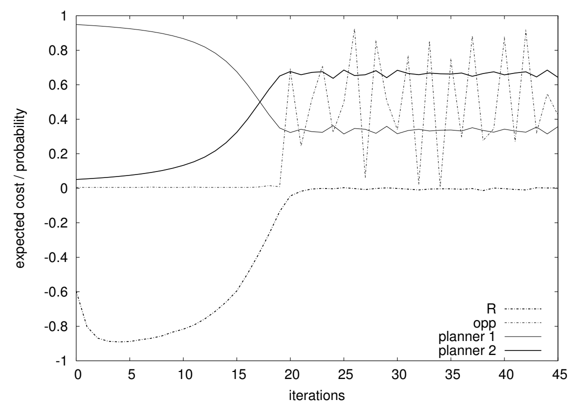

Figure 3. A run of seqRPG on matching pennies
$\alpha_{pla} = .1 \quad \alpha_{opp} = .1 \quad T = 10 \quad t = 60s$

Both cases show a first phase where the planner slowly moves toward its equilibrium position (the strategy of the Nash equilibrium), and a second phase where it slowly oscillates, being kept stable at this position by the opponent's stronger oscillation. The duration of the first phase (about 30s for seqRPG and 5s for simRPG) illustrates the fact that the simultaneous algorithm usually converges much faster (confirmed in all other experiments).

[4] For practical reasons (handling interval constraints easily), the opponents use the GIGA algorithm [17] instead of OL-POMDP. GIGA is not used for the planner because it does not allow the use of a function approximation.
[5] All games considered are zero-sum games.

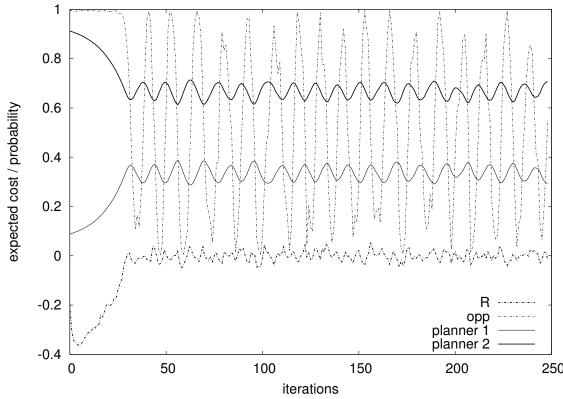

Figure 4. A run of simRPG on matching pennies
$\alpha_{pla} = .0001 \quad \alpha_{opp} = .1 \quad T = 1000 \quad t = 20s$

Note: if the opponent's strategy were stationary, there could be two situations:

- If this strategy is not that of the Nash equilibrium, it can be exploited and the planner would converge to a pure strategy (not the one of the Nash equilibrium).
- Otherwise, the planner has no reason to converge to some particular strategy: all give the same average gain.

Similar results apply to the following problems.

Queueing problem — This second problem leads to a situation rather similar to matching pennies, but shows a practical robust planning problem. It involves two queues of jobs: Q_1 and Q_2, each of size 5 and initially empty. At each time step, the planner has to send a job to one of these queues and a job is taken in one of the queues for processing (Q_1 with probability p), the chosen queue being eventually empty. The opponent controls p. Nobody is aware of the queues' contents.

The reward is $+.1$ for each job processed, and -1 for each job lost due to a queue being full. A goal state is reached when a job is lost or after 50 simulation steps, so that up to 10 jobs can be processed.

Both algorithms learn the appropriate behavior balancing the use of both queues. But, as can be observed on the plots of Fig. 5, the learning is noticeably slowed down (compared to matching pennies). This is due to the much longer duration of the plan, which makes it more difficult for both the planner and the opponent to reinforce decisions properly.

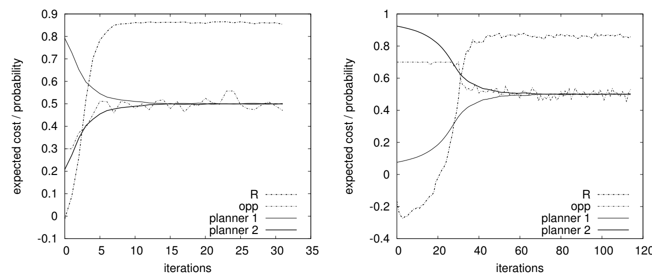

Figure 5. A run of seqRPG and simRPG on the queueing problem
$\alpha_{pla} = 1.|.0001 \quad \alpha_{opp} = .1|.1 \quad T = 10|1000 \quad t = 600|120s$

4.2 Roshambo

Here, we consider the game rock-paper-scissors with the usual payoff matrix:

$$\begin{bmatrix} 0 & +1 & -1 \\ -1 & 0 & +1 \\ +1 & -1 & 0 \end{bmatrix}.$$

Fig. 6 shows the trajectory of both players' policies with different learning rates for the opponents. The upper left triangle is the probability simplex for the opponent, and the lower right triangle is a mirroring probability simplex for the planner. The faster the opponents are, the more reactive they get to the planner's policy, which in turn gets more stable.

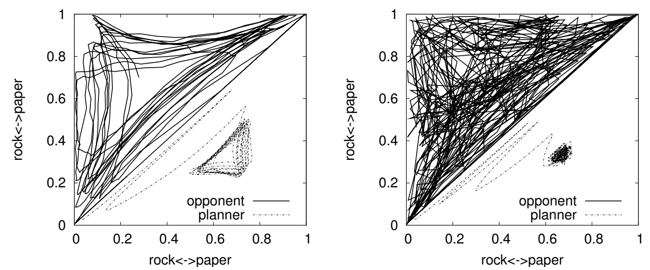

Figure 6. Two runs of simRPG. Only α_{opp} changes.
$\alpha_{pla} = .0001 \quad \alpha_{opp} = .01|.1 \quad T = 1000 \quad t = 90s$

4.3 Problem with 2 maxima

This last problem illustrates the fact that there may be several local optima for the opponent, even for a given policy for the planner. As shown through an MDP formulation on Fig. 7, the planner has indeed no choice of action, and the model has only one parameter p. Assuming that a_0, a_1 and a_2 have respective costs c_0, c_1 and 0, the optimal cost-to-go is (as a function of p):

$$V_{s_0}(p) \quad = \quad \frac{c_0}{p} + \frac{c_1}{(1-p)}.$$

To turn to optimising average reward, we give a fixed horizon (50 simulation steps), so that $R \simeq V_{s_0}(p)/50$. Thus, local maxima are obtained for $p = 0$ or $p = 1$. For practical reasons, we restrict $p \in [.1, .9]$.

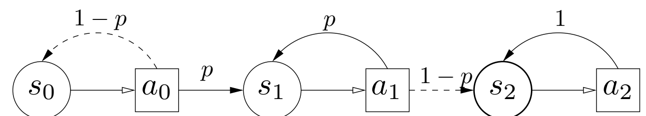

Figure 7. Problem with 2 deterministic optima

To check the behavior of simRPG, we gathered statistics on the number of times it converges to each side of the interval. We made three series of 500 runs lasting 20 seconds each, each ending with $p = .1 \pm .005$ or $p = .9 \pm .005$. A different cost function was used in each series. As the results show it (Table 1), the experiments properly reflect the theoretical probability of converging towards $p = .1$.

Table 1. Percentage of runs converging to $p = .1$
$(c_0 = -.1 \quad \alpha_{pla} = .0001 \quad \alpha_{opp} = .1 \quad T = 1000 \quad t = 20s)$

	$c_1 = c_0$	$c_1 = 2c_0$	$c_1 = 3c_0$
Theory	50.0%	41.4%	36.6%
Experience	47.2%	39.0%	36.0%

5 DISCUSSION AND CONCLUSION

Experimental results confirm that both the sequential and the simultaneous robust policy gradient make it possible to get the plan to converge to an equilibrium. And, as expected, the sequential algorithm is noticeably more time-consuming than the simultaneous one.

The experiments have been conducted on problems chosen to illustrate typical difficulties of robust planning, requiring stochastic policies or with several optima. The algorithm should scale well if we adopt FPG's factored formulation [1]: the memory usage would scale linearly with the number of tasks. But further experiments with large uncertain problems are required to confirm this point.

A difficulty in our generic setting is that, as in Sec. 4.3, there may be several local optima. This is different when Assumption-1 is verified, where there is only one class of equivalent worst models against which all robust plans perform equally well.

Assuming that we have a global optimum, it is possible to converge to a worst model. by exchanging roles between the planner and its opponents in seqRPG or simRPG. The idea is, having found a vector $\vec{\theta}^*$ corresponding to a robust plan, to constrain $\vec{\theta}$ to remain in a neighbourhood of $\vec{\theta}^*$ and stabilise the model with an unstable policy (so that the players are changing sides in seqRPG or simRPG).

Finally, this approach does not appear to be specific to robust planning with uncertain models and could be used for adversarial planning as well. This leads to another direction to explore: the extension of this approach for general-sum games when several agents are planning with different objectives.

Conclusion — Recent works show that model uncertainty is a major issue in decision-theoretic planning, yet most approaches are based on an assumption not verified in many realistic frameworks. We have proposed two algorithms finding robust plans, while usual policy-gradient algorithms can lead to oscillating behaviors in self-play. We demonstrate seqRPG and simRPG on prototypical uncertain problems in a view to provide a better understanding of these problems as well as the algorithms.

ACKNOWLEDGEMENTS

National ICT Australia is funded by the Australian Government. This work was also supported by the Australian Defence Science and Technology Organisation.

REFERENCES

[1] D. Aberdeen, 'Policy-gradient methods for planning', in *Advances in Neural Information Processing Systems 19 (NIPS'05)*, (2005).

[2] D. Aberdeen, S. Thiébaux, and L. Zhang, 'Decision-theoretic military operations planning', in *Proc. of the 14th Int. Conf. on Automated Planning and Scheduling (ICAPS'04)*, (2004).

[3] Anonymous, 'Policy-Gradient for Robust Planning', Technical report, (2006). `http://totor54.free.fr/ecai06-ext.pdf`.

[4] J.A. Bagnell, A. Y. Ng, and J. Schneider, 'Solving uncertain Markov decision problems', Technical Report CMU-RI-TR-01-25, Robotics Institute, Carnegie Mellon U., (2001).

[5] J. Baxter and P. Bartlett, 'Infinite-horizon policy-gradient estimation', *Journal of Artificial Intelligence Research*, **15**, 319–350, (2001).

[6] J. Baxter, P. Bartlett, and Lex Weaver, 'Experiments with infinite-horizon, policy-gradient estimation', *Journal of Artificial Intelligence Research*, **15**, 351–381, (2001).

[7] M. Bowling and M. Veloso, 'Multiagent learning using a variable learning rate', *Artificial Intelligence*, **136**, 215–250, (2002).

[8] O. Buffet and D. Aberdeen, 'Robust planning with (L)RTDP', in *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05)*, (2005).

[9] A. R. Cassandra, *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*, Ph.D. dissertation, Brown U., Dept. of Computer Science, Providence, RI, 1998.

[10] Y.-H. Chang and L.P. Kaelbling, 'Playing is believing: the role of beliefs in multi-agent learning', in *Advances in Neural Information Processing Systems 14 (NIPS'01)*, (2001).

[11] J. Hoffmann and B. Nebel, 'The FF planning system: Fast plan generation through heuristic search', *Journal of Artificial Intelligence Research*, **14**, 253–302, (2001).

[12] Iain Little, Douglas Aberdeen, and Sylvie Thiébaux, 'Prottle: A probabilistic temporal planner', in *Proc. of the 20th American Nat. Conf. on Artificial Intelligence (AAAI'05)*, (2005).

[13] Mausam and D.S. Weld, 'Concurrent probabilistic temporal planning', in *Proc. of the 15th Int. Conf. on Planning and Scheduling (ICAPS'05)*, (2005).

[14] A. Nilim and L. El Ghaoui, 'Robustness in Markov decision problems with uncertain transition matrices', in *Advances in Neural Information Processing Systems 16 (NIPS'03)*, (2004).

[15] S. Singh, M. Kearns, and Y. Mansour, 'Nash convergence of gradient dynamics in general-sum games', in *Proc. of the 16th Annual Conf. on Uncertainty in Artificial Intelligence (UAI'00)*, pp. 541–548, (2000).

[16] H. L. S. Younes and R.G. Simmons, 'Policy generation for continuous-time stochastic domains with concurrency', in *Proc. of the 14th Int. Conf. on Automated Planning and Scheduling (ICAPS'04)*, (2004).

[17] M. Zinkevich, 'Online convex programming and generalized infinitesimal gradient ascent', in *Proc. of the 20th Int. Conf. on Machine Learning (ICML'03)*, (2003).