Rolling Horizon methods for Games with Continuous Actions

Samuel A. Roberts, Spyridon Samothrakis, Diego Perez, Simon M. Lucas

***Abstract*—It is often the case that games have continuous dynamics and allow for continuous actions. For larger games with complicated dynamics, having agents learn offline behaviours is a daunting task. On the other hand, provided a forward model is available, one might try to spread the cost of search in a rolling horizon fashion (e.g. as in example MCTS). In this paper we compare HOLOP, an open loop planning algorithm and a version evolutionary planning using CMA-ES. He show that rolling horizon CMA-ES outperforms HOLOP in two classic benchmark problems (Cart Balancing and ¡problem name¿) and Lunar Lander, a classic arcade game. We conclude that off-the-shelf evolutionary algorithms can be used successfully in a rolling horizon setting.**

I. Introduction

One of the most common uses of function optimisation is for solving control problems. If an agent has access to a generative model of the world (i.e. a simulator) in which it is going to act, a control problem becomes a search/optimisation problem, usually referred to as "simulation based planning". Recently, a family of algorithms, mostly known as Monte Carlo Tree Search (MCTS) [1], has been applied to planning problems of discrete states and actions with considerable success. The focus of these algorithms is to attack extremely large state spaces of perfect information games, where one can sample rewards from the state space easily (e.g. Computer Go [2]). With access to a generative model (from which one can easily sample) and perfect sensor information, the following procedure works well: the agent receives sensor information, formulates a plan of action, performs the first action of that plan, receives a new state, re-formulates a plan, acts again (using the first action of its plan), ad infinitum.

The most important reason for continuous re-planning is the fact that most planning algorithms' computational complexity is linear (or worse) in the number of states, whereas the number of states increases exponentially at each time step. One thus hopes to perform an action that looks good now, act, and replan, effectively creating a smaller problem or a "closer to action" horizon. This kind of behaviour is known as rolling horizon, sample based or model predictive control/planning [3]. Most of these algorithms are not exact; at each time step, due to the large state space, planning takes place in a Monte Carlo fashion, with random "rollouts" (i.e. simulations) guiding the algorithm in a best first search manner. All this planning and replanning is computationally intensive, thus very efficient use of samples should be made. In the case of discrete state and actions, bandit algorithms [4] provided a formidable solution to this problem. An example of exploiting this efficient sampling can be found in Samothrakis et al. [5] where a high-performance MCTS Pac-Man agent was developed using only $50 - 300$ simulations per action (the game simulator requested an action from the controller every 40ms).

Recently, algorithms stemming from, or at least partially inspired by, MCTS aiming at solving planning problems involving continuous states and actions have come to light. A typical such algorithm is Hierarchical Optimistic Open Loop Planing (HOLOP) [6]. HOLOP is based partially on a strong [7] real-valued optimisation algorithm called HOO (for "Hierarchical Optimistic Optimisation"). In this paper two major contributions are made plus a smaller "add-on"; the first contribution is the comparison of two versions of a very popular and strong evolutionary strategy: Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [8] and its Uncertainty Handling version UH-CMA-ES [9] with Truncated Hierarchical Optimistic Optimisation (T-HOO) [7] on the Black Box Optimisation Problem (BBOP) benchmark suit [10]. T-HOO is a faster version of HOO [7], an algorithm reminiscent of Monte Carlo Tree Search. The experiments are conducted using similar conditions to those an agent would face if it was to solve planning problems. Our second contribution is to incorporate CMA-ES and UH-CMA-ES into three versions of an algorithm called EOLOP, for Evolutionary Open Loop Optimistic Planning. This algorithm uses evolution instead of tree search for implementing sample-based control algorithms. We then proceed to test the algorithms proposed on noisy versions of standard reinforcement learning (RL) benchmarks, namely the inverted pendulum and the double integrator problems. Our "extra" contribution is an add-on for evolutionary algorithms (transfered from HOLOP) that can help when the number of function evaluations is very small.

While evolution is commonly used to evolve a reactive neural network controller (in a process commonly called neuro-evolution) for these and other RL problems, it should be noted that evolution is being applied in a very different way here. In this paper, evolution is applied to perform each action given the current state. This approach can only be applied when a generative model is available, but has the advantage of offering immediate good performance without any prior learning. The disadvantage, compared to neuro-evolution, is that every action performed requires CPU time for the simulations. This is described in more detail in section III.

The rest of this paper is structured as follows. Section **??** discusses the relevant base algorithms that are going to be used in this paper, ((Truncated) Hierarchical Online Optimisation [7] and CMA-ES [8]). Section III explains why and how the above algorithms can be used in the context of planning. Section **??** describes how one can incorporate evolution into this, making this our main contribution section from an algorithmic standpoint. In Section **??** the experimental setup is described. In Section **??** a number of experiments are portrayed and analysed. These form the bulk of this paper's contribution. We conclude with a short discussion in Section **??**.

II. Motivation

A. *Solving Continuous Problems with Continuous Actions*

Previous research into using MCTS to provide solutions for continuous physics-based problems have examined cases such as the Physical Travelling Salesman Problem [11]. Within these continuous contexts, discrete actions have proven to be quite effective, especially with use of macro-actions and some degree of higher level planning [12].

Lunar Lander is an excellent test case to study for many reasons, due to its inherently multi-objective nature. With the objectives to both land quickly while also minimising velocity changes, which cost fuel which must be conserved, it has been an interesting scenario to examine the properties and constraints imposed just by the environment alone. While there have been attempts to solve this deceptively complex problem before using discrete macro-actions [13], in this paper the problem is being approached as one that can be better solved with continuous actions.

III. Planning

In this section a formalised version of planing will be presented and an explanation provided on how the algorithms described in the previous section can be used to attack the problem.

A. *Markov Decision Processes*

The main decision theoretic abstraction for planning/control is the Markov Decision Process (MDP) [14]. Formally, an MDP is a tuple $\langle S, A, T, R, \gamma \rangle$, where:

- S is a set of states, $s \in S$, where s^t_j is state j at time t and s'_j is the state j at time $t+1$.
- A is a set of actions, each action named a_j.
- $T : S \times \S \times A \to [0,1]$ is the probability of moving from state s to state s' after action $a \in A$ has taken place. $T(s'|s,a)$ denotes this probability.
- $R : S \to \Re$, $R(s)$ is a reward function at each state.
- γ, a discount factor.

The MDP defines a single agent environment, fully observable to the agent. In an MDP, the Markov property holds (hence the name), which means that all the information an agent needs in order to act is embedded in the current state. A possible route of action an agent might take is known as the *policy* π. A policy is a probabilistic mapping between state and actions, $\pi : S \times A \to [0,1]$, thus $\pi(s,a) \in [0,1]$, $\sum_{a \in A} \pi(s,a) = 1$. The set of all policies is denoted as Π. The goal of an agent in an MDP environment is to maximise its long term value $V(\pi, s_0) = R(s) + \sum\limits_{s \in S} \sum\limits_{a \in A} \gamma \pi(s,a) T(s'|s,a) V(\pi, s')$, by assigning the probability of actions to take at each state.

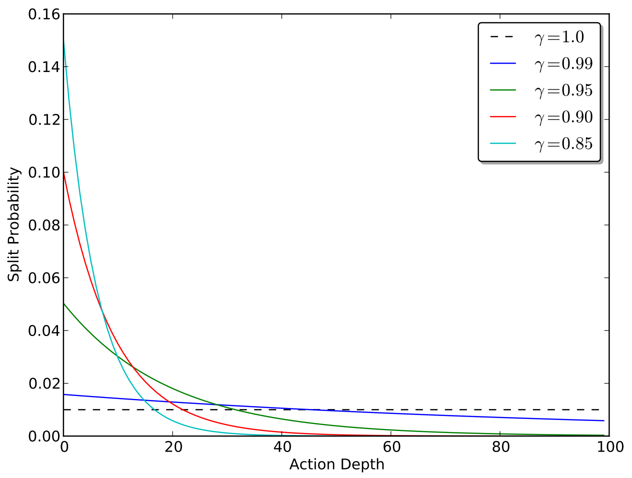


Fig. 1: Examples of how rewards are affected by γ.

B. *Taxonomy of Planning Algorithms*

The terminology of Chang [3] is followed when trying to classify planning algorithms. If the policy $\pi : S \times A \to [0,1]$ is followed, what is known as "closed loop" planning takes place. The agent takes an action, senses its environment, takes another action etc. An alternative to closed-loop planning, "open-loop" planning, requires defining time. An agent sees a "time" when it's about to act instead of a state, i.e it doesn't have access to s_t, but just t. An open loop policy (or plan or control) is when an agent learns a policy with the form $\pi_o : S \times \mathcal{T} \to [0,1]$, where $\mathcal{T}$ is an ordered set of time steps. Thus, an agent takes actions irrespective of the current state it is in. Intuitively this means that a sequence of actions is formed by the agent and the agent will take these actions in sequence (e.g. $(a_1, a_2, a_3, a_4, a_5, a_6, \ldots, a_n)$). Obviously this is not optimal[1], although sometimes is much easier to do.

Another point of interest is planning, and when it happens. In open loop planning (and what is usually termed "planning" without any further qualifications), an agent forms a plan once and follows it until the end.

If the agent replans at every step, discarding or augmenting the plan received from previous steps, it is known as "rolling horizon" planning. The idea here is that, if the agent can plan as well as possible up to a certain point, it can perform an action, move the planning horizon one step forward and replan. For example, Monte Carlo Tree Search for infinite MDPs can be seen as an approximate rolling horizon planning version of $TD(1)$[15]. The term "Simulation based" is used when the planning happens for an MDP that has a tree like structure and rewards are only visible at the end of the tree, a situation common in many games.

When learning how to act using a generative model, one can make a third distinction as to the type of sets represented by A and S. In this paper the focus is on an A and S that

[1]Not optimal in the general case. It is optimal under the condition that the MDP is deterministic, i.e. each action leads to one specific state with probability one, and everything else has a probability zero

come from a metric space. This means that both sets' elements define a notion of distance and, for all practical purposes, have an infinite amount of elements. More specifically, both sets are drawn from a bounded set of real numbers, $\Re^n$.

Since an open loop policy is now a real-valued vector of actions, algorithms like T-HOO and CMA-ES can be used to find such a policy. Hence the title of this paper: Rolling Horizon Open-loop Policy Evolutionary Learning. To the best of our knowledge this is the first time evolution has been used in such a setting. There are examples of closed loop rolling horizon papers, but these are beyond the scope of this article [16], [17].

C. Planning using a Generative Model

The Markov Decision Process simply describes an environment alongside the rewards that said environment provides to an agent. A problem central to Artificial Intelligence is what actions an agent must take to maximise its long term reward. The most basic assumption one can make is that the agent knows nothing about the environment. In this case the agent gets thrown into the environment and interacts ("trains") for some time. After the training is complete, the agent should be able to act intelligently (presumably with some success).

Another approach one can take is to assume that the agent has a copy of the MDP in its "mind" and thus is able to plan accordingly. This is usually termed dynamic programming [18], however it cannot be used in the case of continuous states or actions without the use of a function approximator (FA) or some kind of discretization. Another approach, and one which has made a significant impact recently, arguably presented for the first time by Kearns [19] and popularised mostly as Upper Confidence Bounds for Trees (UCT) [20], is to use a generative model.

A generative model is nothing more than a simulator of future events. This is a less of a requirement than knowing the full model (which entails knowing the full function T), but still requires the agent to have some knowledge of the world. This category of algorithms, which are broadly termed "Rolling Horizon", has the following steps. From the current state, a number of simulations of possible futures is run by executing a set of action sequences (all of this in the "mind" of the agent). The simulations last for a certain number of steps (hence the "rolling horizon") or continue until an end state is encountered (i.e. when the MDP has "absorbing" states that cannot be acted from).

Once a satisfactory plan/policy has been estimated (or the agent runs out of time, iterations etc.) it proceeds by executing the first action of the plan, thus moving to the next state. Once there, the agent performs the same sequence of "mental" actions, (i.e. plans and executes the first action), until some predefined stopping condition. In this paper we will concern ourselves *only* with this method of planning. What changes from method to method is how the plan is formed.

In standard UCT, a procedure similar to temporal difference learning is followed, where one learns the value of each state. An agent then forms a policy by choosing a value greedily. In our case, because our state space has continuous states and actions, this is impossible without the use of a function approximator (or an infinite discretization of space), which is not attempted here. What is attempted though is to learn the policy function directly without reference to state, while planning continuously from the current state. So while the agent forms open-loop plans at each clock tick, in reality it is executing a closed-loop policy. While the strategy is not provably optimal, it can work well in practice [6]. This will be seen by our experimental results (see Section **??**).

Algorithm 1 Rolling Horizon Open Loop Planning

```
define external function FINDOPENLOOPPLAN($depth$) ▷ Try to find the optimal policy. You can use either evolution or HOO here
procedure OOOP
    while $True$ do
        $a \leftarrow$ FINDOPENLOOPPLAN($depth$)[0]    ▷ Execute first action of the policy
        $s \sim T(s'|s,a)$    ▷ Stop only at an absorbing state
```

IV. Methodology

A. Simple Planning Benchmarking Problems

For direct comparison with recent work on sample based planning using trees, in particular Weinstein and Littman [6] and Pazis and Lagoudakis [21], we used the same problems: double integrator, and inverted pendulum. The basic experimental setup followed Weinstein and Littman [6] and Pazis and Lagoudakis [21] closely, except that they used a fixed noise level for each experiment whereas we varied the noise level to explore how performance was affected.

Both problems are modelled using continuous state discrete time simulations. In each case the desired acceleration is corrupted by additive uniform random noise before being limited within the specified range. The noise levels are described in the next section.

1) Double Integrator: The double integrator problem is to control a mass along a single dimension. The state space is 2-dimensional, consisting of (p, v) where p is position and v is velocity. The goal is to change the state from $(1, 0)$ to $(0, 0)$. Both position and velocity are clamped to be within the range -2 to $+2$. At each time step the controller selects the desired acceleration a', which then is noise corrupted and range limited between $-1.5N$ to +1.5N to yield the applied acceleration a.

Euler integration is then used to update the position and velocity, given the time step δ_t:

$$v_{t+\delta_t} = v_t + a\delta_t \quad (1)$$

$$p_{t+\delta_t} = p_t + v_{t+\delta_t}\delta_t \quad (2)$$

At each time step the reward r is:

$$r = -(p^2 + a^2) \quad (3)$$

The problem becomes harder as the time interval δ_t at each action is applied increases, and following Pazis and Lagoudakis we set ($\delta_t = 0.5s$).

2) Inverted Pendulum: The inverted pendulum problem is also known as the pole balancing or cart-pole problem. The goal is to keep the pendulum / pole as upright and as still as possible. This is achieved by accelerating the cart which affects the freely pivoted pendulum.

In this instance of the problem the controller only sees a two dimensional state space, consisting of the angle θ that the inverted pendulum deviates from the vertical, and the angular velocity of the pendulum $\dot{\theta}$. The controller selects the desired angular force F, which is then corrupted by additive uniform random noise and limited in the range $-50N$ to +50N so as to yield the applied angular force u. The angular acceleration $\ddot{\theta}$ is then calculated as follows:

$$\ddot{\theta} = \frac{gsin(\theta) - \alpha ml(\dot{\theta})^2 sin(2\theta)/2 - \alpha cos(\theta)u}{4l/3 - \alpha mlcos^2(\theta)} \tag{4}$$

where g is the acceleration due to gravity ($g = 9.8ms^{-2}$), m is the mass of the pendulum ($m = 2kg$), ($\alpha = 1/(m+M)$), M is the mass of the cart ($M = 8kg$) and l is the length of the pendulum ($l = 0.5m$). The time interval δ_t was set to $0.1s$: i.e. the simulation is updated 10 times per second:

$$\dot{\theta}_{t+\delta_t} = \dot{\theta}_t + \ddot{\theta}\delta_t \tag{5}$$

$$\theta_{t+\delta_t} = \theta_t + \dot{\theta}_{t+\delta_t}\delta_t \tag{6}$$

At each timestep the reward r punishes deviation from the vertical, high speed, and high force:

$$r = -((2\theta/\pi)^2 + \dot{\theta}^2 + (F/50)^2) \tag{7}$$

B. Environment and Physics

1) Environmental Properties: The properties of the base environment of *Lunar Lander* are that it is frictionless, and that it is a two-dimensional plane with horizontal wrapping. This can also be conceptualised as a cylinder. Anything that passes from the left edge of the playing field moves to the right instantaneously, and vice versa.

The other feature of note is the jagged landscape, a simplistic representation of the noisy, crater-filled lunar landscape the game represents. This landscape is constructed as a series of line segments, with each vertex of the jagged landscape being distributed equally horizontally, and randomly vertically.

Each vertex of this landscape is generated sequentially.

2) Spaceship Physics: The spaceship within this game world is modelled as an a circular mass with some basic physical properties, such as a position within the game world $\boldsymbol{s}$, a velocity $\boldsymbol{v}$, an orientation θ, an angular velocity ω and a radius of a bounding sphere r used for collision detection with the landscape.

This collision detection method is based on the lowest point of the spaceship's bounding circle coming into intersection with the highest point of the landscape for the line perpendicular between the ship and the bottom of the playing field, as can be seen in Figure 2.

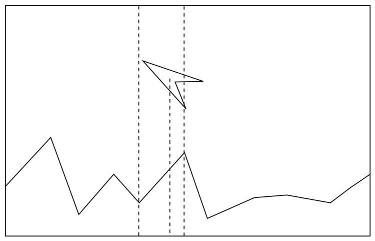


Fig. 2: The point closest to the spaceship on the landscape lies in between two of the defined vertices of the landscape, and requires interpolation to calculate the y co-ordinate from the x co-ordinate of the ship.

As the landscape is stored as a series of line segments, interpolation must be used in the event the ship is not perfectly aligned with one of the landscape axes. The point that lies on the line segment that the ship is being checked against is calculated through simple linear interpolation based on which two vertices the ship is horizontally closest to. For the ship's centre $\boldsymbol{s}$, the left nearest landscape vertex $\boldsymbol{p}^l$ and the right nearest landscape vertex $\boldsymbol{p}^r$, the point of collision $\boldsymbol{p}^c$ against the landscape is calculated as

$$\boldsymbol{p}^c_x = \boldsymbol{s}_x \boldsymbol{p}^c_y = \boldsymbol{p}^l_y + v(\boldsymbol{p}^r_y - \boldsymbol{p}^l_y) \tag{8}$$

where v is a value between 0 to 1 used for interpolation, and can be calculated as follows.

$$v = \frac{\boldsymbol{s}_x - \boldsymbol{p}^l_x}{\boldsymbol{p}^r_x - \boldsymbol{p}^l_x} \tag{9}$$

Collision is then true if the following statement is true:

$$\boldsymbol{s}_y + r \geq \boldsymbol{p}^c_y \tag{10}$$

The ship colliding with the landscape constitutes the end of the *Lunar Lander* game. The conditions surrounding this collision, including speed, orientation of the ship, and fuel used, constitute whether the nature of the collision is a success or a failure.

C. Continuous Action MCTS

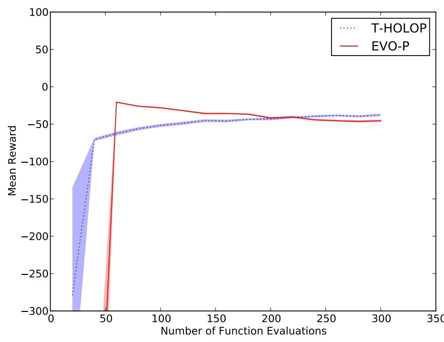
1) Heuristic:

V. Results

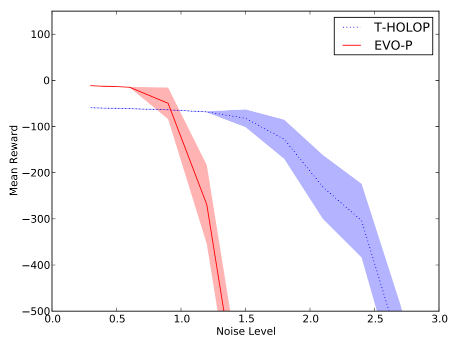
VI. Conclusion

Acknowledgements

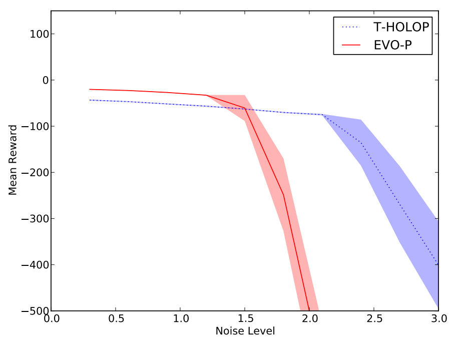
Samuel Roberts is supported by an EPSRC PhD Studentship.



(a) Scores for a noise level of $l = 1$ under variable function evaluations.

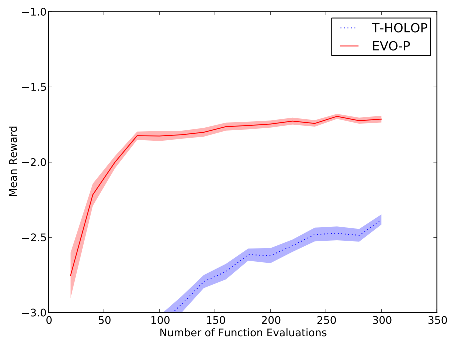


(b) Scores for 50 iterations under variable noise level l. Notice how UH-EOLOP fails to balance the pole.

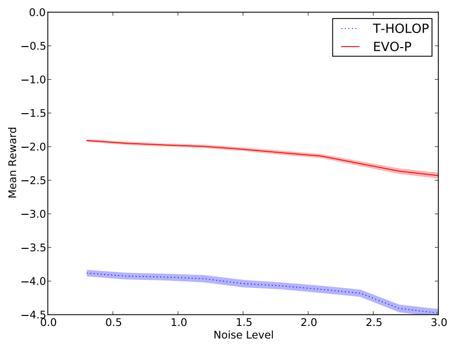


(c) Score for 100 iterations under variable noise level l.

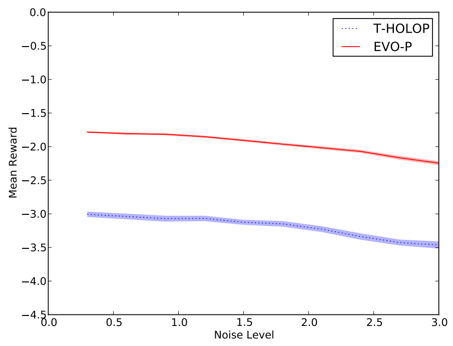
Fig. 3: Performance of all algorithms in the Inverted Pendulum benchmark (error bars for the 95th percentile). Higher scores are better



(a) Scores for a noise level of 1 under variable function evaluations.



(b) Scores for 50 iterations under variable noise level l.



(c) Score for 100 iterations an variable noise level l.

Fig. 4: Performance of all algorithms in the Double Integrator benchmark (error bars for the 95th percentile). Higher scores are better

References

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[2] S. Gelly, Y. Wang, R. Munos, O. Teytaud *et al.*, "Modification of uct with patterns in monte-carlo go," INRIA, Tech. Rep., 2006.

[3] I. Chang, M. Fu, J. Hu, and S. Marcus, *Simulation-based algorithms for Markov decision processes (communications and control engineering)*. Springer Verlag London Limited UK, 2007.

[4] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[5] S. Samothrakis, D. Robles, and S. Lucas, "Fast approximate max-n monte carlo tree search for ms pac-man," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 2, pp. 142–154, 2011.

[6] A. Weinstein and M. Littman, "Bandit-based planning and learning in continuous-action markov decision processes," in *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[7] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvári, "X-armed bandits," *Journal of Machine Learning Research*, vol. 12, pp. 1655–1695, 2011.

[8] N. Hansen, S. Muller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)," *Evolutionary Computation*, vol. 11, no. 1, pp. 1–18, 2003.

[9] N. Hansen, S. Niederberger, L. Guzzella, and P. Koumoutsakos, "A Method for Handling Uncertainty in Evolutionary Optimization with an Application to Feedback Control of Combustion," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 1, pp. 180–197, 2009.

[10] S. Finck, N. Hansen, R. Ros, and A. Auger. (2012, Dec) Real-Parameter Black-Box Optimization Benchmarking 2010: Noisy Functions Definitions. INRIA. [Online]. Available: http://coco.lri.fr/downloads/download11.06/bbobdocnoisyfunctions.pdf

[11] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, and S. M. Lucas, "Solving the physical travelling salesman problem: Tree search and macro-actions," *IEEE Transactions on Computational Intelligence and Games*, 2014.

[12] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Monte carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem," *IEEE Transactions on Computational Intelligence and Games*, 2012.

[13] S. A. Roberts and S. M. Lucas, "Measuring interestingness of continuous game problems," *IEEE Transactions on Computational Intelligence and Games*, 2013.

[14] R. Howard, "Dynamic programming and markov decision processes," *Cambridge, MA*, 1960.

[15] D. Silver, *Reinforcement learning and simulation-based search in computer Go*. University of Alberta, 2009.

[16] Z. Xiang-Yin and D. Hai-Bin, "Differential evolution-based receding horizon control design for multi-uavs formation reconfiguration," *Transactions of the Institute of Measurement and Control*, vol. 34, no. 2-3, pp. 165–183, 2012.

[17] S. Samothrakis and S. Lucas, "Planning using online evolutionary overfitting," in *Computational Intelligence (UKCI), 2010 UK Workshop on*. IEEE, 2010, pp. 1–6.

[18] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. Cambridge Univ Press, 1998, vol. 1, no. 1.

[19] M. Kearns, Y. Mansour, and A. Y. Ng, “A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes,” *Machine Learning*, vol. 49, no. 2, pp. 193–208, Nov. 2002.

[20] L. Kocsis and C. Szepesvári, “Bandit based monte-carlo planning,” *Machine Learning: ECML 2006*, pp. 282–293, 2006.

[21] J. Pazis and M. Lagoudakis, “Binary action search for learning continuous-action control policies,” in *Proceedings of the 26th International Conference on Machine Learning*, 2009, pp. 793–800.