Rolling Horizon methods for Games with Continuous States and 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, possibly with with some added noise. For larger games with complicated dynamics, having agents learn offline behaviours in such a setting is a daunting task. On the other hand, provided a forward model is available, one might try to spread the cost of search/learning in a rolling horizon fashion (e.g. as in Monte Carlo Tree Search). In this paper we compare T-HOLOP (Truncated Hierarchical Open Loop Planning), an open loop planning algorithm at least partially inspired by MCTS with a version evolutionary planning using CMA-ES (which we call EVO-P) in two planning benchmark problems (Cart Balancing and the Double Integrator) and Lunar Lander, a classic arcade game. He show that EVO-P outperforms T-HOLOP in the classic benchmarks, while T-HOLOP is unable to find a solution using the same heuristic. We conclude that off-the-shelf evolutionary algorithms can be used successfully in a rolling horizon setting, and that a different type of heuristics might be needed under different optimisation algorithms.

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 contributions are made; the first contribution is the comparison of two versions of a very popular and strong evolutionary strategy: EVO-P, the open loop planning algorithm using Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [8] with a version of HOLOP, which stands for Hierarchical Open Loop Optimistic Planning [7] on two standard benchmark problems, the double integrator and the reverse pendulum. For speed purposed, we use T-HOO (Truncated Hierarchical Optimistic Optimisation), a faster version of HOO [7], as the engine of HOLOP, hence we name the algorithm used T-HOLOP. Our second contribution is the use of the these algorithms in Lunar Lander, a popular arcade game, coupled with a strong two-stage heuristic. We show that while EVO-P performs reasonably well, T-HOLOP struggles.

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 and in line with what is being done in Perez et al. [9], although this time in a continuous setting.

The rest of this paper is structured as follows. Section II discusses (briefly and epigrammatically) 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 (and thus transformed into HOLOP and EVO-P respectively). In Section IV the experimental setup is described. In Section V 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 VI.

II. ALGORITHMS FOR CONTINUOUS FUNCTION OPTIMISATION

This is the first background section of this paper, which aims to introduce the two core optimisation algorithms on top of which our contributions are made. We will present both algorithms from an implementer's viewpoint, without explicitly justifying the design and mathematical choices behind them, nor going into too much detail. Interested readers should look at the cited bibliography for extensive discussions.

In general, the optimisation problem can be defined as: Given a function $f : \boldsymbol{X} \to Y$, where $X \in \mathcal{R}^n$, with n being the length of vector $\boldsymbol{X}$. The domain $\boldsymbol{X}$ is usually called the search space. Our goal is to find a vector $\boldsymbol{x_0}$, for which $f(\boldsymbol{x}) > f(\boldsymbol{x_0})$ for all x (also known as "minimisation") or $f(\boldsymbol{x}) < f(\boldsymbol{x_0})$ for all x (known as "maximisation"). There are an enormous number of algorithms devoted to this problem, but only two will be presented and compared here, which are both considered state of the art in their respective fields and do not require the calculation of a gradient. If one has an analytic expression of the gradient or if the functions proposed are smooth enough, it might be better to look elsewhere for solutions (e.g. Conjugate Gradient Methods [10]).

A. CMA-ES

The Covariance Matrix Adaptation - Evolutionary Strategy (CMA-ES) is an evolutionary algorithm designed for continuous domains, specially suited for non-linear and non-convex optimization problems [11]. In general, this algorithm is applied to those problems that are not constrained and are made of up to 100 dimensions.

A multivariate normal distribution (MND) is a generalization of the univariate normal distribution, which is a probability distribution that has a bell-shaped probability density function (Gaussian function). A multivariate vector $X = (x_0, x_1, \ldots x_N), x_i \in \mathcal{R}$ is said to have an MND if it satisfies that any linear combination from its components, $w_0 x_0 + w_1 x_1 + \cdots + w_N x_N$, is normally distributed.

CMA-ES creates a population of individuals by sampling from an MND: $\mathcal{N}(m, C)$, which is uniquely defined by the distribution mean $m \in \mathcal{R}^n$ and its covariance matrix $C \in \mathcal{R}^{n \times n}$. The top of the density function, which corresponds to m, also determines the translation of the distribution. The covariance matrix C, positive definite and symmetric, determines the shape of the distribution and its graphical interpretation: it defines an iso-density ellipsoid $\{x_i \in \mathcal{R} | (x-m)^T C^{-1}(x-m) = 1\}$. Through the classic evolutionary setup and by making clever use of population statistics, the algorithm proceeds from generation to generation of sampling until some convergence criteria has been reached.

B. T-HOO

Recently, a family of algorithms based on discretizing the search domain has been proposed. Only one of these algorithms is going to be discussed here, Truncated Hierarchical Optimistic Optimisation (T-HOO) [7]. The T-HOO algorithm was built from the ground up to perform optimisation of noisy functions. It is an iterative, anytime algorithm, which is run for n_0 iterations. The algorithm is based on a number of components. The first one is a data structure called a node and designated ν. The node includes the maximum and minimum values of our search space, vectors $\boldsymbol{x_{max}}$ and $\boldsymbol{x_{min}}$ respectively. The node also has a property called $count$, which initially is set to zero and a property called sum, which, again, is initially set to zero. An unvisited node (i.e. $count = 0$) is called a leaf node. "Splitting" a node is the procedure of creating two children $C(\nu) = \{C_l(\nu), C_r(\nu)\}$. Both children are initialised $\boldsymbol{x_{max}}$ and $\boldsymbol{x_{min}}$ to their parents' nodes values. When there are multiple dimensions to choose from, at at k according to some random distribution. Thus, for child $C_l(\nu)$, $\boldsymbol{x}_{max,k}^{C_l} = \boldsymbol{x}_{min,k}^{C} + \left(\boldsymbol{x}_{min,k}^{C} - \boldsymbol{x}_{max,k}^{C}\right)/2$. For the other child, $\boldsymbol{x}_{min,k}^{C_r} = \boldsymbol{x}_{min,k}^{C} + \left(\boldsymbol{x}_{min,k}^{C} - \boldsymbol{x}_{max,k}^{C}\right)/2$. In other words, the father node is split into two children nodes, choosing which dimension to split uniform randomly (or according to some other criterion). Each node also has an associated value called $B(\nu)$, which is defined recursively as: $B(\nu) = min\{U(\nu), max\{B(C_l(\nu)), B(C_r(\nu))\}\}$. Part of this equation is the term $U(\nu)$, which is defined as in Equation 1:

$$U(\nu) = S(\nu)/N(\nu) + \sqrt{2 ln n_0/N(\nu)} + \upsilon_1 \rho^h \qquad (1)$$

If a dissimilarity metric is defined between different values of $\boldsymbol{X}$ as $\|\boldsymbol{x_1} - \boldsymbol{x_2}\|^\alpha$, one can set $\upsilon_1 = \sqrt{|A|}/2)^\alpha$ and $\rho = 2^{-\alpha/|A|}$. In our experiments $a = 2$. If $N(\nu) = 0$, $B(\nu) = U(\nu) = \infty$, i.e. unvisited nodes have priority. If a node has not been visited then it is called a leaf node. If a node is a leaf node, one can "draw" a sample from it, by uniformly randomly sampling between $\boldsymbol{x_{max}}$ and $\boldsymbol{x_{min}}$.

The algorithm works by starting at the root node (the initial node). Since the leaf node and the root node coincide, HOOT samples from it and splits it. It can now proceed to the second iteration, where the algorithm follows a route using the maximum B-value of its children nodes, which it then samples and splits. The process repeats until the maximum number of iterations n_0 is over. If a certain depth is exceeded there is no point splitting nodes any more (given our iteration budget n_0).

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) [12]. 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_j^t 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.
- $T : S \times S \times A \rightarrow [0, 1]$ 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.
- $R : S \rightarrow \Re$, $R(s)$ 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, $\pi : S \times A \rightarrow [0, 1]$, 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. Note that since here we are dealing with continuous actions and states, sets A and S are metric (and infinitive). Also note that we are NOT dealing with continuous time MDPs in this paper, although there is a notion of time. An example of the effects of the discount factor on rewards can be found in Figure 1.
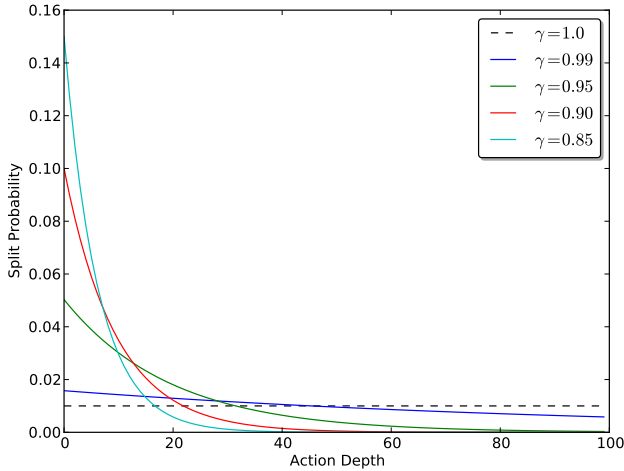

Fig. 1: Examples of how rewards are affected by γ.

B. Taxonomy of Planning Algorithms

The terminology of Chang [3] is followed when trying to classify planning algorithms. If the policy $\pi : S \times A \rightarrow [0, 1]$ 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 $\pi_o : S \times \mathcal{T} \rightarrow [0, 1]$, 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 $TD(1)$[13]. 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 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. 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 [14], [15].

C. Notes on 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 [16], 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

---

[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

one which has made a significant impact recently, arguably presented for the first time by Kearns [17] and popularised mostly as Upper Confidence Bounds for Trees (UCT) [18], 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. The main version of the algorithm used can be seen in Algorithm 1.

In standard UCT, a procedure similar to temporal difference learning is followed (or, more closely, monte carlo methods), 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 V).

Algorithm 1 Rolling Horizon Open Loop Planning

---

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

IV. METHODOLOGY

Our methodology is simple, yet efficient. We test T-HOLOP and EVO-P in two simple benchmark problems (see below for more) and Lunar Lander, a classic arcade game.

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 [19], we used the same problems: double integrator, and inverted pendulum. The basic experimental setup followed Weinstein and Littman [6] and Pazis and Lagoudakis [19] 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:

$$
\begin{aligned}
v_{t+\delta_t} &= v_t + a\delta_t & (2)\\
p_{t+\delta_t} &= p_t + v_{t+\delta_t}\delta_t & (3)
\end{aligned}
$$

At each time step the reward r is:

$$r = -(p^2 + a^2) \qquad (4)$$

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:

$$\ddot{\theta} = \frac{gsin(\theta) - \alpha ml(\dot{\theta})^2 sin(2\theta)/2 - \alpha cos(\theta)u}{4l/3 - \alpha mlcos^2(\theta)} \qquad (5)$$

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:

$$\dot{\theta}_{t+\delta_t} = \dot{\theta}_t + \ddot{\theta}\delta_t \tag{6}$$
$$\theta_{t+\delta_t} = \theta_t + \dot{\theta}_{t+\delta_t}\delta_t \tag{7}$$

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

$$r = -((2\theta/\pi)^2 + \dot{\theta}^2 + (F/50)^2) \tag{8}$$

B. Lunar Lander

In this section we give a brief overview of the Game of Lunar Lander, a popular arcade game.

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.

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 s, a velocity v, an orientation d, 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.
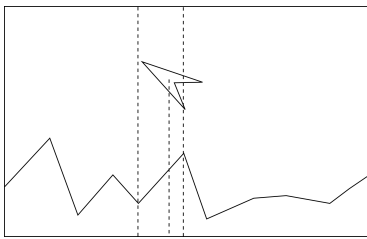

Fig. 2: The point closest to the spaceship on the landscape lies in between two of the defined vertices of the landscape, and requires interpolation to calculate the y co-ordinate from the x co-ordinate of the ship.

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 s, the left nearest landscape vertex p^l and the right nearest landscape vertex p^r, the point of collision p^c against the landscape is calculated as

$$p_x^c = s_x \quad p_y^c = p_y^l + v(p_y^r - p_y^l) \tag{9}$$

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

$$v = \frac{s_x - p_x^l}{p_x^r - p_x^l} \tag{10}$$

Collision is then true if the following statement is true:

$$s_y + r \geq p_y^c \tag{11}$$

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 constitute whether the nature of the collision is a success or a failure. In this experimental setup with accept a velocity of 5 and an angle of < 0.1 radians as successful landing criteria.

An agent can modify the ship directions by applying linear thrust a_T and the angular impulse a_I. The controllers have to identify a vector of real numbers A, which each element of the set $a_i \in [0, 1]$. Each even element of the vector represents the linear thrust, whereas each odd element represents angular velocity. Each linear thrust vector element is transformed by applying as follows:

$$a_I = 5(a_I - 0.5) \quad a_T = 300 a_T \tag{12}$$

Finally each linear thrust and angular impulse is repeated for $X = 7$ actions, the effects of which we are going to explore below.

3) Heuristic: Lunar Lander has a setup where a player gets rewards if and only if it actually lands on the pad. This creates the problem of rollouts (whether used by EVO-P or T-HOLOP) rarely will see a reward. In order to circumvent this, as it is common in other games, we use a heuristic for each state. Initially, we can define a number of vectors that portray the

$$du = [1, 0]$$
$$dl = [1, 2]$$
$$dr = [1, -2] \tag{13}$$

The vectors defined in Equation 13 define three directions for the ship. The first one, du, define a ship standing upright, the second one, dl, define tilting to the left, while the third one dr a ship tilting to the right. We make use of these vectors in order to help align the ship in the right direction, depending on which way the landing point is.
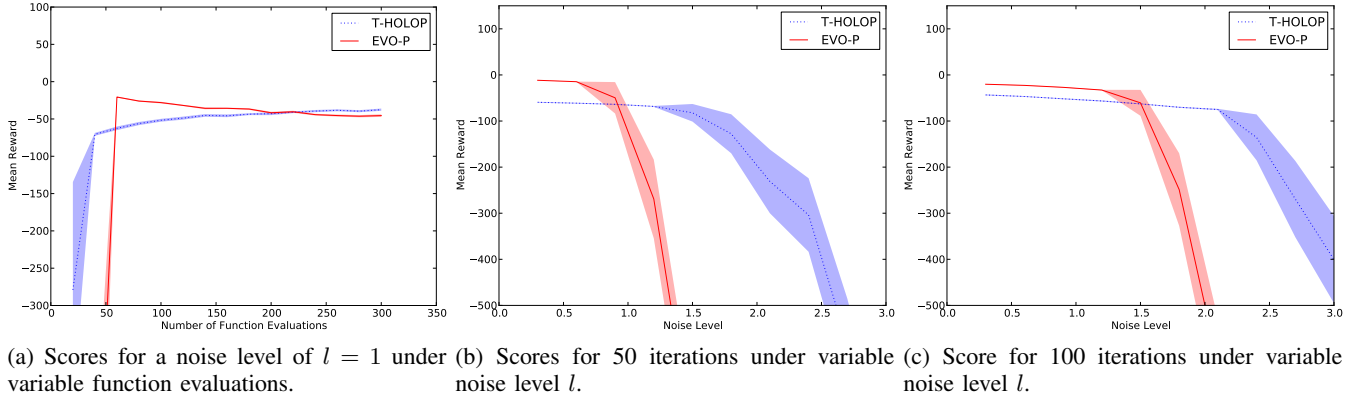
(a) Scores for a noise level of $l = 1$ under variable function evaluations.

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

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

Fig. 3: Performance of both T-HOLOP and EVO-P in the Inverted Pendulum benchmark (error bars for the 95th percentile). Higher scores are better

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

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

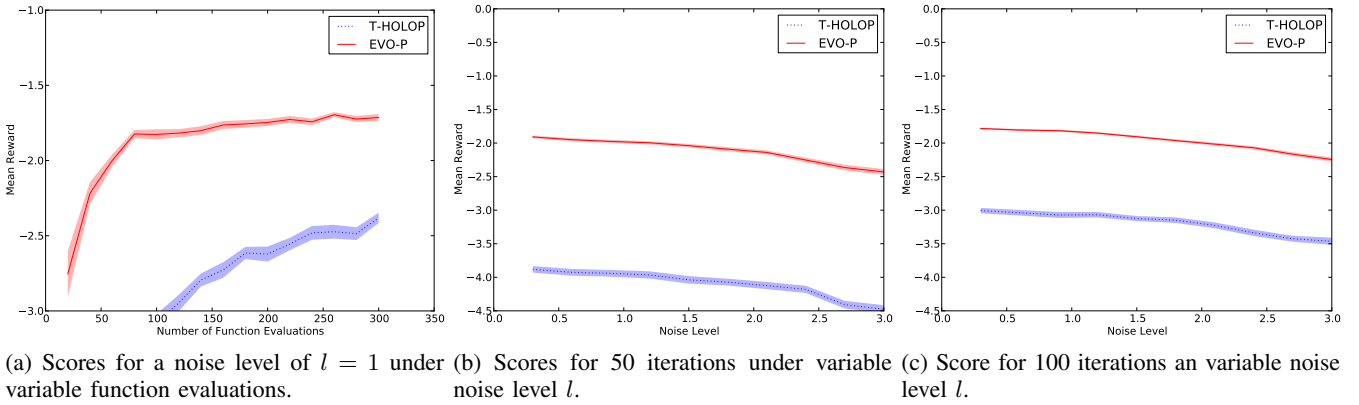(c) Score for 100 iterations an variable noise level l.

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

$$f = \begin{cases} -\sum arccos(\mathbf{dl} \cdot \mathbf{d}/\|\mathbf{dl}\|\|\mathbf{d}\|) & \text{if } \mathbf{np_x} < \mathbf{s_x} \\ -\sum arccos(\mathbf{dr} \cdot \mathbf{d}/\|\mathbf{dr}\|\|\mathbf{d}\|) & \text{otherwise} \end{cases}$$

$$l = -\sum arccos(\mathbf{du} \cdot \mathbf{d}/\|\mathbf{du}\|\|\mathbf{d}\|) - |\mathbf{v_y} - V_y^{max}|$$

$$h = \begin{cases} |-(\|\mathbf{v}\| - C) - |\mathbf{s_x} - \mathbf{np_x}|| - f & \text{if } \|\mathbf{s} - \mathbf{np}\| < D \\ l - |\mathbf{v_x} - C_l| & \text{otherwise} \end{cases}$$

(14)

In the above Equation 14 the constant C denotes the maximum magnitude of speed, which we set to $C = 30$. $D = 60$ is the distance from the landing point np where this heuristic takes precedence. Intuitively, the upper part of the heuristic h tries to get the ship above the landing point as fast as possible while maintaining a certain speed. The second part of the heuristic tries to minimise velocity on the x axis (as the ship tries to land with maximum speed $C_l = 2$), maintain a minimum landing speed on the y axis and get a certain angle (the summation term f) as close to vector du, i.e. help the ship stay upright through the duration of the rollout. It's worth noting the main reason why we use a heuristic of this kind, one that includes not only the final state, but makes use of the path of the state as well, is that there is not enough information to

discern states if one takes into account direction only, without incorporating the angular velocity. The ship might be spinning madly, but the end of the rollout might find it in the correct direction.

V. RESULTS

We present the results of our experiments below.

A. Benchmark Problems

For the second set of experiments we have defined the following variables. The first two involve running the experiments with a fixed rollout depth of 50, i.e. each algorithm at each step plans for 50 steps ahead. We set the noise levels to $0.1(\mathcal{U}(0, 1) - 0.5) * 2l$ uniform noise for the double integrator experiment and $10(\mathcal{U}(0, 1) - 0.5) * 2l$ for the Inverted Pendulum. The variable l is the noise level which defines how noisy the environment is. Initially, we performed two experiments, with a variable number of iterations and a fixed noise level of one.

An example of how the *predicted* reward looks like for a 2-step look-ahead (instead of 50 we use in the experiments) can be seen in Figure **??**. We plotted the full state-space by pushing the internal generative model two steps forward for all

(a) Map A, where the ship is lower then the pad

(b) Map B, an easy map where only the first heuristic is used

(c) Map C, where the ship is close to the top of the screen
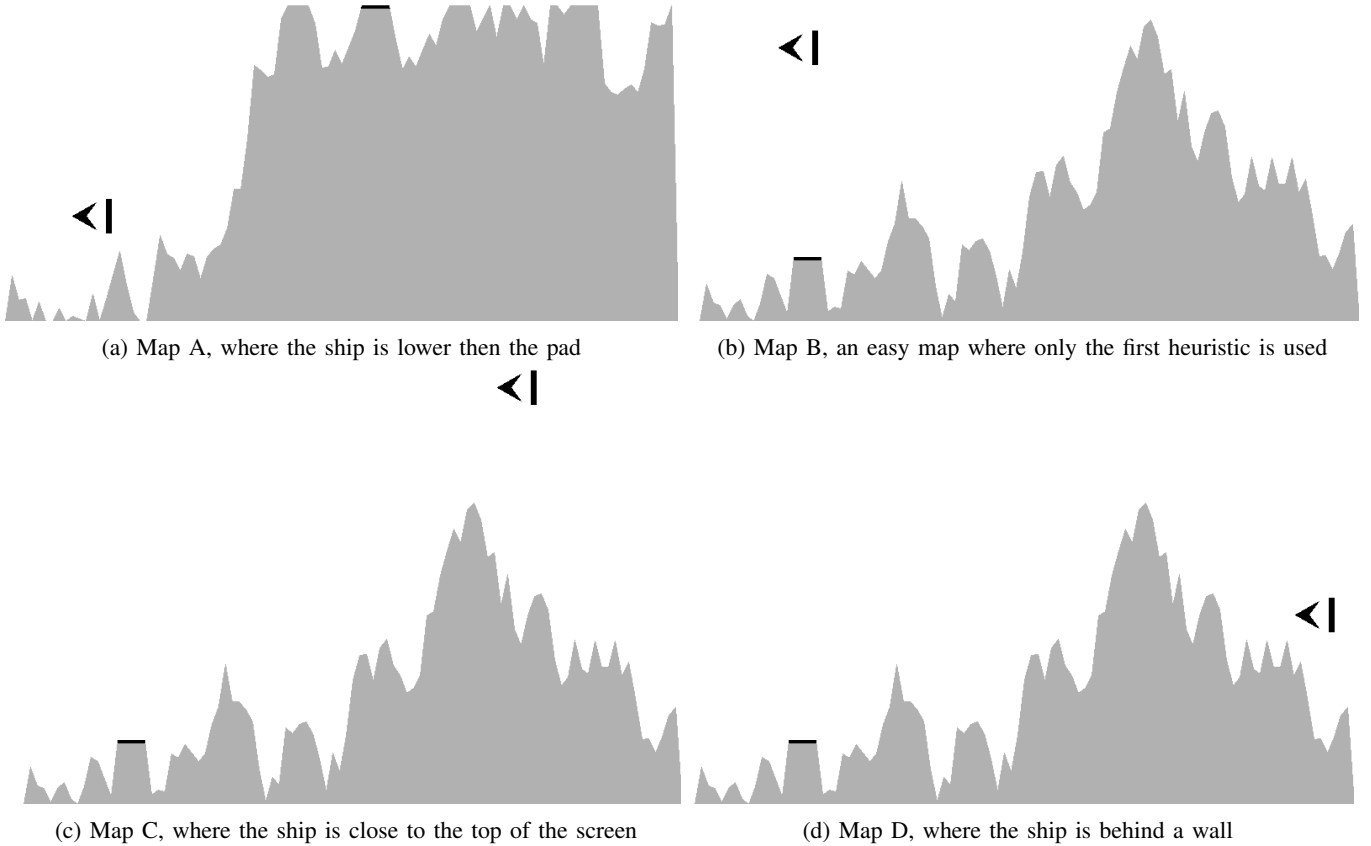

(d) Map D, where the ship is behind a wall

Fig. 5: The four different maps used in our experiments. The maps capture different position/properties

possible actions by discretizing each action. Notice how the predicted reward landscape changes over time as the agent moves. The continuous replanning should aim to perform the action with the highest future reward.

The results of these experiments can be seen in graphs 3a and 4a. These are close to the original experiments of Weinstein and Littman [6], only this time we varied the number of iterations allowed for each algorithm. The original experiment had given the algorithms 200 value iterations. With this number of iterations, for the Inverted Pendulum experiment the mean reported (using HOLOP) was -47.45. We got a better mean using T-HOLOP [2] of -42.94.

EVO-P got as score of -41.69. It also seems to have a drop in performance as we increase function evaluations, presumably because it converges to a false minima without any uncertainty handling mechanism to stop this. This kind of behaviour has been observed previously in Evolutionary Algorithms [20] when having noisy function.

For the double integrator experiment, we had slightly different results. The Weinstein paper reports a mean of -2.72

[2]This was due to a minor difference between our implementation of HOLOP and Weinstein's.

for 200 iterations. Using T-HOLOP we got a better mean of -2.62. A far better result of -1.74 is obtained by EVO-P.

Results for the variable noise levels l (Figures 3b, 3c, 4b, 4c) closely mimic the performance of their constant noise counterparts. For the inverted pendulum, using just 50 iterations (Figure 3b) EVO-P is able to stabilise the pendulum with 50 iterations. This still applies if we increase the number of function evaluations to 100 (Figure 4c). On the other hand, the results start to get closer to the fixed noise level experiments if we increase the number of iterations in the Inverted Pendulum experiment to 100 (Figure 3b).

B. Lunar Lander

Map Name	Success Mean	Success CI	Ticks Mean	Tick CI
Map A	1	0	2925.9	±184.437
Map B	1	0	2733	±159.34
Map C	1	0	2293	±123.917
Map D	0.9	±0.226216	2421.6	±682.893

For lunar lander, we do 10 runs for each map in Figure 5, and the results for EVO-P can be seen in Table **??**. The results include the mean number of successful landings and the mean time it took to land the ship, alongside their confidence

for the 95th interval. For EVO-P we used the default CMA-ES population size, each individual was bound between, each gene had a length of 8 and each action pair $(x[i], x[i+1])$ was repeated 7 times (where $x[i]$ represents thrust and $x[i+1]$ angular thrust). We derived these numbers experimentally and we did not optimise for them (not even implicitly) to any serious degree.

There are a number of things to notice here. The first one is that the only in Map D, where the ship starts the problem facing a wall do we get a failure to land, and this is due to a crash. This has to do with the fact that even a minor thrust can push you a bit forward and drive the ship to a position where there is no longer the possibility of recovery from a crush (or the combination of that recovery is extremely hard to get). The Second thing to observe is that we get there slowly, but this is by design, as we he have made the flying speed of the ship controllable. One can pottentially tweak all the default parameters we have used, plus the ones incorporated in CMA-ES as a kind of meta-heuristic search and achieve much better results. Another thing to note is that EVO-P lands the ship (on average) almost at the same time on MAPS B, C, D, although map has the same landscape as the other two but the ship is much closer. This is due to the fact that the minimum landing speed, set on one of our heuristics, is really low and The final thing we would like to note is that we have only included the results for EVO-P because T-HOLOP failed to land the ship using the same heuristic, and behaved badly. It's worth noting here that T-HOLOP requires rewards between certain bounds and this is not the case with our heuristic. For completeness purposes, we did change the UCB-like node selection policy of T-HOO to e-greedy, in which case we DID get the ship to behave reasonably, albeit the failure to land was evident here as well. This is important we think for a number of reasons. The first one is that it shows that different Heuristics might be needed for different algorithms. Although this might be self-evident, it shows that one must design heuristics with a specific algorithm in mind.

VI. CONCLUSION

We showed that it is doable to use evolution in a rolling horizon setting, making it possible to create agents using off-the-shelf evolutionary algorithms and a generative model. We used an experimental setup that included both simple benchmark problems and a popular arcade game, Lunar Lander. Unfortunately, it seems that different algorithms require different heuristics to work properly in the case that absorbing states are not met during rollouts. In future research we aim to improve MCTS-like algorithms for continuous settings by including a learning module, possibly in the form of actor-critic algorithms. This should automate learning the heuristics (after some ad-hoc played games) and help guide the rollouts towards more sensible solutions, even in the case where heuristics do not favour T-HOLOP.

REFERENCES

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlf-shagen, 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] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling horizon evolution versus tree search for navigation in single-player real-time games," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*. ACM, 2013, pp. 351–358.

[10] J. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[11] N. Hansen, "The CMA evolution strategy: a comparing review," in *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, J. Lozano, P. Larranaga, I. Inza, and Bengoetxea, Eds. Springer, 2006, pp. 75–102.

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

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

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.