Autonomous Agents and Multi-Agent Systems (2023) 37:26
https://doi.org/10.1007/s10458-022-09596-0

Monte Carlo tree search algorithms for risk-aware and multi-objective reinforcement learning

Conor F. Hayes[1] · **Mathieu Reymond**[2] · **Diederik M. Roijers**[2,3] · **Enda Howley**[1] · **Patrick Mannion**[1]

Accepted: 12 December 2022 / Published online: 28 April 2023

Abstract
In many risk-aware and multi-objective reinforcement learning settings, the utility of the user is derived from a single execution of a policy. In these settings, making decisions based on the average future returns is not suitable. For example, in a medical setting a patient may only have one opportunity to treat their illness. Making decisions using just the expected future returns–known in reinforcement learning as the value–cannot account for the potential range of adverse or positive outcomes a decision may have. Therefore, we should use the distribution over expected future returns differently to represent the critical information that the agent requires at decision time by taking both the future and accrued returns into consideration. In this paper, we propose two novel Monte Carlo tree search algorithms. Firstly, we present a Monte Carlo tree search algorithm that can compute policies for nonlinear utility functions (NLU-MCTS) by optimising the utility of the different possible returns attainable from individual policy executions, resulting in good policies for both risk-aware and multi-objective settings. Secondly, we propose a distributional Monte Carlo tree search algorithm (DMCTS) which extends NLU-MCTS. DMCTS computes an approximate posterior distribution over the utility of the returns, and utilises Thompson sampling during planning to compute policies in risk-aware and multi-objective settings. Both algorithms outperform the state-of-the-art in multi-objective reinforcement learning for the expected utility of the returns.

Keywords Multi-objective · Risk-aware · Decision making · Distributional · Reinforcement learning · Monte Carlo tree search

An earlier version of this work was presented as an extended abstract at the *International Conference of Autonomous Agents and Multi-Agent Systems* 2021 [27] and at the *Adaptive and Learning Agents Workshop* 2021 [28]. This article extends our previous work with additional theoretical analysis and new empirical results.

Conor F. Hayes
c.hayes13@nuigalway.ie

1 University of Galway, Galway, Ireland

2 Vrije Universiteit Brussel, Brussels, Belgium

3 City of Amsterdam, Amsterdam, The Netherlands

1 Introduction

In real-world decision making, a policy is often only executed once. For example, consider a government planning to build an off-shore wind farm to generate electricity. To ensure electricity generation is maximised, the wind farm must be located in an area with sufficient wind while also not interfering with any fishing routes or protected marine life. Given the off-shore wind farm will only be constructed once, the government must consider each potential outcome and likelihood of each outcome to ensure an optimal decision can be made.

In reinforcement learning (RL), the expected return is used to make decisions [71]. However, in many scenarios the utility of a user is derived from a single execution of a policy, and, therefore, the utility of the returns must be optimised [61]. For example, in a medical setting a patient may only have one opportunity to select a treatment. In this example, a patient will aim to cure their illness based on a single course of a treatment. As such, the user's utility is derived from the single execution of a policy. Moreover, computing a policy based on applying a utility function to expected return is incompatible with how the user's utility is derived because the expected return considers the average outcome over multiple policy executions. Therefore, the utility of the expectation is computed. In contrast, a policy that maximises utility of the return considers the utility obtained from each individual outcome, which is compatible with how the user's utility is derived. Therefore, the expected utility must be maximised (see Sect. 2.4).

When optimising for expected utility the underlying distribution of the returns must be used differently. Therefore, decisions must be made using the utility of the returns of a full policy. Under these conditions, an agent must be able to sample from the underlying return distribution to calculate the future returns. The agent must also be able to calculate the returns accrued at each timestep. Therefore, we theorise that for an agent to have sufficient critical information at decision time the agent must apply the utility function to the cumulative returns, which is the sum of the accrued and future returns [61].

To calculate the utility, we apply the utility function to the returns where a user's utility function is known a priori. In other words, in the taxonomy of multi-objective sequential decision making [62], we are in the known utility function scenario. When optimising under the expected utility, it is critical to only apply the utility function to the returns of a full execution of a policy [61] because nonlinear utility functions do not distribute across the sum of immediate and future returns [31, 61]. In this case, the agent must know the returns it has already accrued and the future returns before applying the utility function. For example, before the 2008 financial crash, many investment bankers were guaranteed their base salaries regardless of their losses, but their bonuses were dependent on their returns from investments. In the case of an investor incurring a loss, the only policy that would result in a bonus would be one that executes an increasingly risky strategy to win back the losses and receive some bonus.

Learning the utility of the returns is thus naturally risk-aware. Optimising the utility of the sum of the accrued and future returns to make decisions enables an agent to avoid certain undesirable outcomes. Without knowing the accrued returns, an agent cannot understand how future actions could affect the cumulative return. To make decisions that maximise the user's utility, the agent must have information about both the accrued and the future returns.

A further complicating factor is that, in the real world, decision making often involves trade-offs based on multiple conflicting objectives [17, 60, 75]. For example, we may want to maximise the power output of coal-burning electrical generators while minimising CO_2 emissions. Many approaches to multi-objective decision making only consider linear utility functions; this limitation severely restricts the real-world applicability of these methods [76], given that utility in many real-world problems is derived in a nonlinear manner.

In the multi-objective case, optimising under the expected utility is known as optimising the expected scalarised returns (ESR) criterion. For multi-objective reinforcement learning (MORL), the utility function expresses the user's preferences over objectives. If the utility function is linear and is known a priori, it is possible to translate a multi-objective decision problem to its single-objective equivalent. Once translated, we can then apply single objective methods to solve the decision problem. However, if the utility function is nonlinear, as human preferences often are, explicitly multi-objective methods are required to find optimal solutions [62]. The majority of MORL algorithms focus on the scalarised expected returns (SER) criterion. It has been shown that for nonlinear utility functions the policies learned under the ESR criterion and the SER criterion can be different [64]. Futhermore, nonlinear utility functions invalidate the Bellman equation, given nonlinear utility functions do not distribute across the sum of the immediate and future returns, which restricts the number of usable algorithms in this setting [31]. Therefore, to increase MORL's usability in the real world, dedicated multi-objective algorithms for the ESR criterion and the SER criterion that can learn policies for nonlinear utility functions must be formulated. We note that the MORL literature focuses almost exclusively on the SER criterion, leaving the ESR criterion largely understudied with a few exceptions [29, 30, 32, 33, 42, 61, 74].[1]

We propose a novel algorithm that can optimise for nonlinear utility functions for expected utility by taking both the accrued and future returns into consideration. To do so, we define a nonlinear utility function Monte Carlo tree search (NLU-MCTS) algorithm that performs Monte Carlo roll-outs to calculate the future returns while also calculating the accrued returns. Therefore, NLU-MCTS can make decisions using the expected utility of the returns over multiple policies. NLU-MCTS builds upon Monte Carlo tree search and uses UCB to explore during planning.

In sequential decision making settings, one of the fundamental challenges is the exploration versus exploration dilemma [71]. Thompson sampling is an algorithm that has been shown to address this dilemma in bandit settings [15]. Thompson sampling selects actions based on the probability matching principle, where actions are selected stochastically based on the probability of the action being optimal. Thompson sampling has been shown to empirically outperform UCB in bandit settings, and under a wide range of problems shows a more robust convergence compared to UCB [7]. Therefore, to exploit the potential performance gains of Thompson sampling we propose a new algorithm known as distributional Monte Carlo tree search (DMCTS) which computes an approximate posterior distribution over the expected utility over the returns. Using the computed approximate posterior distribution it is possible to use Thompson sampling methods to explore during planning.

Both NLU-MCTS and DMCTS overcome the issues present when making decisions solely with the expected return [11, 38, 53, 70, 80]. As we will show, computing the utility of the returns of a policy is useful when optimising for risk-aware RL and under the MORL ESR criterion, given utility of the returns of a policy contains more information about the range of potential negative and positive outcomes during planning and at decision time. NLU-MCTS achieves good performance in both risk-aware and multi-objective settings,

[1] We will make the distinction between the SER and ESR criterion clear in a later section.

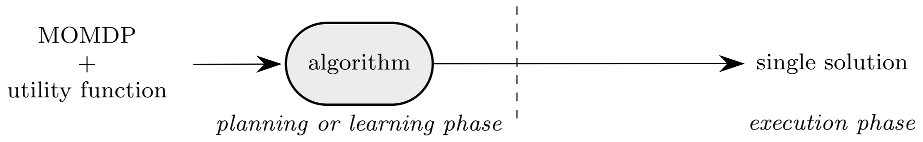

Fig. 1 The known utility function scenario [31]

while DMCTS achieves good performance in risk-aware settings and state-of-the-art performance under multi-objective ESR settings.

2 Background

In this section we introduce necessary background material, including multi-objective Markov decision processes, the known utility function scenario, commonly used optimality criteria in multi-objective decision making, risk-aware utility functions, Bootstrap Thompson Sampling, Expected Utility Policy Gradient and Monte Carlo tree search.

2.1 Multi-objective reinforcement learning

In multi-objective reinforcement learning (MORL), we deal with decision problems with multiple objectives [44, 62], often modelled as a multi-objective Markov decision process (MOMDP). A MOMDP is a tuple, $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma, \mathcal{R})$, where $\mathcal{S}$ and $\mathcal{A}$ are the state and action spaces, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is a probabilistic transition function, γ is a discount factor determining the relative importance of future rewards and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}^n$ is an n-dimensional vector-valued immediate reward function. In MORL, $n > 1$.

2.2 The known utility function scenario

In MORL, an agent seeks to maximise a user's utility function, where a user's utility function describes their preferences over objectives. In certain scenarios the utility function of a user can be known at the time of learning or planning. In the taxonomy of MORL we are deemed to be in the known utility function scenario [31, 62]. When the utility function of a user is known, a single optimal policy can be computed. Figure 1 describes the phases of the known utility function scenario. There are two phases in the known utility function scenario: the planning or learning phase and the execution phase. During the planning or learning phase a multi-objective reinforcement learning or planning algorithm is deployed in the MOMDP to compute a single optimal policy for the known utility function. A single optimal policy is computed once the algorithm has completed planning or learning. The computed policy is then executed during the execution phase.

2.3 Risk-aware utility functions

In single-objective decision making under uncertainty, utility functions are often utilised [79]. In scenarios where risk is considered, utility functions are often used to represent a user's preference for risk. In risk-aware settings the utility function is applied to the returns and the expected utility is maximised. When making decisions in scenarios with risk, a user can be described as risk-averse, risk-seeking or risk-neutral.

A user's preference for risk can be described by the shape of their utility function [5, 21]. The shape of a risk-seeking utility function is convex. For example the nonlinear utility function $u(x) = x^2$ is a risk-seeking utility function given its shape is convex [21, 41]. For a user that is risk-averse their utility function is concave. For example the nonlinear utility function $u(x) = x^{0.5}$ is risk-averse given the utility function has a concave shape from below [21, 36]. In contrast to risk-averse and risk-seeking utility functions, risk-neutral utility functions are linear [36]. A user who has a risk-neutral utility function has no preferences for risk and therefore the utility is a linear function of the returns. For example, $u(x) = x$ is a risk-neutral and linear utility function. In this paper we only consider nonlinear utility functions, therefore we focus on risk-seeking and risk-averse utility functions.

2.4 Scalarised expected returns versus expected scalarised returns

In MORL, the user's utility derives from the vector-valued outcomes (returns). This is typically modelled as a utility function that needs to be applied to these outcomes in one way or another. For this, we consider two choices [31, 62]. Calculating the expected value of the return of a policy before applying the utility function leads to the scalarised expected returns (SER) optimisation criterion:

$$V_u^\pi = u\left(\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathbf{r}_t \mid \pi, \mu_0\right]\right). \tag{1}$$

SER is the most commonly used criterion in the multi-objective (single agent) planning and reinforcement learning literature [62]. For SER, a coverage set is defined as a set of optimal policies for all possible utility functions.

In contrast to the SER criterion, if the utility function is applied before computing the expectation, then the expected scalarised returns (ESR) criterion is being optimised [61]:

$$V_u^\pi = \mathbb{E}\left[u\left(\sum_{t=0}^{\infty} \gamma^t \mathbf{r}_t\right) \mid \pi, \mu_0\right]. \tag{2}$$

Similar to risk-aware settings for single objectives (see Sect. 2.3), the ESR criterion maximises the expected utility. Therefore, the ESR criterion is naturally risk-aware while considering multiple objectives. ESR is the most commonly used criterion in the game theory literature on multi-objective games [59], with some exceptions (e.g. [64]).

2.5 Monte Carlo tree search

One way of approaching a decision problem is to use tree search. Perhaps the most popular of such methods is Monte Carlo tree search (MCTS) [16], which employs heuristic exploration to construct its search tree. MCTS builds a search tree of nodes, where each node has a number of children. Each child node corresponds to an action available to the agent. MCTS has two phases: the planning phase and the execution phase.

In the planning phase the agent implements the following four steps [11]: selection, expansion, simulation and backpropagation. **Selection:** the agent traverses the search tree until it reaches a node for which not all of its possible child nodes have been explored.

Expansion: at a node whose children have not all been expanded, the node must be expanded. The agent creates a random child node and then must simulate the environment for the newly created child node. **Simulation:** the agent executes a random policy through Monte Carlo simulations until a terminal state of the environment is reached. The agent then receives the returns. **Backpropagation:** the agent must backpropagate the returns received at a terminal state to each node visited during selection where a predefined algorithm statistic e.g. UCB [16, 38] is updated. Each step is repeated a specified number of times, which incrementally builds the search tree. Or, as we will discuss in the next subsection, a posterior belief on the returns, from which we can draw actions using Thompson sampling [8].

During the execution phase the agent must select a child node, corresponding to an action and associated state transition, to traverse to next. The agent evaluates the statistic at each node that is reachable from the root node and moves to the node which returns the maximum value. Once the execution phase has completed, the agent repeats the planning phase.

As already highlighted MCTS makes decisions and explores based on a predefined algorithm statistic. One such version of MCTS is UCT [38] which uses the following formula to derive the optimal action at decision time while also incorporating exploration during learning:

$$v_i + C \times \sqrt{\frac{ln(N)}{n_i}}, \tag{3}$$

where v_i is the approximated value of the node i, n_i is the number of the times the node i has been visited and N is the total number of times that the parent of node i has been visited. C is a hyperparameter that can be tuned for exploration, however C is often set to $\sqrt{2}$.

2.6 (Bootstrap) Thompson sampling

As previously mentioned, during the planning phase of MCTS, we can use Thompson sampling to take exploring actions [8]. However, it is not always possible to get an exact posterior. In this case a bootstrap distribution over means can be used to approximate a posterior distribution [20, 49]. Eckles et al. [18, 19] use a bootstrap distribution to replace the posterior distribution used in Thompson Sampling. This method is known as Bootstrap Thompson Sampling (BTS) [18] and was proposed in the multi-arm bandit setting. The bootstrap distribution contains a number of bootstrap replicates, $j \in \{1, \dots, J\}$, where J is a hyper-parameter that can be tuned for exploration. For a small J, BTS can become greedy. A larger J value increases exploration, but at a computational cost [18].

Each bootstrap replicate, j, in the bootstrap distribution contains two parameters, α_j and β_j, where $\frac{\alpha_j}{\beta_j}$ is an is an estimate of replicate j's expected utility. At decision time, to determine the optimal action the bootstrap distribution for each arm, i, is sampled. The observation for the corresponding bootstrap replicate, j, is retrieved and the arm with the maximum expected utility is pulled [18].

The distribution which corresponds to the maximum arm is randomly re-weighted by simulating a coin-flip (commonly known as sampling from a Bernoulli bandit) for each bootstrap replicate, j, in the bootstrap distribution (see Algorithm 1). If the coin-flip is heads, the α and β parameters for j is re-weighted.[2] To do so, the return is added to the α_j value and 1 is added to β_j [18].

Algorithm 1: Bootstrap Thompson Sampling Update

```
1 for $j \in J$ do
2     sample $d_j$ from Bernoulli(1/2)
3     if $d_j = 1$ then
4         $\alpha_j \leftarrow \alpha_j + u(\mathbf{r})$
5         $\beta_j \leftarrow \beta_j + 1$
6     end
7 end
```

Bootstrap methods with random re-weighting [65] are more computationally appealing as they can be conducted online rather than re-sampling data [52]. BTS addresses problems of scalability and robustness when compared to Thompson Sampling [18]. Furthermore, bootstrap distributions can approximate posteriors that are difficult to represent exactly.

2.7 Expected utility policy gradient

We now introduce Expected Utility Policy Gradient (EUPG) [61], a state-of-the-art MORL algorithm for ESR that we will use as a benchmark algorithm in our experiments. EUPG is an extension of Policy Gradient [72, 83], where Monte Carlo simulations are used to compute the returns and optimise the policy. EUPG calculates the accrued returns, $\mathbf{R}_t^-$, which is the sum of the immediate returns received as far as the current timestep, t. EUPG also calculates the future returns, $\mathbf{R}_t^+$, which is the sum of the immediate returns from the current timestep, t, to the terminal state. Using both the accrued and future returns enables EUPG to optimise over the utility of the full returns of an episode, where the utility function is applied to the sum of $\mathbf{R}_t^-$ and $\mathbf{R}_t^+$.

In policy gradient the policies are adapted towards the attained utility by gradient descent. For EUPG the utility of the sum of the accrued and future returns is calculated inside the loss function, which results in the following:

$$\mathcal{L}(\pi) = -\sum_{t=0}^{T} u(\mathbf{R}_t^- + \mathbf{R}_t^+) \log(\pi_\theta(a|s, \mathbf{R}_t^-, t)). \tag{4}$$

Roijers et al. [61] demonstrated for the ESR criterion the accrued and future returns must be considered when learning in order to learn a good policy. Applying this consideration to EUPG, the algorithm achieves the state-of-the-art performance under the ESR criterion. In this paper, we use the same method of adding past and future returns together before applying the utility inside of the search scheme of our novel DMCTS algorithm.

[2] Updating the distribution in this way is known as "double-or-nothing" or online half sampling [18]. It is important to note that the absolute scale of the weights does not matter for most estimators [18]. In the literature various other weight distributions have been used. For example, Rubin [65] uses a Bayesian bootstrap which uses exponential weights. While this overcomes some numerical problems, it requires updating all replicates and therefore can be more computationally expensive. For an extensive study on weight distributions for bootstrapping the sample mean see Owen and Eckles [51].

3 Expected scalarised returns

In the known utility function scenario, there are two phases: the planning phase and the execution phase. During the learning phase a policy is computed and returned to the user. After planning has completed the user executes the computed policy during the execution phase. In scenarios where the utility of a user is derived from the single execution of a policy, the expected scalarised returns (ESR) criterion must be optimised. Under the ESR criterion, the user will only execute the computed policy once in the execution phase.[3] The majority of RL research focuses on the SER criterion [54, 58], while the ESR criterion has been largely overlooked with some exceptions [32, 34, 42, 61, 74]. Additionally, the majority of RL research only considers linear utility functions. However, in the real world, utility functions can be nonlinear. A potential reason why the RL community has focused on linear utility function is that nonlinear utility functions invalidate the Bellman equation, given nonlinear utility functions do not distribute across the sum of the immediate and future returns [61],

$$\begin{aligned} &\max_{\pi} \mathbb{E}\left[u\left(\mathbf{R}_t^- + \sum_{i=t}^{\infty} \gamma^i \mathbf{r}_i\right) \middle| \pi, s_t\right] \\ &\neq u(\mathbf{R}_t^-) + \max_{\pi} \mathbb{E}\left[u\left(\sum_{i=t}^{\infty} \gamma^i \mathbf{r}_i\right) \middle| \pi, s_t\right], \end{aligned} \tag{5}$$

where u is a nonlinear utility function[4] and $\mathbf{R_t^-} = \sum_{i=0}^{t-1} \gamma^i \mathbf{r}_i$. It has also been shown that for nonlinear utility functions the policies computed under the ESR criterion and the SER criterion can be different [59]. Therefore, to enhance RL's usability in real-world problem domains, new methods must be formulated that can compute policies for the ESR criterion and the SER criterion for nonlinear utility functions.

When making decisions under the SER criterion, the expected returns is computed before the utility function is applied. A decision is then selected based on the scalar utility of the expectation [77]. SER methods learn policies that optimise a user's utility function over multiple policy executions. Therefore, a user will execute a policy computed under the SER criterion multiple times during the execution phase. Under the SER criterion, making decisions on an expected value vector is optimal [62]. However, a user optimising for the ESR criterion may only have one opportunity to execute a policy. Therefore, making decisions based on a single expected value vector is not sufficient because the user must have sufficient critical information available about each potential return vector and the associated likelihood [32, 33]. Therefore, applying the utility function to each return vector before computing the expectation ensures the user has taken into consideration each potential outcome a policy may have and the associated utility. We outline two new algorithms that compute policies under the ESR criterion in Sects. 4 and 5.

[3] It is important to note that in order to compute policies during the planning phase multiple policy executions must take place. However, it is important to remember that the computed policy may then be executed once (ESR) or multiple times (SER) during the execution phase.

[4] For nonlinear utility functions the accrued returns must be included in the state, s_t [61].

4 Monte Carlo tree search for nonlinear utility functions

To compute policies for the ESR criterion when the utility function is nonlinear and known a priori [31], we present a novel Monte Carlo tree search algorithm, known as NLU-MCTS. As shown by Roijers et al. [61], in order to compute optimal policies under the ESR criterion, both the accrued and future returns must be taken into consideration before applying the utility function. Therefore, an algorithm must either maintain a distribution over the returns or have some method which allows the agent to sample from the underlying return distribution of the environment. NLU-MCTS utilises the latter, by performing Monte Carlo simulations to compute the future returns. Usually in single objective MCTS an expectation of the returns is maintained at each chance node and the agent seeks to maximise the expectation. When the utility function is nonlinear, making decisions based on the expected returns does not account for the potential undesired outcomes a decision may have. For risk-aware RL and MORL under the ESR criterion, we need to be able to make decisions with sufficient information to avoid undesirable outcomes and exploit positive outcomes. Our key insight is that computing the utility of the cumulative returns, the returns received from executing a policy, can be used to replace the expected future returns (of vanilla MCTS) at each node. We outline our algorithm for single-objective risk-aware RL and MORL that can compute policies for the ESR criterion.

Before we outline how the accrued and future returns are computed, we must describe the structure of the search tree constructed by NLU-MCTS. Under the ESR criterion, the environment must be stochastic, where the state transitions or reward function are stochastic. To handle this uncertainty, NLU-MCTS builds an expectimax search tree using the same planning phase as MCTS (see Sect. 2.5). A search tree is a representation of the state-action space that is incrementally built via the steps of the underlying MCTS algorithm. An expectimax search tree [78] uses both decision and chance nodes. Figure 2 describes a search tree constructed by NLU-MCTS which contains both decision and chance nodes. Each decision node represents a state, action and reward of a MOMDP, where each decision node has a child chance node per action. In this paper we examine environments with stochastic rewards. Each chance node represents the state and action of a MOMDP. At each chance node, the environment is sampled. For NLU-MCTS, if a new observation-reward combination is generated when sampling the environment, a new child decision node is created. This process repeats as the agent traverses the search tree. It is important to note that each chance node and its parent decision node share the same state and action. A child decision node is only created when a new observation-reward combination is received when sampling the environment. To build and traverse a search tree similar to MCTS, NLU-MCTS uses the following phases: selection, expansion, simulation and backpropagation (Sect. 2.5).

Now that the structure of the underlying search tree has been outlined it is possible to describe how the cumulative returns and future returns are calculated. The accrued returns is the sum of returns the NLU-MCTS algorithm receives during the execution phase from timestep 0, t_0, to timestep, $t-1$, where $\mathbf{r}_t$ is the reward vector received at each timestep,

$$\mathbf{R}_t^- = \sum_{t_0}^{t-1} \mathbf{r}_t.$$

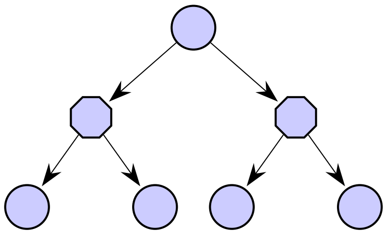
Fig. 2 A representation of a search tree constructed using NLU-MCTS for a problem with stochastic rewards and two actions. The search contains both decision nodes, represented by circular nodes, and chance nodes, represented by octagons

Given we utilise the underlying planning phases of Monte Carlo tree search, we can use the simulation phase to compute the future returns. As already mentioned during the simulation phase the agent performs a random rollout, also known as a Monte Carlo simulation, until a terminal state. NLU-MCTS utilises Monte Carlo simulations of the environment until a terminal state is reached. Therefore, the future returns can be computed from Monte Carlo simulations performed at each node during planning. Taking this into consideration the future returns, $\mathbf{R}_t^+$, is the sum of the rewards received when traversing the search tree during the planning phase and Monte Carlo simulations from timestep, t, to a terminal node, t_n,

$$\mathbf{R}_t^+ = \sum_{t}^{t_n} \mathbf{r}_t. \tag{6}$$

Finally, before the utility function is applied the cumulative returns must be calculated. The cumulative returns, $\mathbf{R}_t$, is the sum of the accrued returns, $\mathbf{R}_t^-$, and the future returns, $\mathbf{R}_t^+$,

$$\mathbf{R}_t = \mathbf{R}_t^- + \mathbf{R}_t^+. \tag{7}$$

In other words, the cumulative returns is the returns received from a full policy execution. Once the cumulative returns, $\mathbf{R}_t$, have been calculated, it is possible to compute the utility of the returns, $u(\mathbf{R}_t)$, to optimise for the ESR criterion.

As already highlighted, NLU-MCTS builds an expectimax search tree and utilises both decision and chance nodes. Over multiple iterations of the planning phase, NLU-MCTS constructs a search tree using the selection, expansion, simulation and backpropagation phases used by traditional MCTS [70]. We outline the NLU-MCTS algorithm in Algorithm 2.

Firstly, NLU-MCTS utilises the selection phase (Algorithm 3, see Fig. 3), where the agent traverses the search tree starting at the current root decision node [68]. During the selection phase, we utilise outcome selection for chance nodes and action selection for decision nodes. When the agent arrives at a chance node, we perform outcome selection where the agent simulates the environment model (Algorithm 5). The agent then moves to the child decision node corresponding to the observation-reward combination received from the simulation [68]. When the agent arrives at a decision node, n_d, the agent must decide which of its child chance nodes, C_{n_d}, to select. Therefore, NLU-MCTS selects the chance node, n_c, which maximises the UCB term:

$$\text{BestChild} = arg\ max_{n_c \in C_{n_d}} \text{UCB}(n_d, n_c) \tag{8}$$

the UCB term is defined as follows:

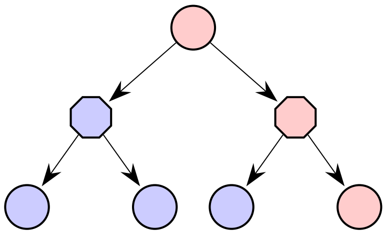
Fig. 3 During the selection phase, NLU-MCTS starts at the root node and traverses down the search tree (nodes highlighted in red). The agent traverses the search tree until a leaf decision node is found

$$\mathrm{UCB}(n_d, n_c) = \frac{v_{n_c}}{N_{n_c}} + C \times \sqrt{\frac{ln(N_{n_d})}{N_{n_c}}}, \tag{9}$$

where v_{n_c} is the total utility of the child node n_c, N_{n_c} is the number of times child node n_c has been visited, $\frac{v_{n_c}}{N_{n_c}}$ is the expected utility of the child node n_c, C is an exploration value, and N_{n_d} and N_{n_c} are the number of times n_d and n_c have been visited respectively. Equation 9 ensures that the agent explores areas of the tree which have not been visited often while also ensuring that the agent exploits nodes which have good returns. The agent then traverses to the chance node corresponding to the best action. The agent continues to traverse the search tree until a decision node is encountered which has not had all of is children expanded. The agent then progresses to the expansion phase (Algorithm 4) where the selected decision node is utilised. It is important to note that, as the agent traverses the search tree, the future returns, $\mathbf{R}_t^+$, is being computed incrementally.

During the expansion phase (Algorithm 4, see Fig. 4), the agent considers a decision node selected during the previous phase which has not had all of its children expanded. There are three steps to the expansion phase. Firstly, for the decision node, a child chance node corresponding to a previous remaining action is created for a randomly selected action. Secondly, the agent simulates the environment model for the newly created chance node. Finally, for the previously created chance node, the agent creates a child decision node corresponding to the observation-reward combination received. It is important to note that both a chance node and a decision node are generated during the expansion phase. The newly created decision node is then utilised in the next phase, known as the simulation phase.

After expansion, the created decision node must be simulated. Figure 5 highlights the simulation phase (Algorithm 6) for NLU-MCTS. When a decision node is simulated, a random rollout is executed. During the rollout, a random policy is followed until a terminal state is reached. Once the simulation has completed, the cumulative returns, $\mathbf{R}_t$, can be computed. The future returns, $\mathbf{R}_t^+$, is equal to the sum of the rewards received when traversing the search tree and the returns from the random rollout in the simulation phase. The cumulative returns, $\mathbf{R}_t$, is then computed by adding both the accrued returns, $\mathbf{R}_t^-$, and the future returns, $\mathbf{R}_t^+$. We note that $\mathbf{R}_t$ is the same for every node during backpropagation.

Figure 6 and Algorithm 7 outlines the backpropagation phase of NLU-MCTS. Once the simulation phase has completed, the cumulative returns, $\mathbf{R}_t$, is backpropagated to each node visited during the previous phases of the search tree. As the agent backpropagates the cumulative returns, the agent updates the required statistic for each node.

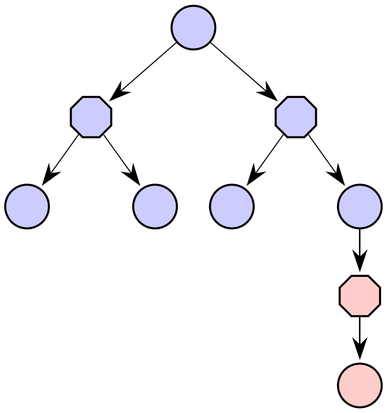

Fig. 4 During the expansion phase of NLU-MCTS (nodes highlighted in red), a child chance node is created. The newly generated chance node simulates the environment and creates a child decision for the corresponding reward received

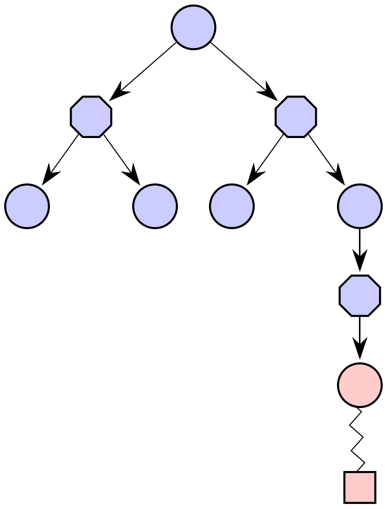

Fig. 5 During the simulation phase of NLU-MCTS (nodes highlighted in red), the decision node generated in the expansion phase executes a random policy until a terminal state. Finally, the cumulative returns $\mathbf{R}_t$ is computed

Under the ESR criterion, the utility of the cumulative returns, u($\mathbf{R}_t$), is computed during the backpropagation phase[5] by applying the known utility function, u, to the cumulative returns, $\mathbf{R}_t$. Therefore during backpropagation, the statistics at chance node are updated by updating the total utility, v, of the node as follows:

$$v_{n_c} \leftarrow v_{n_c} + u(\mathbf{R}_t). \tag{10}$$

The visit count for both chance node and decision nodes is also updated as follows:

$$N_{n_c} \leftarrow N_{n_c} + 1, \tag{11}$$

[5] To compute policies under the ESR criterion it is also possible to backpropagate the utility of the cumulative returns, u($\mathbf{R}_t$). The relevant statistics can then be updated using the utility of the cumulative returns.

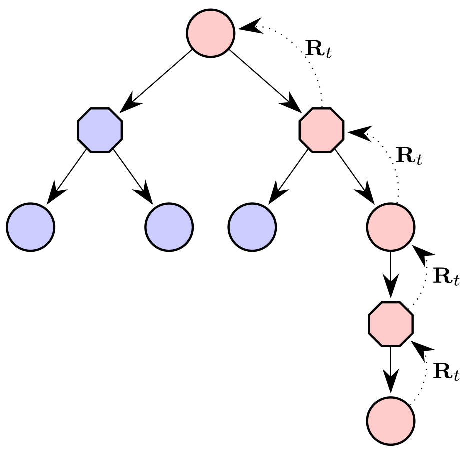

Fig. 6 During the backpropagation phase, the cumulative returns, $\mathbf{R}_t$, is backpropagated to each node visited during the planning phase

$$N_{n_d} \leftarrow N_{n_d} + 1. \tag{12}$$

The NLU-MCTS algorithm runs each step of the planning phase (selection, expansion, simulation and backpropagation) a specified number of times. We denote the number of times the planning phase is run as n_{exec}. Once the NLU-MCTS algorithm has run the planning phase an n_{exec} number of times, the algorithm returns the best action to take from the current root node, n_r. Under the ESR criterion, the best action, a^*, can be calculated by evaluating the expected utility of each of the current root nodes, n_r, children, C_{n_r} and taking the action which returns the maximum expected utility as follows:

$$a^* = \arg\max_{n \in C_{n_r}} \frac{v_n}{N_n}. \tag{13}$$

Algorithm 2: Monte Carlo Tree Search for Nonlinear Utility Functions

```
1  **Input** : $\mathrm{N}_{root}$ ← Root node; $\mathbf{R}_t^-$ ← Accrued returns
2  **Output**: Action $a$
3  **while** *Not out of computation* **do**
4      N ← $N_{root}$
5      $\mathbf{R}_t^+$ ← Future returns with 0 value entry per objective
6      N, $\mathbf{R}_t^+$ ← **Selection**(N, $\mathbf{R}_t^+$)
7      $\mathbf{R}_t^+$ ← **Simulation**(N, $\mathbf{R}_t^+$)
8      $\mathbf{R}_t \leftarrow \mathbf{R}_t^+ + \mathbf{R}_t^-$
9      **Backpropagate**(N, $\mathbf{R}_t$)
10 **end**
11 bestAction ← **calculateBestAction**($\mathrm{N}_{root}$) (Equation 13)
12 **return** bestAction
```

Algorithm 3: Selection

```
1  Input: N ← Node in the tree; $\mathbf{R}_t^+$ ← Future returns
2  if N is terminal then
3      return N, $\mathbf{R}_t^+$
4  end
5  if N is a chance node then
6      N, $\mathbf{R}_t^+$ ← Sample(N, $\mathbf{R}_t^+$)
7  end
8  if N has children to expand then
9      N*, $\mathbf{R}_t^+$ ← Expansion(N, $\mathbf{R}_t^+$)
10     return N*, $\mathbf{R}_t^+$
11 end
12 N ← BestChild (Equation 8)
13 Selection(N, $\mathbf{R}_t^+$)
```

Algorithm 4: Expansion

```
1  Input : N ← Node in the tree; $\mathbf{R}_t^+$ ← Future returns
2  N* ← a new child chance node for a remaining action
3  add N* to N's children
4  N*, $\mathbf{R}_t^+$ ← Sample(N*, $\mathbf{R}_t^+$)
5  return N*, $\mathbf{R}_t^+$
```

Algorithm 5: Sample

```
1  Input: N ← Chance Node; $\mathbf{R}_t^+$ ← Future returns
2  observation, reward ← simulate agent environment for N
3  $\mathbf{R}_t^+ \leftarrow \mathbf{R}_t^+ + \text{reward}$
4  for DN in N.Children do
5      if (DN.observation, DN.reward) = (observation, reward) then
6          return DN, $\mathbf{R}_t^+$
7      end
8  end
9  DN ← N create child decision node (observation, reward)
10 return DN, $\mathbf{R}_t^+$
```

Algorithm 6: Simulation

```
Input : N ← Node in the tree; s ← Node.state; $\mathbf{R}_t^+$ ←
    Future returns
while s is not terminal do
    a ← random action
    s, $\mathbf{r}_t$ ← env(s, a)
    $\mathbf{R}_t^+ \leftarrow \mathbf{R}_t^+ + \mathbf{r}_t$
end
return $\mathbf{R}_t^+$
```

Algorithm 7: Backpropagate

```
Input : N ← Node in the tree; $\mathbf{R}_t$ ← Cumulative returns
while N is not null do
    UpdateStatistics(N, $\mathbf{R}_t$) (Equations 10, 11 and 12)
    N ← N.parent
end
```

5 Distributional Monte Carlo tree search

Monte Carlo tree search for nonlinear utility functions (NLU-MCTS) utilises the UCB statistic to explore during planning. However, Thompson sampling methods have been shown to outperform UCB methods in bandit settings [15, 66]. Therefore, to exploit the potential performance increases associated with Thompson sampling methods, we present a novel distributional Monte Carlo tree search algorithm (DMCTS) that learns a posterior distribution over the expected utility of the returns.

Algorithm 8: Distributional Monte Carlo Tree Search

```
Input : $N_{root}$ ← Root node; $\mathbf{R}_t^-$ ← Accrued returns
Output: Action a
while Not out of computation do
    N ← $\mathrm{N}_{root}$
    $\mathbf{R}_t^+$ ← Future returns with 0 value entry per objective
    N, $\mathbf{R}_t^+$ ← Selection(N, $\mathbf{R}_t^+$)
    $\mathbf{R}_t^+$ ← Simulation(N, $\mathbf{R}_t^+$)
    $\mathbf{R}_t \leftarrow \mathbf{R}_t^+ + \mathbf{R}_t^-$
    Backpropagate(N, $\mathbf{R}_t$)
end
bestAction ← calculateBestAction($\mathrm{N}_{root}$)
return bestAction
```

Algorithm 9: Selection

```
1  Input: N ← Node in the tree; $\mathbf{R}_t^+$ ← Future returns
2  if N is terminal then
3      return N, $\mathbf{R}_t^+$
4  end
5  if N is a chance node then
6      N, $\mathbf{R}_t^+$ ← Sample(N, $\mathbf{R}_t^+$)
7  end
8  if N has children to expand then
9      N*, $\mathbf{R}_t^+$ ← Expansion(N, $\mathbf{R}_t^+$)
10     return N*, $\mathbf{R}_t^+$
11 end
12 N ← ThompsonSampling
13 Selection(N, $\mathbf{R}_t^+$)
```

Firstly, it is important to discuss how DMCTS (Algorithm 8) builds an underlying search tree. DMCTS builds an expectimax search tree using the same planning phase as NLU-MCTS (see Sect. 4). However, DMCTS takes a distributional approach to decision making.

DMCTS aims to maintain a posterior distribution over the expected utility of the returns at each chance node. However, because the utility function may be nonlinear, a parametric form of the posterior distribution may not exist. Since a bootstrap distribution can be used to approximate a posterior [20, 49], it is much more suitable to maintain a bootstrap distribution over the expected utility of the returns at each chance node.

Each bootstrap distribution contains a number of bootstrap replicates, $j \in \{1, \dots, J\}$ [18] (see Sect. 2.6). It is important to note the number of bootstrap replicates, J, is a hyperparameter that can be tuned for exploration [18]. Each bootstrap replicate, j, in the bootstrap distribution has two parameters, α_j[6] and β_j, where $\frac{\alpha_j}{\beta_j}$ is the expected utility for replicate j. On initialisation of a new node, for each bootstrap replicate, j, the parameters α_j and β_j are both set to 1. Moreover, α_j can be set to positive or negative values to increase initial exploration without a computational cost. Figure 7 outlines a bootstrap distribution learned by the DMCTS algorithm. For ESR settings, the expected utility of each bootstrap replicate, j, can be computed as follows:

$$\mathbb{E}(u(j)) = \frac{\alpha_j}{\beta_j}. \tag{14}$$

It is important to note that, similarly to NLU-MCTS, DMCTS requires the utility function of the user to be known a priori. The bootstrap distribution is updated during the back-propagation phase of the DMCTS algorithm.

[6] In this work our use of α differs slightly from that of Kaptein and Eckles [18]. We utilise α to track the sum of the utility, which can then be utilised to compute the expectation. Whereas, Kaptein and Eckles utilise α as a count for the returns of a Bernoulli bandit.

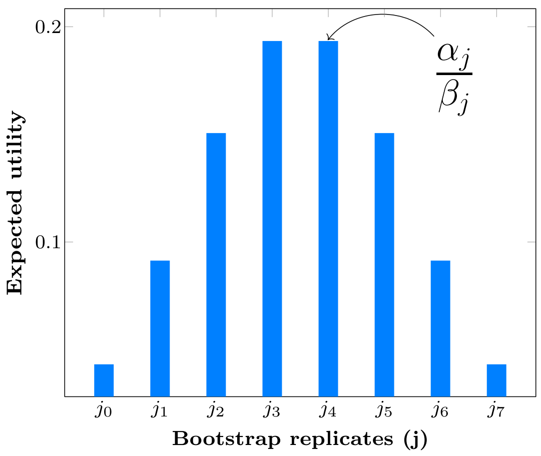

Fig. 7 A bootstrap distribution learned by DMCTS with the number of bootstrap replicates, J, set to 8. The expected utility for each bootstrap replicate, j, can be calculated by $\frac{\alpha_j}{\beta_j}$. For example, the expected utility for bootstrap replicate j_4 can be calculated as follows: $\mathbb{E}(u(j_4)) = \frac{\alpha_{j_4}}{\beta_{j_4}}$

Algorithm 10: Backpropagate

```
1 Input : N ← Node in the tree
2 Input : $\mathbf{R}_t$ ← Cumulative returns
3 while N is not null do
4     UpdateDistribution(N, $\mathbf{R}_t$)
5     N ← N.parent
6 end
```

During the backpropagation phase (Algorithm 10) the cumulative returns is backpropagated and the bootstrap distribution at each chance node is updated. Algorithm 11 outlines how a bootstrap distribution for a node is updated for the ESR criterion. In this paper, we do not use discounting as we perform evaluations only on finite horizon tasks. We note that DMCTS can easily be adapted to discounted settings. At chance node, i, for each bootstrap replicate, j, a coin flip is simulated (See Algorithm 11, Line 4). If the result of the coin flip is equal to 1 (heads), α_{ij} and β_{ij} are updated:

$$\alpha_{ij} \leftarrow \alpha_{ij} + u(\mathbf{R}_t)$$
$$\beta_{ij} \leftarrow \beta_{ij} + 1$$

To select actions while planning (Algorithm 9), we use the previously computed statistics. At each timestep the agent must choose which action to execute in order to traverse the search tree (as outlined in Algorithm 12). At decision node n, we select an action by sampling the bootstrap distribution at each child chance node, i. For each sampled bootstrap replicate, j, the α_{ij} and β_{ij} values are retrieved and $\frac{\alpha_{ij}}{\beta_{ij}}$ is computed. Since the following approximation is true,

$$\frac{\alpha_{ij}}{\beta_{ij}} \equiv \mathbb{E}[u(\mathbf{R}_t^- + \mathbf{R}_t^+)], \tag{15}$$

by maximising over i in Eq. 15, we select an action corresponding to j approximately proportional to the probability of that action being optimal–per the Bootstrap Thompson

Sampling exploration strategy. The agent then executes the action, a^*, which corresponds to the following:

$$a^* = \arg\max_i \frac{\alpha_{ij}}{\beta_{ij}}. \tag{16}$$

We note that, at execution time, we can calculate the best action (Algorithm 8, Line 11) by simply selecting the overall maximising action by averaging over all the acquired data, thereby maximising the ESR criterion:

$$ESR = \mathbb{E}[u(\mathbf{R}_t^- + \mathbf{R}_t^+)]. \tag{17}$$

Algorithm 11: UpdateDistribution

```
1 Input: i ← Node in the tree; $\mathbf{R}_t$ ← Cumulative Returns
2 J ← node.bootstrapDistribution
3 for j, ..., J bootstrap replicates do
4     Sample $d_j$ from Bernoulli($\frac{1}{2}$)
5     if $d_j$ = 1 then
6         $\alpha_{ij} \leftarrow \alpha_{ij} + u(\mathbf{R}_t)$
7         $\beta_{ij} \leftarrow \beta_{ij} + 1$
8     end
9 end
```

Algorithm 12: ThompsonSample

```
1 Input: n ← Node in the tree
2 Require: $\alpha$, $\beta$ prior parameters
3 $\alpha_{ij} := \alpha$, $\beta_{ij} := \beta$ {For each n child, $i$, and each bootstrap replicate, $j$ }
4 for i in n.children do
5     Sample $j$ from uniform 1, ..., $J$ bootstrap replicates
6     Retrieve $\alpha_{ij}$, $\beta_{ij}$
7 end
8 maxChild = $\arg\max_i \frac{\alpha_{ij}}{\beta_{ij}}$
9 return maxChild or maxChild.action
```

Using the outlined algorithm, DMCTS is able to learn policies for risk-aware settings and under ESR for multi-objective settings. In Sect. 6, we evaluate DMCTS for risk-aware settings and multi-objective settings for the ESR criterion.

6 Experiments

In order to evaluate NLU-MCTS and DMCTS, we test both algorithms in multiple settings. Firstly, we perform an ablative study to outline the effect on computation and performance the J parameter has when computing the BTS distribution for DMCTS. We then evaluate NLU-MCTS and DMCTS in a risk-aware setting. Finally, we evaluate both algorithms in multi-objective settings under the ESR criterion. In multi-objective settings, we test our algorithms on variants of standard benchmark problems from the MORL literature.

We also evaluate NLU-MCTS and DMCTS against two other state-of-the-art RL algorithms: Expected Utility Policy Gradient (EUPG) [61] and C51 [10]. EUPG is the only MORL algorithm that can compute policies under the ESR criterion and is therefore the state-of-the-art performance in this setting [61]. We use C51 as a baseline algorithm for our evaluation of DMCTS given C51 is a distributional RL algorithm and has achieved state-of-the-art performance [10].

At each timestep for NLU-MCTS and DMCTS, the planning phase is performed multiple times before an action is selected during the execution phase. To fairly evaluate all other algorithms against NLU-MCTS and DMCTS, we have altered each benchmark algorithm to have the same number of policy executions of each environment at each timestep as NLU-MCTS and DMCTS. At each timestep, each algorithm gets n_{exec} full policy executions worth of learning from that state and timestep onward. Therefore, if $n_{exec} = 10$, NLU-MCTS and DMCTS perform the planning phase ten times before selecting an action. To ensure a C51 and EUPG get the same opportunity to learn, both algorithms are altered to execute a policy n_{exec} number of times from the current state. For the other algorithms (except NLU-MCTS and DMCTS) this has the effect of increasing the learning speed. The number of policy executions n_{exec} varies for each problem domain. All experiments are averaged over 10 runs.

6.1 Ablation study

Before we evaluate both NLU-MCTS and DMCTS in risk-aware and multi-objective sequential decision making problems, we empirically evaluate how the Bootstrap Thompson Sampling (BTS) parameter settings affect performance and run time. We also provide a visualisation that shows how a BTS distribution is updated over time to estimate the underlying posterior distribution over the expected utility. Finally, we evaluate the performance of DMCTS under different J values in a MOMDP, to highlight how the selection of the J value can effect performance in sequential settings.

6.1.1 Bootstrap Thompson sampling *J* values and runtime

To illustrate how a BTS distribution evolves over time, we update a single BTS distribution based on the returns of a simple multi-objective bandit. In this setting the bandit has one arm, where there is a 0.5 chance of receiving the following return: $\mathbf{r} = [1, 1]$, and a 0.5 chance of receiving the following return: $\mathbf{r} = [0, 0]$. The returns are then scalarised using the following utility function:

$$u = r_1 r_2, \tag{18}$$

where r_1 and r_2 are the returns for objective 1 and objective 2 respectively. In this example, expected utility is 0.5.

Using this bandit we update a single BTS distribution and show how the distribution evolves over a number of updates using the utility of the returns. Figure 8 outlines how a BTS distribution with 25 bootstrap replicates evolves after 1, 8, 32, 128, 250 and 500 updates.

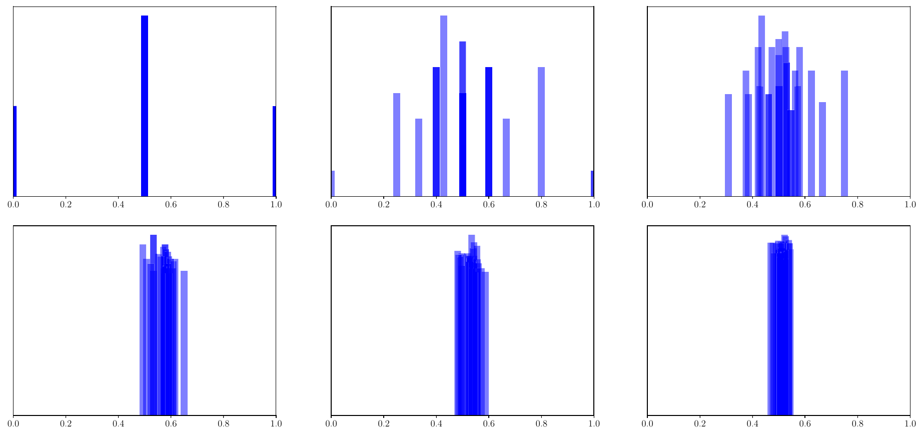

Fig. 8 A BTS distribution after 1, 8, 32, 128, 250 and 500 updates. After 500 updates the distribution converges to the correct expected utility, where expected utility is on the x-axis

Next, we investigate the computational run time for a BTS distribution with varying number of replicates, J. To evaluate the run time for each chosen J value we compute the time in seconds taken to perform 1, 000 updates of a BTS distribution. This experiment was performed 10 times for each J value and the average run time was computed. To evaluate the run time we use the following J values: 10, 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1000 and present the results in Fig. 9.

Figure 9 shows that the run time in seconds increases linearly with the number of replicates J. Therefore, the hyperparameter J can have an impact on the run time of the algorithm and therefore should be taken into consideration in order to optimise performance. Next we will evaluate the performance of a BTS distribution for multiple J values in a multi-objective multi-armed bandit setting. By comparing run time and performance it should be possible to determine which J values can be selected for good performance and efficiency.

6.1.2 Bootstrap Thompson sampling *J* values & performance

To investigate the effect the hyperparameter J on the performance of DMCTS we consider a multi-objective multi-armed bandit (MOMAB) setting where a BTS distribution is utilised per arm to determine which arm is optimal for a given utility function.

We utilise a MOMAB setting from the literature [63], with 5 arms, and each of the following ground truth mean vectors: (0, 0.8), (0.4, 0.4), (0.8, 0.0) and (0.9, 0.1). Each reward distribution is multi-variate Gaussian with correlations 0 and in-objective variance 0.0005 [63]. We utilise the following utility function:

$$u = 6.25\ max(r_0, 0)\ max(r_1, 0). \tag{19}$$

In this setting, the arm with mean vector (0.4, 0.4) is optimal and returns an expected utility of 1. We run BTS for 10, 000 trials for the following J values: 10, 100, 500 and 1000.

Figure 10 presents the results for each J value in the MOMAB setting. In this case it is clear that the choice of J value has little impact on the algorithms ability to compute the optimal utility. Therefore, given the computational results presented in Fig. 9 a lower J value may be preferred.

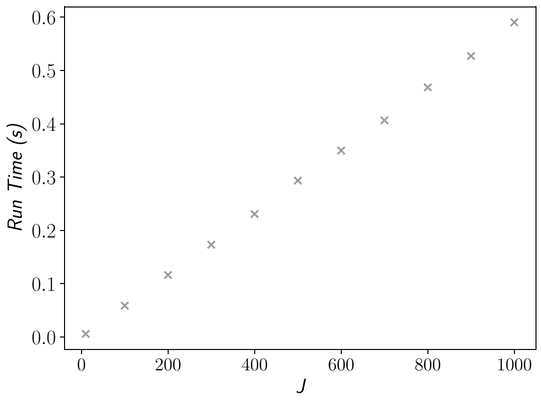

Fig. 9 The run time is seconds required to complete 1000 updates of a BTS distribution for different J values. The run time required increases linearly with the increase in the J value

Although the results presented in Fig. 10 show that the choice of J has little impact on the BTS distributions, it has been shown by Eckles and Kaptein [18] that J values lower than 100 lead to higher levels of regret in single-objective bandit settings. Therefore, we aim to utilise a J value of around 100 for DMCTS given the run time for J values of around 100 are relatively low. Such values also provide good performance while avoiding the limitations highlighted by Eckles and Kaptein [18]. However, we acknowledge that the J value is problem dependent and certain problems may require a higher J.

6.1.3 Bootstrap Thompson sampling *J* values in MOMDPs

To evaluate the selection of the J parameter for the BTS distribution has on the performance of DMCTS, we run DMCTS using different J values in a MOMDP. To do so, we have utilised a random MOMDP from the literature [61]. The random MOMDP is configurable based on the requirements of the experiments, where the numbers of states, actions, objectives, timesteps, and possible successor states can be determined a priori. The random MOMDP can then be initialised for each experiment by selecting a consistant seed. We generate a random MOMDP with 20 states, 2 actions, and 2 objectives. The transition function $T(s, a, s')$ is generated using $N = 8$ possible successor states per action, with random probabilities drawn from a uniform distribution [61]. We use the following nonlinear utility function:

$$u = r_1^2 + r_2^2. \tag{20}$$

We evaluate DMCTS using the following J values of 1, 2, 10, 100, 500 and 1000 for the BTS distributions. Figure 11 outlines the results from the random MOMDP. Utilising a J value of 1 has an impact on performance, given DMCTS with J set to 1 achieves a lower utility compared to the other parameter settings. As we increase the J value to 2 we can see that performance begins to improve. However, for a very low J value ($J = 1$ or $J = 2$) DMCTS will select actions greedily and will not explore the environment enough to obtain a good utility. As we increase the J value we can see that the performance increase. Once the J value is set to 10, DMCTS has a large increase in performance. Similarly, once the J value increases to 100 we can see even better performance. However, we do not see any more performance increases for DMCTS in the random MOMDP when we set the J value to a higher value. When the J value is set to 500 or 1000 the performance does not increase

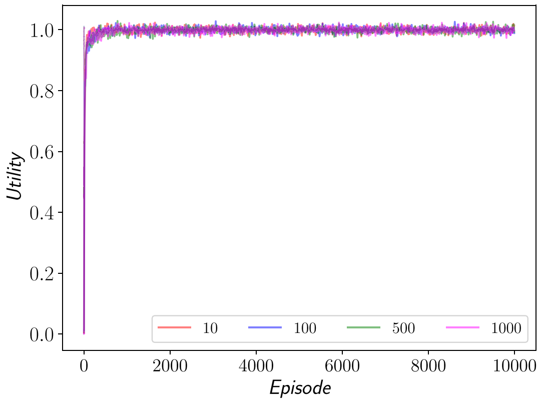

Fig. 10 The performance of BTS algorithm is a multi-objective multi-armed bandit setting for different J values. For each J value the algorithm converges to the optimal utility of 1

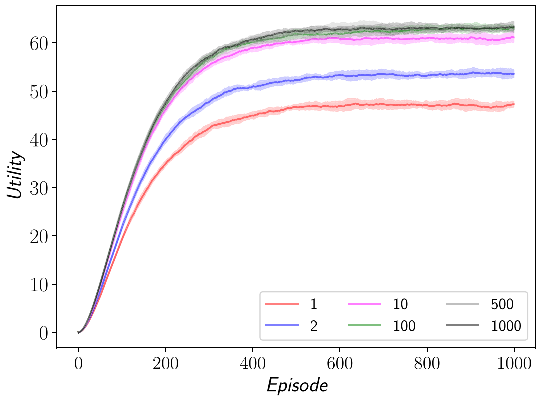

Fig. 11 Evaluation of different J values in a random MOMDP with 20 states, 2 actions, 2 objectives and 8 successor states reachable from each state

relative to $J = 100$. However, the computational cost of updates the BTS for higher J values increases. Therefore, it is important to ensure that the J value is set sufficiently high for exploration, while also avoiding J values with a high computational cost. Therefore, it may be important to tune the J value depending on the evaluation setting.

6.2 Risk-Aware MDP

Before testing NLU-MCTS and DMCTS on benchmark problems from the MORL literature, we evaluate both algorithms in a risk-aware problem domain under the ESR criterion. Shen et al. [69] define a Risk-Aware MDP where an agent must decide from a number of stocks in which to invest. The underlying MDP which has 4 actions (each action is a monetary amount, in Euros, of investment) and 7 states. At each timestep the agent must select a monetary amount to invest in the stock for a given state. We can invest €0, €1, €2 or €3 in a stock at each timestep. Each stock has a probability of making a profit and a probability of making a loss where the agent's return is the action multiplied by the stock price. All remaining implementation details can be found in the work of Shen et al. [69]. In risk-aware decision making a user can be risk seeking, risk averse or risk neutral. A user's preference for risk is described by their individual utility function, which can often be nonlinear. Given risk-based decision making scenarios are ubiquitous in the real world [23] it is important that algorithms can compute policies for risk-aware nonlinear utility functions. Therefore, to highlight the usability of NLU-MCTS and DMCTS in risk-aware decision making scenarios, we evaluate NLU-MCTS and DMCTS using the outlined Risk-Aware MDP. For the Risk-Aware MDP we use a nonlinear risk-seeking and risk-averse utility functions to evaluate the performance of NLU-MCTS and DMCTS.

6.2.1 Risk-seeking utility function

Firstly, we evaluate DMCTS and NLU-MCTS in the Risk-Aware MDP using the follow risk-seeking utility function:

$$u(x) = (max(0, x))^2. \tag{21}$$

In the Risk-Aware MDP, utilising the risk-seeking utility function presented in Sect. 2.3, would reward the agent with a positive utility when the returns are negative. Therefore, we have used the utility function in Eq. 21 to ensure that negative returns are not seen as a positive outcome by the agent. The nonlinear utility function outlined in Eq. 21 is risk-seeking given the shape of the utility function is convex.

For all experiments in the Risk-Aware MDP with the risk-seeking utility function, the parameter n_{exec} is set to 10 for each algorithm and each experiment lasts for 1000 episodes. For DMCTS we set the number of bootstrap replicates, J, for the bootstrap distribution as follows: $J = 500$. For NLU-MCTS we set $C = \sqrt{2}$.

Figure 12 describes the experimental results for the risk-seeking utility function in the Risk-Aware MDP. While both algorithms learn good stable policies, DMCTS achieves a higher utility when compared to NLU-MCTS for the risk-seeking utility function.

6.2.2 Risk-averse utility function

Secondly, we evaluate DMCTS and NLU-MCTS using the following risk-averse utility function:

$$u(x) = x^{\frac{1}{2}}. \tag{22}$$

The utility function in Eq. 22 is risk-averse given the shape of the utility function is concave.

For all experiments in the Risk-Aware MDP with the risk-averse utility function, the parameter n_{exec} is set to 10 for each algorithm and each experiment lasts for 1000 episodes. For DMCTS we set the number of bootstrap replicates, J, for the bootstrap distribution as follows: $J = 500$. For NLU-MCTS we set $C = \sqrt{2}$. In the Risk-Aware MDP the returns can be negative. Therefore, for the risk-averse utility function, we add 150 the returns because this is the minimum returns that the agent can achieve.

Figure 13 shows that both NLU-MCTS and DMCTS can compute good policies for the risk-averse utility function. DMCTS achieves a higher utility when compared to NLU-MCTS.

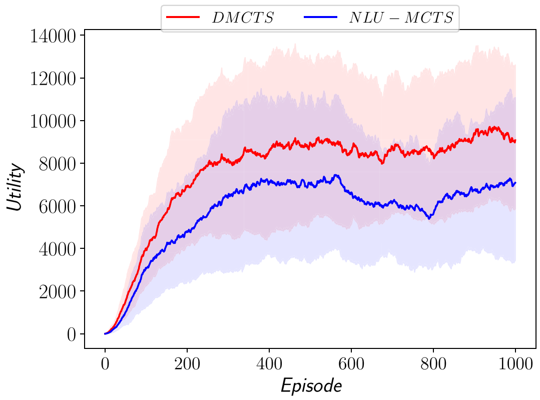

Fig. 12 Results from the Risk-Aware MDP environment where DMCTS is evaluated against NLU-MCTS using a risk-seeking utility function. DMCTS achieves a higher utility compared to NLU-MCTS for a risk-seeking utility function

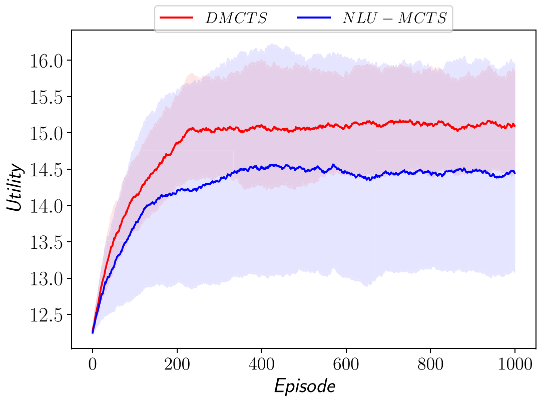

Fig. 13 Results from the Risk-Aware MDP environment where DMCTS is evaluated against NLU-MCTS using a risk-averse utility function. DMCTS achieves a higher utility compared to NLU-MCTS for a risk-averse utility function

6.2.3 Discussion of experimental results for risk-aware MDP

Both NLU-MCTS and DMCTS learn good policies for the risk-seeking and risk-averse utility functions. However, DMCTS achieves a higher utility in both settings. It is important to note that both the risk-seeking and risk-averse utility function are nonlinear. Therefore for any algorithm to compute good policies the cumulative returns must be taken into consideration. Both NLU-MCTS and DMCTS compute policies based on the expected utility by computing the cumulative returns. If the cumulative returns were not taken into consideration we would expect the utility obtained by both algorithms to be lower, given that nonlinear utility functions do not distribute across the sum of the future and immediate returns. Therefore, for risk-aware settings it is important that the cumulative returns are calculated before the utility function is applied. Taking this approach ensures that an agent can make decisions with knowledge of how future outcomes may affect utility. This is important in risk-aware settings given an agent may only have one opportunity to execute a policy, and having access to the expected utility of a policy ensures sufficient information is available to the agent so utility can effectively be optimised.

While both NLU-MCTS and DMCTS learn good polices for both risk-aware utility functions, DMCTS achieves a higher utility. The key difference between NLU-MCTS and DMCTS is how each algorithm explores during planning. NLU-MCTS utilises UCB while DMCTS uses the Thompson sampling method, Bootstrap Thompson sampling. In the bandit literature Thompson sampling methods have been shown to empirically outperform UCB [15]. Thompson sampling selects actions proportional to the probability of an action being optimal [67]. Therefore, by maintaining an approximate posterior distribution via a bootstrap distribution, and using Thompson sampling to sample from each approximate posterior distribution at each chance node to select actions, DMCTS can exploit the performance gains of Thompson sampling to achieve a higher utility than NLU-MCTS.

6.3 Multi-objective MDPs

To evaluate NLU-MCTS and DMCTS in multi-objective settings under the ESR criterion, we use a number of problem domains. Firstly, we evaluate NLU-MCTS and DMCTS in the Fishwood problem [61], given this is one of the very few domains for which ESR results have been published. Secondly, we evaluate NLU-MCTS and DMCTS in the Renewable Energy Dynamic Economic Emissions Dispatch (REDEED) problem domain[7].

6.3.1 Fishwood

Fishwood is a multi-objective benchmark problem proposed by Roijers et al. [61]. In Fishwood the agent has two states: in the woods or at the river. The goal of the agent is to catch fish and collect wood. The Fishwood environment is parameterised by the probabilities of successfully obtaining fish and wood at these respective states. In this paper we use the following values: at the river the agent has a 0.25 chance of catching a fish and in the woods the agent has a 0.65 chance of acquiring wood. For every fish caught, two pieces of wood are required to cook the fish, which results in a utility of 1. The goal in this setting is to maximise the following nonlinear utility function:

$$u = \min\left(\texttt{fish}, \left\lfloor \frac{\texttt{wood}}{2} \right\rfloor\right). \tag{23}$$

As demonstrated by Roijers et al. [61], to maximise utility in Fishwood it is essential that both past and future returns are taken into consideration when learning. For example, if there are 5 timesteps remaining and the agent has received 2 pieces of wood, the agent should go to the river and try to catch a fish to ensure a utility of 1 [61].

We evaluate NLU-MCTS and DMCTS in the Fishwood domain against Expected Utility Policy Gradient (EUPG) [61] and C51 [10]. EUPG achieves state-of-the-art results in the Fishwood problem under ESR [61]. C51 [10] is a distributional deep reinforcement learning algorithm that achieved state-of-the-art results in the Atari game problem domain.

For C51 the learning parameters were set as follows: $V_{min} = 0$, $V_{max} = 2$, $\epsilon = 0.01$, $\gamma = 1$ and $\alpha = 0.0001$. For DMCTS we set the number of bootstrap replicates, J, in the bootstrap

[7] It is important to note, in Sect. 6.3 we evaluate NLU-MCTS and DMCTS (both model-based algorithms) against a number of model-free algorithms. Therefore, to fairly evaluate model-based and model-free approaches, both NLU-MCTS and DMCTS maintain the search tree across episodes. This has the effect of both algorithm need less simulations during experimentation.

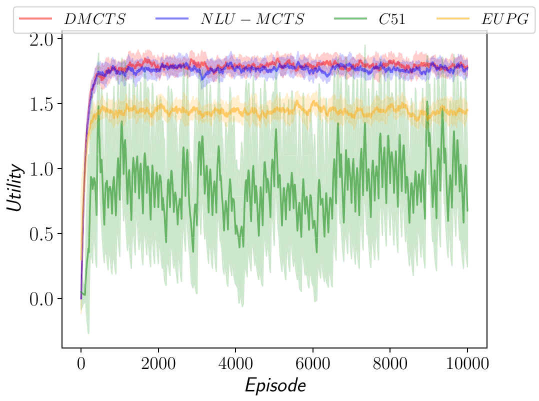

Fig. 14 Results from the Fishwood environment where DMCTS achieves state-of-the-art performance in a multi-objective setting over EUPG

distribution as follows: $J = 100$. For NLU-MCTS we set $C = \sqrt{2}$. EUPG is conditioned on the accrued returns and the current timestep, t. We set $n_{exec} = 2$ and run each experiment for 10, 000 episodes where each episode has 13 timesteps.

As shown in Fig. 14, the utility for C51 fluctuates throughout experimentation and it fails to learn a consistent policy. Given C51 does not take the accrued returns into consideration during learning the utility function is applied directly to the reward received by the agent. The reward received by an agent in the Fishwood domain can be [0, 1] or [1, 0]. Thereby applying the utility function, presented in Eq. 23, to the reward the C51 agent can only receive a utility of 0. DMCTS, NLU-MCTS and EUPG all take the accrued and future returns into consideration and can learn better policies when compared to C51.

DMCTS, NLU-MCTS and EUPG outperform C51. DMCTS and NLU-MCTS achieve a higher utility when compared to EUPG. All algorithms, except C51, use Monte Carlo simulations of the environment and optimise over the expected utility of the returns of a full episode. Although EUPG uses Monte Carlo simulations of the environment, policy gradient algorithms are sample inefficient. DMCTS and NLU-MCTS are sample efficient given both algorithms utilise the planning phase steps, which has been shown to be sample efficient [4, 14].

In the Fishwood environment, the agent is not guaranteed to obtain a fish or a piece of wood. For an action in a particular state the agent may need multiple simulations to understand the underlying distribution of the stochastic rewards. Both DMCTS and NLU-MCTS build a search tree, which enables the agent to re-sample the environment at each chance node during learning. However, DMCTS achieves an overall higher utility when compared with NLU-MCTS despite both algorithms utilising repeated sampling at each chance node and Monte Carlo simulations.

6.3.2 Renewable energy dynamic economic emissions dispatch

Next, we evaluate NLU-MCTS and DMCTS in a complex problem domain with a large state action space. Renewable Energy DEED (REDEED) is a variation of the traditional DEED problem [9]. In REDEED, the power demand for a city must be met over 24 h. To supply the city with sufficient power, a number of generators are required. There are 9 fossil fuel-powered generators, including a slack generator and 1 generator powered by renewable energy which is generated by a wind turbine. The optimal power output for each generator was derived by Mannion et al. [43] and the derived values are used for the both the fossil fuel generators and the renewable energy generator. In this example, Generator 3 is controlled by an agent, Generator 1 is a slack generator and Generator 4 is powered by a wind turbine.

In this setting we imagine a period of 24 h and for each hour we receive a weather forecast for a city. For hours 1–15, the weather is predictable and the optimal power values derived by Mannion et al. [43] can be used to generate power. From hours 16–24, a storm is forecast for the city. During the storm, both high and low levels of wind are expected and the weather forecast impacts how much power the wind turbine can generate. At each hour during the storm, there is a 0.15 chance the wind turbine will produce 25% less power than optimal, a 0.7 chance the wind turbine will produce optimal power and a 0.15 chance the wind turbine will produce 25% more power than optimal. In the REDEED problem we aim to learn a policy that can ensure the required power is met over the entire day while reducing both the cost and emissions created by all generators.

The goal is to maximise the following nonlinear utility function under the ESR criterion,

$$R_{+} = \prod_{o=1}^{O} f_{o}, \tag{24}$$

where f_o is the objective function for each objective, $o \in O$ [26, 43].

The following equation calculates the local cost for each generator n, at each hour m:

$$f_c^L(n,m) = a_n + b_n P_{nm} + c_n (P_{nm})^2 + |d_n sin\{e_n (P_n^{min} - P_{nm})\}|. \tag{25}$$

Therefore the global cost for all generators can be defined as:

$$f_c^G(m) = \sum_{n=1}^{N} f_c^L(n,m). \tag{26}$$

The local emissions for each generator, n, at each hour, m, is calculated using the following equation:

$$f_e^L(n,m) = E(a_n + b_n P_{nm} + \gamma_n (P_{nm})^2 + \eta \exp \delta P_{nm}). \tag{27}$$

Therefore the global emissions for all generators can be defined as:

$$f_e^G(m) = \sum_{n=1}^{N} f_e^L(n,m). \tag{28}$$

It is important to note the emissions for the generator controlled by the wind turbine are set to 0.

If the agent exceeds the ramp and power limits a penalty is received. A global penalty function f_p^G is defined to capture the violations of these constraints,

$$f_p^G(m) = \sum_{v=1}^{V} C(|h_v + 1|\delta_v). \tag{29}$$

Along with cost and emissions, the penalty function is an additional objective that will need to be optimised. Some parameters for this problem domain have not been included, all

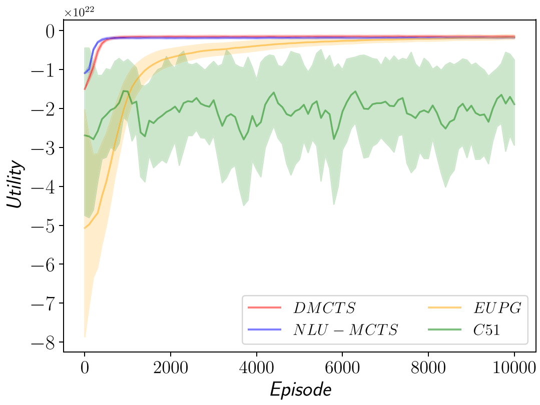

Fig. 15 Results from the REDEED environment DMCTS outperforms EUPG, C51 and NLU-MCTS. DMCTS achieves a higher utility compared to other algorithms used throughout experimentation in the REDEED domain under the ESR criterion

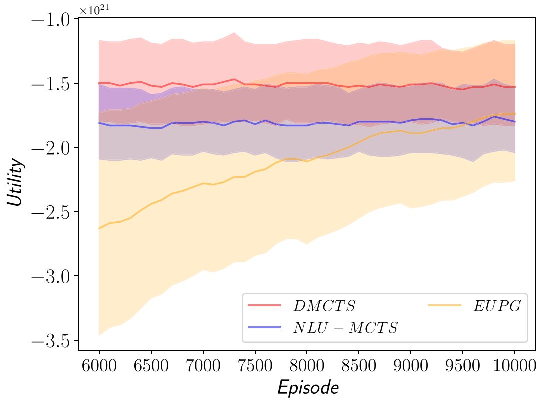

Fig. 16 Results from the final 4000 episodes of the REDEED environment to highlight how DMCTS outperforms NLU-MCTS, EUPG and C51

equations and parameters absent from this paper that are required to implement this problem domain can be found in the works of Basu [9] and Mannion et al. [43].

To evaluate NLU-MCTS and DMCTS in the REDEED domain, we compare against EUPG and C51. For DMCTS we set the number of bootstrap replicates, J, for the bootstrap distribution as follows: $J = 100$. For NLU-MCTS we set $C = \sqrt{2}$. For C51 the learning parameters were set as follows: $V_{min} = -8e^{22}$, $V_{max} = 0$, $\epsilon = 0.01$, $\gamma = 1$ and $\alpha = 0.0001$. For the REDEED problem the agent learns for 10,000 episodes and $n_{exec} = 2$ for each algorithm.

As seen in Fig. 15, DMCTS outperforms EUPG, NLU-MCTS and C51 in the REDEED domain. C51 struggles to learn a consistent policy and C51's utility fluctuates throughout experimentation. The hyper-parameters chosen for C51 provide good performance but are difficult to tune. Although the learning speed of EUPG is slow, EUPG achieves a higher utility than C51.

Both DMCTS and NLU-MCTS learn good policies faster than EUPG. MCTS algorithms are much more sample efficient when compared to policy gradient algorithms like EUPG. Figure 15 highlights the difference in sample efficiency of DMCTS, NLU-MCTS and EUPG given the differences in the number episodes required for each of the aforementioned algorithms to compute stable policies for the defined nonlinear utility function.

DMCTS, NLU-MCTS and EUPG all learn stable policies. However, DMCTS achieves a higher utility when compared to NLU-MCTS and EUPG. DMCTS converges to a policy with an average utility of -1.54×10^{21}. In comparison, NLU-MCTS converges to a policy with an average utility of -1.80×10^{21}, while EUPG converges to a stable policy with an average utility of -1.75×10^{21}. Given the scale of the utility computed throughout REDEED experimentation, it is difficult to see the final difference in utility in Fig. 15. Therefore, to highlight the difference in utility between DMCTS, NLU-MCTS and EUPG we have plotted the final 4, 000 episodes in Fig. 16. It is important to note, given C51 has performed poorly in the REDEED domain, we have not included C51 in Fig. 16. For the highlighted episodes in Fig. 16, it is clear that DMCTS achieves a higher utility when compared to both NLU-MCTS and EUPG.

The REDEED environment has a large state action space with complex returns. Although C51 has achieved state-of-the-art results in the Atari environment [10], C51 fails to learn any meaningful policy for REDEED. We hypothesise that a reason for poor performance is C51's inability to learn a distribution over the full returns and the level of discretisation of the distribution. The distribution for C51 uses 51 bins to discretise the algorithm's categorical distribution. In the work presented by Bellemare et al. [10] the number of bins is set to 51. While this provides good performance, Bellemare et al. [10] highlight that increasing the number of parameters may lead to increased performance. However, we fix the number of bins to 51 to remain consistant with the literature, given the potential added performance when increasing the number of bins has not been thoroughly explored. The results presented in this paper for C51 show this parameter setting is sub-optimal in scenarios where the returns are not simple scalars over small ranges. The results present in Fig. 15 show that C51 struggles to scale to large problem domains with complex returns over large ranges.

6.3.3 Discussion of experimental results for multi-objective MDPs

In multi-objective settings both NLU-MCTS and DMCTS learn good policies for the specified nonlinear utility functions. Similarly to the Risk-Aware MDP, applying the utility function to the cumulative returns (rather than just the expected future return) ensures that both NLU-MCTS and DMCTS can learn good policies. It is clear from the performance of C51 that applying the utility function to the cumulative returns is crucial for good performance in multi-objective settings when the utility function is nonlinear.

As previously highlighted, the difference between NLU-MCTS and DMCTS is the method used to explore during planning. NLU-MCTS uses UCB to determine which action to take during planning. UCB selects actions deterministically based on the expected utility and an exploration bonus [6, 66]. In contrast DMCTS selects actions using Thompson sampling, which stochastically samples from the underlying approximate posterior distribution (BTS distribution) and selects the action proportional to the probability of the action being optimal [15, 18]. In the bandit literature Thompson sampling has been shown to empirically outperform UCB [15, 66, 67]. Monte Carlo tree search methods utilise independent nodes, therefore we can consider each node itself to be a bandit. In this case, we expect Thompson sampling methods to also outperform UCB in sequential settings. Our findings for sequential settings in risk-aware and multi-objective settings are consistent with prior bandit literature that suggests that Thompson sampling can outperform UCB [15].

Additionally, UCB makes highly pessimistic assumptions regarding the underlying reward/return distributions, in order to guarantee a bound on the regret of its action selection procedure [66]. For known parametric distributions, tighter bounds have been proven using tighter upper confidence bounds (e.g. for Gaussian reward distributions [66]). However, doing something similar in our setting isn't opportune, because even if the return distributions are nicely parametric, the nonlinear transformation resulting from the application of the utility function would no longer allow for a closed-form distributions [66]. As such, we are either stuck with highly pessimistic assumptions (and therefore suboptimal performance) or we need to have a different method. Bootstrap distributions and the resulting Bootstrap Thompson sampling algorithm for action selection is able to approximate, and effectively exploit knowledge about the utility distributions, regardless of the shape of this underlying distribution.

6.4 Nonlinear utility functions

During experimentation DMCTS has been evaluated using previously defined utility functions for each experimental benchmark. To show that DMCTS can learn a good policy for any nonlinear utility function we have evaluated DMCTS in the Fishwood problem domain using four nonlinear utility functions under the ESR criterion. The following nonlinear utility functions are used to evaluate DMCTS in the Fishwood domain:

$$\begin{aligned}
u_1 &= max(\frac{r_1}{2}, \frac{r_2}{2}),\\
u_2 &= \frac{r_1}{2} + r_2^4,\\
u_3 &= min(\frac{r_1}{2}, \frac{r_2}{4}),\\
u_4 &= r_1^2 + r_2^2,
\end{aligned}$$

where r_1 is the returns received for the fish objective and r_2 is the returns received for the wood objective.

For this demonstration, we set $n_{exec} = 2$ and each experiment lasts 10, 000 episodes. For DMCTS we set the number of bootstrap replicates, J, for the bootstrap distribution as follows: $J = 100$.

In Fig. 17, for each utility function we have scaled the utility between 0 and 1. For the scaled utility, 1 represents the maximum utility and 0 represents the minimum utility

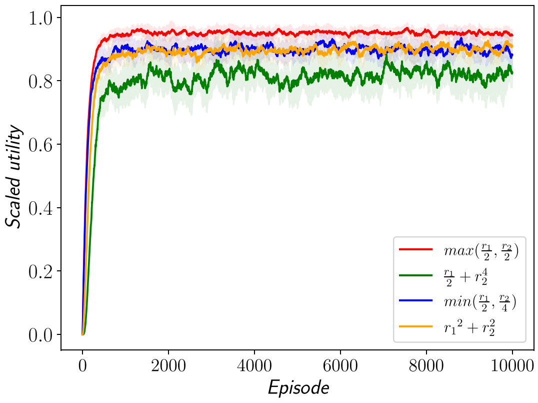

Fig. 17 Results from the Fishwood environment where DMCTS is evaluated against multiple nonlinear utility functions

obtained by DMCTS. We have scaled the utility to show the performance of DMCTS for each utility function on a single plot.

Figure 17 outlines the performance of DMCTS when optimising for each nonlinear utility function. It is clear from Fig. 17 that DMCTS converges to a good policy for each utility function. Therefore, DMCTS can learn a good policy for each of the outlined nonlinear utility functions and is not limited to the utility functions associated with predefined benchmark problems. The ability of DMCTS to learn a good policy for a range of nonlinear utility showcases how DMCTS could potentially be used in real-world scenarios, where different decision makers may have very different nonlinear utility functions for the same problem.

7 Related Work

Many risk-aware RL approaches seek to learn policies to maximise the expected return. Some research in this area focuses on learning policies which maximise the expected exponential utility [46]. Other approaches take the weighted sum of the return and risk into consideration when learning policies [22, 24]. Although most risk-aware RL approaches aim to maximise the expected utility, they often do not take into consideration the utility of the return of a full episode. It is also important to note that little research exists where decisions are made based on a learned distribution over the expected returns [47, 48] for risk-aware RL.

Many MCTS methods have developed for situations involving reward uncertainty. For example, Tesauro et al. [73] take a Bayesian approach to UCT with Gaussian approximation. Their method backpropagates probability distributions over rewards. To select actions Tesauro et al. [73] use UCB1 while taking the distributions into consideration. Cazenave and Saffidine [13] define a MCTS algorithm that takes into account the bounds on the possible values of a node to select nodes for exploration. They apply their algorithm to problems that have more than two outcomes and show that taking the bounds into consideration can increase performance. Kaufmann and Koolen [37], and Huang et al. [35] also have developed MCTS algorithms which can compute policies for settings with reward uncertainty.

As previously highlighted, the majority of RL research focuses on the SER criterion. Multi-objective MCTS (MOMCTS) [80] was shown to be able to learn a coverage set under SER. However, MOMCTS can only learn a coverage set in deterministic environments. Convex Hull MCTS [53] is able to learn the convex hull of the Pareto front but focuses solely on linear utility functions. A number of other multi-objective MCTS methods exist [39, 56, 57], but no method has previously been shown to learn the Pareto front for both deterministic and stochastic environments for any unknown utility function. In contrast to the SER criterion, no method exists that can learn a set of optimal policies under the ESR criterion in sequential settings. Hayes et al. [33] compute a set of ESR non-dominated return distributions, known as the ESR set, in a multi-objective multi-armed bandit setting. However, the method proposed by Hayes et al. [33] cannot be applied to sequential decision making problems. An interesting opportunity for future work is the possibility of building on the methods of Wang and Sebag [80] and Painter et al. [53] to extend DMCTS to learn the optimal coverage set under both SER and ESR for any unknown utility function.

Zhang et al. [87] compute a multi-variate distribution over the returns for RL settings. While this work considers reward vectors, we believe this algorithm will suffer from similar limitations to traditional RL algorithms when applied to nonlinear utility functions. The method proposed by Zhang et al. [87] does not take the accrued returns into consideration. Therefore we believe that such an approach would fail to achieve a high utility [61]. However, this method could be an interesting starting point for developing model-free multi-objective distributional RL algorithms.

A key argument in this paper is that the expected utility of the future returns under ESR must be replaced with a posterior distribution over the expected utility of the returns. Bai et al. [8] extend MCTS to maintain a distribution at each node using Thompson Sampling as an exploration strategy. However, the work presented in this paper is significantly different. In their work, Bai et al. [8] do not learn a posterior distribution over the expected utility of the return, apply their work to multi-objective settings, or incorporate the accrued returns as part of their algorithm. It is also important to note the C51 algorithm proposed by Bellemare et al. [10] achieves state-of-the-art performance in single-objective settings and learns a distribution over the future returns. Abdolmaleki et al. [1] learn a distribution over actions based on constraints set per objective. This approach ignores the utility-based approach [62] and uses constraints set by the user to learn a coverage set of policies where the value of constraints is dependent on the scale of the objectives. Abdolmaleki et al. [1] claim setting the constraints for this algorithm is a more intuitive approach when compared to setting weights for a linear utility function. We theorise that if the user's utility function is nonlinear, this approach would fail to learn a coverage set.

8 Conclusion and future work

In this paper we propose a novel Monte Carlo tree search algorithm that can compute good policies for nonlinear utility functions (NLU-MCTS). We then extend NLU-MCTS, to define a new distributional Monte Carlo tree search (DMCTS) algorithm. Both NLU-MCTS and DMCTS are able to learn good policies in MORL settings, under the ESR criterion for nonlinear utility functions in problem domains with stochastic rewards. DMCTS replaces the expected utility of the future returns with a bootstrap distribution over the utility of the returns, and achieves state-of-the-art performance in MORL domains under the ESR criterion. We achieve this by using a bootstrap distribution as an approximate posterior over the expected utility of the returns of the episode. It is our hope that this paper will inspire further work on algorithms that replace the expected returns with a distribution over the expected utility of the returns for risk-aware and ESR settings.

Although DMCTS achieves state-of-the-art performance under the ESR criterion, as the size of the problem domain increases we expect the bootstrap distribution may encounter limitations. In order to apply DMCTS to real-world problem domains like, [3], distributions like Dirichlet [50] may be better suited to high dimensional state spaces especially when dealing with multi-variate scenarios when learning policies for the SER criterion.

We also aim to extended both NLU-MCTS and DMCTS to learn policies in multi-objective environments with continuous state spaces. For example, Abels et al. [2] define a multi-objective benchmark problem known as Minecart that has a continuous state space. In the future we plan to extend DMCTS to learn policies in problem domains with continuous action spaces [85].

DMCTS initialises the α and β values for each bootstrap replicate to 1. However, not much is known about the impact of such an initialisation. It is possible that varying this value could increase performance. Therefore, we plan to investigate the potential performance gains possible from altering this parameter on initialisation.

In the work of Martin et al. [45] stochastic dominance [25, 40, 84] is used to determine optimal actions by comparing learned categorical distributions over the returns. Martin et al. [45] demonstrate good performance in risk-based scenarios. Hayes et al. [32, 33] present ESR dominance as a dominance criteria for distributional multi-objective decision making under the ESR criterion. In future work we aim to utilise stochastic dominance with DMCTS for single objective risk-based problems. We also plan to extend DMCTS to utilise ESR dominance as a dominance criterion for multi-objective decision making problems under the ESR criterion.

In this paper, the utility function is known a priori. In different MORL scenarios, the utility function can be unknown at the time of learning or planning [31, 59, 62]. In these scenarios, an algorithm must recover a coverage set of optimal policies. Under the SER criterion many methods have been developed that compute sets of optimal policies [12, 55, 81, 82, 86]. For example, multi-objective MCTS [80] can learn a coverage set for deterministic environments under SER. Currently, no method exists that can compute a set of optimal policies for the ESR criterion by interacting with the environment in a sequential setting. In future work, we aim to extend our DMCTS algorithm to be able to learn coverage sets for unknown utility functions under the ESR criterion and the SER criterion for stochastic environments.

Acknowledgements Conor F. Hayes is funded by the University of Galway Hardiman Scholarship. This research was supported by funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" program.

Funding Open Access funding provided by the IReL Consortium.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Abdolmaleki, A., Huang, S. H., Hasenclever, L., Neunert, M., Song, H., Zambelli, M., Martins, M. F., Heess, N., Hadsell, R., & Riedmiller, M. A. (2020). A distributional view on multi-objective policy optimization. ArXiv.
2. Abels, A., Roijers, D. M., Lenaerts, T., Nowé, A., & Steckelmacher, D. (2019). Dynamic weights in multi-objective deep reinforcement learning. In *International conference on machine learning* (pp. 11–20). PMLR.
3. Abrams, S., Wambua, J., Santermans, E., Willem, L., Kuylen, E., Coletti, P., et al. (2021). Modelling the early phase of the Belgian covid-19 epidemic using a stochastic compartmental model and studying its implied future trajectories. *Epidemics, 35*, 100449. https://doi.org/10.1016/j.epidem.2021.100449

4. Abramson, B. (1987). The expected-outcome model of two-player games. *Ph.D. thesis*, Columbia University.
5. Arrow, K. J. (1965). *Aspects of the theory of risk-bearing*. Yrjo Jahnssonin Saatio: Yrjo Jahnsson lectures.
6. Auer, P. (2002). Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research 3*(Nov), 397–422.
7. Bai, A., Wu, F., Zhang, Z., & Chen, X. (2014). Thompson sampling based monte-carlo planning in pomdps. In *Twenty-fourth international conference on automated planning and scheduling*.
8. Bai, A., Wu, F., Zhang, Z., & Chen, X. (2014). Thompson sampling based monte-carlo planning in pomdps. In *Proceedings of the twenty-fourth international conference on international conference on automated planning and scheduling, ICAPS'14* (pp. 29–37). AAAI Press.
9. Basu, M. (2008). Dynamic economic emission dispatch using nondominated sorting genetic algorithm-ii. *International Journal of Electrical Power and Energy Systems, 78*, 140–149.
10. Bellemare, M. G., Dabney, W., & Munos, R. (2017). A distributional perspective on reinforcement learning. In *Proceedings of the 34th international conference on machine learning-volume* (Vol 70, pp. 449–458). JMLR. org.
11. Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., et al. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games, 4*(1), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810
12. Bryce, D., Cushing, W., & Kambhampati, S. (2007). Probabilistic planning is multi-objective. *Technical Report* ASU-CSE-07-006, Arizona State University.
13. Cazenave, T., & Saffidine, A. (2010). Score bounded monte-carlo tree search. In *International conference on computers and games* (pp. 93–104). Springer.
14. Chang, H. S., Fu, M. C., Hu, J., & Marcus, S. I. (2005). An adaptive sampling algorithm for solving Markov decision processes. *Oper. Res., 53*(1), 126–139. https://doi.org/10.1287/opre.1040.0145
15. Chapelle, O., & Li, L. (2011). An empirical evaluation of Thompson sampling.
16. Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search.
17. Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., & Hester, T. (2021). Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *In Machine learning*. https://doi.org/10.1007/s10994-021-05961-4
18. Eckles, D., & Kaptein, M. (2014). Thompson sampling with the online bootstrap. arxiv:abs/1410.4009.
19. Eckles, D., & Kaptein, M. (2019). Bootstrap Thompson sampling and sequential decision problems in the behavioral sciences. SAGE Open, *9*(2).
20. Efron, B. (2012). Bayesian inference and the parametric bootstrap. *The Annals of Applied Statistics, 6*(4), 1971–1997. https://doi.org/10.1214/12-AOAS571
21. Friedman, M., & Savage, L. J. (1948). The utility analysis of choices involving risk. *Journal of Political Economy, 56*(4), 279–304. http://www.jstor.org/stable/1826045.
22. Geibel, P., & Wysotzki, F. (2005). Risk-sensitive reinforcement learning applied to control under constraints. *Journal of Artificial Intelligence Research, 24*(1), 81–108.
23. Gerber, H. U., & Pafum, G. (1998). Utility functions: from risk theory to finance. *North American Actuarial Journal, 2*(3), 74–91.
24. Gosavi, A. (2009). Reinforcement learning for model building and variance-penalized control. In *Winter simulation conference, WSC '09* (pp. 373–379). Winter Simulation Conference.
25. Hanoch, G., & Levy, H. (1969). The efficiency analysis of choices involving risk. *The Review of Economic Studies, 36*(3), 335–346. http://www.jstor.org/stable/2296431.
26. Hayes, C. F., Howley, E., & Mannion, P. (2020). Dynamic thresholded lexicograpic ordering. In *Proceedings of the adaptive and learning agents workshop at AAMAS 2020.*
27. Hayes, C. F., Reymond, M., Roijers, D. M., Howley, E., & Mannion, P. (2021). Distributional monte carlo tree search for risk-aware and multi-objective reinforcement learning. In *Proceedings of the 20th international conference on autonomous agents and multiagent systems* (pp. 1530–1532).
28. Hayes, C. F., Reymond, M., Roijers, D. M., Howley, E., & Mannion, P. (2021). Risk-aware and multi-objective decision making with distributional monte carlo tree search. In *Proceedings of the adaptive and learning agents workshop at AAMAS 2021).*
29. Hayes, C. F., Roijers, D. M., Howley, E., & Mannion, P. (2022). Decision-theoretic planning for the expected scalarised returns. In *Proceedings of the 21st international conference on autonomous agents and multiagent systems* (pp. 1621–1623).
30. Hayes, C. F., Roijers, D. M., Howley, E., & Patrick, M. (2022). In *Adaptive and learning agents workshop (AAMAS: Distributional multi-objective value iteration, 2022).*

31. Hayes, C. F., Rădulescu, R., Bargiacchi, E., Källström, J., Macfarlane, M., Reymond, M., Verstraeten, T., Zintgraf, L. M., Dazeley, R., Heintz, F., Howley, E., Irissappane, A. A., Mannion, P., Nowé, A., Ramos, G., Restelli, M., Vamplew, P., & Roijers, D. M. (2022). A practical guide to multi-objective reinforcement learning and planning. *Autonomous Agents and Multi-Agent Systems, 36*(1), 26. https://doi.org/10.1007/s10458-022-09552-y
32. Hayes, C. F., Verstraeten, T., Roijers, D. M., Howley, E., & Mannion, P. (2021). Dominance criteria and solution sets for the expected scalarised returns. In *Proceedings of the adaptive and learning agents workshop at AAMAS 2021.*
33. Hayes, C. F., Verstraeten, T., Roijers, D. M., Howley, E., & Mannion, P. (2022). Expected scalarised returns dominance: A new solution concept for multi-objective decision making. *Neural Computing and Applications*, 1–21.
34. Hayes, C. F., Verstraeten, T., Roijers, D. M., Howley, E., & Mannion, P. (2022). Multi-objective coordination graphs for the expected scalarised returns with generative flow models. *European workshop on reinforcement learning (EWRL).*
35. Huang, R., Ajallooeian, M. M., Szepesvári, C., & Müller, M. (2017). Structured best arm identification with fixed confidence. In S. Hanneke, L. Reyzin (Eds.) *Proceedings of the 28th international conference on algorithmic learning theory, proceedings of machine learning research* (Vol. 76, pp. 593–616). PMLR. https://proceedings.mlr.press/v76/huang17a.html
36. Karni, E., & Schmeidler, D. (1991). Utility theory with uncertainty. *Handbook of Mathematical Economics, 4*, 1763–1831.
37. Kaufmann, E., & Koolen, W. M. (2017). Monte-carlo tree search by best arm identification. *Advances in Neural Information Processing Systems, 30.*
38. Kocsis, L., & Szepesvári, C. (2006). *Bandit based Monte-Carlo planning*, pp. 282–293.
39. Lee, J., Kim, G. h., Poupart, P., & Kim, K. E. (2018). Monte-carlo tree search for constrained pomdps. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.) *Advances in neural information processing systems* (Vol. 31, pp. 7923–7932). Curran Associates, Inc.
40. Levy, H. (1992). Stochastic dominance and expected utility: Survey and analysis. *Management Science, 38*(4), 555–593.
41. Machina, M. J. (1987). Choice under uncertainty: Problems solved and unsolved. *The Journal of Economic Perspectives, 1*(1), 121–154. http://www.jstor.org/stable/1942952.
42. Malerba, F., & Mannion, P. (2021). In *Multi-objective decision making workshop (MODeM: Evaluating tunable agents with non-linear utility functions under expected scalarised returns, 2021.*
43. Mannion, P., Devlin, S., Duggan, J., & Howley, E. (2018). Reward shaping for knowledge-based multi-objective multi-agent reinforcement learning. *The Knowledge Engineering Review, 33*, e23. https://doi.org/10.1017/S0269888918000292
44. Mannion, P., Heintz, F., Karimpanal, T. G., & Vamplew, P. (2021). Multi-objective decision making for trustworthy ai. In *Multi-objective decision making workshop (MODeM).*
45. Martin, J., Lyskawinski, M., Li, X., & Englot, B. (2020). Stochastically dominant distributional reinforcement learning. In *International conference on machine learning* (pp. 6745–6754). PMLR.
46. Moldovan, T., & Abbeel, P. (2012). Risk aversion in Markov decision processes via near-optimal Chernoff bounds. *Advances in Neural Information Processing Systems, 4,* 3131–3139.
47. Morimura, T., Sugiyama, M., Kashima, H., Hachiya, H., & Tanaka, T. (2010). Nonparametric return distribution approximation for reinforcement learning. In *ICML* (pp. 799–806).
48. Morimura, T., Sugiyama, M., Kashima, H., Hachiya, H., & Tanaka, T. (2010). Parametric return density estimation for reinforcement learning. In *Proceedings of the twenty-sixth conference on uncertainty in artificial intelligence, UAI'10* (pp. 368-375). AUAI Press, Arlington, Virginia, USA.
49. Newton, M., & Raftery, A. (1994). Approximate Bayesian inference by the weighted likelihood bootstrap. *Journal of the Royal Statistical Society Series B-Methodological, 56,* 3–48.
50. Olkin, I., & Rubin, H. (1964). Multivariate beta distributions and independence properties of the wishart distribution. *The Annals of Mathematical Statistics*, 261–269.
51. Owen, A. B., & Eckles, D. (2012). Bootstrapping data arrays of arbitrary order. *The Annals of Applied Statistics, 6*(3), 895–927. http://www.jstor.org/stable/41713508.
52. Oza, N. C., & Russell, S. (2005). *Online Bagging and Boosting, 3*, 2340–2345.
53. Painter, M., Lacerda, B., & Hawes, N. (2020). Convex hull monte-carlo tree-search. In *Proceedings of the thirtieth international conference on automated planning and scheduling* (pp. 217–225), Nancy, France, October 26-30, 2020. AAAI Press.
54. Pan, A., Xu, W., Wang, L., & Ren, H. (2020). Additional planning with multiple objectives for reinforcement learning. *Knowledge-Based Systems, 193,* 105392.

55. Parisi, S., Pirotta, M., & Restelli, M. (2016). Multi-objective reinforcement learning through continuous pareto manifold approximation. *Journal of Artificial Intelligence Research, 57*, 187–227.
56. Perez, D., Mostaghim, S., Samothrakis, S., & Lucas, S. (2015). Multiobjective Monte Carlo tree search for real-time games. *IEEE Transactions on Computational Intelligence and AI in Games, 7*(4), 347–360.
57. Perez, D., Samothrakis, S., & Lucas, S. (2013). Online and offline learning in multi-objective monte carlo tree search. In *2013 IEEE conference on computational inteligence in games (CIG)* (pp. 1–8).
58. Perny, P., & Weng, P. (2010). On finding compromise solutions in multiobjective Markov decision processes. *ECAI, 215,* 969–970.
59. Rădulescu, R., Mannion, P., Roijers, D. M., & Nowé, A. (2020). Multi-objective multi-agent decision making: a utility-based analysis and survey. *Autonomous Agents and Multi-Agent Systems, 34*(10).
60. Reymond, M., Hayes, C. F., Willem, L., Rădulescu, R., Abrams, S., Roijers, D. M., Howley, E., Mannion, P., Hens, N., Nowé, A., & Libin, P. (2022). Exploring the pareto front of multi-objective covid-19 mitigation policies using reinforcement learning. arXiv preprint arXiv:2204.05027.
61. Roijers, D. M., Steckelmacher, D., & Nowé, A. (2018). Multi-objective reinforcement learning for the expected utility of the return. In *Proceedings of the adaptive and learning agents workshop at FAIM 2018*.
62. Roijers, D. M., Vamplew, P., Whiteson, S., & Dazeley, R. (2013). A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research, 48*, 67–113.
63. Roijers, D. M., Zintgraf, L. M., Libin, P., Reymond, M., Bargiacchi, E., & Nowé, A. (2021). Interactive multi-objective reinforcement learning in multi-armed bandits with gaussian process utility models. In F. Hutter, K. Kersting, J. Lijffijt, & I. Valera (Eds.), *Machine learning and knowledge discovery in databases* (pp. 463–478). Springer International Publishing.
64. Rădulescu, R., Mannion, P., Roijers, D. M., & Nowé, A. (2019). Equilibria in multi-objective games: A utility-based perspective. In *Adaptive and learning Agents workshop (at AAMAS 2019).*
65. Rubin, D. B. (1981). The bayesian bootstrap. *The Annals of Statistics, 9*(1), 130–134. http://www.jstor.org/stable/2240875.
66. Russo, D., & Van Roy, B. (2014). Learning to optimize via posterior sampling. *Mathematics of Operations Research, 39*(4), 1221–1243.
67. Russo, D. J., Van Roy, B., Kazerouni, A., Osband, I., & Wen, Z. (2018). A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning 11*(1), 1–96.
68. Shen, W., Trevizan, F., Toyer, S., Thiébaux, S., & Xie, L. (2019). Guiding mcts with generalized policies for probabilistic planning. *HSDIP, 2019*, 63.
69. Shen, Y., Tobia, M. J., Sommer, T., & Obermayer, K. (2014). Risk-sensitive reinforcement learning. *Neural Computation, 26*(7), 1298–1328.
70. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*.
71. Sutton, R. S., & Barto, A. G. (1998). *Introduction to reinforcement learning* (Vol. 135). MIT Press Cambridge.
72. Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th international conference on neural information processing systems, NIPS'99* (pp. 1057-1063). MIT Press.
73. Tesauro, G., Rajan, V. T., & Segal, R. (2010). Bayesian inference in monte-carlo tree search. In *Proceedings of the twenty-sixth conference on uncertainty in artificial intelligence, UAI'10* (pp. 580–588). AUAI Press.
74. Vamplew, P., Foale, C., & Dazeley, R. (2021). The impact of environmental stochasticity on value-based multiobjective reinforcement learning. *In Neural Computing and Applications*. https://doi.org/10.1007/s00521-021-05859-1
75. Vamplew, P., Smith, B. J., Kallstrom, J., Ramos, G., Radulescu, R., Roijers, D. M., Hayes, C. F., Heintz, F., Mannion, P., Libin, P. J. K., Dazeley, R., & Foale, C. (2022). Scalar reward is not enough: A response to silver, singh, precup and sutton (2021). *Autonomous Agents and Multi-Agent Systems, 36*(2), 41. https://doi.org/10.1007/s10458-022-09575-5
76. Vamplew, P., Yearwood, J., Dazeley, R., & Berry, A. (2008). On the limitations of scalarisation for multi-objective reinforcement learning of pareto fronts. In *Australasian joint conference on artificial intelligence* (pp. 372–378). Springer.

77. Van Moffaert, K., Drugan, M. M., & Nowé, A. (2013). Scalarized multi-objective reinforcement learning: Novel design techniques. In *2013 IEEE symposium on adaptive dynamic programming and reinforcement learning (ADPRL)* (pp. 191–199).
78. Veness, J., Ng, K. S., Hutter, M., Uther, W., & Silver, D. (2011). A Monte-Carlo aixi approximation. *Journal of Artificial Intelligence Research, 40*(1), 95–142.
79. Von Neumann, J., & Morgenstern, O. (1947). *Theory of games and economic behavior*, 2nd rev.
80. Wang, W., & Sebag, M. (2012). *Multi-objective Monte-Carlo tree search*. (pp. 507–522) PMLR, Singapore Management University, Singapore.
81. White, D. (1982). Multi-objective infinite-horizon discounted Markov decision processes. *Journal of Mathematical Analysis and Applications, 89*(2), 639–647.
82. Wiering, M. A., Withagen, M., & Drugan, M. M. (2014). Model-based multi-objective reinforcement learning. In *2014 IEEE symposium on adaptive dynamic programming and reinforcement learning (ADPRL)* (pp. 1–6). IEEE.
83. Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning, 8*(3–4), 229–256.
84. Wolfstetter, E. (1999). *Topics in microeconomics: Industrial organization, auctions, and incentives*. Cambridge University Press. https://doi.org/10.1017/CBO9780511625787
85. Xu, J., Tian, Y., Ma, P., Rus, D., Sueda, S., & Matusik, W. (2020). Prediction-guided multi-objective reinforcement learning for continuous robot control. In *Proceedings of the 37th international conference on machine learning*.
86. Yang, R., Sun, X., & Narasimhan, K. (2019). A generalized algorithm for multi-objective reinforcement learning and policy adaptation. In *Advances in neural information processing systems* (pp. 14636–14647).
87. Zhang, P., Chen, X., Zhao, L., Xiong, W., Qin, T., & Liu, T. Y. (2021). Distributional reinforcement learning for multi-dimensional reward functions. *Advances in Neural Information Processing Systems, 34*, 1519–1529.