Adapt the Nested Rollout Policy Adaptation to solve stochastic problem

Thomas Petiteau

november 20 2020

Contents

1 Introduction

The Monte Carlo Tree Search (MCTS) is a family of algorithm that have successfully applied to a lot of problems and games [2]. In particular, the Nested Rollout Policy Adaptation (NRPA) has been recently used to solve optimization problem such as Traveling Salesman Problem with Time Window (TSPTW)[5] and it obtains state of the art result on many. It has found new records in Morpion Solitaire and crosswords puzzle. But it only performs well on non stochastic problem which compose a small portion of more practical importance. In this paper, we present an adaptation of the NRPA to stochastic context resulting in a new algorithm we call Stochastic Nested Rollout Policy Adaptation (SNRPA). We will first present the MCTS and then the NRPA. We will then proceed to explain the difficulty of solving stochastic problems with the NRPA. Finally, we will introduce the new SNRPA and we will try it on instances of the Tactical Wildfire Management (TWM) problem with other algorithms to compare their performance.

2 Monte Carlo Tree Search algorithm

MCTS algorithm is a family of algorithm based on Monte-Carlo simulations and a tree seach procedure. Monte-Carlo simulations were first used in 1993 [3] for the game of Go. In 2006 [6], MCTS most popular algorithm called Upper Confidence bound applied to Trees (UCT) was introduced. It used multi armed bandit in the tree search procedure to keep a balance between exploitation and exploration of the tree. Which, respectively means to study more a well known and good strategy and explore new stategies. Nowadays, MCTS algorithms have been used in many domains [2] and still give state of the art result in many of them. For example, it is a core component of algorithms such as AlphaZero [8].

Here, we are interested in solving optimization problems, that can also be seen as games or puzzle. Moreover, we have a particular interest in stochastic problems. In this section, we will discuss the different classic components of a MCTS algorithm. We will first discuss the preliminaries to understand the MCTS.

2.1 Markov Decision Process

Optimization problem solved using MCTS algorithm are modelised as a Markov Decision Process (MDP). it modelises the problem as a set of states and actions that can be applies or played on a state. A state contains all the information necessary to understand the situation meaning that the past state are irrelevant. In this section,

we will give a vulgarize explanation of what a game is and a more definition of a MDP.

2.1.1 Sequential problem modelization

Let's consider a single player basic game called the Left Most Problem (LMP). Its rules are simple:

- Each turn the player can choose *left* or *right*.
- A player choosing *left* gain one point.
- A player choosing *right* gain no point.
- The game ends after N turn.

The goal of the game is to have as much point as possible at the end. It is a very simple problem but is perfect to explaine the key concept. A state of the game can be represented by two elements: the score of the player and the number of turn that passed. So if we define the state $s = (6, 10)$ it means that the player has 6 points and has played 10 turns. Past states are irrelevant to describe s, it contains all information necessary to describe the situation. Some states are called terminal because they correspond to state where the game has ended. Here, all the states where the player has played N turns are final or terminal.

The actions the player can choose are *left* and *right*. If the player applies *left* to s then, we will obtain a new state $s' = (7, 11)$. The transition from s to s' is always the same if we apply *left* to s.

The goal being to obtain the most point as possible, a player will learn to always pick *left*. This is called a policy or, more conviniently, a strategy: it dictates the action according to a situation. In other, if I give s to a player following the policy to always go right, he will choose *right* and will go to state $s'' = (6, 11)$.

This simply explains the key concepts of a MDP. You have states on which you apply actions. Applying certain actions on certain states will give you a new state. And states have what is called a reward, which is the number of point the player scored.

2.1.2 Definition

Now for a more formal definition, a MDP is composed of:

- S a set of state.

- A a set of actions
  - $A_s \subseteq A$ are the legal actions or actions that can be applied to s
  - If $A_s = \emptyset$ then it means that s is a final or terminal state
- $P_a(s, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that when applying action a to state s, we obtain state s'
- $R_a(s, s')$ is the reward obtained when transitioning from s to s' by applying action a

When optimizing a MDP, we search for the best policy, that means the policy that will give the best expected reward possible. A policy $\pi(s)$ is a distribution of probability over all the elements of A_s. A policy that maximizes the reward is called an optimal policy and is often noted π^*.

A MDP represents a stochastic game if the probabilities given by $P_a(s, s')$ are not equal to 1. That is to say, if the transitions between the states are uncertain, therefore random, the modelized problem is stochastic.

2.2 Key steps of a MCTS

A MCTS algorithm such as UCT follows four steps that are repeated indefinitely. Theses steps are the selection, the expansion, the simulation and the backpropagation. Here, we will give a quick explanation of each of these steps. But, before going in, we have to explain what the tree used in MCTS approach looks like.

A MCTS tree is a tree as defined in graph theory which is not mandatory to explain here. At the top of the tree, we find the root state, which is the current state of the game, the one for which we search the best action. Each node of the tree is a state and nodes are connected by actions.

During the selection the algorithm will descend in the tree from node to node, starting from the root node. At one point, according to a policy, it will stop on a node corresponding to a state s and begin the expansion phase.

The expansion phase corresponds to the addition of a new node on the tree. It applies an action $a \in A_s$ chosen according to a policy. Then, it adds the resulting state, let's say s' to the tree.

The simulation is a Monte-Carlo simulation. It is used to evaluate the quality of a node for a given player. It consists in playing the game randomly from state s' until a terminal state is reached. No modification is done to the tree. The simulation is called a rollout because it is a random playout. It follows what is called the default policy which chooses the action uniformly.

The backpropagation is the last phase. The result of the rollout is transmitted back to the top of the tree. It follows back the way chosen at the selection. We can then have statistics on each node which, often, are useful to create the policy used during the selection and expansion.

All theses steps are repeated over and over again. The number of repetition is often limited by a search time which is set beforehand. Based on the tree obtained at the end, the player can choose which action apply on the root state. Often, the search is repeated on a new tree each turn to avoid the algorithm to be stuck on the same strategy. erge

3 The Nested Rollout Policy Adaptation

The Nested Rollout Policy Adaptation (NRPA)[7] is a Monte Carlo Tree Search (MCTS) algorithm that has been used to solve optimization problem with state of the art results like the Traveling Salesman Problem with Time Window (TSPTW)[5]. Howether, it is not a traditional MCTS algorithm as it does not follow the classic steps of these algorithms: selection, expansion, simulation and backpropagation (see section 2.2). Even the tree procedure is very different from other MCTS algorithm. The NRPA learns a playout policy used during the rollouts. Actions are associated to weights and sampled during rollout according to the exponential of this same weight. Doing so, it find the best sequence of actions it can and modifiy its policy to adapt it to the best known sequence. In the end, it returns the best sequence found.

This algorithm is very effective to solve non stochastic optimization problem and has found new records for game like Morpion Solitaire. In this section, we will explain the NRPA in three parts: the playout, the adaptation and the core function.

3.1 NRPA playout

A NRPA playout is used to generate and evaluate a sequence of actions. So a sequence $S = a1, a2, a3, \ldots$ is a list of actions that lead from the root state to a terminal one. To S, we can attach a score equal to the sum of rewards obtained by applying it to the root state. As the problem is not stochastic, therefore, S can be apply any number of time and will always give the same resuls.

The algorithm in charge of the rollout is given in algorithm 1. The algorithm first initialize the sequence it's going to create (line 2). Then if it reaches a terminal state, it returns the sequence alongside the score obtained. Otherwise, it repeats the following. It sums the exponential of the weights attached to each legal actions for the current state (line 8-10). Then it samples a move among the legal ones according to the exponential of the weight divided by the sum it just computed. This basically means that, a move attached to a big weight will more likely be picked. It then plays the chosen action and update the sequence by adding the move at the end.

3.2 NRPA adapt

The NRPA adapt algorithm is used to adapt a policy to a given sequence. We explained in section 3.1 that the policy is used during rollout to select actions and generate sequence. Adapt a policy π to a sequence S means to give the policy a greater tendency to generate S. More formerly, is Ω is the state of all sequences,

Algorithm 1: The NRPA playout algorithm

```
1  playout(state, policy)
2  sequence ← []
3  while true do
4  |  if state is terminal then
5  |  |  return (score (state), sequence)
6  |  end
7  |  z ← 0.0
8  |  for m in possible moves for state do
9  |  |  z ← z + exp(policy [code(m)])
10 |  end
11 |  move ← choose a move with probability
   |    exp(policy [code(move)]) / z
12 |  state ← play(state, move)
13 |  sequence ← sequence + move
14 end
```

then a policy π can be seen as a probability distribution over Ω. If π is adapted to S resulting in π' then $P(S|\pi) \leq P(S|\pi')$.

The adapt algorithm is given in algorithm 2. The algorithm starts by copying the policy (line 2) which can be avoided as explained with the generalized form of the NRPA[4]. Then, for each move of the sequence (line 4), it starts by increasing the weight associated to the current move (line 5). Next, it computes the sum of weights, taken to the exponential, of all legal moves for the current state (line 7-9). Finally, it updates the weight of all considered actions according to the exponential of their weight divided by the sum (line 10-12). The sum of weights is important to maintain the sum of probability to 1.

The adapt algorithm use a argument α that represents the learning step of the adapt algorithm. The bigger it is, the more closer the policy will be to the sequence after it has been adapted. α is problem-dependent and must be tune in order to obtain the best possible results. It represents one of the difficulty of the NRPA.

3.3 NRPA core algorithm

The core function of the NRPA aims to learn a policy in order to find the best sequence it can. The algorithm is given in algorithm 3. It is a recursive function taking a level and a policy and returning a sequence alongside its score. Each level greater

Algorithm 2: The NRPA adapt algorithm

```
1  adapt(policy, sequence)
2  polp ← policy
3  state ← root
4  for move in sequence do
5  |  polp [code [move]] ← polp [code(move)] + α
6  |  z ← 0.0
7  |  for m in possible moves for state do
8  |  |  z ← exp(policy [code(m)])
9  |  end
10 |  for m in possible moves for state do
11 |  |  polp [code(m)] ← polp [code(m)] - α * exp(policy [code(m)]) / z
12 |  end
13 |  state ← play(state, move)
14 end
15 policy ← polp
```

than 0 learns a policy and search a sequence. At level 0, the function computes a rollout according to the policy it received. Each level obtain N sequence by recursive calls. The best sequence is stored and, at each step, the policy is adapted to it.

To give more details, the algorithm starts by checking its level and, if it is zero, it computes a playout (line 2-4). If the level is greater, it performs N iterations (line 6). It obtains a new sequence by recursive call (line 7) and proceeds to compare it to the best known one (line 8-11). The policy received by copy (it is important to mention it because otherwise, nesting the call would have no use) is then adapted to the best sequence (line 12).

3.4 NRPA, a story of code

As seen in the playout (see section 3.1) and the adapt (see section 3.2), the NRPA uses a function named *code*. This function is essential because it changes what the algorithm learns entirely. This function takes an action (and can also take a state) and returns a number corresponding to it. It can be viewed as an hash function for moves. It helps to store the weight inside the policy that can be seen as a list. The size of this list will be the number of code possible but that is just one way to see it or implement it.

Algorithm 3: The NRPA algorithm

```
1  NRPA(level, policy)
2  if level == 0 then
3  |  return playout(root, policy)
4  end
5  bestScore ← −∞
6  for N iterations do
7  |  (result, new) ← NRPA(level- 1, policy)
8  |  if result ≥ bestScore then
9  |  |  bestScore ← result
10 |  |  seq ← new
11 |  end
12 |  policy ← adapt(policy, seq)
13 end
14 return bestScore, seq
```

When we say that it changes the knowledge acquired by the NRPA it is undertone. Let's consider the Left Most Problem (LMP) as defined in section sub:mdp and examine two way of coding the actions.

$$code(a) = \begin{cases} 1 & \text{if } a = left \\ 0 & \text{otherwise} \end{cases}$$

In this first case, the code is produces from the action itself. Going to the left is 1 and going to the right is 0. This means that there is only 2 codes possible; therefore, their are two weights to adjust. The policy will be binary and only learn to go left or right. This is very adapted to the problem since the question does not change according to the number of turn already played.

$$code(a, s = (r, d)) = d * 2 + \begin{cases} 1 & \text{if } a = left \\ 0 & \text{otherwise} \end{cases}$$

In this second case, there are $2 \times N$ codes and the same number of weight to adjust. This means that, at each level of deepness, the algorithm will wonder if it is better to go left or right. If he knows that, at the root level, it is always better to go left, he does not apply it on other levels. This is a real problem in the modelization of the problem since it does not capture well what the algorithm have to learn. That said, even with this version, it probably find the best solution but in more time.

The codes are an essential part of the NRPA since they modelize the knowledge the algorithm is acquiring. Bad codes will result in bad learning and bad learning results in bad results.

4 Adapt the NRPA to stochastic problem

As explained in section 3, the Nested Rollout Policy Adaptation (NRPA) is an algorithm that obtains state of the art results on different optimization problem. All these problems have in common to be non stochastic problem. It means that if we apply action a on state s and we obtain the state s' then, when repeating the same process, we will always obtain s'. In a stochastic context, this assurance does not exist. Meaning that, when applying a on s we may obtain s' but also s^* or s'' or any others, depending on the problem.

In this section, we are going to present how we tried to adapt the NRPA to solve stochastic problem. We will first discuss a bit more what a stochastic problem is by considering a small example in order to highlight the difficulty to solve one. Then we will explain why the NRPA is not well suited to solve this kind of problem by identifying the main issues. Then, we will present how we attended to these issues by explaining our new algorithm the Stochastic Nested Rollout Policy Adaptation (SNRPA).

4.1 Stochastic problem and how to solve them

A stochastic problem, modelized as a Markov Decision Process (MDP) (see section 2.1) is a MDP in which $P_a(s, s') \leq 1$. Meaning that the transition between two states s and s' by action a is not guaranteed to succeed. Therefore, applying any action to any state may, each time, results in different state.

To be clearer, let's consider a small problem. The rules are simple:

- The game is played in a grid of dimension $W \times H$.
- The player starts on a random cell of the grid.
- Each turn the player can go *up* , *right* , *down* or *left*.
- When choosing a direction, the player may walk one, two or three cells in that direction with a unknown probability.
- The player can not exit the grid and is stuck at the edge if he tries.
- Each turn, the player freeze a bit and gain -1 point.
- An exit is placed on a random cell of the grid.
- The goal is for the player to start a turn on the exit.

Take the time to understand these rules, they are not very complex. A state of the game can be represented by the coordinates of the player. It is a stochastic or random game because, when applying any action to any states, we do not know which state we can obtain. Exception due to states where the player is on the edges of the grid and try to go beyond. Now imagine a state where the player is on the next next to the exit. If he chooses to go toward the exit he might go beyond it.

The difficulty of a stochastic problem is that a sequence S of n actions may not contain enough action to solve the game. Furthermore, many stochastic problems, once modelized as a MDP tends to posses a very large set of states. For some of them, if we were to see the entire tree of the game, the ramification may be too large. That is not say that for some non stochastic problem, the number of states is not immensely large. In the game of Go, which is not stochastic, the number of state is 3^{361} which is enormous. Some problem even have an infinity amount of states.

So the real problem is really the fact that sequence, when applied to a state may not always lead to terminal state. Also, in some problems, actions can be illegal so a sequence $S = a1, a2$ when applied from state s might lead to a final state s^* but also to another state for which th$a2$ is illegal. This poses a new problem as to how define a sequence in a stochastic context. For some problem, sequences that always lead from root state to a terminal one exists but, in general, it is unlikely. Furthermore, we are more interested in solving more difficult problem.

4.2 The NRPA to solve stochastic problem

Let's consider the NRPA as described in the section 3 and apply it to stochastic problem. Obviously, the NRPA is not suited for this king of problem. But we want to highlight the main issues to explain how we try to attend to them. In the rest of this section, we will separate each issue and open the discussion around them. We will also present different ideas that we had to fix them.

4.2.1 A sequence of action in stochastic context?

The first issue we pointed in section 4.1 is the fact that sequence are not certain anymore. The NRPA aims to find the best sequence possible that start from root to any terminal state. But if a sequence may not lead to the same outcome, or worst, happen to lead to illegal actions, it is a real problem.

The concept of sequence is therefore dangerous to use in stochastic problem as they tend to not have optimal sequence or policy.

4.2.2 Evaluation of a sequence

The NRPA evaluates its sequences by Monte-Carlo simulation in its playout (see section 3.1). It attaches the result of the simulation which follows a policy to the sequence it evaluates. Storing the best sequence it finds, the NRPA tends to be stubborn. Once it has found a good sequence, its policy will generate similar sequences. It might never find a better sequence because it will be stuck on exploitation of the best known sequence. This allows to perform very well in non stochastic problem as a sequence always produce the same results.

Evaluate a sequence of a stochastic problem might result in a high score just by fluke. If it happens, then the algorithm might stuck to this sequence which is just badly evaluate because the expected reward of the sequence may be very different.

4.2.3 Policy over action for a given state

The basic policy of the NRPA gives probability distribution over actions for every state. Of course, as seen in section 3.4 it really depends on the definition given to the *code* function. But, a stochastic problem may present states that are not often meet. You may very well encounter states one and only time due to the low probability in the transition between states and the size of the state space. Therefore, coding your action based on too much information from the state is a bad idea because the algorithm might not learn anything at all.

The codes must then contains global information on the current state. Even better, they should capture the nature of the action and its consequences on any state.

4.2.4 Summary

The NRPA is a very precise algorithm based on the fact that sequence can be replayed and always give the same results. It seems to lack a layer of abstraction necessary when dealing with stochastic problem. Indeed, these problems place the player in new state and situation they might never have encountered before. Therefore, their knowledge must, in some way, be useful to them in any given situation. That is a point the NRPA misses depending on how coded the actions are.

4.3 The Stochastic Nested Rollout Policy Adaptation

In this section, we will present our new algorithm called the SNRPA. It is an adaptation of the NRPA to solve stochastic problem. It tends to the issues we highlighted

in section 4.2 while still keeping the key concepts of the NRPA. In the rest of this section, we will present the different algorithm composing the SNRPA and will explain them.

4.3.1 SNRPA generate sequence

The NRPA generates its sequences through one and only one playout. Therefore it can generate and evaluate the sequence in the same time. We will see in section 4.3.2 that the SNRPA evaluates sequence differently. We then decided to split the two process.

In the SNRPA a sequence is an ordered list of codes containing all possible codes. A sequence is generated from a policy that define a weight for each possible codes. The sequence is created by random sampling without replacement so that the codes appear one and only once in each sequence.

The algorithm to generate the sequence is given in algorithm 4. The algorithm starts by initializing the sequence to build (line 2). Then it will repeat a random sampling on all the codes (line 3). To sample, it first compute the sum of exponential weights of all codes that are not yet in the sequence (line 5-9). Then it picks a code among the ones not in the sequence according to the exponential of the weight divided by the sum it just computed. Finally, it adds the chosen code to the sequence.

Algorithm 4: The SNRPA generate sequence

```
1  generateSequence(policy)
2  sequence ← []
3  for i ← 0 to numberCode do
4  |  z ← 0.0
5  |  for code in possible codes do
6  |  |  if code is not in sequence then
7  |  |  |  z ← z + exp(policy [code])
8  |  |  end
9  |  end
10 |  code ← choose a code not in sequence with
   |    probability exp(policy [code]) / z
11 |  sequence ← sequence + code
12 end
13 return sequence
```

We changed the concept of the sequence to attend to the issue presented in

section 4.2.1. A sequence being a list of all codes, it contains every possible actions in a way. We will see in section 4.3.2 that the way to use sequence has also changed.

4.3.2 SNRPA playout

The playout function is used to evaluate a sequence. To avoid any random high score, we compute an average for the sequence by computing its score on multiple playout. The score attached to the sequence is then its expected score.

The algorithm for the playout is given in algorithm 5. It starts by generating the sequence to evaluate (line 2). Then it continues by computing the average score for the sequence (line 3-4). It plays *nbPlayout* playout which is an hyperparameter of the SNRPA. At each turn, it finds the first legal code in the sequence for the current state. A code is legal if the action behind is legal and the information on the state the code contains are true for the current state. We give more details on codes in section 4.3.5. This may seem cryptic but we will give an example in section 5.2. It repeats the process till it reaches a terminal state. Finally, it returns the sequence with its expected score.

Algorithm 5: The SNRPA playout algorithm

```
1  playout(state, policy)
2  sequence ← generateSequence(policy)
3  sumScore ← 0
4  for i ← 0 to nbPlayout do
5  |  copyState ← state
6  |  while copyState is not terminal do
7  |  |  code ← choose the first legal code in
   |  |   sequence
8  |  |  move ← decode(code)
9  |  |  copyState ← play(copyState, move)
10 |  end
11 |  sumScore ← sumScore + score(copyState)
12 end
13 score ← sumScore/ nbPlayout
14 return (score, sequence)
```

4.3.3 SNRPA adapt policy

Adapting a policy to a sequence is a similar process in both the NRPA and the SNRPA. It increases the weight of codes at the beginning of the sequence and reduce the weights of codes at the end. To say it otherwise, the more to the left a code is in the sequence, the more its weight will increase. It firstly adapt the policy to the entire sequence before repeating the process without considering the first code and so on.

The adapt algorithm is given in algorithm 6. It iterates through the sequence index from 0 (the beginning) till the penultimate one (line 2). It then increases the weight of the current code in the sequence (line 3). Then computes the sum of the exponential of the weights of all codes to the right of the current code in the sequence (line 5-7). Finally it decreases the weight of the same codes according to the exponential of their weight divided by the sum (line 8-10).

Algorithm 6: The SNRPA adapt algorithm

```
1  adapt(policy, sequence)
2  for codeIndex ← 0 to sequence.size() - 2 do
3  |  policy [sequence [codeIndex]] ← policy [sequence
   |   [codeIndex]] + α
4  |  z ← 0.0
5  |  for codeIndexBis ← codeIndex to
   |   sequence.size() do
6  |  |  z ← exp(policy [sequence [codeIndexBis]])
7  |  end
8  |  for codeIndexBis ← codeIndex to
   |   sequence.size() do
9  |  |  policy [sequence [codeIndexBis]] ← policy
   |  |   [sequence [codeIndexBis]] −α *
   |  |   exp(policy [sequence [codeIndexBis]])
   |  |                    z
10 |  end
11 end
```

This function could be written as a recursive one calling itself once the update is done with the sequence minus its first code. Since the sequence contains all the codes, there is no need for a copy of the policy like in the adapt algorithm of the NRPA (see section 3.2).

4.3.4 SNRPA core function

The core function of the SNRPA is the same as the NRPA's one. The algorithm is still given in algorithm 7. Please see section 3.3 for more details on this function.

Algorithm 7: The SNRPA algorithm

```
1  SNRPA(level, policy)
2  if level == 0 then
3  |  return playout(root, policy)
4  end
5  bestScore ← −∞
6  for N iterations do
7  |  (result, new) ← SNRPA(level- 1, policy)
8  |  if result ≥ bestScore then
9  |  |  bestScore ← result
10 |  |  seq ← new
11 |  end
12 |  policy ← adapt(policy, seq)
13 end
14 return bestScore, seq
```

4.3.5 SNRPA code and decode

The SNRPA relies a lot on the *code* and a new *decode* functions. While the NRPA is on the level of states and actions, the SNRPA is on the code level. Therefore, it has a new layer of abstraction. It means that the codes must contain all the information needed from the states and actions while encapsulating them. A code must contain the consequences the action can have on the state. That is why we introduced a *decode* function that give the action hidden behind a code. We will give an example of action coding in section 5.2. Codes are still the tricky, problem-dependent part of the SNRPA.

5 Tactical Wildfire Management

The Tactical Wildfire Management (TWM) is an optimization problem that has many real life applications. As we have these last few years, giant forest fire are more and more vivid, frequent and dangerous. This problem modelize such a fire and the teams of firefighters that must extinguished it. We use the modelization presented by Bertsimas et al.[1]. As they say themselves, this problem is of practical use and is challenging but enough simple to test the Stochastic Nested Rollout Policy Adaptation (SNRPA).

We will first present the modelization of the TWM and we will proceed to the codification of the actions we used for the SNRPA.

5.1 Definition of the TWM problem

Our modelization is practically the same as the one defined by Bertsimas et al.[1] with minor changes. The TWM is set on a squared grid of cells X. Each cell $x \in X$ is defined by two attributes:

- $B(x)$ is a boolean indicating if the cell x is on fire.
- $F(x)$ is an integer representing the amount of fuel on cell x. The more fuel, the longer it burns.

A state is then represented by the values of these two attributes for all $x \in X$. A state is considered final when no cell is burning anymore.

We have a set T of firefighter teams which we consider identical. An action consists in placing each team $t \in T$ setting the attribute $a^i \in X$. Every team can be assigned to any cell and can even be assigned to the same cells.

A burning cell consumes its fuel at a constant rate. If a cell has no more fuel, it extinguishes. The burning rate constitutes the time unite of the game. Meaning that, each turn, all burning cells consume one fuel, therefore:

$$F_{t+1}(x) = \begin{cases} F_t(x) & \text{if } \neg B_t(x) \vee F_t(x) = 0 \\ F_t(x) - 1 & \text{otherwise} \end{cases}$$

The evolution of the attribute $B(x)$ is stochastic. It is set on a terminal state machine. This transition model is given by the following equation defining ρ_1 and ρ_2 and the figure 1.

$$\rho_1 = \begin{cases} 1 - \Pi_y(1 - P(x,y)B_t(x)) & \text{if } F_t(x) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\rho_2 = \begin{cases} 1 & \text{if } F_t(x) = 0 \\ 1 - \Pi_i(1 - S(x)\delta_x(a^i)) & \text{otherwise} \end{cases}$$

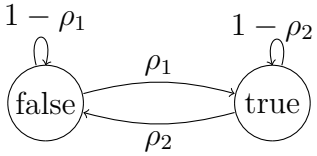

Figure 1: $B(x)$ transition model

$P(x, y)$ represents the probability that y ignites x if y is burning. We will only consider case where only neighbors cell can ignite each other. $S(x)$ is the probability that a firefighter team extinguish the cell x. $\delta_x(a^i)$ equals 1 if firefighter team i has been placed on cell x and 0 otherwise.

The reward $R(x)$ is always negative and is the cost of the fire on cell x. The reward at time t is therefore $\Sigma_{x \in X} B_t(x)R(x)$. The score of a final state is the sum of all rewards.

5.2 How code TWM actions

In this section we will discuss of how we codify actions for the TWM as defined in section 5.1. The TWM is a multi-agent problem. Each team of firefighter can be control on their own. We decided to have them play in sequence and not in parallel. Meaning that, we identify each team by a number and then, team 1 play, then team 2, etc…This way, the algorithm can place team one by one. It is important to do so because teams can be stacked on the same cell to extinguish it faster or can be spread across the grid to cover the fire.

Each cell $x \in X$ can be seen as a pair of coordinates (i, j). Each cell can then be associated to an index k such as $k = (j \times W + i)$. As said before each firefighter team $t \in T$ is associated to a number o which represents its turn order from 0 to $|T| - 1$. So we define the function *code* as follows:

$$code(x, s_n) = k_x + (o_n + |T|B_t(x)) * |X|$$

where x is the cell on which we want to send the next team $tinT$ and s_t is a state of the game at turn n.

With this definition a code holds three information:

- which cell $x \in X$ is targeted
- which team is send to the cell
- is the cell burning or not

Therefore, the SNRPA can make the difference between action for each team of firefighters and action targeting burning cells or not.

6 Experiments

We use a custom implementation in C++ of all algorithm.

We used almost the same setup of the first grid presented by Bertsimas et al.[1]. Only we considered one precise setup to compare multiple algorithms. We consider a Tactical Wildfire Management (TWM) problem on a 8×8 grid. We consider 2 firefighter team. The rewards of cells start at -1 on the bottom left corner and increase by the by -1 for each cell between a cell and the bottom left corner according to the manhattan distance. Only neighbors cell can ignite each other with a probability of 0.06. Firefighter teams have a 0.8 probability to extinguish any burning cell. We then follow the following procedure to obtain a root state

1. Initialize all the cells with 66 fuel.
2. Ignite the bottom left corner.
3. Let the fire propagate with no cost for 66 turn.
4. Scale down the amount of fuel on each cell by .06 to avoid too long computation

We use the same algorithm on 100 different root states, obtain by the previous procedure. We then aggregate the results together to have an average value. We used different algorithms:

- random: select a random action among legal ones every turn.
- Upper Confidence bound applied to Trees (UCT) with 60 seconds of search time and an exploration parameter c equal to 1.
- Generalized Rapid Action Value Estimation (GRAVE) with 60 seconds of search time, an *bias* of 0.01 and a *ref* equal to 50.
- Nested Rollout Policy Adaptation (NRPA) with a root level 2 and 25 iterations per level.
- Stochastic Nested Rollout Policy Adaptation (SNRPA) with a root level 2 and 25 iterations per level and 100 playount at level 0.

All algortihms are rerun every turn, allowing us to use NRPA by taking the first action in the returned sequence. We also included a version of the SNRPA called *srnpa_os* which is not rerun every turn but run only once on the root state. We then

take the returned sequence and use it just like in a SNRPA playout. This allow us to use the SNRPA with a level of 3 and 50 iteration per level with 100 playout at level 0 while still having a reasonable time of computation.

The figure 2 presents the aggragated results of a 100 run for each algorithm with the parameters presented before. The blue and yellow bar shows respectively the average highest reward obtainable from the root state and the average lowest reward obtainable from the root state. These two values are computed before running the algorithm by considering the best case scenario and the worst one.

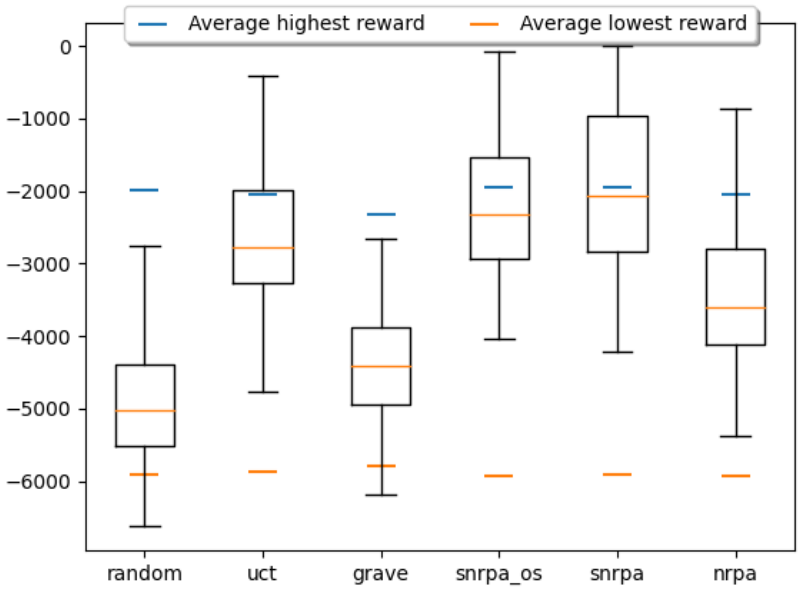

Figure 2: Results on 100 simulation for different algorithm as described in section **??** for different algorithms

The results show that the SNRPA performs significately better than other algorithm. We find strange that GRAVE algorithm performs so poorly and we extend the possibility of a non-wanted behaviour due to a code mistakes but we could not find one. The NRPA still performs better than the random algorithm which suggest that good sequences might be close to what the SNRPA find.

We wanted to see how the number of playout at level 0 impact the performance of the SNRPA. We run, on the same setup as experiment 1 (see section **??**), 100 times the SNRPA with the same parameters. Only one had 100 playout at level 0

and the other had 1000.

The results are shown in figure 3. It shows that a bigger number of playouts tends to improve vastly the performance of the algorithm for a computation cost that does not increase that vastly. The playout of the SNRPA are much simpler than the playout of the NRPA and are computed much more faster.

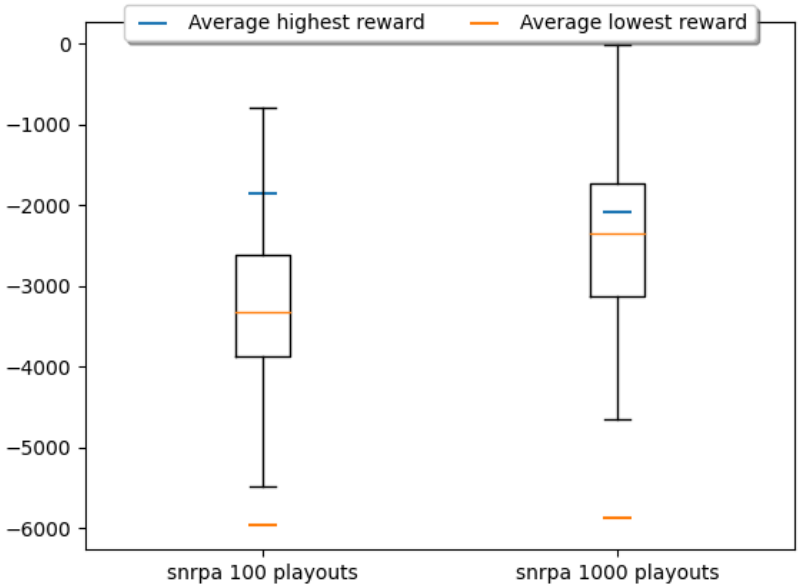

Figure 3: Comparison between a SNRPA with 100 playout at level 0 and a SNRPA with a 1000

The SNRPA seems to perform very well on this setup of grid. It would be interesting to test it on other setup and problems, even on non stochastic ones where the NRPA perform very well. By lack of computational power and time we will most likely presents these tests later on another paper.

7 Conclusion

In this paper, we presented the NRPA, a Monte Carlo Tree Search (MCTS) algorithm, giving state of the art results on many optimization problem. We presented the SNRPA, an adaptation of the NRPA to solve stochastic problem. Our results show that the SNRPA performs very well on the TWM problem. Future works include the test on other stochastic problems and on non stochastic problems to compare the SNRPA to the NRPA on territory dominated by the NRPA. We would also want to present more computational demanding results to see if the gap of performance between the algorithms is more significant.

References

[1] Dimitris Bertsimas et al. "A Comparison of Monte Carlo Tree Search and Mathematical Optimization for Large Scale Dynamic Resource Allocation". In: (May 2014).

[2] C. B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[3] Bernd Brügmann. "Monte carlo go". In: (Nov. 1993).

[4] Tristan Cazenave. *Generalized Nested Rollout Policy Adaptation*. 2020. arXiv: 2003.10024 [cs.AI].

[5] Tristan Cazenave and Fabien Teytaud. "Application of the Nested Rollout Policy Adaptation Algorithm to the Traveling Salesman Problem with Time Windows". In: *Learning and Intelligent Optimization*. Ed. by Youssef Hamadi and Marc Schoenauer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 42–54. ISBN: 978-3-642-34413-8.

[6] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Proceedings of the 17th European Conference on Machine Learning*. ECML'06. Berlin, Germany: Springer-Verlag, 2006, pp. 282–293. ISBN: 354045375X. DOI: 10.1007/11871842_29. URL: https://doi.org/10.1007/11871842_29.

[7] Christopher Rosin. "Nested Rollout Policy Adaptation for Monte Carlo Tree Search." In: Jan. 2011, pp. 649–654. DOI: 10.5591/978-1-57735-516-8/IJCAI11-115.

[8] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.