Planning Algorithms: An Overview

Zamboni Riccardo

e-Novia, Italy, ricc.zamboni@gmail.com

Keywords: Planning, MCTS, UCT, OLOP, UGapE

Abstract

Planning algorithms are gaining increasing interest, as they are a powerfull framework to deal with decision processes, and consist of a nice intersection between games and more general reinforcement learning, leading to super-duper results as well [1]. Here is a list of interesting results.

1 Introduction

The decision making problem in planning consists in an exploration phase followed by a recommendation. First, the agent explores freely the set of possible sequences of actions, (using a budget of n actions, virtually). Then the agent makes a recommendation on the first action to play. The goal of the agent is to find the best way to explore its environment (first phase) so that, once the available resources have been used, he is able to make the best possible recommendation on the action to play in the environment (second phase). Starting from a simple RL framework to make things more clear, a planning is conceived as when an agent uses some generative model of the environment to acquire more knowledge about the best policy to be followed, and then act accordingly. Planning algorithms are based on many different settings, and they can provide open-loop policies (independent from the state in which the agent is), and closed-loop policies. We will take into account various settings for the planning framework, starting from the most common formulation of the problem, the multi-armed bandit case.

2 Introduction: Multi-Armed Bandits

2.1 Overview

The main question is: What is the best strategy to achieve highest long-term rewards? This is the main question as RL, but imagine there are actually no states, and the reward itself coincides with a state. This scenario is well represented by playing to several slot machines, whose reward is unknown a priori.

A naive approach can be that you continue to playing with one machine for many many rounds so as to eventually estimate the "true" reward probability according to the law of large numbers. However, this is quite wasteful and surely does not guarantee the best long-term reward, having focused for so long on a potentially sub-optimal, or even not optimal choice.

A multi-armed bandit can be described as a tuple of $\langle \mathcal{A}, \mathcal{R} \rangle$, where:

- We have K machines with reward probabilities, $\theta_1, \cdots, \theta_K$
- At each time step t, we take an action a on one slot machine and receive a reward r.
- A is a set of actions (i.e. pulling one specific arm). The value of action a is the expected reward, $Q(a) = E[r|a] = \theta$. If action at the time step t is on the i-th machine, then $Q(a_t) = \theta_i$
- R is a **reward function**. In the case of Bernoulli bandit, we observe a reward r in a stochastic fashion. At the time step t, $r_t = R(a_t)$ may return reward 1 with a probability $Q(a_t)$ or 0 otherwise.

As we said, this is a simplified version of Markov decision process, as there is no state S, and yet it well describes many decision processes. If we knew the optimal action with the best reward, then the goal would be to minimize the potential regret or loss by not picking the optimal action. The optimal reward probability θ^* of the optimal action a^* is:

$$\theta^* = Q(a^*) = \max_{a \in A} Q(a) = \max_{1 \leq i \leq K} \theta_i$$

Our loss function is the total regret we might have by not selecting the optimal action up to the time step T:

$$\mathcal{L}_T = E[\sum_{t=1}^{T} (\theta^* - Q(a_t))]$$

2.2 Upper Confidence Bounds and UCB1

An usefull concept which is often taken into account when dealing with planning and Bandits is the one of the confidence bounds, and in particular the upper one. Random exploration make us to try out options that we have not known much about. But we may end up trying a bad action which we already knew about as well. To avoid such inefficient exploration, one approach is to be optimistic about options with high uncertainty and thus to prefer actions for which we haven't had a confident value estimation yet. Or in other words, we favor exploration of actions with a strong potential to have a optimal value (optimism).

The **Upper Confidence Bounds** (UCB) algorithm measures this potential by an upper confidence bound of the reward value,$U^t(a)$, so that the true value is below with bound $Q(a) \leq Q^t(a)+U^t(a)$ with high probability. The upper bound $U^t(a)$ is a function of $N_t(a)$ as well. A larger number of trials $N_t(a)$ should give us a smaller bound $U^t(a)$.

In UCB algorithm, we always select the greediest action to maximize the upper confidence bound:

$$a_t^{UCB} = argmax_{a \in A} Q^t(a) + U^t(a)$$

The critical part is how to estimate the upper confidence bound. We will not take into account the Bayesian case (we do not want to assign any prior knowledge on how the distribution looks like and we do not use inference-like steps as in Thompson Sampling). However, we can get help from **Hoeffding's Inequality**, a theorem applicable to any bounded distribution (as in the Bernoulli case).

Let $X_1, \cdots, X_t$ be i.i.d. (independent and identically distributed) random variables and all bounded. The empirical mean is $\bar{X}_t = \frac{1}{t}\sum_{\tau=1}^{t} X_\tau$. Then for $u > 0$, we have:

$$\mathcal{P}[E[X] > \bar{X}_t + u] \leq e^{-2tu^2}$$

Given one target action a, let us consider:

- $r_t(a)$ as the random variables,
- $Q(a)$ as the true mean,
- $Q^t(a)$ as the sample mean,
- u as the upper confidence bound, $u = U_t(a)$

Then we have,

$$\mathcal{P}[Q(a) > Q^t(a) + U_t(a)] \leq e^{-2tU_t(a)^2}$$

We want to pick a bound so that with high chances the true mean is blow the sample mean + the upper confidence bound. Thus $e^{-2tU_t(a)^2}$ should be a small probability p:

$$e^{-2tU_t(a)^2} = p \text{ Thus, } U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

It is a smart choice to reduce the threshold p in time, as we want to make more confident bound estimation with more rewards observed. Set $p = t^{-4}$ and we get **UCB1 Algorithm** as discussed in [2]:

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}} \text{ and } a_t^{UCB1} = \arg\max_{a\in\mathcal{A}} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}$$

3 Monte-Carlo Tree Search and UCT

In the development of planning algorithms, the approach based on the Monte-Carlo exploration has gained much attention. In general, Monte Carlo tree search (MCTS) algorithms employ strong search methods to identify the best available action in a given situation. They gain knowledge by simulating the given problem and thus require at least its generative model. The MCTS planning process incrementally builds an asymmetric search tree and it improves when increasing the number of iterations. An MCTS iteration usually consists of four consecutive phases:

1. selection of actions already memorized in the tree (descent from the root to a leaf node) with a **tree policy**
2. expansion of the tree with new nodes with a **action-selection policy**
3. playout, i.e., selection of actions until a terminal state is reached with a **default policy**
4. **backpropagation** of the feedback up the tree

After the planning, the starting move leading most frequently to a win is played. The efficiency of Monte-Carlo search heavily depends on the way moves are sampled during the episodes. A pseudo-code for Monte-Carlo Search is showed in 1.

```
procedure MonteCarloSearch(position)
repeat
    search(position, rootply)
until Timeout
return bestMove(position);

function search(position, depth)
if GameOver then
    return GameResult
end if
m := selectMove(position, depth);
v := −search(position after move m, depth + 1);
Add entry (position, m, depth, v, . . .) to the TT
return v;
```

Figure 1. MCTS Pseudo-Code

UCT is a Monte-Carlo search algorithm with a specific action-selection policy based on the UCB1 algorithm, modelling the action-selection problem as a separate multi-armed bandit for every (explored) internal node. In order to find the best move in the root, we have to determine the best moves in the internal nodes as well (at least along the candidate principal variations). Since the estimates of the values of the alternative moves rely on the estimates of the values of the (best) successor nodes, we must have small estimation errors for the latter ones. The problem reduces to getting the estimation error decay quickly. In order to achieve this, the algorithm must balance between exploration and exploitation, as guaranteed from the UCB1 framework. The pseudo-algorithm of the UCT is showed in 2 as from [3].

4 Temporal Differences Extensions and SARSA-UCT(λ)

It is possible to include temporal differences formalisms in the MCTS framework. In particular, it is possilbe to make assumptions on the missing values of playout states (and actions) in a similar way as assumptions can be made on the optimistic initial values $V_{init}(s)$ and $Q_{init}(s; a)$. We introduce the notion of a **playout value function**, which replaces the missing value estimates in the non-memorized part of the state space

```
function selectMove(position, depth)
nMoves := # available moves in position
nsum := 0 {nsum will contain # times the descendants of position are considered}
for i := 1 to nMoves do
   Let tte[i] be the TT entry matching the ith descendant of position
   if the entry tte[i] is invalid then
      return random move in position
   end if
   nsum := nsum + tte[i].n {tte[i].n = # times the ith descendant is considered}
end for
maxv := −∞;
for i := 1 to nMoves do
   if tte[i].n = 0 then
      v := +∞ {Give high preference to an unvisited descendant}
   else
      v := tte[i].value + √(2 ∗ ln(nsum)/tte[i].n)
   end if
   if v > maxv then
      maxv := v
      Let m be the ith move
   end if
end for
return m
```

Figure 2. UCT Pseudo-Code

with playout values $V_{playout}(s)$ and $Q_{playout}(s;a)$. The focus on the non-memorizing part as well comes from the fact that this is a peculiarity of the MCTS framework, not represented in the RL one. Then, a TD back-up can be introduced in the usual MCTS case as shown in the pseudo-code 3, taken from [4].

```
procedure BACKUPTDERRORS(episode)
   δ_sum ← 0                                   ▷ cumulative decayed TD error
   V_next ← 0
   for i = LENGTH(episode) down to 1
      (s, R) ← episode(i)
      if s is in tree
         V_current ← tree(s).V
      else                                     ▷ assumed playout value
         V_current ← V_playout(s)
      δ ← R + γV_next − V_current               ▷ single TD error
      δ_sum ← λγδ_sum + δ                       ▷ decay and accumulate
      if s is in tree                          ▷ update value
         tree(s).n ← tree(s).n + 1
         α ← 1/tree(s).n                       ▷ example of MC-like step-size
         tree(s).V ← tree(s).V + αδ_sum
      V_next ← V_current                       ▷ remember the value from before the update
```

Figure 3. TD Backup Pseudo-Code

5 The other side of the coin: Best-Arm Identification and UGapE

As we have seen planning methods make often use of a Monte-Carlo formalism. Yet MC backups might lead to an update with high variance, due to the long run of the simulation, moreover such methods often assume stationarity of the reward distributions over the tree, which might be not always the case. On the other side of the coin, in order to take the best decision at the root, it might be sufficient to be able to select simply the best action at the root itself. This problem falls under the class of best-arm identification. A proper planning is actually going to balance between this two needs, of a long run outcome (Monte Carlo), and of the short-run outcome (Best-Arm selection). We will briefly discuss the latter case. Unlike the standard multi-armed bandit problem, where the goal is to maximize the cumulative sum of rewards obtained by the planner, in this problem the planner is evaluated on the quality of the arm(s) returned at the end of the exploration phase. This problem has been studied in the literature from two different perspectives: fixed budget and fixed confidence, and the best choice can be given with some fixed confidence ϵ over a subset of the possible arms m, leading to the so-called (ϵ, m)-optimality. The definitions are the same as the standard multi-armed settings, with the expeption of the extension to m best- arm mean and simple regret of the actions in a set S with respect to the m best arms:

$$\mu_{(m)} = \max_{k \in A}^{m} \mu_k, \; r_m = \max_{k \in A}^{m} r_k = \mu_{(m)} - \min_{k \in S} \mu_k$$

In the best arm case, by assuming boundedness of the distributions of the rewards of each action, the **UGapE** algorithm was shown to be particularly effective [5], both in a fixed budget (sampling efficiency) and fixed confidence (simple regret efficiency) case. One of the upsides of this algorithm is that its core structure is actually the same in both cases, and the selection function is the one adapted to the objective. After having played each arm at least once, the next choices follow a **selection function** based on the upper confidence concept. At each time step t, UGapE first uses the observations up to time $t-1$ and computes an upper confidence index $B_k(t) = \max_{i \neq k}^{m} U_i(t) - L_k(t)$ or each arm $k \in A$, where

$$U_k(t) = \hat{\mu}_k(t-1) + \beta_k(t-1), \; L_k(t) = \hat{\mu}_k(t-1) - \beta_k(t-1)$$

where $\beta_k(t-1)$ is a confidence interval specific for the fixed budget and fixed confidence cases, but it is pretty similar to the UCB case, and $U_k(t)$ and $L_k(t)$ are high probability upper and lower bounds on the mean of arm k, μ_k, after $t-1$ rounds.

The index $B_k(t)$ is an upper-bound on the simple regret r_k of the kth arm. An additional index for a set S can be defined as $B_S(t) = \max_{i \in S} B_i(t)$. Similar to the arm index, B_S is also defined in order to upper-bound the simple regret r_S with high probability.

After computing the arm indices, UGapE finds a set of m arms $J(t)$ with minimum upper-bound on their simple regrets. From $J(t)$, it computes two arm indices

$$u_t = arg \max_{j \notin J(t)} U_j(t), \; l_t = arg \min_{i \in J(t)} L_i(t)$$

where in both cases the tie is broken in favor of the arm with the largest uncertainty β.

Arms l_t and u_t are the worst possible arm among those in $J(t)$ and the best possible arm left outside $J(t)$, respectively, and together they represent how bad the choice of $J(t)$ could be.

Finally, the algorithm selects and pulls the arm $I(t)$ as the arm with the large uncertainty β among the two, observes a sample $X_I(t)$ from the related distribution of reward of the arm, updates the mean and the number of the pulls for the selected arm, and so on.

This is repeated until a termination condition is met.

6 Open Loop Planning: OLOP

While the previous settings can be called closed loop settings, because the action depends somehow on the states, there is another possible case, called open-loop planning, where the class of considered policies (i.e. sequences of actions) are only function of time (and not of the underlying resulting states). This open-loop planning is in general sub-optimal compared to the optimal (closed-loop) policy (mapping from states to actions). However, here, while the planning is open-loop, the resulting general policy is closed-loop (since the chosen action depends on the current state). In such cases, optimism has been shoown as extremely effective as well. In the case of open-loop planning, upper confidence bounds are assigned to all sequences of actions, and the exploration expands further the sequences with highest UCB. In this framework, it is critical to understand that the value of a sequence of action being defined as the sum of discounted rewards along the path, thus the rewards obtained along any sequence provides information, not only about that specific sequence, but also about any other sequence sharing the same initial actions. The OLOP algorithm [6] is designed to use this property as efficiently as possible, to derive tight upper-bounds on the value of each sequence of actions. In particular, since the value of a sequence is the sum of discounted rewards, one would like to explore more intensively the sequences starting with actions that already yielded high rewards. OLOP proceeds as follows. It assigns upper confidence bounds (UCBs), called B-values, to all sequences.of actions, and selects at each round a sequence with highest B-value. At time $m = 0$, the B-values are initialized to $+1$ to guarantee optimistic exploraion. Then, after episode $m \geq 1$, the B-values are defined as follows: For any $1 \leq h \leq L$, for any $a \in A^h$, let

$$T_a(m) = \sum_{s=1}^{m} \mathcal{I} a_{1:h}^s = a$$

be the number of times we played a sequence of actions beginning with a. Now we define the empirical average of the rewards for the sequence a as:

$$\hat{\mu}_a(m) = \frac{1}{T_a(m)} \sum_{s=1}^{m} Y_h^s \mathcal{I} a_{1:h}^s = a$$

if $T_a(m) > 0$, and 0 otherwise. The corresponding upper confidence bound on the value of the sequence of actions a is by definition:

$$U_a(m) = \sum_{t=1}^{h} (\gamma^t \hat{\mu}_{a_{1:t}}(m) + \gamma^t \sqrt{\frac{2logM}{T_{a_{1:t}}(m)}}) + \frac{\gamma^{h+1}}{1-\gamma}$$

if $T_a(m) > 0$ and $+1$ otherwise. Now that we have upper confidence bounds on the value of many sequences of actions we can sharpen these bounds for the sequences by defining the B-values as:

$$B_a(m) = \inf_{1 \leq h \leq L} U_{a_{1:h}}(m)$$

At each episode with highest B–value, observes the rewards Y_t^m provided by the environment, and updates the B–values. At the end of the exploration phase, OLOP returns an action that has been the most played.

References

[1] D. Silver *et al.*, "Mastering the game of go without human knowledge,"

[2] L. Weng. `https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutio` `html`.

[3] L. Kocsis *et al.*, "Improved monte-carlo search."

[4] T. Vodopivec *et al.*, "On monte carlo tree search and reinforcement learning."

[5] V. Gabillon *et al.*, "Best arm identification: A unified approach to fixed budget and fixed confidence."

[6] S. Bubeck *et al.*, "Open loop optimistic planning."