0.1 Monte Carlo Search

Monte Carlo Search (MCS) is a family of search techniques that have had much success, particularly within the fields of games, planning and optimization. Generally speaking, MCS is a best-first search strategy that combines random sampling in order to determine an optimal action given a specific domain. In order to describe the ideas of MCS it is useful to be able to describe the domain concretely. To this extent, it is necessary to define the *search domain* (**Undefined reference**), this notation will be used throughout the remainder of this thesis.

The search domain $\mathbb{S}$ is defined as a 5 - tuple $=S, S_T, A, f, R$ where:

- S - the set of states.
- $S_T \subseteq S$ - the terminal states.
- A - the possible actions.
- $f : S \times A \to S$ - the state transition function.
- $R : S \to \mathbb{R}^k$ - the utility function.

Within the domain, an *agent* progresses between states by performing an action.The state $s_0 \in S$ is the unique starting state. At each state s_i, an action $a \in A$ is chosen and the agent progresses to the next state $s_{i+1} = f(s_i, a)$. After performing an action, the agent recieves a reward determined by the utility function R. Though an agent can obtain a reward for entering any state, rewards are usually only defined for terminal states. These rewards reflect the utility of a certain state and are generally normalized to values between $[0, 1]$. Since it is typically the case that not all actions are valid within a particular state, one can define a function $\alpha : S \to 2^A$ which given a state s returns all *legal* actions within that state.

Within the context of search, the idea is to find a sequence of actions $(a_0, a_1, ..., a_t)$ that leads to a final state $s \in S_t$ such that $R(s)$ is maximized. Such a sequence is thus called an *optimal strategy* or *policy*. Algorithms such as MCS thus aim to provide approximations for an optimal strategy by estimating the true value of an action through random simulation and then using the actions with the best values in the strategy. The simplest example of MCS called *Flat Monte Carlo* is illustrated in figure 1. Given a starting state, the optimal action to play is determined by, for each possible actions, making random playouts until a terminal state is reached, evaluating those states, and consequently propogating that value back through the tree. The optimal action is then the one which has the largest propogated value.

Algorithm 1 Flat MCS

```
1:  procedure FlatMCS(s)
2:      max ← −∞
3:      for a ∈ α(s) do
4:          val ← R(randomRollout(f(s, a)))
5:          if val > max then
6:              a_best ← a
7:              max ← val
8:      return a_best
9:  procedure randomRollout(s)
10:     s_cur ← s
11:     while s_cur ∉ S_T do
12:         s_cur ← f(s_cur, getRandom(α(s_cur)))
13:     return s_cur
```

The algorithm described in 1 is highly dependant on the random rollouts and thus it is highly likely that the action obtained is not the optimal action. MCS techniques, in general, thus take advantage of repeated simulations in order to obtain more valid approximations to the values of the various actions.

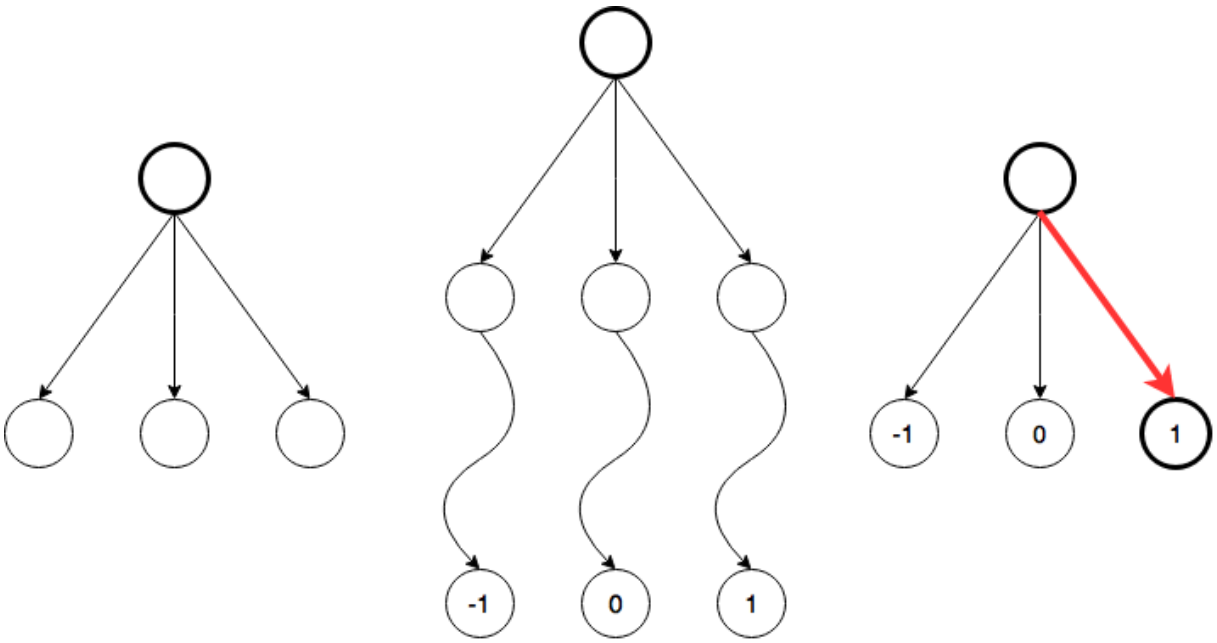

Figure 1: *Example of flat MCS. The right-most action is rated as the best and thus the player chooses to play it.*

0.1.1 Nested Monte Carlo Search

Nested Monte Carlo Search (NMCS) is a variant of MCS that uses recursive layers of random simulations in order to determine an optimal move(**Undefined reference**). This algorithm (see Algorithm 2) is typically used for 1-player games and thus instead of searching for the best action at a given state, it searches for the whole policy $\pi_{nmcs} = \{s_0, \dots, s_t\}$ [1] where s_0 is the initial state and s_t is a possible terminal state. The level n of NMCS defines the depth of recursion that is used in the search. For example a level-0 NMCS algorithm obtains the policy just by performing random rollouts from the initial state until it reaches a terminal state, lines 6 - 10 in the algorithm. NCMS defines utility in terms of policies $R(\pi_{ncms}) := R(s_t)$, i.e, the utility of a policy is the utility of its terminal state. Therefore, NCMS effectively searches to directly find the most optimal terminal state. At higher levels, a level n NCMS algorithm searches for the optimal policy by, for each possible successor state, determining the optimal policy through a level $n-1$ NCMS, lines 14 - 19. The state corresponding to the best optimal policy is chosen as the next state, and the algorithm continues. This idea is illustrated in Figure 2.
Nested Monte Carlo Search has proven to be an effective algorithm within the domain of single player games such as Morpion, Same Game and Crossword puzzle generation(**Undefined reference**). It is because of this success that it is believed that NMCS would be suitable for the problem of first species counterpoint generation.

0.1.2 Nested Rollout Policy Adaptation

Nested Rollout Policy Adaptation (NRPA) is a method based on NMCS created by Rosin (**Undefined reference**). The algorithm is similar in the sense that it performs nested searches and at level - 0 performs a full playout. The main difference is that, apart from directly traversing the search tree based on the results of previous nested calls, NRPA adapts the rollout policy based on the scores obtained from the nested calls. The idea is that, as search progresses, the policy is updated such that good or promising moves are given increasing probabilities. The algorithm is described in 3.

The algorithm is called with two parameters, n and p, the search level and the initial *policy vector*. The policy vector represents the likelihood of an action being selected at a specific state. Therefore, the policy vector must contain a value for each action at each state. These state action pairs are encoded

[1] Here, some liberty is taken in deviating from the definition of the policy π as a function. Since NCMS deals with single-player games, it is sufficient to describe π as a set of states

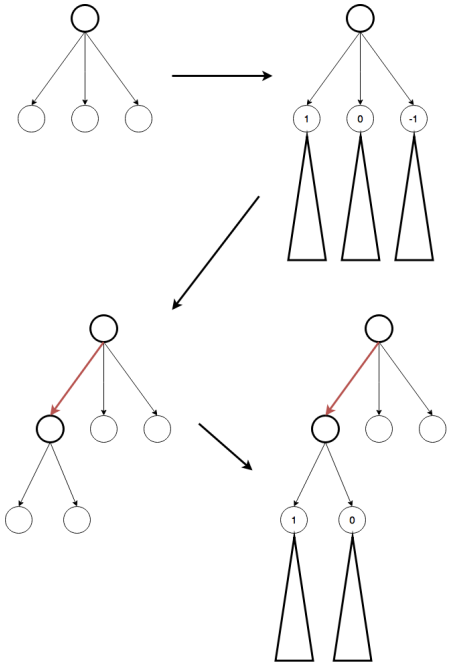

Figure 2: *Illustration of the n-level NMCS algorithm. The triangles represent an n-1 level NMCS search.*

Algorithm 2 Level-n NMCS

```
1:  procedure NCMS(n, s)
2:      ply ← 0
3:      π ← ∅
4:      if n = 0 then
5:          π ← {s}
6:          while s ∉ S_T do
7:              s ← f(s_cur, GETRANDOM(α(s)))
8:              π.ADD(s)
9:              ply ← ply + 1
10:         return (π, R(π))
11:     else
12:         max ← −∞
13:         while s ∉ S_T do
14:             for a ∈ α(s) do
15:                 (π_temp, score) ← NCMS(n − 1, f(s, a))
16:                 if score > max then
17:                     max ← R(π_temp)
18:                     π.ADD(s, ply)
19:                     π.ADD(π_temp, ply + 1)
20:             s ← π.GET(ply)
21:         return (π, R(π))
```

using the function CODE(s, s'), where s represents the current state and s' is the resultant state after applying a specific action. At level 0, NRPA performs a random playout where each action is chosen proportional to the values in the policy vector, lines 5 - 9. At higher levels, the algorithm searches for N results from $n-1$ level searches, choosing the best sequence based on its evaluation, lines 12 - 17. The current best sequence is then used to update the values in the policy vector using the ADAPT(p, π) procedure. For all states in the sequence, their weight in the policy vector is increased by a set amount

Algorithm 3 Level-n NRPA

procedure NRPA(n, p)
  if n $= 0$ **then**
    $s \leftarrow \text{root}(), ply \leftarrow 0, \pi \leftarrow \emptyset$

    while $s \notin S_T$ **do**
      $s \leftarrow f(s, \text{GETRANDOM}(\alpha(s), p)$
      $\pi.\text{ADD}(s)$
      $ply \leftarrow ply + 1$
    return $(\pi, R(\pi))$
  else
    $max \leftarrow -\infty$
    for N iterations **do**
      $(\pi_{temp}, score) \leftarrow NRPA(n-1, p)$

      if $score \geq max$ **then**
        $max \leftarrow score$
        $\pi \leftarrow \pi_{temp}$

      $p \leftarrow \text{ADAPT}(p, \pi)$
    return $(\pi, R(\pi))$

procedure ADAPT(p, π)
  $s \leftarrow \text{root}(), ply \leftarrow 0, p' \leftarrow p$

  while $ply < \pi.\text{LENGTH}()$ **do**
    $s' \leftarrow \pi.\text{GET}(ply)$
    $p'[\text{CODE}(s, s')] \leftarrow p'[\text{CODE}(s, s')] + \beta$

    $z \leftarrow \sum_{a \in \alpha(s)} \exp(p[code(s, f(s, a))])$

    for $a \in \alpha(s)$ **do**
      $p'[code(s, f(s, a))] \leftarrow p'[code(s, f(s, a))] - \frac{\beta}{z} \exp(p[code(s, f(s, a))])$

    $s \leftarrow s'$
  return p'

β.