2018

Game Changing Technology

COMP3702 – Assignment 2

Team WLS

1. Markov Decision Process (MDP) Problem

As the agent knows the state exactly, this is not a partially observable MDP problem.

1.1 State Space S

$S =${cell location $L = [1\,..\,N]$, movement situation $M = \{\text{moveable, slip, breakdown}\}$, car C, driver D, tire type TI, tire pressure $TP = \{50\%, 75\%, 100\%\}$, terrain TE, fuel $F = [0\,..\,50]$, time $TM = [0\,..\,maxT]$}

1.2 Action Space A

$A =${move, where the distance moved (k) is not certain but $k \in A_1$ and $A_1 = [-4\,..\,5] \cup \{\text{slip}, \text{breakdown}\}$, change car A_2, change driver A_3, change tire type A_4, add fuel A_5, change tire pressure A_6, $A_7 = A_2 \cap A_3$, $A_8 = A_4 \cap A_5 \cap A_6$}

Note that A_8 is optional for COMP3702 and thus omitted in further discussions. If the required fuel usage for the moving action is greater than the amount of fuel at current state, then the action will be truncated.

1.3 Transition Function T

$$T(s', a, s) = P(s'|s \cap a)$$

Transitions of state components except cell location and movement situation are deterministic, for instance $a \in A_2$ will result in the definite change of C to a particular value in the next state. For the indeterministic parts with $m \in M$, $l \in L$ and $k \in A_1$:

$$l = l + k, m = \text{moveable}, k \notin \{\text{slip}, \text{breakdown}\}$$

$$m = k, k \in \{\text{slip}, \text{breakdown}\}$$

It has been assumed that conditional independence of C, D and TI given k and prior distributions of those parameters and of k are uniform. Additionally, if k is not slip, then it has been assumed that the probability with respect to TP and TE is uniform. With car c, driver d, tire type ti, tire pressure tp and terrain te:

$$P(s'|s \cap a) = \begin{cases} 1, a \in A - \{A_1\} \\ P(a = k|C = c, D = d, TI = ti, TP = tp, TE = te), a \in \{A_1\} \end{cases}$$

For readability, let $P_{movek} = P(a = k|C = c, D = d, TI = ti, TP = tp, TE = te)$. Then:

$$P_{movek} = \frac{P(C = c|a = k)P(D = d|a = k)P(TI = ti|a = k)P(TP = tp, TE = te|a = k)P(a = k)}{\sum_{k \in A_1} P(C = c|a = k)P(D = d|a = k)P(TI = ti|a = k)P(TP = tp, TE = te|a = k)P(a = k)}$$

According to Bayes' Theorem, some of the terms can be directly calculated from input information:

$$P(C = c|a = k) = \frac{P(a = k|C = c)P(C = c)}{P(a = k)}$$

$$P(D = d|a = k) = \frac{P(a = k|D = d)P(D = d)}{P(a = k)}$$

$$P(TI = ti|a = k) = \frac{P(a = k|TI = ti)P(TI = ti)}{P(a = k)}$$

, where $P(C = c) = \frac{1}{|C|}$, $P(D = d) = \frac{1}{|D|}$, $P(TI = ti) = \frac{1}{|TI|} = \frac{1}{4}$ and $P(a = k) = \frac{1}{|A_1|} = \frac{1}{12}$. TP and TE affects slip cases, thus with the input information $P(a = \text{slip}|TE = te, TP = 50\%)$:

$$P(a = \text{slip}|TE = te, TP = 75\%) = 2 * P(a = \text{slip}|TE = te, TP = 50\%)$$

$$P(a = \text{slip}|TE = te, TP = 100\%) = 3 * P(a = \text{slip}|TE = te, TP = 50\%)$$

$$P(a = k|TE = te, TP = tp) = \frac{1 - P(a = \text{slip}|TE = te, TP = tp)}{|A_1| - 1}, k \neq \text{slip}$$

, and the probability associated with TP and TE is:

$$P(TP = tp, TE = te|a = k) = \frac{P(a = k|TE = te, TP = tp)P(TE = te, TP = tp)}{P(a = k)}$$

, where $P(TE = te, TP = tp) = P(TE = te) = \frac{1}{|TE|}$ since probabilities for all three pressures are known.

Moreover, some values of k may cause out-of-bound issues, therefore the corresponding probabilities will be added to the probability of the extrema k (i.e. leading to the edge location l, either 1 or N) at that state instead.

1.4 Reward Function R

$$R(s) = \begin{cases} w * (maxT - t), s = goal \\ 0, s \neq goal \end{cases}$$

, where: $s \in S$; w is a positive constant; and t is the cumulative time required to reach the current state, which can consist of 1 for normal forward/backward moving, fuel amount divided by 10 for adding fuel, and specified recovery durations for slip and breakdown respectively.

2. Methodology

Aiming for level 4, an online method Monte Carlo Tree Search (MCTS) is used due to the size of A. The overall data structure is an AND-OR Tree, with OR nodes being states and with AND nodes being actions. An OR node has AND node children according to A, while an AND node has OR node children in terms of T. Four major components of this algorithm and their integration are discussed below.

2.1 Selection

In the Upper Confidence Bounds applied to Trees algorithm shown below, $\boldsymbol{Q(s,a)}$ is the exploitation term and $\sqrt{\frac{\ln(\mathrm{Ns})}{\mathrm{N(s,a)}}}$ is the exploration term, where $\boldsymbol{Q(s,a)}$ is the expected reward received from this state ***s*** with action ***a***, $\boldsymbol{N(s,a)}$ is the number of times action ***a*** has been visited and ***Ns*** is the number of times this node has been visited. Thus, actions with higher reward or fewer visits are preferred in selection, where ***e*** is the tunable constant between them.

Algorithm 1: Select Action (member of OR node, denoted as select())

Input: Hash Map $MANQ$ (as a member of OR node), with key as action a and with value in tuples $(N(s,a), Q(s,a))$; Exploration factor e; Number of visits of the OR node Ns (as a member of OR node)

```
for all the a ∈ MANQ.keySet() do
        pair ← MANQ.get(a)
        N(s,a) ← pair[0]                // first value of the tuple
        if N(s,a) = 0 then              // ensure every child has been visited at least once, so return unvisited child
                return a
        end if
        aMax ← ∅
        uct ← −1
        Q(s,a) ← pair[1]                // second value of the tuple
        pi ← Q(s,a) + e * sqrt(ln(Ns) / N(s,a))
        if pi > uct then
                aMax ← a
                uct ← pi
        end if
end for
return aMax
```

Output: The action selected with maximum Upper Confidence Bound $aMax$

2.2 Expansion

This part is relatively simple and denoted as expand(). When the search (detailed algorithm will be in Section 2.5) has reached (via a selected action a) a leaf OR node that is neither a goal (i.e. $l = N$) nor the node at $maxT$, the corresponding child AND node is created with grandchildren OR nodes. If $a \in A_1$ (move), there will be multiple grandchildren per P_{movek}, otherwise only one grandchild will be created. The move grandchildren are stored in a Hash Map MOP as keys with the corresponding P_{movek} as values. For all types of OR grandchildren, a list of available actions LA is picked from A for future expansions, which excludes any unchanged current parameters and out-of-fuel situations. In addition, some valid actions of changing parameters will be pruned from LA if their heuristic based on expected value is much worse (i.e. less than a negative threshold) than current. The equation for heuristic is:

$$E = -TM_{slip} * P_{slip} - TM_{breakdown} * P_{breakdown} + \sum_{k \in A_1 - \{\text{slip,breakdown}\}} k * P_{movek}$$

Lastly, the AND child is stored and returned. The parent OR node (i.e. grandparent of the AND node's OR children) uses a Hash Map MAC with a as key and child anc as value to store such information.

2.3 Simulation

All OR children of an AND node will undergo simulation (shown in Algorithm 3 later). The algorithm to simulate from a OR node is shown as below. Note the algorithm is greedy that it will select action A_1 (or A_5 if necessary) all the time to the end (either reach max time or reach goal), which boosts the speed and accuracy of the rollout.

Algorithm 2: Simulate States (denoted as simulate())

```
Input: List of available actions LA as a subset of A; The OR node itself ont; Discount factor γ
a ← LA.get(rand.nextInt(|LA|))
an ← new(AndNode(parent ← ont, action ← a))
on ← ∅          // declare a placeholder for the AND node's child
count ← 1
while TRUE do
    // Sample from the AND node's children where MOP is the same as described in Section 2.2.
    // When generating AND node's children, the states associated are updated implicitly
    MOP ← an.getChildren()
    p ← rand.nextDouble()
    for all the onc ∈ MOP.keySet() do
        if p ≤ MOP.get(onc) then
            on ← onc
            break
        end if
    end for
    if on = GOAL or on.TM = maxT then
        return γ^count * on.rs   // get the immediate reward of the OR node weighted by discount factor
    end if
    a ← LA.getGreedyAction()    // Now a can only be either from A_1, or from A_5 if not enough fuel
    an ← new(AndNode(parent ← on, action ← a))
    count ← count + 1
end while
Output: The reward from this simulation
```

2.4 Backpropagation

After the simulation for each OR child node in the AND node, the expected reward will be calculated based on the probability of each OR child node.

This stage updates the $\boldsymbol{Q(s,a)}$ and $\boldsymbol{N(s,a)}$ recursively starting from the parent OR node and all the way up to the root which will be used for reward comparisons in the overall search.

Algorithm 3: Backpropagate to Root (denoted as backpropagate())

Input: The current OR node on; Last action a; Intermediate value q; Discount factor γ

$pair \leftarrow on.MANQ.\text{get}(a)$ // the same MANQ member described in Algorithm 1

$qn \leftarrow \frac{pair[1]*pair[0]+q}{pair[0]+1}$

$MANQ.\text{put}(a, (pair[0]+1, qn))$

$ns \leftarrow ns + 1$ // the same ns member described in Algorithm 1
$an \leftarrow on.parent$
if $an \neq \emptyset$ **then** // the current OR node is not root yet
  $onn \leftarrow an.parent$
  backpropagate($onn, an.$getAction(), $q * \gamma$)
end if
Output: $\emptyset$ // this algorithm only updates the value in relevant OR nodes

---

2.5 Overall Search

The algorithm below brings the four components together, with a heuristic at the end to prevent actions other than A_1 that will not vary the rewards too much.

Algorithm 4: Monte Carlo Tree Search (Step Level)

Input: Problem spec ps; Timeout t (real-world time for 1 game time step, default to 15 s)
$s \leftarrow$ getInitialState(ps)
$root \leftarrow$ new($OrNode(parent \leftarrow \emptyset, state \leftarrow s, rs \leftarrow 0)$)
while timer() $< t$ **do**
  $on \leftarrow root$
  $a \leftarrow root.select()$
  $an \leftarrow root.MAC.get(a)$ // the same MAC described in Section 2.2
  $bool \leftarrow$ FALSE
  while $an \neq \emptyset$ **do**
    // first sample an OR node from the AND node
    $MOP \leftarrow an.$getChildren() // the same MOP described in Algorithm 2
    $p \leftarrow$ rand. nextDouble()
    for all the $onc \in MOP.$keySet() **do**
      if $p \leq MOP.$get(onc) **then**
        $on \leftarrow onc$
        break
      end if
    end for
    // The sampled OR node (also as a leaf) satisfies stopping conditions
    if $on =$ GOAL **or** $on.TM = maxT$ **then**
      $onn \leftarrow an.parent$
      backpropagate($onn, an.$getAction(), $on.rs$)
      $bool \leftarrow$ TRUE
    end if
    $a \leftarrow on.$select()
    $an \leftarrow root.MAC.$get(a)
  end while
  if $bool$ **then** // not need to simulate, and backpropagate again for stopping OR nodes
    continue
  end if
  $an \leftarrow on.$expand()
  $MOP \leftarrow an.$getChildren()
  $q \leftarrow 0$
  for all the $onc \in MOP.$keySet() **do**
    $q \leftarrow q + onc.$simulate() $* MOP.$get(onc) // calculate the value
  end for
  $onn \leftarrow an.parent$
  backpropagate($onn, an.$getAction(), q)
end while
// finally, select the action with maximum reward

$reward \leftarrow -1$
$aMax \leftarrow \emptyset$
for all the $a \in root.MANQ.\mathrm{keySet}()$ **do**
    $Q(s,a) \leftarrow root.MANQ.\mathrm{get}(a)[1]$
    if $Q(s,a) > reward$ **then**
        $reward \leftarrow Q(s,a)$
        $aMax \leftarrow a$
    end if
end for
if $aMax \notin A_1$ **then** // this is useful for level 3 and above
    $minNsa \leftarrow root.MANQ.\mathrm{getMinNsa}()$
    $maxNsa \leftarrow root.MANQ.\mathrm{getMaxNsa}()$
    if $\frac{maxNsa}{minNsa} < \mathrm{RATIO\_THRES}$ **then** // meaning similar rewards across changing parameters
        $aMax \leftarrow LA.\mathrm{getMoveAction}(0)$ // pick the first move action from available actions instead
    end if
end if
return $aMax$
Output: Optimal action to perform at the current game time step

3. Complexity Analysis

3.1 Time complexity

Each time the MCTS performs a cycle (a sampling), $\boldsymbol{Q(s,a)}$ of the root node will be more and more precise, which means more samples in a single run will contribute to a more accurate optimal policy. As a result, the time complexity for each sampling should be as fast as possible.

MCTS algorithm is an effective way to solve MDP problem with large state space. However, the time complexity of MCTS highly depends on the action space and the maximum step in this game.

The branching factor in each OR node is equal to the size of the action space. A large branching factor means the MCTS algorithm will take longer to process the selection stage as the calculation of $\boldsymbol{UCT}$ needs to traverse through each action from the action space. This can be seen from Figure 1, where the graphs are the sampling status for Level-3 and Level-4 with the same configuration. The number at the bottom line is the overall samples performed in a run. The Level-4 case has 6307 samples, only about half of the samples in the Level-3 case due to the increase of the action space.

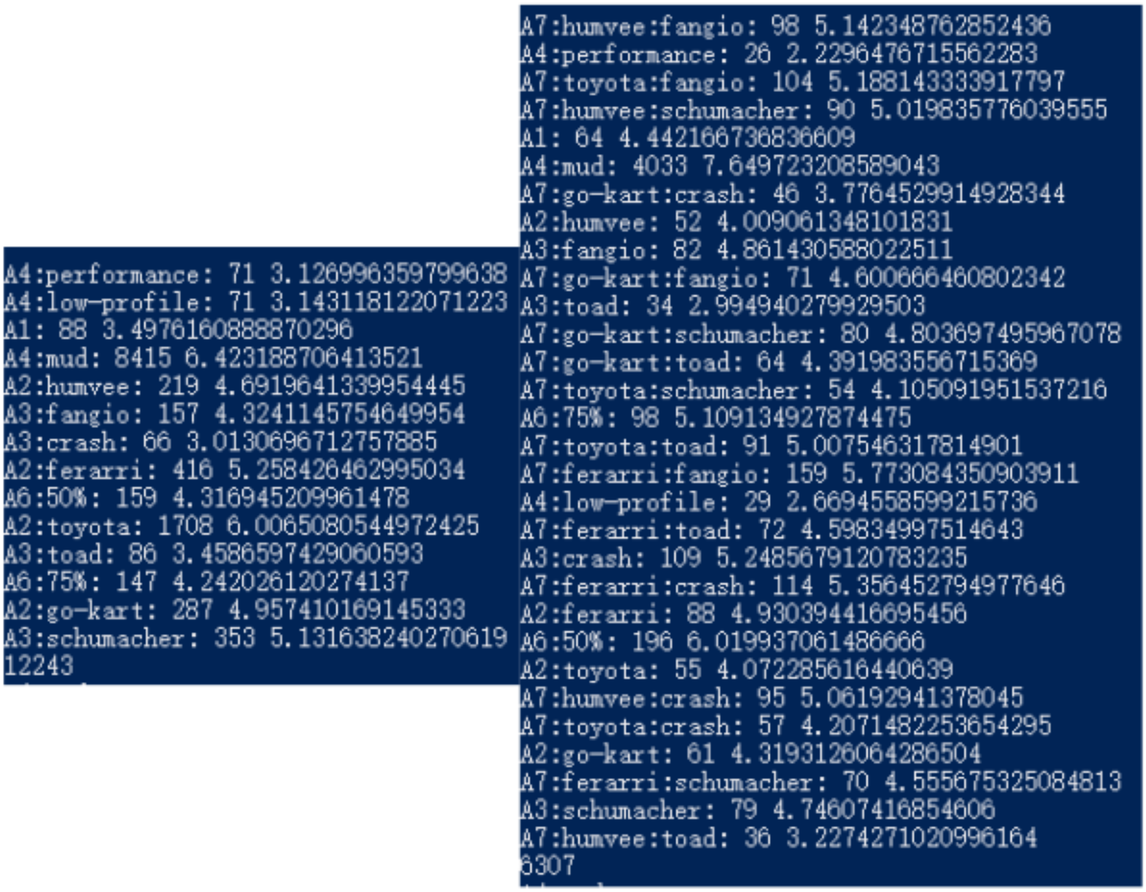

Figure 1 – Comparison Test for Different Action Space with the same Configuration

(Left – Level 3, Right – Level 4)

Another factor is the maximum step in the game. Each time when the MCTS reaches a leaf node, it will perform a simulation from that node and the agent needs to simulate the game until the end. A larger maximum step means it can take longer to simulate the game to the end, which will increase the time used in a single run. This can be seen in Figure 2, where the graphs are the sampling status for two Level-3 cases with different maximum steps. The case with 60 maximum steps has more samples than the case with 90 maximum steps (18103 samples > 12556 samples).

```
A4:performance: 150 2.784731528060352
A4:low-profile: 152 2.7984115874154516
A1: 340 3.648697775879342
A4:mud: 8721 5.005374998727201
A2:humvee: 864 4.279698027428925
A3:fangio: 623 4.091961179351237
A3:crash: 263 3.411956305938836
A2:ferarri: 972 4.34103241842327
A6:50%: 573 4.040461269713038
A2:toyota: 2178 4.670895462242599
A3:toad: 202 3.1381261517268992
A6:75%: 500 3.945030743677784
A2:go-kart: 658 4.125855209551241
A3:schumacher: 1907 4.624601879779482
18103
```

```
A4:performance: 108 3.815478121838906
A4:low-profile: 99 3.691555692886392
A1: 100 3.6961214294881097
A4:mud: 6386 6.390693560675424
A2:humvee: 193 4.568423501959846
A3:fangio: 101 3.7269810538717554
A3:crash: 106 3.7898841503317855
A2:ferarri: 348 5.120230075991452
A6:50%: 110 3.8556605564381243
A2:toyota: 3649 6.253131665702069
A3:toad: 56 2.6473572263403855
A6:75%: 231 4.75422260051723
A2:go-kart: 114 3.9024990330867237
A3:schumacher: 955 5.773318888598599
12556
```

Figure 2 – Comparison Test for Different Max Step with the same Configuration

(Left – 60 Steps, Right – 90 Steps)

3.2 Memory complexity

Generally, the memory of MCTS grows linearly as each time when only one node is added to the tree. However, a list of viable AND node or actions that it can choose needs to be stored at each OR node in the MCTS, whose size depends on the action space again, indicating that a larger action space leads to a greater memory usage. Figure 3 shows different memory consumptions when running Level-3 and Level-4 cases with same configuration and the Level-4 uses slightly more memory space than the Level-3 case.

Name	Status	18% CPU	39% Memory	0% Disk	0% Network	0% GPU
Windows PowerShell (3)		16.7%	877.6 MB	0 MB/s	0 Mbps	0%

Name	Status	17% CPU	33% Memory	0% Disk	0% Network	0% GPU
Windows PowerShell (3)		16.8%	912.9 MB	0 MB/s	0 Mbps	0%

Figure 3 – Comparison Test for Different Action Space with the same

(Up – Memory Usage for Level 3, Down – Memory Usage for Level 4)

3.3 Reward correctness analysis

Unlike value iteration or policy iteration, the optimal policy given by MCTS is not always correct. The correctness of the optimal policy depends on the number of samples generated in a run. More samples will contribute to a more appropriate policy; therefore, the extreme case is that the MCTS runs for an infinite time, and generates infinite number of samples, the reward for each action will be the same as the result generated from value iteration.

A larger action space will lead to a larger branching factor of the tree, which will require more samples to give a good estimate of the reward for each action. However, a larger action space will increase the time complexity of the algorithm. Therefore, problem with a large action space will need much more time to gain a good estimation of the reward.

This can be seen from the Figure 4 with two special Level-3 cases, when the action change pressure has been removed. The rest of choices are identical and as a result, the optimal policy would always be move as the move can give one step advantage than other actions. The first Level-3 is normal one while the second Level-3 supports actions up to A7 (code has been modified to support this function). Each line in the graph represents the type of action and its corresponding $\boldsymbol{N(s,a)}$ and $\boldsymbol{Q(s,a)}$ and the optimal policy is printed at the last line.

MCTS can find out the one step advantage in the first case but not in the second, as the size of the action space is too big to explore within the same time period (15 seconds).

```
A3:mushroom: 1061 5.069561294501282
A4:performance: 1015 5.0463651898298885
A4:low-profile: 1165 5.111547635789456
A1: 1626 5.251303347910105
A4:mud: 1062 5.069681466015891
A2:humvee: 1236 5.139002507331007
A3:schumacher: 1013 5.051021901737363
A3:crash: 1287 5.15724280330146
A2:ferarri: 982 5.031219080883444
A2:go-kart: 978 5.029322967863 6315
A2:toyota: 1326 5.167643573102156
A3:anakin: 1436 5.202468927137329
14187
A1
```

```
A7:humvee:crash: 140 5.6614790491639475
A7:toyota:schumacher: 90 5.0002375636634016
A7:toyota:crash: 97 5.107096076184477
A7:go-kart:schumacher: 104 5.226493030715887
A2:toyota: 114 5.361682677964763
A4:low-profile: 89 4.990764676009779
A2:go-kart: 133 5.551520814017238
A3:anakin: 113 5.321328698369222
A7:go-kart:crash: 112 5.338292932508454
A1: 150 5.681253283028962
A3:crash: 173 5.843194939865423
A7:ferarri:schumacher: 140 5.610463294230548
A7:toyota:mushroom: 105 5.210347292165121
A7:ferarri:anakin: 154 5.718614850097511
A7:ferarri:crash: 91 5.045454394296713
A4:performance: 121 5.4351356403975615
A7:humvee:anakin: 106 5.225812953489576
A4:mud: 164 5.80054763184407
A7:humvee:mushroom: 111 5.328206214004433
A2:humvee: 107 5.2393752756066805
A7:toyota:anakin: 147 5.674554629268652
A2:ferarri: 129 5.514841671721106
A7:go-kart:mushroom: 142 5.633048329399356
A7:ferarri:mushroom: 121 5.4361071078495575
A3:schumacher: 151 5.684787344692021
A7:humvee:schumacher: 108 5.28019242325429
A3:mushroom: 131 5.514736640463072
A7:go-kart:anakin: 113 5.344684636045306
3456
```

Figure 4 – Comparison Test for Performance

(Left – Action up to A6, Right – Action up to A7)

However, if the second case is run for 2 minutes, the one step advantage can be found out. (as shown in Figure 5)

```
A7:humvee:crash: 1170 5.381103017460542
A7:toyota:schumacher: 1014 5.314014872387082
A7:toyota:crash: 1166 5.3802331124922
A7:go-kart:schumacher: 1132 5.367370388119745
A2:toyota: 952 5.282068594056984
A4:low-profile: 906 5.255769900058345
A2:go-kart: 775 5.170811746965504
A3:anakin: 715 5.120106278273316
A7:go-kart:crash: 825 5.204091452285094
A1: 1201 5.394651843231025
A3:crash: 929 5.265948076782212
A7:ferarri:schumacher: 940 5.275923823671131
A7:toyota:mushroom: 1074 5.340493289857599
A7:ferarri:anakin: 845 5.219331747101565
A7:ferarri:crash: 1190 5.391494753826045
A4:performance: 868 5.231973919215599
A7:humvee:anakin: 899 5.248393857511226
A4:mud: 957 5.281149884897172 5
A7:humvee:mushroom: 734 5.134866964561954
A2:humvee: 1194 5.391103284713638
A7:toyota:anakin: 865 5.230337457147237
A2:ferarri: 1069 5.337856947624869
A7:go-kart:mushroom: 821 5.199332985568 03
A7:ferarri:mushroom: 1092 5.347963255219398 5
A3:schumacher: 863 5.230729508524016
A7:humvee:schumacher: 733 5.138951591271 51
A3:mushroom: 1056 5.333626265679871
A7:go-kart:anakin: 945 5.277717244689544
26930
A1
```

Figure 5 – Running Case 2 for 2 minutes

Overall, the performance of MCTS depends on the number of samples performed. More samples mean more information, each of the sample can provide new information to the tree. Therefore, the estimate will get more and more accurate over time.

Approach to improve the MCTS algorithm in this case would be either using a heuristic in the selection stage to provide more samples on the actions which are more worth exploring or using a more accurate simulation policy that can estimate the real value of the node accurately as the policy used in this problem is greedy.