Game Changing Technology

COMP3702 ARTIFICIAL INTELLIGENCE

Yunke Qu 44631604
Jamieson Lee 43923674
William Tam 42371308

Problem Definition

Markov Decision are formally defined as a 4-tuple (S, A, T, R), i.e. the state space, action space, transition function, and reward function. Our game world is a 1 dimensional problem of size N. For simplicity, we will define the problem based on level 5.

State Space

The state space consists of all the possible configuration and positions relating to the race vehicle. The state space can be denoted as

S = {cell index, is in slip condition or not, is in breakdown condition or not, car type, driver, tire type, fuel level, tire pressure level}, where

car type ∈ {mazda, toyata, Ferrari, Humvee, go-kart, etc},

driver ∈ {stig, Schumacher, Anakin, mushroom, crash, etc},

terrain ∈ {dirt-straight-hilly, dirt-straight-flat, dirt-slalom-hilly, dirt-slalom-flat, asphalt-straight-hilly, asphalt-straight-flat, asphalt-slalom-hilly, asphalt-slalom-flat, etc},

tire ∈ {all_terrain, mud, low_profile, performance},

tire pressure ∈ {50%, 75%, 100%},

fuel level ∈ [0, 50].

Action Space

The action space consists of 8 actions labelled from 1 to 8.

A = {A1, A2, A3, A4, A5, A6, A7, A8}, where

A1 = "continue driving",

A2 = "change car type"

A3 = "change the driver"

A4 = "change tire(s)"

A5 = "add fuel"

A6 = "change tire pressure"

A7 = "change car type or driver"

A8 = "change tire(s) or add fuel or change tire pressure".

Note that each action is actually not a single action. For example, A2 "change car type" may change the car type to any car type from the set *{mazda, toyata, Ferrari, Humvee, go-kart}*.

Transition Function

Maps current state So to a new state Si given an action A. The transition function contains non-deterministic and deterministic properties which are dependent on what type of action was given to the agent.

Non-Deterministic Transitions: T_{ND} (S, A1)

Transitional probabilities are calculated when the agent is given a non-deterministic action to move and are dependent on the given configuration of the current state. This implies that the current state and future states are independent from past states satisfying 1st order Markov properties.

Deterministic Transitions: T_D (S, A2 – A8)

As these actions have deterministic properties, the transition function is just direct mapping of the current action applied to the next state. e.g.

TD(S1, A2, So) = S1{new car type, driver, terrain, tire, tire pressure, fuel}

Reward Function

The reward function (w_i) is made up of two parts; the terminal state value and the heuristic value: $w_i = W_{TS} + W_{HEURISTIC}$. This was done in order to assure the agent would reward states that were likely to reach the goal state, were close to a goal state, and had a high probability of moving forward

Terminal State Value (W_{TS})

During the simulation phase the agent loops over random actions T times where T is the maximum possible number of steps remaining to reach the goal. If the agent reaches the goal state in this period the terminal state value is $\frac{3}{2}$, that is, $W_{TS} = \frac{3}{2}$. If the loop reaches time-out the terminal state value is –1 ($W_{TS} = -1$). This gave the agent incentive to reward states that were likely to reach the goal state.

Heuristic Value ($W_{HEURISTIC}$)

The heuristic value was calculated using the statistical expected move value from the current cell (the distance you are expected to move from the cell). The formula below shows part of the heuristic value:

$$\text{expMoves} = -4 \times \mathbf{P}(\text{move } -4) + -3 \times \mathbf{P}(\text{move } -3) + \ldots + 5 \times \mathbf{P}(\text{move } 5)$$

$$w_{\text{HEURISTIC}} = \begin{cases} 2 - \frac{\text{maxiumum remaining steps to goal}}{\text{maximum steps total}} & \text{expMoves} > 0 \\ -1 & \text{expMoves} \leq 0 \\ 0 & w_{TS} = -1 \end{cases}$$

Note that if the node is not a terminal state the heuristic value is not required. The use of expected number of moves served to encourage the agent to choose the best configuration, and the use of relative distance to the goal gave the agent incentive to move forward. In practice, we observed that without the expected number of moves, the agent would perform A1 repeatedly, while without the relative distance , the agent would repeated perform A2 to A7 a large majority of the time. Using the chosen reward function gave the agent incentive to reward states that were closest to the goal state, with the highest probabilities of moving forward.

Methodology & Search Algorithm

Monte Carlo Tree search (MCTS) is a search method known for its implementation in popular games such as Chess and Go. It is an algorithm that tries to determine the best set of moves down a decision tree with high branching and will only search nodes that seem promising. There are four main stages in the Monte Carlo Tree Search algorithm. The first stage is tree traversal or node selection, followed by node expansion, rollout (random simulation) and finally back-propagation.

Selection & Node Expansion

In the tree traversal stage - starting with the root node, recursively select optimal child nodes until a leaf node is found. The selection process is determined by the UCB1 (Upper Confidence Bound) policy implemented in node selection:

$$UCB = \frac{w_i}{n_i} + c\sqrt{\frac{ln\, t}{n_i}}$$

Where w_i is the number of wins achieved at the node, n_i is the number of times the node has been visited, c is a tune-able bias factor and t is the total number of times the parent node has been visited. We keep selecting optimal (highest UCB) child nodes until a leaf node is reached. This selection policy encourages a balance between exploitation and exploration so that no state will be ignored and prioritises branches that maximise reward. Figure 1 below shows a flow diagram for the tree traversal and node expansion phase:

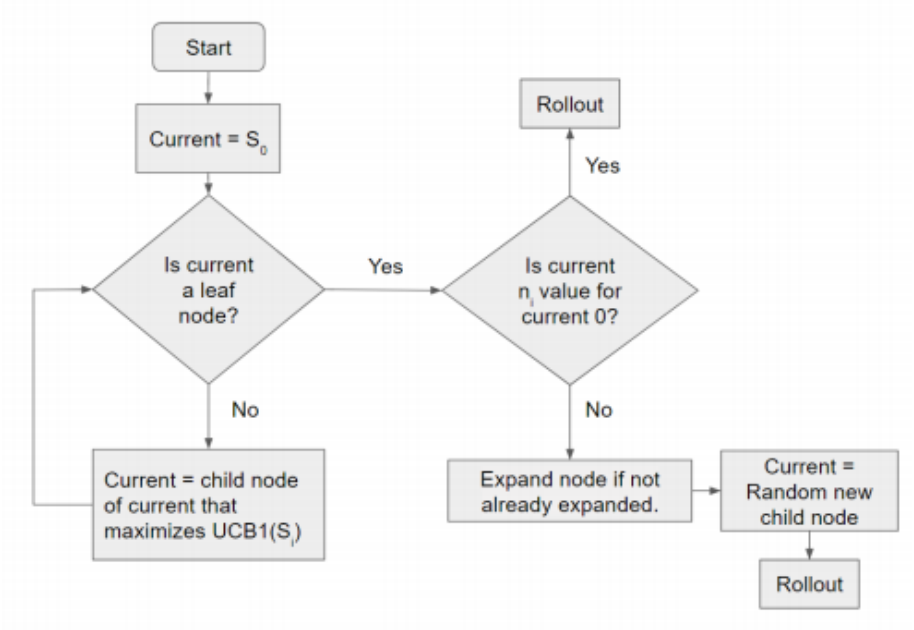


Figure 1: Flow Diagram

Rollout / Simulation

During the rollout phase the algorithm randomly chooses successive child nodes to simulate a random path from the selected node until either the terminal state is found or a timeout is reached. The reward given by the reward function is back-propogated up the tree after rollout. As discussed previously this involves rewards such that states that were likely to reach the goal state, were close to a goal state, and had a high probability of moving forward had the highest w_i value back-propagated. Pseudo code for the rollout phase is shown below:

1. **while** *time out not reached*:
2. *tempState* ← *new random state from current tempState*
3. **if** *tempState is terminal state*:
4. *RETURN* $w_{TS} + w_{HEURISTIC}$
5. *RETURN -1*

Back-Propagation

In the back propagation stage, the simulation reward at the selected node and the sequence of nodes along the path back to the root is stored. Each node contains two values, an estimated value from the simulation results and the number of times that the node has been visited. These values are used in the tree traversal stage (via UCB). To back propagate the value up the tree - the current value is added to the parent node's estimated value, and the parent nodes visit count is incremented (implemented using a recursive function).

Time & Space Complexity

Time Complexity

Reasoning

As discussed above, in order to choose an optimal action at a given state, we first initialized an empty tree structure. After this, we repeatedly performed *selection*, *expansion*, *simulation*, and *backpropagation* operations for a certain amount of time, which was 15 seconds in our game. Then we could select the best action to take. Finding the next optimal step takes $O(n*b*h)$ time, where n is the number of iterations, b is the average branching factor, and d is the height of the tree.). Note that b is approximately equal to the state space.

Since each iteration contains four phases, we know:

$$\textit{Time per iteration} = \textit{selection} + \textit{expansion} + \textit{simulation} + \textit{back-propagation}$$

In the selection phase, we start from the root, iterate over all its children and go to its most promising child, i.e. the child node with the highest UCB scores. This takes O(b) time. Then we go to its most promising child node in the same way. This is repeatedly done for d times until we get to a leaf. Therefore, the overall time complexity of selection phase is O(b*h).

In simulation, we repeatedly and randomly go to one of its children of a node until we get to a terminal state or the maximum number of steps allowed has been reached. This process takes O(h), where h is the height of that tree. Intuitively, h is proportional to track size N and the number of allowed time steps. Thus, we can increase the height of a tree by increasing N or the number of the allowed number of time steps.

And finally in backpropagation, the time complexity of this phase can be easily computed as O(h), since the number of times we need to push the value of a node to its parent is proportional to the height of the tree.

Summing up the time of the four phases, we get

$$\textit{Time per iteration} = O(bh + b + h + h) \approx O(bh)$$

And the overall time complexity of n iterations is

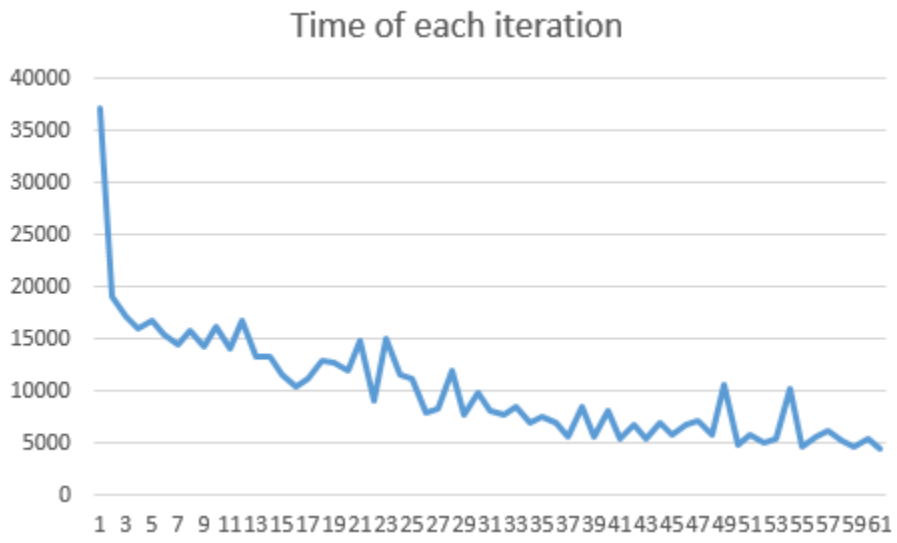
$$n * O(bd) = O(nbh)$$

Experiments

First, we want to investigate the relation between the average running time per iteration and the height of the tree created in that iteration. To do that, we want the branching factor to be constant and only change the height of the tree in every iteration. We set the track size N to be 30, the number of time steps allowed to be 90.The time allowed for considering each move to be 1 second. That is we allow the iteration to continue for 1 second. By keeping track of the number of iterations that happened within 1 second, the average running time is calculated as

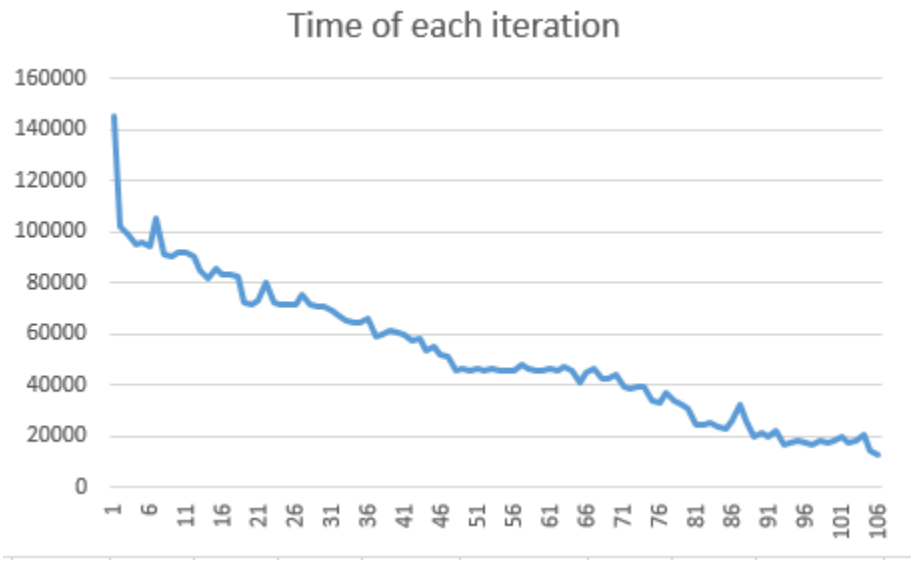
1000 000 000 / # iterations within 1 second,

then plot the average time used in each iteration and the time steps.

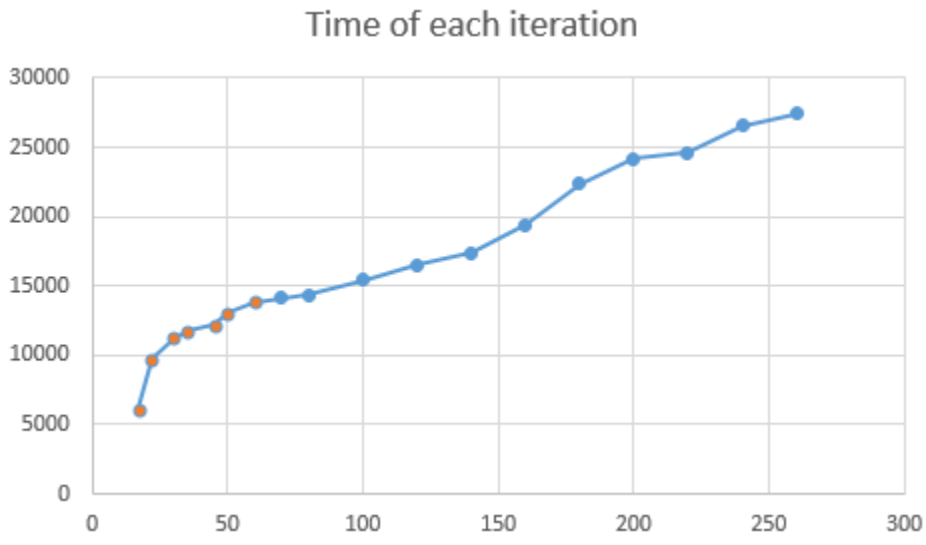


The graph above reveals the relation between the average running time of one iteration and the height of the tree on level four. In this case, the branching factor b is constant, which is 30 in our implementation. The agent solved the game within 61 time steps. At the initial state, each iteration took more than 35000 nanoseconds. We initiate a new empty search tree after taking the selected step. As the agent got increasingly closer to the goal state after taking a series of actions, the height of the new search tree is predicted to decrease, which results in a linear decline of the time of each iteration.

Similarly, we can get more iteration samples if we set the number of allowed time steps to be positive infinity. As the height of the search tree decreases, the time of each iteration declines almost linearly. This suggests the average running time of each iteration is O(h), where h is the height of the search tree.



Next we look at the relation between the branching factor and the average running time of each iteration. We tested the program using input cases of different problem sizes and got the data below.



This graph gives information on the relation between the branching factor and the average time per iteration. Similar to the method used before, we count the number of iterations within one second. Then the average time per iteration can be calculated as

1000 000 000 / number of iterations.

We observe that the average run time of each iteration is roughly in a linear relation with the branching factor b when b is greater than 65, which is the section plotted by blue dots.

The game is solvable when the average branching factor is no greater than 65. When the branching factor is less than 65, the branching factor b and the height of the search tree h are varying simultaneously. Since each iteration takes O(b*h) time, the complexity of each iteration cannot be linear when b is less 65. However, note that if the track size N and the total number of steps allowed remain constant, there should be an upper bound for the height of the search tree, regardless of the increase of b. So as b increases to infinity, h will become negligible compared to b. When this happens, the average run time per iteration will be in an approximate linear relation with b, which is well suggested by the graph.

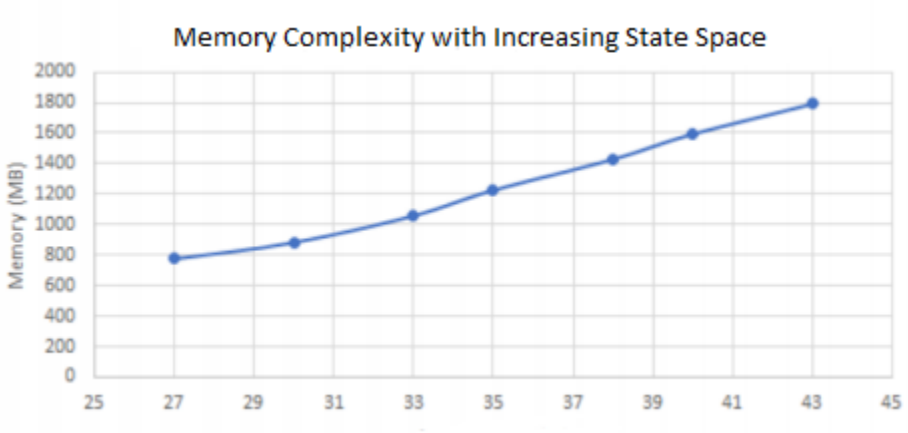
Space Complexity

Reasoning

The space usage of one iteration O(m), where m is the overall size of the state and action space. The search tree stores almost all the possible action and state nodes in the game. Therefore, the space usage in one iteration should be in a linear relation with the size of the state and action space.

On the other hand, since the program always initiates a new search tree after taking an action, by the time the program completes, only a series of state-action pairs will be stored in the memory. These pairs represent the policy or solution to the game, namely the action to take given a state S, mapping each state in the state space to an action. Therefore, the space usage by the time the program completes is O(s), where s is the size of the state space.

Experiments



We tested the program with input with state spaces of varying sizes. Then we output the memory usage of the program by the time it completes. The graph clearly proves the linear relation between overall memory usage and the size of the state space.