THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

COM7702 - Artificial Intelligence

ASSIGNMENT 2 REPORT

BotMate Team:

- 45510760 - **Wenhui Han**
- 45270916 - **Canggih Pramono Gultom**
- 44907635 - **Vu Anh LE**

Table of contents

1 . MDP Design Problem

In this assignment, a program is designed to provide the estimated best instructions in a sequence for a car to travel through terrains of varying types and reach the goal within given steps. Since the action “continue moving” may result in a range of possible outcomes, stochasticity is introduced into the problem. The MDP problem can be defined as follows:

1.1. State Space

A state for this MDP problem captures information available to the agent at current step. It incorporates state of the car (car type, tyre model, fuel level, slip, breakdown, tyre pressure), the driver and the position of current terrain in the whole map. Using parameters denoted below:

- **pos** - The current position in the whole map, range from 1 to N
- **isSlip** - Boolean value indicating whether the car is currently in slip condition
- **isBrkDn** - Boolean value indicating whether the car is in breakdown condition or not
- **carType** - The current car type
- **fuelLvl** - The current fuel level, range from 0 to 50
- **tyrePrs** - The current tyre pressure, with 3 options being 50%, 75%, 100%
- **driver** - Current driver of the car
- **tyreMdl** - Current tyre model, options include all-terrain, mud, low-profile, etc...

The state can be denoted as:

$$S_i = \{pos_i, isSlip_i, isBrkDn_i, carType_i, fuelLvl_i, tyrePrs_i, driver_i, tyreMdl_i\}$$

The count of possible states for each level is calculated as per table below:

	Pos	Slip	BrkDn	Car	Fuel	Press.	Driver	Tyre	Total States
Level 1	10	2	2	2	1	1	2	4	640
Level 2	10	2	2	3	50	3	2	4	144.000
Level 3	30	2	2	5	50	3	5	4	1.800.000
Level 4	30	2	2	5	50	3	5	4	1.800.000
Level 5	30	2	2	5	50	3	5	4	1.800.000

Table 1.1.1 state space for varying levels

1.2. Action Space

- Action includes varying numbers of options based on input level, and each action will incur a cost in time measured by steps provided by input file.
- Some actions are only allowed in certain conditions. Eg. The car can perform A1 (continue moving) only if the fuel level is sufficient to support such an action.
- Apart from A1, all other actions are deterministic, that is, the results of such actions are fully predictable and non-stochastic.
- Some action items can be further split into smaller categories. Eg. Change tyre pressure can be split into: change tyre pressure to 50%, 75% and 100% level.

Possible actions are listed as follows with respective conditions

Action No.	Action Description	Sub Types	Action Result	Allowed in Condition
A1	Continue moving	No	The car may move forward, backward or remain at same position because of slip or breakdown condition.	Sufficient fuel level; Not in slip condition;Not in breakdown condition
A2	Change car	Yes (Toyota, Mazda, etc.)	Switch to another car from options	
A3	Change driver	Yes (eg. Change to Max / Tim)	Switch to another driver from options	
A4	Change tyres	Yes (all-terrain, low-profile, etc.)	Switch to another tyre model from options	
A5	Add fuel	Yes (amount of Fuel to add)	Increased fuel level	When fuel level is not 100%
A6	Change tyre pressure	Yes (50%; 75% or 100%)	Change to another fuel pressure from options	
A7	A2 & A3	Yes	Change car type and driver from options	
A8	A4 & A5 & A6	Yes	Change tyre, add fuel and change tyre pressure from options	When fuel level is not 100%

Table 1.2.1 Action summary

The count of possible actions available for each level is calculated as

*A7=A2*A3; A8=A4*A5*A6*

	A1	A2	A3	A4	A5	A6	A7	A8	Total Actions
Level 1	1	2	2	4	0	0	0	0	9
Level 2	1	3	2	4	5*	3	0	0	18
Level 3	1	5	5	4	5*	3	0	0	23
Level 4	1	5	5	4	5*	3	25	0	48
Level 5	1	5	5	4	5*	3	25	60	108

** Add Fuel is discretized into 10 unit for each step, so it is reduced to 5 possible actions*

1.3. Transition Function

The transition is deterministic for all actions excluding A1 (continue moving). Assume S_i={pos_i, $isSlip_i$, $isBrkDn_i$, $carType_i$, $fuelLvl_i$, $tyrePrs_i$, $driver_i$, $tyreMdl_i$}, the transition matrix for action A2 – A8 is:

Action	Action Name	Current State	Next State
A2	Change Car	{,,,$carType_i$,$fuelLvl_i$,,,,}	{,,,$carType_j$,$fuelLvl_{max}$,,,,}
A3	Change Driver	{,,,,,,$driver_i$,}	{,,,,,,$driver_j$,}
A4	Change Tyre	{,,,,,,,$tyreMdl_i$}	{,,,,,,,$tyreMdl_j$}
A5	Add Fuel	{,,,,$fuelLvl_i$,,,,}	{,,,,$fuelLvl_i$,,,,}
A6	Change Pressure	{,,,,,$tyrePrs_i$,,}	{,,,,,$tyrePrs_j$,,}
A7	A2 & A3	{,,,$carType_i$,$fuelLvl_i$,,$driver_i$,}	{,,,$carType_j$,$fuelLvl_{Max}$,,$driver_j$,}
A8	A4 & A5 & A6	{,,,,$fuelLvl_i$,$tyrePrs_i$,,$tyreDdl_i$}	{,,,,$fuelLvl_j$,$tyrePrs_j$,,$tyreDdl_j$}

Omitted parameters are unchanged parameters.

The transition is non-deterministic for action A1 (continue moving), denoted as :

$$T(s, A_1, s') = P[S_{i+1} = S' \mid S_t = s, A_t = A_1]$$

That is, the next state of the action “continue moving” can be described as a probability distribution over 12 possible outcomes for a given state, with each probability calculated based on input slip factor of car, driver, tyre, pressure and terrain.

1.4. Reward Function

The reward function is based on how far the agent can move forward from the previous condition (eg: current car, current driver, and current fuel level). When the agent reaches the goal, it will be rewarded with extra bonus that is equivalent to being able to move 5 cells ahead. If the Agent cannot reach the goal in the given step count during simulation, it will not receive any bonus. The reward function is calculated as :

$$Reward\ Point = (Current\ Position - Start\ Position) + Goal\ Bonus$$

The default policy of rollout simulation implemented is to continue moving until out of fuel, so the fuel level is also been taken to account.

2. BotMate Agent at Conceptual Level

2.1. Data Structure

Every action is stored in a **TreeNode** data structure, which is a generic class containing properties including action, parent node, children nodes, value and visitCount. The states will not be stored in our data structure, instead, it will be created on-the-fly by the simulator with actions stored in parent nodes. The states consist of position, tyre pressure, driver, tyre model, fuel, slip or breakdown information.

```
public class TreeNode {

    private Action action;
    private TreeNode parent;
    private List<TreeNode> children;
    private double value;
    private int visitCount;
```

- **value** - The value of a node which is updated when back-propagation method is called
- **visitCount** - Records the times the current node is visited
- **parent** - A reference to the parent node from the current node
- **children** - A collection of all the children of the current node

2.2. Monte Carlo Tree Search

Game-changing problem is solved by using an online method using MCTS because of the enormous state space and action space from input of higher levels. As mentioned previously, the count of states is around 1.800.000 from level 3 to 5, requiring huge computational resources using the offline method. Four core components of our MCTS implementation is described as per below:

2.2.1. Selection

Selection is a function to choose the best action based on the upper confidence bound (UCB) value. The node with maximum UCB value will be selected.

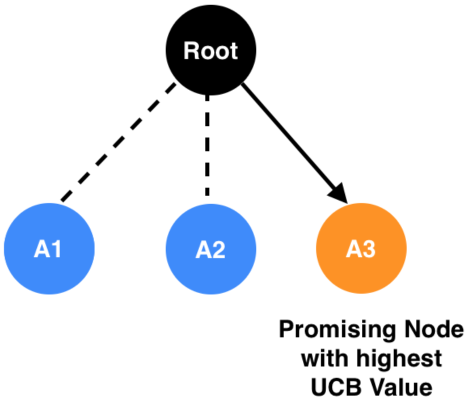


Figure 2.2.1.1: Select the promising node based on UCB value

PseudoCode for selection phase:

```
Algorithm selectPromisingNode(currentNode)
selectedNode <- first child of current node
bestValue <- selectedNode.getValue()
a <- currentNode.visitCount
for each childNode in currentNode.getChildren()
    if childNode.visitCount == 0 return childNode
    b <- childNode.visitCount
    UCBValue = childNode.getValue() + CONSTANT * sqrt(log(a/b))
    if UCBValue >= bestValue
        selectedNode <- childNode
        bestValue <- UCBValue
return selectedNode
```

Upper Confidence Bound (UCB value) for every node is calculated with this formula, then the agent selects the node with the highest UCB Value.

$$UCBValue = childValue + CONSTANT + \sqrt{\frac{\log(parentVisitCount)}{childVisitCount}}$$

The UCB value will be stored in every node after back-propagation from a leaf node.

2.2.2. Expansion

Expansion is a function to add children nodes to a particular node based on available actions at current state.

PseudoCode for expansion phase:

```
Algorithm expandNode(leafNode)
currentState <- simulator.getCurrentState()
for each action in generateActions(currentState)
    newNode <- new treeNode(action)
    newNode.setParent(leafNode)
    leafNode.addChild(newNode)
```

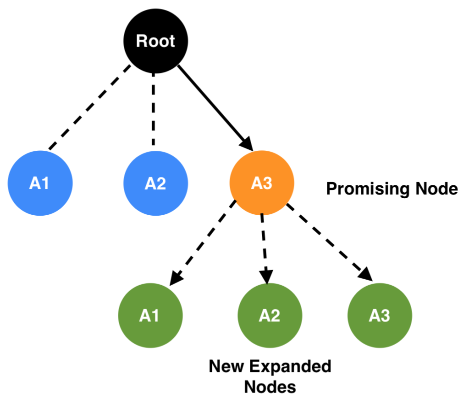


Figure 2.2.2.1 : Expansion to create new children nodes

2.2.3. Simulation

Simulation is a function to simulate a newly added node to estimate its value. In this implementation, the default policy is to always move forward until it reaches the goal, or run out of fuel, or reaching max step. The value of current node is measured by the distance covered by the car within simulation steps until out of fuel.

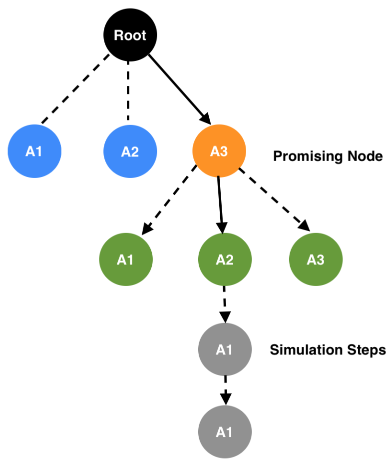


Figure 2.2.3.1 : Simulation from the current state

PseudoCode for simulation phase:

```
Algorithm rollOut(LeafNode, remainingStep)
currentState <- simulator.getCurrentState()
car <- getCurrentCar(currentState)
while simulator.getStep() <= remainingStep
    terrain <- getCurrentTerrain(currentState)
    if currentState.getFuel() >= getFuelUsage(car, terrain)
        currentState <- sim.step(new Action(ActionType.MOVE))
    else
        break
    if isGoalState(currentState)
        return ps.getN() - startPos + CAR_MAX_MOVE;
return currentState.getPos() - startPos;
```

2.2.4. Back-propagation

Back-propagation is a function to update the value of the predecessor nodes from children nodes. The value of a parent node is updated whenever new iteration finishes upon its children. Formula for such a relation is shown as :

$$Parrent\ Value = \frac{Parent\ Value \times Parent\ Visit + Child\ Value}{Parent\ Visit + 1}$$

The Back-propagation process is illustrated as follow

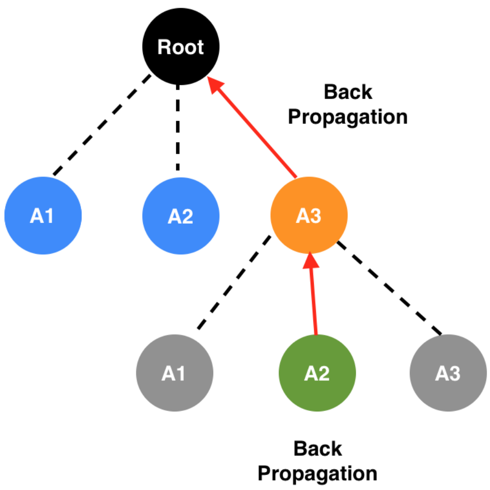


Figure 2.2.4.1 : Backpropagation is a function to update every value of nodes

Back-propagation function is called after a rollout simulation. This algorithm iterates through the ascendant nodes of the leaf node to update their values.

PseudoCode for back-propagation phase:

```
Algorithm backPropagation(leafNode)
currentNode <- leafNode
while currentNode != NULL
    currentNode.visitCount <- currentNode.visitCount + 1;
    q <- currentNode.getValue
    parentNode = currentNode.getParent
    if parentNode != NULL
        c <- parentNode.getVisitCount()
        parentNode.Value <- (parentNode.getValue * c + q) / (c + 1)
        currentNode = parentNode;
```

3. Time and Memory Complexity Analysis

3.1. Time Complexity

3.1.1. Time Complexity for One Iteration Calculation

Let **a** be the number of Possible Actions in action space, let **t** be the maximum allowed steps and let **d** be the depth of the tree. We can calculate the computational step for one iteration from the following components:

- Select Node: takes **a * d** primitive operations (for each level, select one node and iterate over all its children to select the promising node)
- Expand Node: takes **a** primitive operations (create new node for every action)
- Simulation: takes **t** primitive operations, one for each default action until it reach the maximum number of step (worst case)
- Back-propagation: takes **d** primitive operations (iterate over one ascendant at once)

So the time complexity for one iteration would be

$$Number\ of\ Calculation = a \times d + a + t + d$$

Since the **Depth of the tree** is **Expected** to be equal to the **Number of Max Steps,** one iteration will run in the time complexity of **O(a*t)** .

3

3.1.2. Overall Time Complexity

The overall time complexity is calculated by measuring the execution time that is required to solve different levels of the game. We compare the overall time consumption of different levels to explore how time can be affected by increased state space and action space(problem size).

Screenshot for sample taking as below:

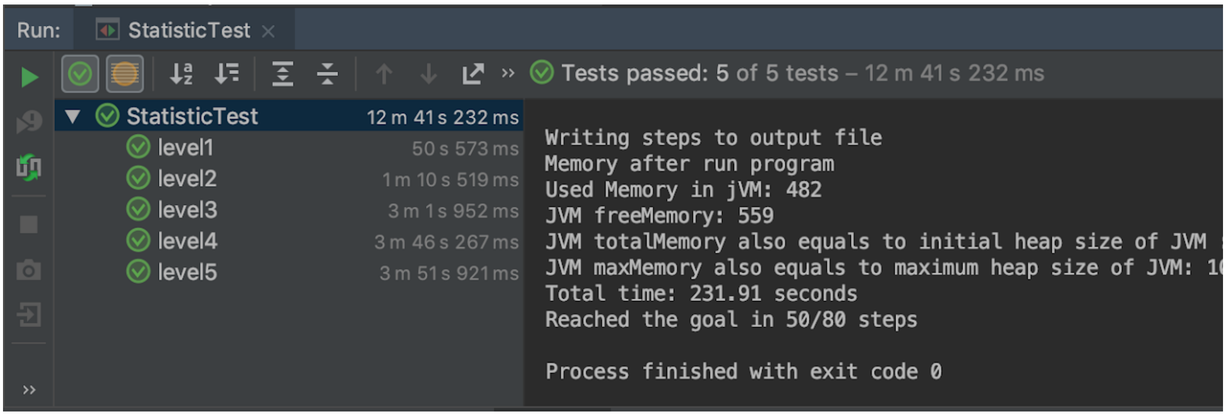


Figure 3.1.2.1 : One of sample iteration for every level.

	Level 1		Level 2		Level 3		Level 4		Level 5	
#Iter.	**Time**	**Steps**	**Time**	**Steps**	**Time**	**Steps**	**Time**	**Steps**	**Time**	**Steps**
1	55.3s	11	65.2s	19	108s	21	299s	67	314s	69
2	75.1s	21	68.7s	15	149s	33	204s	47	353s	79
3	60.1s	12	55.2s	11	191s	45	192s	43	156s	31
4	60.7s	12	70.4s	18	185s	44	238s	55	358s	75
5	45.4s	9	130s	25	213s	47	173s	40	215s	45
6	55.5s	12	75.8s	19	201s	44	227s	44	199s	40
7	50.3s	10	55.5s	11	141s	30	176s	41	317s	64
8	60.2s	13	200s	17	167s	35	183s	40	236s	49
9	65.3s	13	60.4s	12	177s	35	192s	42	213s	45
10	50.3s	11	45.2s	9	183s	38	262s	53	259s	59
Average	**57.8s**	**12.4**	**82.6s**	**15.6**	**171s**	**37.2**	**214s**	**47.2**	**262s**	**55.6**

Table 3.1.2.1 Average time and steps for every level

The table above contains the execution results for all levels , with 10 times iteration for every level by 5 second planning time.

As expected, the increase in problem size also increases steps to reach the goal as well as the overall execution time.

As the action planning time is restrained by parameter Exploration Time, the total execution time does not reflect the complexity within one complete MCTS iteration. Hence further analysis over one iteration is critical for understanding how problem size impacts the complexity of an MCTS iteration. One MCTS iteration is finishes when the one set of selection, expansion, evaluation and back-propagation steps are done.

To estimate the time spent over one iteration, our approach is to count total iteration of a single execution and the total time spent over the iterations. Then

$$Time\ Per\ Iteration = \frac{Total\ Run\ Time}{Total\ Iteration\ Count}$$

For each difficulty level, the program is executed 10 times to obtain an average value. The sample data is taken across inputs from all levels. For instance, our level 5 experimental record is:

Level 5 problem with 5 second planning time(on a 16G RAM macbook)

Execution (lv5)	Cell Count	Iteration Count	total time (sec)	time per iteration (nano sec)	total steps	max step	peak memory usage	reached goal
1st	30	59,491,507	195.63s	3288ns	48	80	2948	yes
2nd	30	68,330,421	225.41s	3298ns	50	80	2902	yes
3rd	30	56,525,746	175.66s	3107ns	41	80	2838	yes
4th	30	54,932,175	175.98s	3203ns	37	80	2965	yes
5th	30	71,138,430	230.85s	3245ns	49	80	3071	yes
6th	30	72,097,015	225.83s	3132ns	51	80	2887	yes
7th	30	77,296,369	245.64s	3177ns	52	80	2933	yes
8th	30	74,546,293	235.66s	3161ns	54	80	2985	yes
9th	30	71,846,865	230.77s	3211ns	49	80	3110	yes
10th	30	86,369,589	276.21s	3198ns	59	80	2897	yes
Avg	**30**	**69,257,441**	**221.76s**	**3202ns**	**49**	**80**	**2954**	**100%**

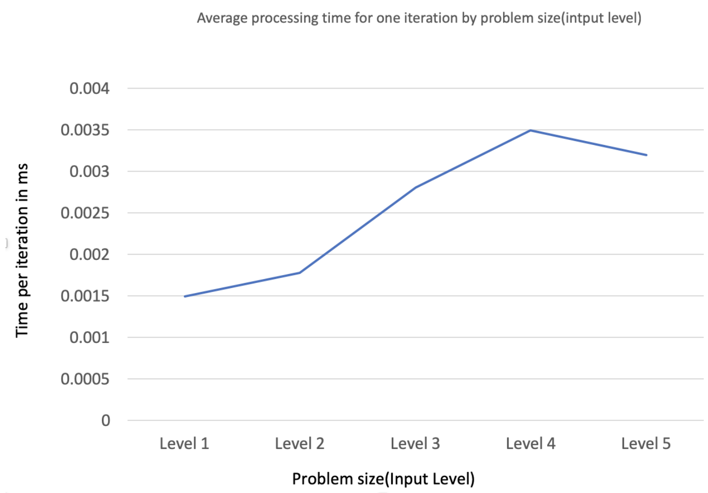
Table 3.1.2.2 Average time for one iteration in level 5

After collecting data from all input levels, here is the relation of problem size(level) and time consumption for one iteration:

Level	Average Iteration	Average Time	Avg Time per Iteration	Average Steps	Average Memory	Reach Goal Percentage
1	38.482.466	57.681s	1498ns	12.1	3303MB	100%
2	34.741.950	62.154s	1789ns	13.8	2982MB	100%
3	56.764.422	159.89s	2816ns	36.1	3329MB	100%
4	60.184.897	210.71s	3501ns	46.2	3030MB	100%
5	69.257.441	221.76s	3202ns	49.0	2953MB	100%

Table 3.1.2.3 The memory consumption with 5 second planning time

Based on the data collected above, the chart is plotted to illustrate the relation between average time for one iteration of different input levels:



It can be observed from the chart that in general the time consumption for one iteration on average increases along with problem size, reaching maximum(at around 0.0035ms) on level 4 and then drops slightly on level 5.

One possible reason for time consumption decrease on level 5 input is that the level 5 input file has maximum step as 80 while the level 4 maximum step is 90. In our implementation,the rollout simulation step count (rollout depth) is proportional to the maximum step from input, so the agent is constructing and traversing more nodes during the rollout in level 4 than level 5, thus impacting the time spent on each iteration as it needs to traverse longer distance.

To prove this, the previous level 5 is modified with max step as 90. After multiple executions, the average time on one iteration for this modified level 5 input is 3754 ns. The new plot with modified level 5 input file is:

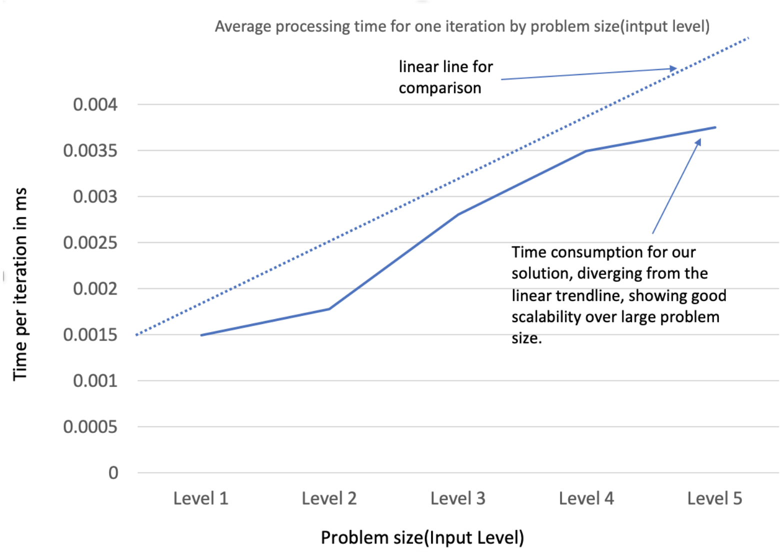


Figure 3.1.2.2 : Time consumption chart after using consistent max step input.

As observed, the average time consumption for one MCTS iteration gradually increases with the problem size consistently.

3.2. Memory Complexity

3.2.1. Memory Complexity for One Iteration Calculation

For any iteration, the agent has to store all data of the MCTS tree which consists of all the explored tree node, the number of tree node can be calculated using this formula:

Let **a** be the number of Possible Actions in action space, let **t** be the maximum allowed steps and let **d** be the depth of the tree. We can calculate the computational step for one iteration

$$Number\ of\ Nodes = \ Number\ of\ Action^{(Depth\ of\ Tree+1)} \ = a^{(d+1)}$$

Since the depth of the MCTS tree is **Expected** to be the **Number of Max Steps** so that one iteration will have the space complexity of **O(a^{t})**

3.2.2. Overall Memory Complexity

The overall memory complexity is the complexity of the memory that is run 10 times in different level of the game (we did 10 times because we need to avoid the outlier that representing our program). We just make an experiment in 10 times iteration and shows how the different size of state and action which are related with different level will increase memory consumption.

Screenshot for sample taking as below:

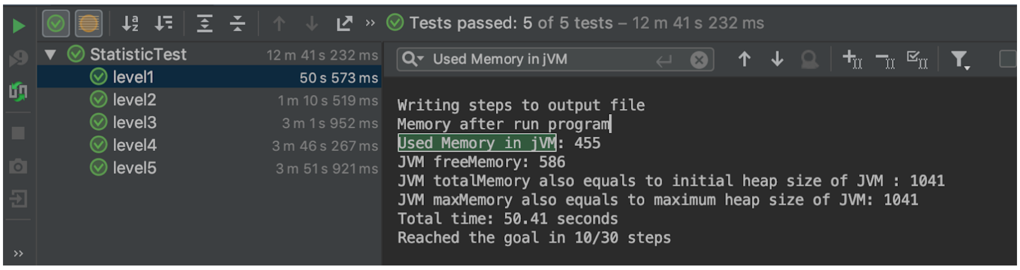


Figure 3.2.3.1 : Memory consumption on level 1 for the first iteration, it used 455 MB

Level	# 1	#2	#3	#4	# 5	#6	#7	#8	#9	#10	Average
1	442	519	538	495	457	597	472	541	393	546	**501 MB**
2	565	652	437	476	571	633	534	496	995	618	**597 MB**
3	648	757	676	545	686	535	524	497	511	722	**610 MB**
4	628	653	728	902	773	540	731	452	767	704	**687 MB**
5	503	419	740	749	493	496	411	731	566	602	**571 MB**

Table 3.2.3.1 The average memory comparison for every level with the given input

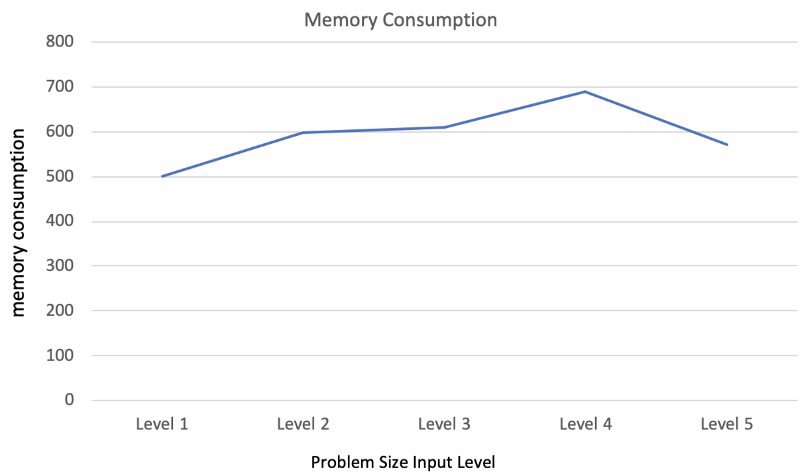
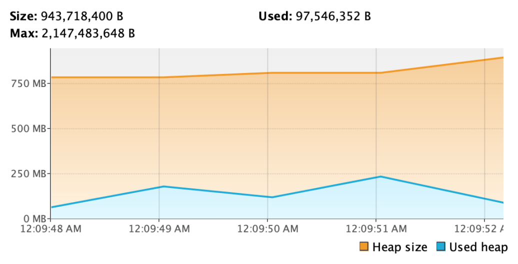
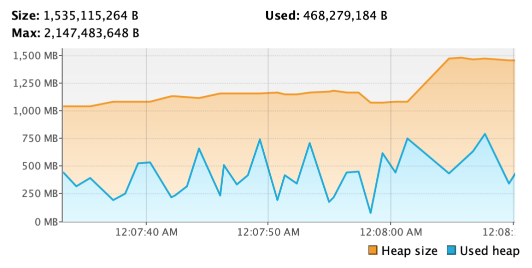
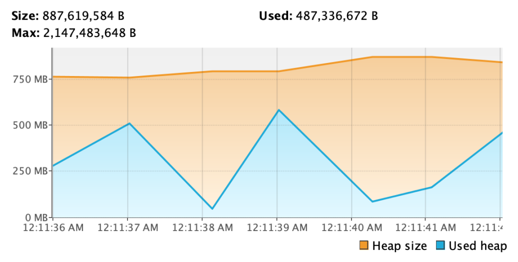
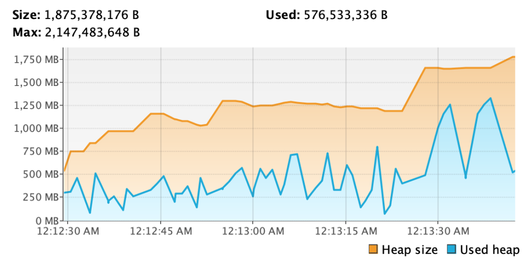
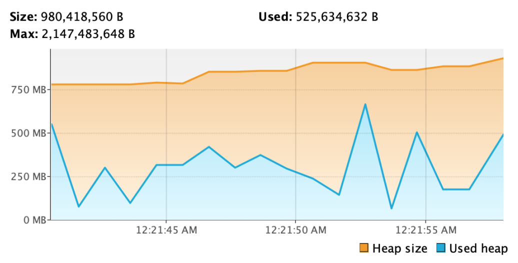
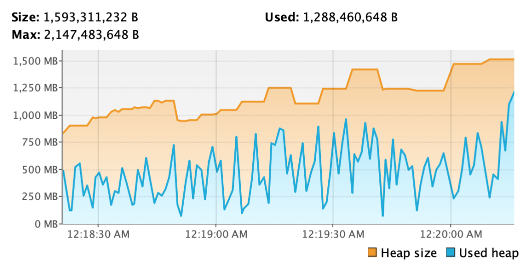
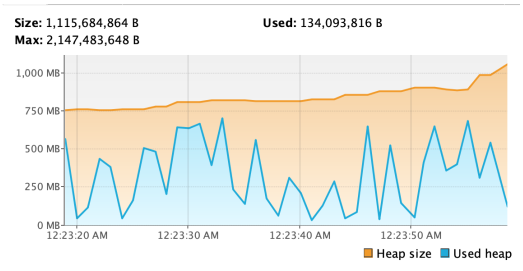
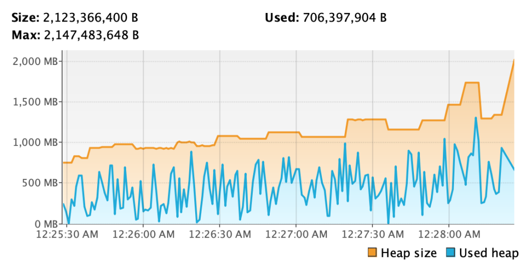


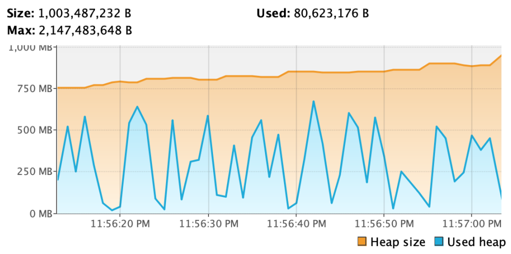
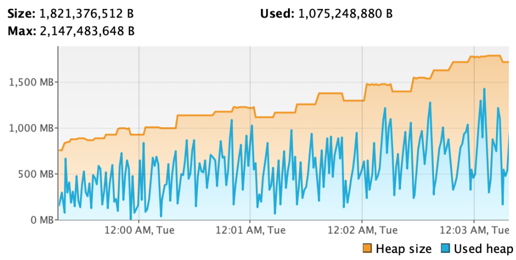
Figure 3.2.3.2 : The graph of memory consumption based on level in 10 times iteration.

The graph shows that memory consumption for each level. As can be seen on from the graph the memory consumption increased steadily between level 1 and level 4. After that memory consumption will decrease on level 5 because the maximum of step in level 5 is different with maximum step in level 4. In our experiment, we test the test case which is given by tutor where the maximum step for level 4 is 90 and level 5 is 80. By testing this test case our program or agent will take the least step on level 5, so memory consumption for level 5 will smaller than level 4.

3.2.4. Memory Consumption Charts

We used Java Virtual VM to record the memory consumption when solve the problems from level 1 to level 5 with two different planning time (1 and 5 seconds), the graph shows that the memory consumption is increase when the state space and action space increase (by increase the level).

<table>
<tr><th>Level</th><th>1 Second Planning Time</th><th>5 Second Planning Time</th></tr>
<tr><td>1</td><td>
</td><td>
</td></tr>
<tr><td>2</td><td>
</td><td>
</td></tr>
<tr><td>3</td><td>
</td><td>
</td></tr>
<tr><td>4</td><td>
</td><td>
</td></tr>
</table>

5		

3.3. Performance with Respect to Varying Planning Time

In the assignment, time consumption over decision for action is controlled by parameter EXPLORATION_TIME. Increasing time limit for one action decision will also increase the iteration times for that decision, thus the decision is made with more information. The following experiments are run in a 8GB computer.

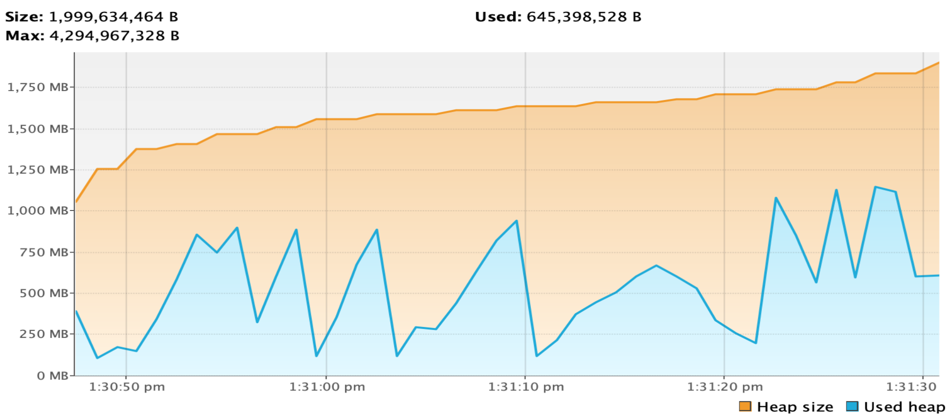


Figure 3.3.1 : Memory consumption when planning time 1 sec, it used around 645 MB

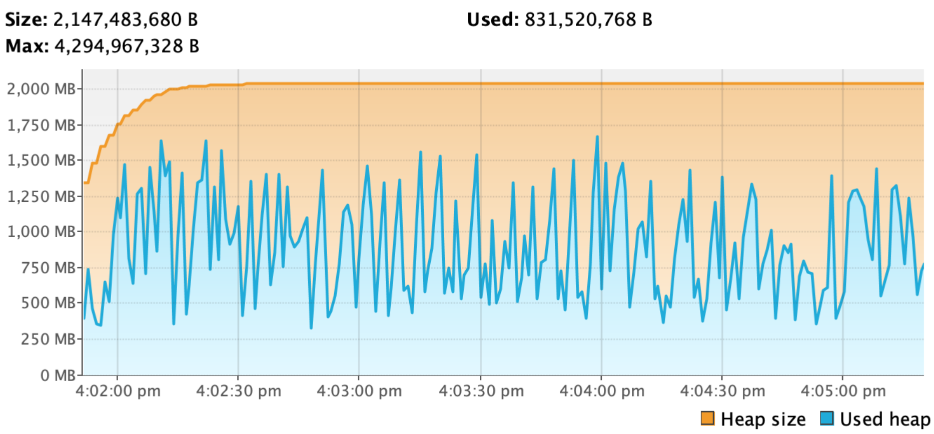


Figure 3.3.2 : Memory consumption when planning time 5 sec, it used around 831 MB

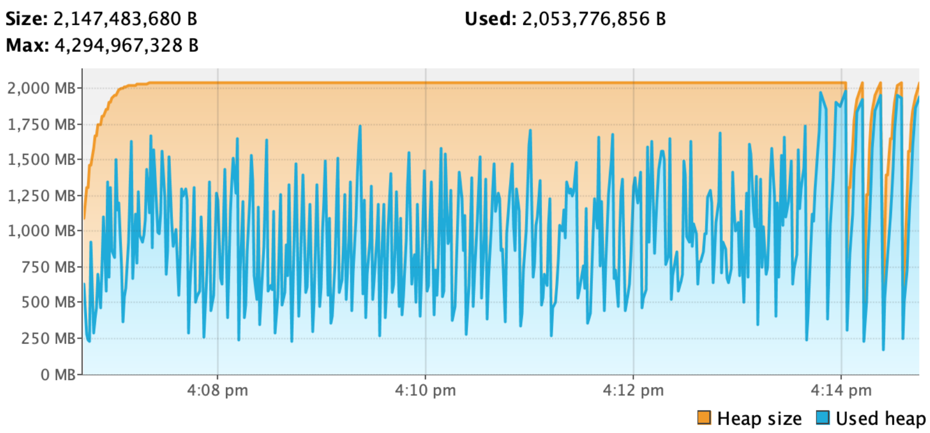
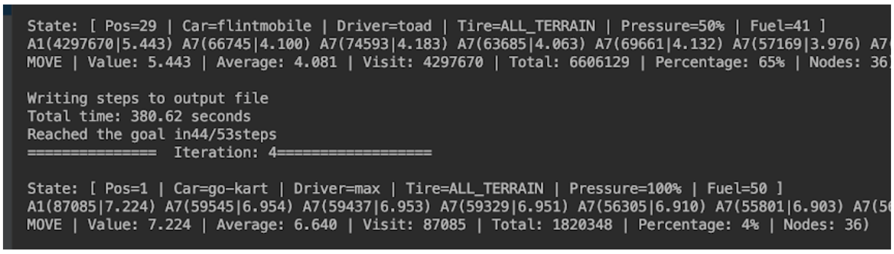


Figure 3.3.3 : 10 secs planning time result, it used around 1,000 MB memory

In this section, we want to compare the success rates given by various planning time. In this case the planning time will be set to 1 second, 5 second and 10 second. This program is run 10 times and we take the average number of step and the average of the programs will reach its goal. This experiment only runs at level 5 because we want to see results from difficult cases to show how success rates are affected by 1 second, 5 second and 10 second respectively.

Screenshot for sample taking as below:



```
State: [ Pos=29 | Car=flintmobile | Driver=toad | Tire=ALL_TERRAIN | Pressure=50% | Fuel=41 ]
A1(4297670|5.443) A7(66745|4.100) A7(74593|4.183) A7(63685|4.063) A7(69661|4.132) A7(57169|3.976) A7
MOVE | Value: 5.443 | Average: 4.081 | Visit: 4297670 | Total: 6606129 | Percentage: 65% | Nodes: 36

Writing steps to output file
Total time: 380.62 seconds
Reached the goal in44/53steps
===============  Iteration: 4=================

State: [ Pos=1 | Car=go-kart | Driver=max | Tire=ALL_TERRAIN | Pressure=100% | Fuel=50 ]
A1(87085|7.224) A7(59545|6.954) A7(59437|6.953) A7(59329|6.951) A7(56305|6.910) A7(55801|6.903) A7(5
MOVE | Value: 7.224 | Average: 6.640 | Visit: 87085 | Total: 1820348 | Percentage: 4% | Nodes: 36)
```

Figure 3.3.1.1 : Sample iterations

For every one iteration, we will only make maximum step up to 53 because we want to make the problem more difficult for our agent. Furthermore, we want to make clear comparison between planning time for each iteration by increasing the difficulty. 53 is the number of steps that come from the average step when the iterates program is 10 times.These table is result from solving level 5 problem with different planning time.

Iteration	1 second		5 seconds		10 seconds	
#1	Pass	48/53 steps	Pass	50/53 steps	Pass	37/53 steps
#2	Pass	52/53 steps	Failed	N/A	Pass	31/53 steps
#3	Pass	50/53 steps	Pass	41/53 steps	Pass	37/53 steps
#4	Pass	43/53 steps	Pass	42/53 steps	Pass	44/53 steps
#5	Failed	N/A	Pass	45/53 steps	Pass	38/53 steps

#6	Pass	51/53 steps	Pass	45/53 steps	Pass	29/53 steps
#7	Failed	N/A	Pass	43/53 steps	Pass	44/53 steps
#8	Pass	51/53 steps	Pass	37/53 steps	Pass	40/53 steps
#9	Failed	N/A	Pass	43/53 steps	Pass	45/53 steps
#10	Failed	N/A	Failed	N/A	Pass	38/53 steps
Avg	**60%**	**49/53 steps**	**80%**	**43/53 steps**	**100%**	**37/53 steps**

Table 3.3.1.1 The success rates and the number of steps with various planning time

Based on the last row of the table it can be observed that the increase of the planning time will affect the result positively in two aspects:

1. Rate of reaching the goal.
2. Average step for reaching the goal

To avoid exceeding time limit, only 1, 5, 10 second planning is included in the chart. It is expected that the steps for reaching goal would converge to optimal if the planning time reaches certain value (elbow point). However, convergence is not observed in this experiment even if 15-second planning (with average steps reaching goal being 32) is included into the scope.