# AN EXTENSION TO MONTE CARLO TREE SEARCH IN CONTINUOUS ACTION SPACE WITH REINFORCEMENT LEARNING PERSPECTIVE

DONGQING XIA

Abstract. For self-navigation in a dynamic environment, path planning needs to figure out a series of actions towards the predetermined destination. As for the need to generate a smooth trajectory, search in continuous action space is in great need. We present a heuristic-based planning algorithm able to produce a sequence of adaptive and optimal actions in such setting. It combines insights and therefore benefits from Monte Carlo Tree Search, kernel regression, and bellman equation in reinforcement learning domain. We present theoretical analysis of the algorithm, experimental results on a 2D simulator where an agent car searches a path to avoid obstacles and move smoothly towards the target point.

## 1. Introduction

Planning for systems operating in the real world involves dealing with a number of challenges not faced in many simpler domains. Firstly, the real world is an inherently uncertain and dynamic place; accurate models for planning are dif?cult to obtain and quickly become out of date. Secondly, when operating in the real world, time for deliberation is usually very limited; agents need to make decisions and act upon these decisions quickly. Fortunately, a number of researchers have worked on these challenges. To cope with imperfect information and dynamic environments, reinforcement learning policy network and efficient replanning algorithms are two areas that have been developed to be promising solutions based on dynamic environment. In policy network domain, agent can have an adaptive policy (even in POMDP); however, training an appropriate policy network can be difficult. Planning algorithms such as A$*$, D$*$ and their extensions are efficient and more controllable. These algorithms maintain optimal solutions for a fraction of the computation required to generate such solutions from scratch. However, when the planning problem is complex, it may not be possible to obtain optimal solutions within the deliberation time available to an agent. Recently, Master has shown a possibility in a combination in action planning and value network with a tool of Monte Carlo Tree Search.

As of now, there has been relatively little interaction between these three areas of research. Reinforcement learning utilize bellman equation to iterate an optimal value-estimate or policy solution. It usually loses sub-optimal information for future planning. Planning algorithms provides some optimal and sub-optimal solutions in path planning, but they needs a raster map to search. Meanwhile, the action

Received by the editors April 24, 2017.

space is discrete and fixed, while most of the path planning problem in real world needs to explore in continuous action space such as velocity and steer in driving. Monte Carlo Tree Search shows its power in finite horizon planning problem, but it pays less attention in considering new action based on sampled ones. There is an extension in MCTS which combines a kernel regression in action generation. However, in these two papers, the estimator for specific action-state value matches the conditions where two or more agents are in a game state, and it is not suitable for a closely related sequential decisions for one agent. For example, our current work focuses on an agent (with partial observability) wants to make an sequential decision to carry out a smooth trajectory to avoid obstacles and other cars in a short period of time. In this paper, we present a heuristic-based, Monte Carlo Tree Search algorithm that bridges the gap between these three areas of research. Our algorithm, V-KRUCT, which means an variation of Upper confidence Bound applied to Tree based on Kernel Regression, allows the agent to search an optimal action sequence to move forwards to the predetermined destination point.

This paper is organised as follows. We begin by discussing necessary background knowledge, focusing on Monte Carlo Tree Search, A$*$ path planning algorithm, kernel regression and reinforcement learning bellman equation. Next, we introduce our novel algorithm, V-KRUCT. We demonstrate the characteristics of our algorithms and benefits of the approach through experimental results and conclude with discussion and suggestions in future work.

## 2. **Background**

We start by introducing mainstream searching algorithms such as Monte Carlo Tree Search and A-star. Then we describe core modules in our V-KRUCT algorithm.

### 2.1. **Monte Carlo Tree Search.**

Monte Carlo Tree Search (MCTS) is a simulation-based search method to find an optimal decision sequence within finite horizon settings. At every decision node, a fixed number of simulation are requested. Each simulation process contains 4 setps, *selection*, *expansion*, *evaluation*, and *backup*. Simulation starts by visiting nodes in the tree. A selection function is used to selected the node, which contains the information in this state along with the action. State is then transmitted into the respective successor state by taking the action. Once a visited node does not have any immediate child, simulation starts to expand the tree by adding a child node to this visited node. Then a *rollout* policy is called to evaluate how good this child node is (i.e how good this successor state is). Rollout policy can be either a random action selection, or a fast rule-based selection, or even a mixture of the two approaches. And rollout out policy can be applied from this staring node to a terminal state, or, just a fix number of steps if simulation budget is limited. Evaluation feedback value is updated in this child node, and then the information stored in the tree is updated by backup process.

The most common selection function for MCTS is Upper Confidence Bounds Applied to Trees (UCT). Each node maintains a mean value of the state, $v$, and also the number of times it has been visited in the past, $n$. Selection function is usually expressed in this way:

$$\arg\max v_i + C\sqrt{\frac{\log\Sigma_j n_j}{n_i}} \tag{2.1.1}$$

C is a domain determined constant, which serves to compute one side confidence interval based on Chernoff-Hoeffding bound. It controls the exploration-exploitation ratio.

The success of MCTS is due to its good usage of computation budget. It prefers to explore the nodes with a higher state value and thus have a larger possibility to discover the optimal decision sequence. However, child node is added by rules determined by human in the every first place. For example, in the game of Go, only the places, which hold the policy probability more than 0.1, can be added as leaves to the tree. In this way, the pure Monte Carlo Tree Search doesn't have abilities to generate new actions beyond human knowledge. Meanwhile, when Monte Carlo Tree Search is applied to a problem containing continuous action space, either we can use limited default action, or naive discretization. However, the first approach lose a lot of optimal possibilities in action space, and the second approach can result in a difficulty in deepening within a tree.

## 2.2. **A-star Path Planning.**

A-star solves problems by searching among all possible paths to the solution (goal) for the optimal one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It starts from a specific node of a graph and then it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node. At each iteration of its main loop, A-star needs to determine which of its partial paths to expand into a longer paths. It does so based on an estimate of the expense needed to go to the goal node. Specifically, A-star selects the path that minimizes the following expense expression:

$$f(n) = g(n) + h(n) \tag{2.2.1}$$

$n$ is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A-star use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest $f(n)$ value is removed from the queue, the $f$ and $g$ values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower $f$ value than any node in the queue (or until the queue is empty). The $f$ value of the goal is then the length of the shortest path. The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

As for A-star algorithm, it is useful in finding optimal path to predetermined goal node. However, for each decision node, agent's action set is still predetermined based on some set of rules in the context. A-star has no ability to help generate new actions in continuous action space or problem in high dimensionality. A-star also needs global information to estimate the cost so as to find out the optimal solution. For partial observable navigation, it has limitations. Meanwhile, every A-star loop will be terminated with an optimal path to goal node or a not reachable signal under an static context. Once an optimal path is found, the agent can executed the action sequence. But as an example of driving, we need to adapt to changes in real world. Actually, we need to find an algorithm which has ability to search under a changing condition.

## 2.3. **Kernel Regression.**

Kernel Regreesion is a nonparametric method for estimating the conditional expectation of a real-valued random vatriable from data. In its simplest form, it estimates the expected value of a point as an average of the values of all points in the data set, weighted based on a typically non-linear function of the distance from the point. The function defining the weight given a pair of points is called the kernel and funther denoted $\mathcal{K}$. For a data set $(x_i, y_i)_{i=0}^n$, the estimated expected values is:

$$\mathbb{E}(y|x) = \frac{\sum_{i=0}^n \mathcal{K}(x, x_i) y_i}{\sum_{i=0}^n \mathcal{K}(x, x_i)} \tag{2.3.1}$$

The kernel is typically a smooth symmetric function, such as Gaussian probability density function. However, asymmetric kernel functions can also be used in well-motivated cased. An important quantity related to kernel regression is kernel density, which quantifies the amount of relevant data available for a specific point in the domain.

## 2.4. **Reinforcement Learning.**

As for heuristic information based searching, we need to determine a goal function (expense) to optimize. Also, we base on these statistics to estimate a state value so that we can selection some actions and find a path. We need to determine some cost function along with the estimate funtion for future cost. And for every new data comeing from environment, we need to update the cost in precessor nodes. Here, we get inspiration from reinforcement learning basis, Bellman equation, to propose a new way to evaulate an action-state value. Bellman equation express the relationship in immediate reward and mean return in the following way:

$$Q(s, a) = E_s[r + \gamma \cdot Q(s', a')|s, a] \tag{2.4.1}$$

Through the Q value iterations in the tree, we can select a sequence of optimal actions to achieve high mean return.

## 3. **Kernel Regression UCT Variation**

The main idea behind KR-UCT is to increase the number of children of a node as it is repeatedly visited.

**Algorithm 1** Variation of Kernel Regression UCT

```
1:  function V-KRUCT($simulator$, $thisNode$)
2:      if simulator(thisNode.info) is terminal then
3:          return thisNode.reward, thisNode.visitNum, $false$
4:      end if
5:      if thisNode.depth is maxDepth then
6:          return thisNode.value, thisNode.visitNum, $false$
7:      end if
8:      $expanded \leftarrow false$
9:      if thisNode.children $\leq$ len(action considered at this node) then
10:         add resulting states as children to thisNode for $a \in$ action set $\mathbf{A}$
11:     end if
12:     totalVisitNum $= \sum$ child.visitNum
13:     selectedNode $= \arg\max_{a \in \mathbf{A}} \mathbb{E}(\text{v}|\text{a}) + C\sqrt{\frac{\log(\text{totalVisitNum})}{\text{a.visitNum}}}$
14:     if $\sqrt{\text{totalVisitnum}} \leq |\mathbf{A}|$ then
15:         $value$, $visitNum$, $expanded \leftarrow$ V-KRUCT($simulator$, $selectedNode$)
16:     end if
17:     if not $expanded$ then
18:         $newAction \approx$ selectedNode.action + random process $\mathbb{P}$
19:         $newNode \leftarrow$ resulting info by taking $newAction$ to thisNode.children
20:         add initial actions to $newNode$
21:         for $i = 0 \rightarrow rolloutTimes$ do
22:             $rv$ += ROLLOUT(simulator, newNode, steps)
23:         end for
24:         newNode.value $\leftarrow$ newNode.reward + $\gamma \frac{rv}{rolloutTimes}$
25:         newNode.visitNum $\leftarrow 1$
26:         selectedNode.value $\leftarrow \omega\cdot$newNode.value + $(1\text{-}\omega)\cdot$selectedNode.value
27:         selectedNode.visitNum += 1
28:     end if
29:     thisNode.value = thisNode.reward + $\gamma \cdot \frac{\sum_{i \in \text{child}} v_i \cdot n_i}{\sum_{j \in \text{child}} n_j}$
30:     thisNode.visitNum += 1
31:     return thisNode.value, thisNode.visitNum, $true$
32: end function
```

3.1. **Selection.**
At each decision node (usually starts from tree's head), our algorithm uses idea of UCB formula to select amongst already explored nodes. Each node maintains the number of visits, $n_i$ and an estimate of state value, $v_i$, along with some other neccessary information, such as immediate reward and action (which taken from parent node to THIS child node) etc. When checking $\mathbb{E}(v_{N_t}|a)$, means that, checking for its respective child node value $\mathbb{E}(v_{N_{t+1}})$ by taking action $a$. For the clearity of the expression, we regard $\mathbb{E}(v_{N_t}|a)$ as $\mathbb{E}(v)$. We incorporate idea of reinforcement learning to evaluate a state's value from immediate reward. As nodes in trees are visited through every simulation process, the $n_i$ and $v_i$ information in visited nodes

and related nodes are updated.

$$\mathbb{E}(v) = \text{reward} + \gamma \cdot \frac{\sum_{i \in \text{child}} v_i \cdot n_i}{\sum_{j \in \text{child}} n_j} \tag{3.1.1}$$

$$n_i \leftarrow n_i + 1 \tag{3.1.2}$$

So in selection process, we refer to equation (2.1.1) and select the node for further exploration. When a leaf node in a tree is visited, we initialize its value as its immediate reward. We tend to explore a certain successor node with maximum reward first. However, with the value updated from its successor nodes, in another iteration of simulation, this node might be selected again. The scaling constant $C$ serves the same role as in vanilla UCT, controlling the tradeoff between exploring less visited actions and refining the value of more promising node. As we can see, when some nodes maintaining similar values, the node with less visit number tend to have a larger bonus and have a larger possibility to be the next node for exploration. (This constant $C$ should be experimentally tuned for each specific domain.)

## 3.2. **Expansion.**

After selecting successor node with the highest UCB value, the algorithm continues by either improving the estimated value of the selected node by recursing on its outcomes (line 15), or improving the estimated value of the action by adding a new child node (line 17-28). This decision is based on some sub-linear functions (line 14). As showed in algorithm 1, we can expect the algorithm first select a seemingly optimal node out from default action set to begin a sequence estimation as the tree grows deeper. When the total visit number exceed a bound, we tend to generate more new actions at this node (we refer this to a process of widening). In the event of reaching a terminal state, within the tree, a new action is always added at the leaf's parent (line 17). Also, in some settings, we might not reach terminal state within our finite searching horizon. In case the tree unlimitedly grows deeper (as at every new node it satisflies line 14), we determine a maximum depth in tree. When leaf's depth reaches maximum depth, then a new action is always added at the lead's parent as well. When adding a new successor node by taking a new action, we add an random process to vary a little bit from the so far best selected action. In this way, we explore the continuous action space at the specific node. After identify a new action, the succesor state is added to this node's children list, and new node is then evaluated with fast rollout policy.

## 3.3. **Simulation and Back-propagation.**

When a new successor node is added to the tree, either a complete simulation (to terminal state), or an imcomplete one (fixed simulation steps) is carried out (line 23). We can use a totally random selection as rollout policy, or rule-based policy. The value $rv$ can be viewd as mean return of successor state of this new node, so that we update newNode's value in line 24, using idea of value iteration from reinforcement learning. Clearly, we initialize the visit number of new node to be 1 (line 25). Since, we generate new action refering to some selected actions, so we want to update this strongly-related node to share statistics (line 26-27). After all, we update parent nodes in an incremental way (line 28-29).

## 3.4. **Final Selection.**

After the computational budget is exhausted, we still need to use the results of the search to select an action to execute. It is common to use the action with the highest samlped mean along with a promising bonus at the root. However, we select the action at the root with the greatest lower confidence bound (LCB) using this equation:

$$\arg\max v_i - C\sqrt{\frac{\log\Sigma_j n_j}{n_i}} \tag{3.4.1}$$

The constant $C$ acts as a tradeoff between chossing actions that have been sampled the most (i.e., are the most certain) and actions that have high estimated value. This constant does not have to be the same as the constant used in the upper confidence bound calculation. Actually during final selection, we can make one or more decisions, depending on the tree's depth. What's worthy to notify is that, when the goal node is reached, the tree should be cleared.

## 4. **Evaluation**

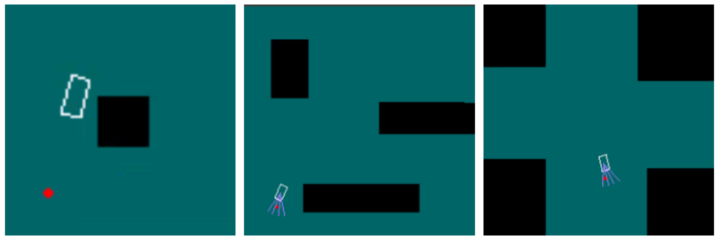
We testify our algorithm under the partial observable navigation setting. Specifically, we focus on planing for autonumous driving. Although the dynamics of driving is complex, we can simplify the environment into some core modules and carry out insightful experiments.

### 4.1. **Simulator.**

Our simulator is developed simply on pygame module. An environment consist some obstacles and a small car agent, as showed in figure. Agent (we refer to this car) has limited statistics from lidar sensors in 360 degrees. During searching process, simulator provides an immediate reward to agent, which is a scalar. Agent gets larger reward when either getting closer to target point, or angle difference between car's driving direction and target direction (a vector out from car's geometry centor to target point) getting smaller. However, car agent can't separate this scalar to extract concrete imformation about the target point so that statically planning in global view is not possible. In searching process,Here, we set the maximum detection range to be 25 pixels. Also, agent can have a speed which ranges from -5 to 5 pixel per step, and a steer which ranges from -$\frac{\pi}{8}$ to $\frac{\pi}{8}$. Rollout policy is set to be a mixture of random action selection and rule-based policy. This proportion is controlled by a constant $\epsilon$, the idea borrowed from reinforcement learning's tradeoff between exploration and exploitation. In our algorithm, $\epsilon$ is 0.7, meaning that 70% of the time we use random action selection while 30% of the time, we use rule-based avoidance policy on the agent.

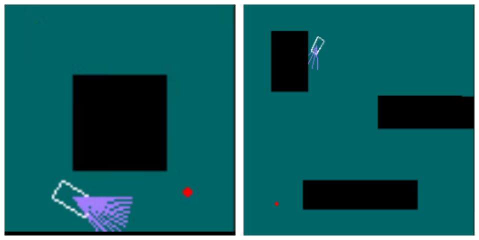
### 4.2. **Reinforcement Learning in Searching.**

In Fig.3 we observe that first of all, car agent can't directly make a steer towards the target point due to its own action space limitation. So it explore another way to avoid obstacles. In first stage of simulation, value of action node is initialzed as immediate reward, however, after several steps agent hit the obstacles, so the the seccessor stages' value is updated with a large negative reward by back-propagation following equation (3.1.1). Then in the next round of V-KRUCT simulation, the left turn action is no longer the best action, so that agent choose to move forwards



(a) simple (b) maze (c) cross

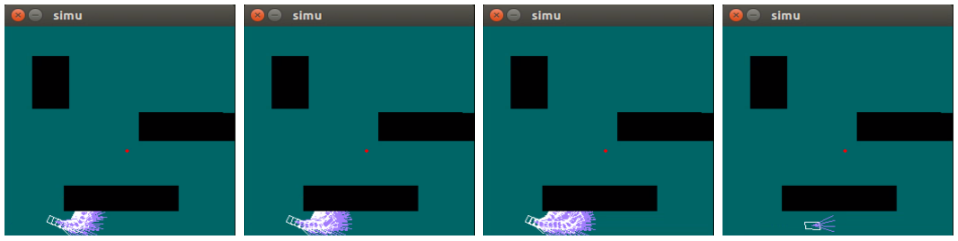
Figure 1. 2D-simulator is developed based on pygame module. User can change border information and playground size in the simulator. (a) Simple border in window size as 100x100 pixel. (b) A more complex version of maze in window size of 255x255 pixel. (c) User can also design a cross road in the 255x255 window.



(a) simple (b) maze

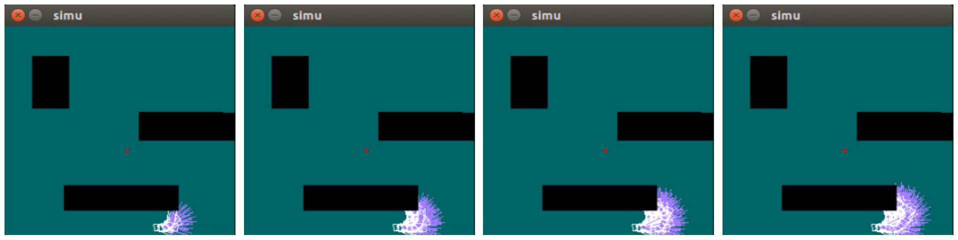
Figure 2. Car agent has a virtual lidar sensor which can help angent observe 25 pixel in 360 degree around the car geometry center. Simulator provides an immediate reward to agent, which is a scalar. Agent gets larger reward when either getting closer to target point, or angle difference between car's driving direction and target direction (a vector out from car's geometry centor to target point) getting smaller. However, car agent can't separate this scalar to extract concrete information about the target point so that statically planning in global view is not possible. Here, only [-30°, 30°] sensorary statistics in car's polar coordinate are displayed in the images. (a) Simple border in window size as 100x100 pixel. (b) For simplicity, only show 3 sensorary feedback, -30°, 0°, -30°

and then try turnning left (which you can observe simulation trajectory in simu-2 in Fig.3) Agent still hit the obstacles. However, this time, agent live for a longer time, so with the $\gamma$ parameter, terminal negative feedback is updated with a less weight to the action of moving forwards. So next round agent tends to move forwards for a longer time, and then at some point still want to turn left. After computation budget is exhausted, agent uses search tree results to make a sequence of decision, i.e. moving forwards for several steps, as showed in 'step' in Fig.3. Although agent doesn't have a optimal path to target point, it can still maintain an sub-optimal solution to avoid obstacles and try searching path in next round. Same story in

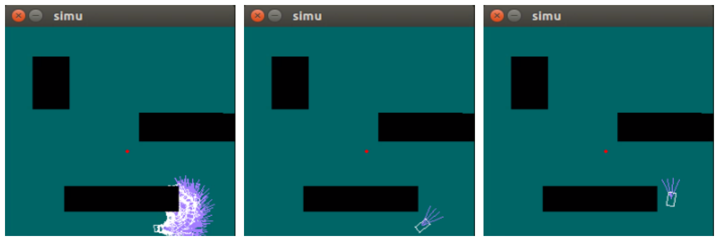


(a) simu-1 (b) simu-2 (c) simu-3 (d) step

FIGURE 3. Agent maintains an sub-optimal solution to avoid obstacles by simulation and back-propagation in the tree.



(a) simu-1 (b) simu-2 (c) simu-3 (d) simu-4



(e) simu-5 (f) step-1 (g) step-2

FIGURE 4. Agent search in its action space and tends to make a soft turn to reach the target point.

Fig.4. We can observe the agent's simulation starts from the action which could receive largest immediate reward. However, with tree search and update, agent tend to make a sequence of decisions to maximize mean return.

4.3. **Exploration in Continuous Action Space.**
Since we have set a sublinear function as the bound to decide deepening or widening in the tree, we observe agent can explore a sequence of optimal decision and generate some other optimal nodes once it obtains some seemingly optimal solution sequence. For example, we can see the simulation trajectory in Fig.5(a). Agent samples distribute near actions of moving forwards and actions of turning right, because actions of turning left result in a hit into obstables. And from Fig.5 we observe more samples of moving forwards than moving rightwards. We only add a default action set, (moving directly forwards, moving rightwards with maximum steer to be $\frac{\pi}{8}$ and leftwards as well), as initialization to each node when it is a leaf. However, due to

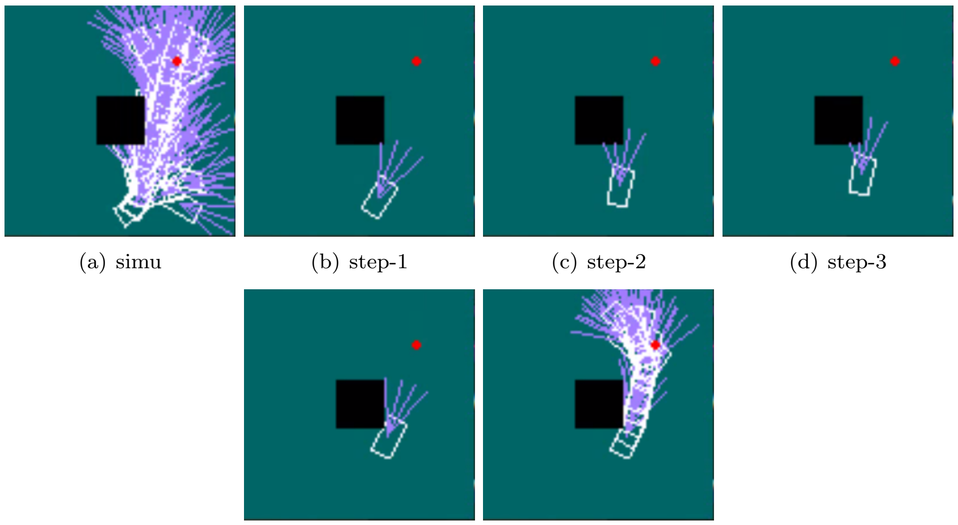


Figure 5. Addition of random process generates new actions so that agent can have a soft trajectory towards the target point.

an addition in random process, we can observe the moving forwards trajectory are with some small steer, which would result in a soft trajectory. Then after 4 step (which are shown in Fig.5), car agent can tune its own driving direction towards the target point. Agent also hold a policy network to generate its own policy (but sometimes not optimal). However, it gives more candidates to generate optimal solutions.

## 5. Discussion

Mainly we discuss some hyperparameters of this searching algorithms, such as constants in action exploration and final selection, $\epsilon$ in rollout policy and decision bound selection. Also, we suggest some directions for future work, such as random process selection and generic algorithms in action generations etc.

### 5.1. Hyper-parameters.

Constants in action exploration and final selection can be different. Usually, we select a positive constant $C$ in exploration so that according to equation (3.1.1), the node with less visit number can have a higher selection bonus even though it doesn't have a high enough value $v$. However, in the final selection, we can use the negative $C$ to be the constant. We tend to choose the node with a combination of high enough state value $v$ and a large visit number (which means it is of great certainty.) However, it should be a fine-tuned constant C, otherwise it could not serve these functionalities.

$\epsilon$ in rollout policy is now set to be 0.7. It means in 70% of the time, agent randomly select an action to simulate how good a state is. For example, if a place is far away from obstacles, then random action selection can also result in a good enough state value. If we use a rule-based policy, for example, tend to move towards the open space based on statistics received from the virtual sensor, then the narrowness of the space might not be discovered by simulation. However, a totally random action selection might need a great amount of experiments to reveal approximately true state value. To use rule-based policy can avoid hitting and accelerate path searching. Generally speaking, an $\epsilon$ which is greater than 0.5 means using more than half of the time for random simulation approaching the approximate state value.

Decision bound now is set to be a sub-linear function, e.g. $\sqrt{\text{totalVisitnum}} \leq |\mathbf{A}|$. Generally, this decision bound determine the tree's width and degree of exploration in continuous action space. If a bound is rather small (i.e. some constants) with respective to fast growing visit number, then according to our algorithm, the tree can will become widening in an rather early stage without enough simulation on default action set. However, if a bound is a function and grows faster than the node's visit number in some ways, node can not generate useful new action child node at any time.

5.2. **Future Work.**
Future work can be done in several related areas.

So far, we choose Gaussian random process to be base of our generator of new action and the kernel for regression as well. However, kernel can be selected to fit the uncertainty in certain context. Meanwhile, in the algorithm we only append the new action node value to the selected node, on which it is based and generated. These two nodes have stronger relationship within the action space, so their node value can be shared to some degree. However, we consider some other nodes can also be neighbours to these two nodes so that implement an KNN kernel regression here is also reasonable (based on consideration of computation budget).

As for generation of new action, we will consider to add in generic algorithm in an early stage. As for an continuous action space but with only sparse default actions to be considered during initialization, using kernel to generate new actions result in a loss in exploration in great action space. We observer, new actions distribute closely to default actions. To implement an generic algorithm during an early stage can generate a wider distribution of new action candidates for further refining using kernel method in action generation.