ECE 276B
Project 3

Motion Planning

Due Date: March 9, 2018

Professor: Nikolay Atanasov

Chuqiao Song

A53239614

Introduction

Sampling-based searching RRT*

Instead of using searching-based planning to solve shortest path, the sampling-based planning is another way to solve this problem, which is well-suited for high-dimensional planning as it is faster and require less memory than search-based planning in lots of domains. However, not like searching-based planning guaranteeing to find optimal path, the sampling based-planning only provides suboptimal bounds on the solution. In the first section, I implemented the RRT star algorithm to solve the shortest problem.

Balancing inverted pendulum:

In section two, the task is about solving an optimal control problem in order to balance an inverted pendulum. It is an infinite horizon dynamic programming, which means the time sequence is infinite. Therefore, I discrete the states and control spaces to formulate a finite-state MDP problem, and then solve it use value iteration and policy iteration.

Problem Formulation

1) Sampling-based searching RRT*

Building configuration space: Given the maze configuration input file, which specifies the corner coordinates of the obstacle, the configuration space should be build based on these points.

Implementing RRT* algorithm: Given the start point and terminal point, there are basically 3 steps for each iteration to solve this shortest path problem, sampling, extending, and rewiring. For sampling, try to get random sampling point Xrand in free space for each iteration. In extend step, given the sampled Xrand, Xnearest around it has to be found to do steering to get Xnew ε distance. Then, a checking step has to be implemented to see whether there is point, Xmin, with accumulated smallest cost and connect it with Xnew to form an edge. Finally, in rewiring step, trying to find nodes in graphs that Xnew can be its parent. For each edges connection and steering, the collision checking has to be implemented, to see whether these actions are allowable.
Visualization of RRT*: For each edge found in RRT, it has to be visualized with in real time.

2) Balancing inverted pendulum:

Formulating MDP problem: Given inverted pendulum model (should be formulated), the states and control space has to be discretized. Additionally, the transition probability matrix representing P (s'| s, a) for each state and action (control) has to be found with shape of (A, S, S'). Finding cost matrix, C, representing g (s, a) of the cost from one state to another state with its action, with shape (s, a) is another thing to formulate MDP problem.

Value iteration: The value iteration has to be implemented to find optimal action for each state by doing iteration to let optimal value function V(i) to converge to J*(i) with minimum value selection in each iteration.

Policy iteration: There are two steps to implement policy iteration, policy evaluation to find total cost $J^{\pi}(i)$ given specific policy π by solving linear system of equation, and policy improvement to find min value and its new stationary policy π', and it has to be iterated all above two steps until getting converge.

Technical Approach

1) Sampling-based searching RRT*

In general, the RRT* algorithm is better than original RRT start, as the shortest path find in this way is more directed. The 'edges' is set of tuples storing two points is with its parent node and child node, and we terminate the searching when there is an edge and its children node is the target node. Therefore, by knowing this relationship, the tracing back function can be implemented to find the shortest path.

A. Building configuration space: In this project, I am given the corner coordinates of each obstacle in maze. To generate the maze and prepare for collision checking step, I use a python directory called shapely to formulate each obstacle. Also, to visualize the maze and path finding I use pygame with some scaling and shift for each real coordinate.

B. Implementing RRT* algorithm

Addition to collision checking, shapely package is being used to draw the obstacles and the checking line either for Xnearest and Xnew, and Xnear and Xnew. Then the intersection function in the package is used to check if the edge has collision with the obstacles.

For the sampling Xrand, I randomly generate the coordinate in uniform distribution with the real map size (-15, 15) by using random.uniform(-15,15) for x and y separately. Also, to apply somehow directed sampling method, for each 1000 iteration I set the terminal point to be Xrand.

In the extend step, which includes finding Xnearest point, steering ε=1.5 distance to get Xnew point. To find the nearest neighbor, I search all of the vertex in my Vertex set which is generated by Xnew point from previous iterations. After getting Xnew, the collision checking is implemented here to say whether the edge connect Xnew and Xnearest is allowable. If not redo the whole iteration again, which means sampling again to get another Xrand. Otherwise, implement Xnear function to find all near points around the Xnew within some specific Radius = 2 in my case, and use these nodes to check whether there is a Xmin exist here can replace Xnearest to connect the Xnew, Xmin to Xnew edge. The minimum acculumated path cost of the nodes in Xnears is being searched, so that the corresponding Xmin can be found.

RRT*: Extend Step

- Generate a new potential node x_{new} identically to RRT
- Instead of finding the closest node in the tree, find all nodes within a neighborhood $\mathcal{N}$
- Let $x_{nearest} = \arg\min_{x_{near} \in \mathcal{N}} g_{x_{near}} + c_{x_{near}, x_{new}}$, i.e., the node in $\mathcal{N}$ that lies on the currently known shortest path from x_s to x_{new}
- Add node: $\mathcal{V} \leftarrow \mathcal{V} \cup \{x_{new}\}$
- Add edge: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(x_{nearest}, x_{new})\}$
- Set the label of x_{new} to $g_{x_{new}} = g_{x_{nearest}} + c_{x_{nearest}, x_{new}}$

In rewiring step, the main idea is to find whether the Xnew node can replace the father of the node in Xnear\{Xmin} set to lower the accumulate path cost of these node, and if there is, disconnect the edge between xnear and its parent, and replace its parent by Xnew to get new edge.

RRT*: Rewire Step

- Check all nodes $x_{near} \in \mathcal{N}$ to see if re-routing through x_{new} reduces the path length (**label correcting!**):
- If $g_{x_{new}} + c_{x_{new}, x_{near}} < g_{x_{near}}$, then remove the edge between x_{near} and its parent and add a new edge between x_{near} and x_{new}

To visualize the edge found by RRT, the pygame is being implemented with some scale and shift to get draw the edge in real time in black, and for the rewired line, the write line is drawn to cover the previous line.
Following is the main step for RRT* algorithm.

Algorithm 1: Body of RRT and RRG Algorithms

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$; $i \leftarrow 0$;
while $i < N$ **do**
  $G \leftarrow (V, E)$;
  $x_{\text{rand}} \leftarrow \texttt{Sample}(i)$; $i \leftarrow i + 1$;
  $(V, E) \leftarrow \texttt{Extend}(G, x_{\text{rand}})$;

Algorithm 4: $\texttt{Extend}_{RRT^*}$

$V' \leftarrow V$; $E' \leftarrow E$;
$x_{\text{nearest}} \leftarrow \texttt{Nearest}(G, x)$;
$x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x)$;
if $\texttt{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
  $V' \leftarrow V' \cup \{x_{\text{new}}\}$;
  $x_{\min} \leftarrow x_{\text{nearest}}$;
  $X_{\text{near}} \leftarrow \texttt{Near}(G, x_{\text{new}}, |V|)$;
  for *all* $x_{\text{near}} \in X_{\text{near}}$ **do**
    if $\texttt{ObstacleFree}(x_{\text{near}}, x_{\text{new}})$ **then**
      $c' \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$;
      if $c' < \texttt{Cost}(x_{\text{new}})$ **then**
        $x_{\min} \leftarrow x_{\text{near}}$;
  $E' \leftarrow E' \cup \{(x_{\min}, x_{\text{new}})\}$;
  for *all* $x_{near} \in X_{\text{near}} \setminus \{x_{\min}\}$ **do**
    if $\texttt{ObstacleFree}(x_{\text{new}}, x_{\text{near}})$ *and* $\texttt{Cost}(x_{\text{near}}) > \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}}))$ **then**
      $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E' \leftarrow E' \setminus \{(x_{\text{parent}}, x_{\text{near}})\}$;
      $E' \leftarrow E' \cup \{(x_{\text{new}}, x_{\text{near}})\}$;
return $G' = (V', E')$

2) Balancing inverted pendulum:

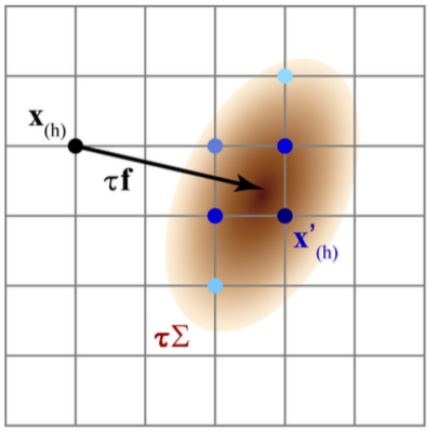
Formulating MDP problem: The first thing I do is to formulate the inverted pendulum with physical meaning parameters (a=1, b=0, σ= (2*2 identity matrix), k=1, r=1, γ=0.3) with its corresponding stage cost in continuous time. X1 is for state of pendulum angle, X2 is for state of angular velocity, u is for control input, and ω is for motion noise.

$$d\mathbf{x} = f(\mathbf{x}, u)\, dt + \sigma\, d\omega \qquad f(\mathbf{x}, u) := \begin{bmatrix} x_2 \\ a \sin x_1 - b\, x_2 + u \end{bmatrix}$$

$$g(\mathbf{x}, u) = 1 - \exp(k \cos x_1 - k) + \frac{r}{2} u^2$$

The second thing I do here is to discretize the states and control space, and the paramters of this model is list here [δt=0.1, n_1=80, n_2=20, n_u=50, θ_max = pi, v_max=8, u_max=8], where δt is the time interval, n1, n2, and n_u is corresponding the number of discretized grid points for angle of pendulum, angular velocity, and control space. For θ_max, v_max, u_max, they the range of angle of pendulum [-θ_max, θ_max] and angular velocity [-v_max, v_max], and control [-u_max, u_max] which I want to discretize. In general, the way I

discretize the space following by equation $\frac{2v_{max}}{n_2}\delta t \approx \frac{2\pi}{n_1}$, and apply similar reasoning to velocity and control discretization. Also, the transition probability can be generated, $p_f(x'|x,u) \sim N(x + f(x,u)\delta_t, \sigma\sigma^T\delta_t)$, to form transition matrix P, shape (A, S, S), with some sampling technique. I calculate the probability for all the states with given mean as well as covariance, and I select a circle of region where the probability of states in that region are greater than some threshold. Therefore, a region of samples with their probabilities of this samples can be get, to normalize them I can form transition probability. As the stage cost function is deterministic, I do not have to discretize it, and stage cost matrix G, shape (A, S), can be directly generated for given the discretized space, g(x, u) δt.



Then, by using formulated MDP and policy iteration or value iteration, I can obtain the optimal control strategy defined on the grid.

Implementing value iteration

To implement this value iteration, I first initialize the $V_0 = 0$ vector and calculate the V_{k+1} iteratively until V_k converges to J*(i), or in other word, $|V_{k+1} - V_k|$ < threshold = (1- discount factor)/ discount factor. Following is the procedure without discount factor. Basically, in each epoch, I iteratively try all of the controls with its corresponding transition matrix and its stage cost G and try to find a policy in one epoch that minimize the accumulated cost.

- **Value Iteration** (VI): applies the DP recursion with an arbitrary initialization $V_0(i)$ for all $i \in \mathcal{X} \setminus \{0\}$:

$$V_{k+1}(i) = \min_{u \in \mathcal{U}(i)} \left[g(i,u) + \sum_{j=1}^{n} P_{ij}^{u} V_k(j) \right], \qquad \forall i \in \mathcal{X} \setminus \{0\}$$

- VI requires an infinite number of iterations for $V_k(i)$ to converge to $J^*(i)$
- In practice, define a threshold for $\|V_{k+1}(i) - V_k(i)\|$ for all $i \in \mathcal{X} \setminus \{0\}$

Implementing policy iteration

For the policy iteration, in each epoch there are two steps, the first one is policy evaluation to compute $J^{\pi}(i)$ by solving linear system equation with $J^{\pi}(i) = (I - discount * P)^{-1} g(i, ui)$. The second one is policy improvement to compute the stationary policy π' based on calculated $J^{\pi}(i)$ from evaluation part. Basically, this step is similar to value iteration part finding minimum value and corresponding policy. The following is the policy iteration without discount factor.

- **Policy Iteration** (PI): iterates the following two steps over policies π instead of values/cost-to-go:
  1. **Policy Evaluation**: Given a policy π, compute J^{π} by solving the linear system of equations:

$$J^{\pi}(i) = g(i, \pi(i)) + \sum_{j=1}^{n} P_{ij}^{\pi(i)} J^{\pi}(j), \qquad \forall i \in \mathcal{X} \setminus \{0\}$$

  2. **Policy Improvement**: Obtain a new stationary policy π':

$$\pi'(i) = \arg\min_{u \in \mathcal{U}(i)} \left[g(i, u) + \sum_{j=1}^{n} P_{ij}^{\pi(i)} J^{\pi}(j) \right], \qquad \forall i \in \mathcal{X} \setminus \{0\}$$

- Repeat the two steps above until $J^{\pi'}(i) = J^{\pi}(i)$ for all $i \in \mathcal{X} \setminus \{0\}$

Implement interpolation and try other constant combination: By using interpolation, I can extend the control input from discrete space to continuous space, so that make control smoother.

Results and Discussion

1) Sampling-based searching RRT*

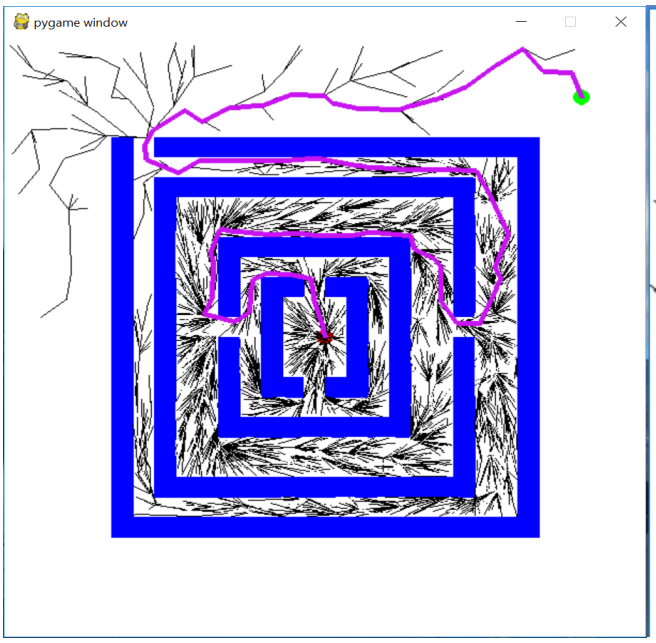


Figure1

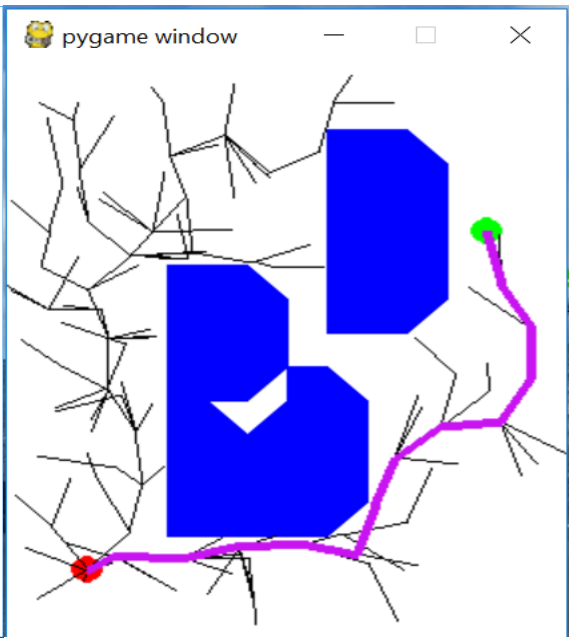


Figure2

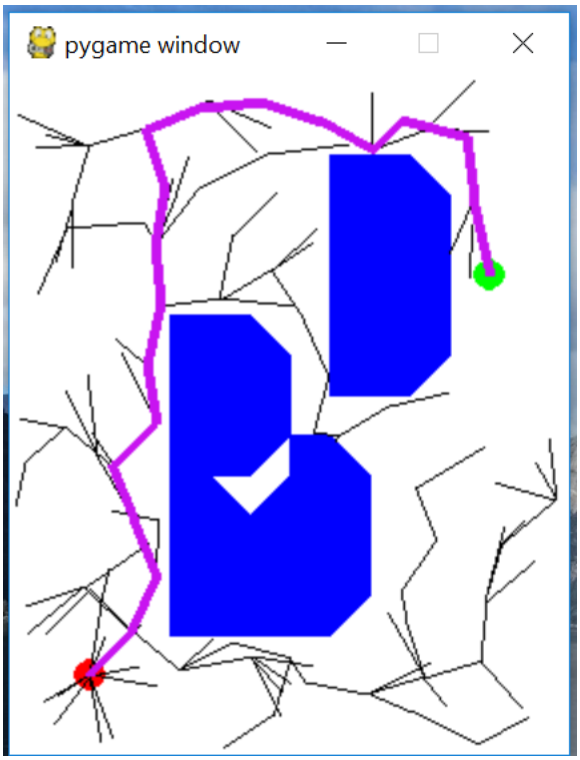


Figure1

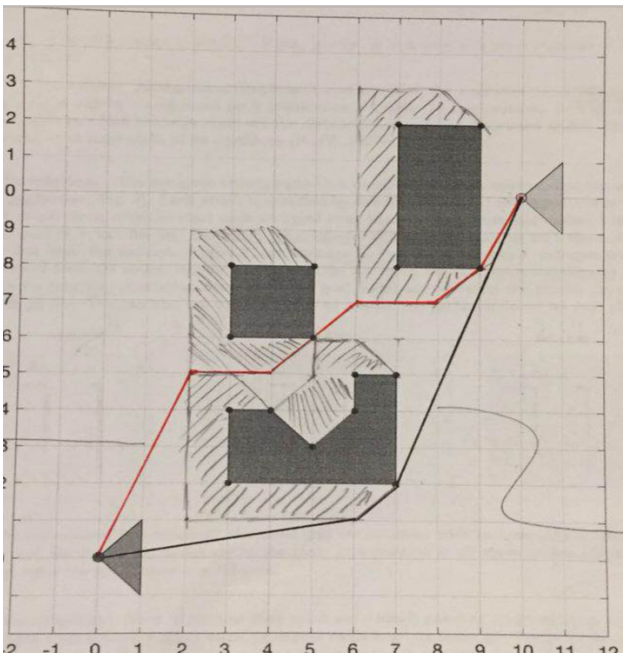


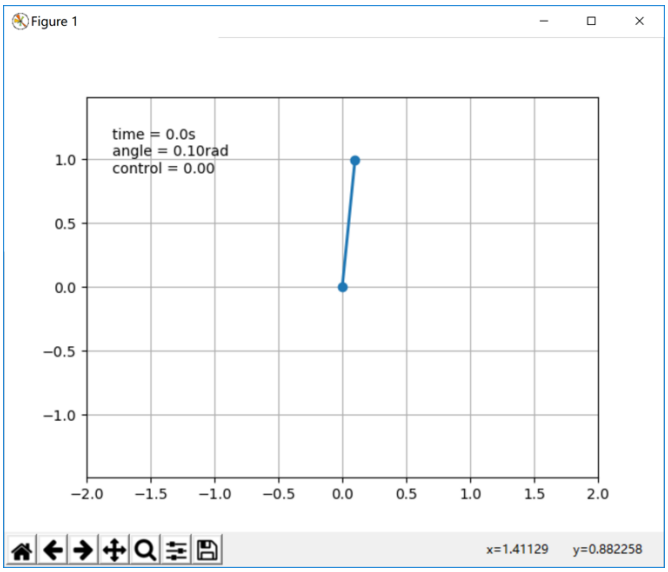
Figure4

For RRT*, I set searching radius for Xnear = 2, and epsilon = 1.5 for steering, and for input file maze and triangular, it can both find the suboptimal path, like show in figure1, figure2, and figure3. In general, there are three loops which are computation expensively for finding nearest, near, and updating edges, for the edges searching part in my code, I think the K-d tree search can be implemented to decrease the computation cost, but I did not do that. For the triangle input file, it can be realized that the sampling-based searching RRT* is not guaranteed to find shortest path; Instead, it can only guarantee to find suboptimal path. Therefore, sometimes the path is find like figure2 and sometimes find path like figure3, and very small chance to find the shortest path which we find use A * like figure4. In general, this is the trade-off between searching based algorithm like A* which guarantee to find optimal solution but running out of time in high dimension; however, sampling-based searching works well in high dimension case, and in large map. In general, for the maze map, it takes around 15000 iterations to find path, and for the triangle map, it takes around 200 iterations to find path.

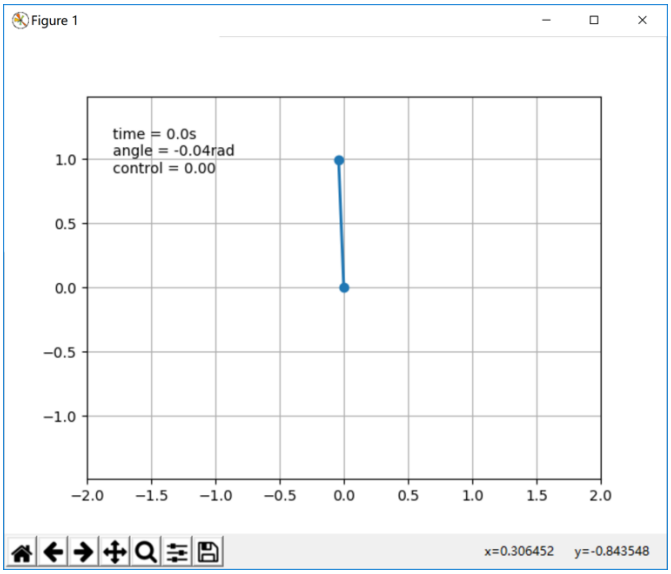
2) Balancing inverted pendulum

In general, with (a=1, b=0, σ= (2*2 identity matrix), k=1, r=0.01, γ=0.3) and (δt=0.1, n_1=80, n_2=20, n_u=50, θ_max = pi, v_max=8, u_max=8) defined, the inverted pendulum can be balanced vertically. With increasing the noise, it gets hard to balance the pendulum, and the number of value iteration and policy iteration to get converge also increase. Additionally, policy iteration always converges faster than value iteration, but with more complexity in each iteration than in value iteration. With above setting and discount factor = 0.9, value iteration takes around 286 iterations to converge, but the policy iteration only takes 13 iterations to converge.

To some extent, denser discretization is, the smooth control I can get; however, with denser discretization would increase the computation cost, which makes the generating probability transition and value as well as policy iteration slow. Somehow, the storage of np.array cannot resist this denser discretization.



Policy iteration



value iteration

The above two graphs are the results for policy iteration and value iteration, which can balance the pendulum vertically. In my case, the result of policy iteration balancing is a little oscillated than the result of value iteration. Also, by decreasing value of r from 1 to 0.01, the balancing becomes smoother. For large discount factor, it would make the converge slower than small discount factor, and sometimes makes the pendulum unbalanced.