3. Exercises:

3.1. Solving 8-Puzzle with Dijkstra's Algorithm:

3.1.1. The number of tiles configuration is $9! = 362,880$. Although this is an enormous number because we want to solve the single pair shortest path problem there are some redundant sates which not belong to the requested path, therefore, we can crate the state online, during Dijkstra iteration, that way we will create only the required state to solve our problem.

3.1.2. *Code is submitted.*

3.2. Solving 8-Puzzle with A*:

3.2.1. We define a Manhattan distance on a puzzle state by given examine the minimum number of movements to get from one state to another for example, if number "2" is on position (1,1) the Manhattan distance to move it to (2,3) is $|2-1|+|3-1|=3$ and sum for all numbers in the puzzle but "0" because if we arrange all 8 numbers 0 must be in its place.

3.2.2. *Code is submitted.*

3.2.3. Yes, **the heuristic is matter**, because it effects on the number of states the algorithm is going to create, the more it is close to real distance it is better. We change the heuristic to "Air distance heuristic", basically it is taking each coordinate delta power it by 2 sums for all coordinates and square root the sum. The heuristic is admissible because, its value is for sure less than Manhattan distance and the Manhattan distance is admissible because it is presenting the minimum changes that needed to be done from one state to another without blocking considerations.
For Manhattan distance A* evolves 222 states and for air distance: 327, therefore it is clearly that Manhattan distance is better, it is closer to real distance.

3.2.4. Initial state is:

8	7	6
1	0	5
2	3	4

Goal state is:

1	3	4
8	2	0
7	6	5

Actions need to be done:
d-> l-> u-> r-> r-> d-> l-> u-> u-> r-> d-> d-> l-> u-> u-> l-> d-> d-> r-> u-> u-> l-> d-> r-> r

Dijkstra solving time: 10.768 seconds and 149,326 states.
A* solving time: 0.186000 seconds and 2,058 states.

3.2.5. The parameter α effects the heuristic function in a way to get it closer to the real distance. For $\alpha = 0$ the heuristic function is basically the constant zero, therefore, the algorithm behaves like Dijkstra.
For $0 < \alpha \leq 1$ if the original heuristic is admissible therefore the weighted one is also admissible, but in that case the parameter has no contribution, in the contrary it makes the function less closer to real.
But for $\alpha > 1$ we can't claim any more that we hold with admissible heuristic. Notice that the heuristic might stays admissible for some $\alpha > 1$ values.
For $\alpha \to \infty$ the heuristic function value is infinity for all states and that makes it not admissible for sure.

3.2.6. For $\alpha = 0$, the heuristic is the constant zero, it doesn't give any extra information to A* therefore the algorithm behaves like Dijkstra, will find the shortest path.
For $0 \leq \alpha \leq 1$ if the original heuristic is admissible therefore the weighted one is also admissible, the algorithm will find the shortest path. Notice that if the original is admissible the weighting is making the algorithm to evolve more states, so it is no needed. But if the original is not admissible the weighting might help to make the heuristic admissible and therefore to find the shortest path.
As said before, $\alpha \to \infty$ the search algorithm assume that for any pair of states the distance between them is ∞ therefore it will find some path and probably not the optimal, and also crate large number of states.

3.3. Solving Cart-Pole with LQR:

3.3.1.

```
def get_A(cart_pole_env):
    '''
    create and returns the A matrix used in LQR. i.e. x_{t+1} = A * x_t + B * u_t
    :param cart_pole_env: to extract all the relevant constants
    :return: the A matrix used in LQR. i.e. x_{t+1} = A * x_t + B * u_t
    '''
    g = cart_pole_env.gravity
    pole_mass = cart_pole_env.masspole
    cart_mass = cart_pole_env.masscart
    pole_length = cart_pole_env.length
    dt = cart_pole_env.tau
    moment_of_inertia = pole_mass * (pole_length ** 2)

    A_bar = np.array([[0, 1, 0, 0],
                      [0, 0, ((pole_mass * g) / cart_mass), 0],
                      [0, 0, 0, 1],
                      [0, 0, (g / pole_length) * (1 + (pole_mass / cart_mass)), 0]])
    A = np.add(moment_of_inertia, np.multiply(dt, A_bar))

    return A


def get_B(cart_pole_env):
    '''
    create and returns the B matrix used in LQR. i.e. x_{t+1} = A * x_t + B * u_t
    :param cart_pole_env: to extract all the relevant constants
    :return: the B matrix used in LQR. i.e. x_{t+1} = A * x_t + B * u_t
    '''
    g = cart_pole_env.gravity
    pole_mass = cart_pole_env.masspole
    cart_mass = cart_pole_env.masscart
    pole_length = cart_pole_env.length
    dt = cart_pole_env.tau

    B_bar = np.array([[0],
                      [1 / cart_mass],
                      [0],
                      [1 / (cart_mass * pole_length)]])

    B = np.multiply(dt, B_bar)

    return B
```

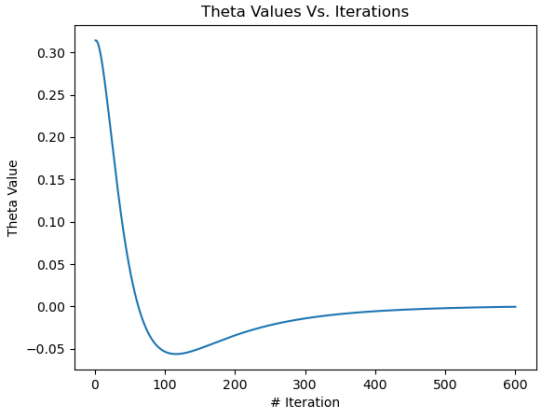
3.3.2. We decided the Q matrix should be $\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ and R is 1, therefore it creates a cost function of $\dot{x}_t^{\,2} + \theta^2 + \dot{\theta}_t^{\,2} + u_t^{\,2}$. The motivation behind it is:

- Makes it more expansive to be in large angles, means make the controller to avoid large θ values.
- We don't take price of the x location because it doesn't important in order to stabilize the mass on the pole.
- Because it is very delicate to stabilize to mass over the pole, we would like to make the changes over time, aka velocity and angular velocity, very costly in that way the controller should avoid moving the cart very fast.

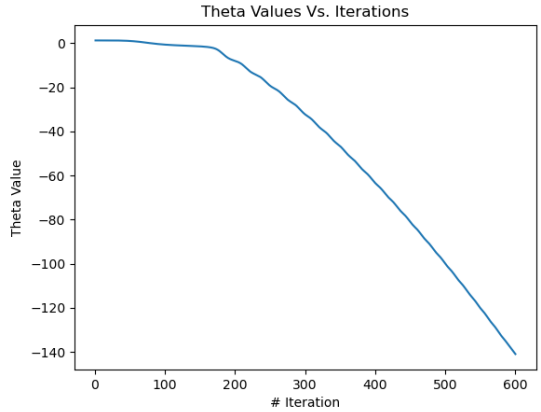
3.3.3. We found the for initial θ of $\theta_{unstable} = 0.4 \cdot \pi = 1.25664$ the controller cannot stabilize the mass and it loses control, we think this happens

because the linearization assumptions do not apply for large theta's values.

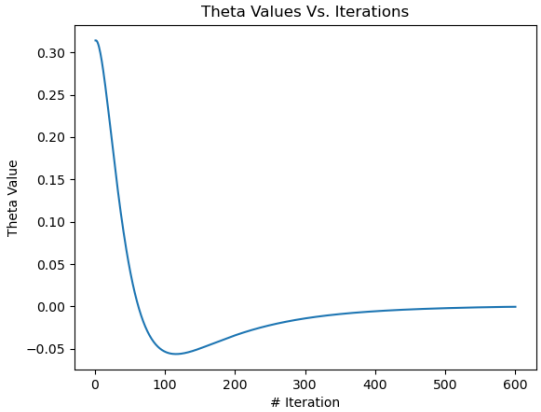
Plot for $\theta = 0.1 \cdot \pi$:



Plot for $\theta_{unstable} = 0.4 \cdot \pi$:



Plot for $0.5 \cdot \theta_{unstable} = 0.2 \cdot \pi$:

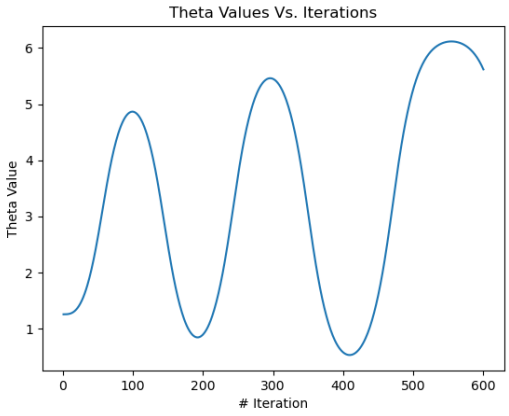


As we can see for $\theta_{unstable} = 0.4 \cdot \pi$ the value of theta is not going to zero but losing control because the controller is trying to stabilize it by moving the cart fast but when the cart moves fast it becomes more difficult to stabilize the mass. In the other graphs we can see that the controller decisions lead to stabilization of the mass.

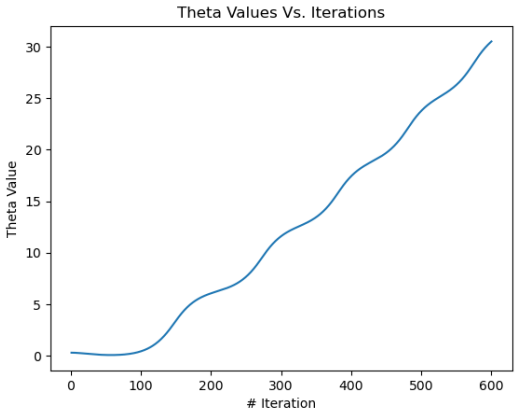
3.3.4. Plots for feedforward control:

Plot for $\theta = 0.1 \cdot \pi$:

Plot for $\theta_{unstable} = 0.4 \cdot \pi$:



Plot for $0.5 \cdot \theta_{unstable} = 0.2 \cdot \pi$:

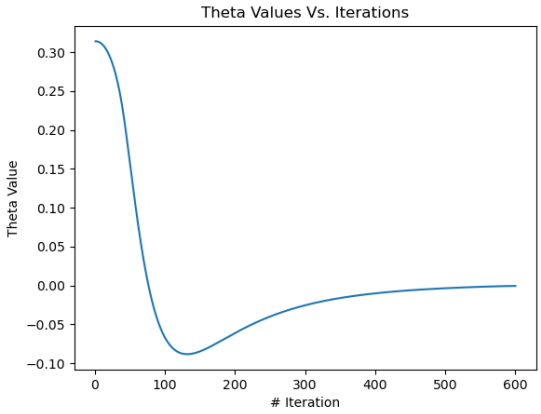


As we can see clearly the LQR feedback control is better than feedforward control, because in LQR we linearize the dynamic of the system to $x_{t+1} = Ax_t + Bu_t$ around $\theta = 0$, but the dynamic is different in the simulator which calculate the change in state like the real physics of the system. Therefore, for a given action there is a delta (aka noise) between the next state the LQR calculates and the simulator calculates which means the paths

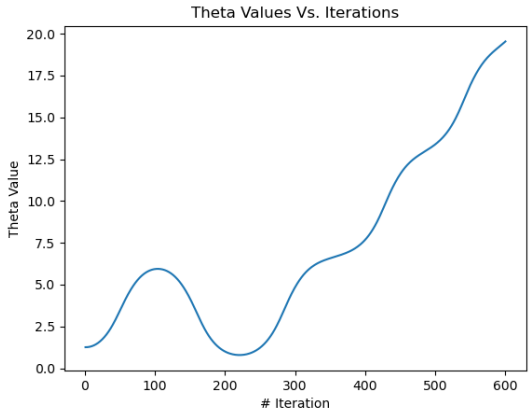
of states is not the same.

3.3.5. Force limitation:

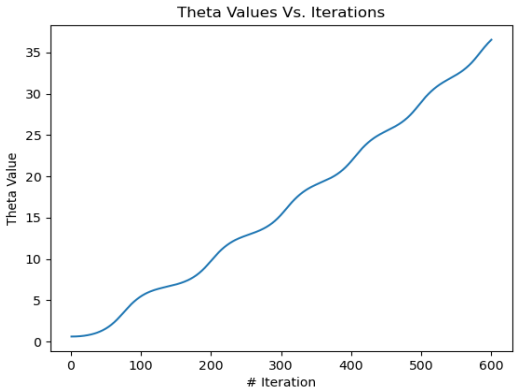
Plot for $\theta = 0.1 \cdot \pi$:



Plot for $\theta_{unstable} = 0.4 \cdot \pi$:



Plot for $0.5 \cdot \theta_{unstable} = 0.2 \cdot \pi$:



In order to stabilize the mass with initial $\theta = 0.1 \cdot \pi$ we didn't need to change out Q or R matrices. Unlike before the controller doesn't success to stabilize the mass with initial $0.5 \cdot \theta_{unstable} = 0.2 \cdot \pi$, the explanation for that is that it might need more force which is now limited respected to cost function. But when we tried to increase R in

respect to Q, it helped to increase $\theta_{unstable}$ it makes the control more cautious with the control signals (smaller control signals fit with new limitations).