1. **Exercises:**
   1. **Solving 8-Puzzle with Dijkstra's Algorithm:**
      1. The number of tiles configuration is . 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.
      2. *Code is submitted.*
   2. **Solving 8-Puzzle with A\*:**
      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 and sum for all numbers in the puzzle but "0" because if we arrange all 8 numbers 0 must be in its place.
      2. *Code is submitted.*
      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.
      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.

* + 1. The parameter  effects the heuristic function in a way to get it closer to the real distance. For  the heuristic function is basically the constant zero, therefore, the algorithm behaves like Dijkstra.  
       For 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 we can't claim any more that we hold with admissible heuristic. Notice that the heuristic might stays admissible for somevalues.  
       For the heuristic function value is infinity for all states and that makes it not admissible for sure.
    2. For , the heuristic is the constant zero, it doesn’t give any extra information to A\* therefore the algorithm behaves lkike Dijkstra, will find the shortest path.  
       For 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, 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.
  1. **Solving Cart-Pole with LQR:**

**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

* + 1. We decided the Q matrix should be and R is 1, therefore it creates a cost function of . The motivation behind it is:
* Makes it more expansive to be in large angles, means make the controller to avoid large  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 it changes over time very costly that way the controller should avoid moving the cart.