**046203 – Computer Exercise 1**

**Deterministic Planning Algorithms**

Dar Arava - 205874951

Yair Nahum - 034462796

**3.1 Solving 8-Puzzle with Dijkstra’s Algorithm**

1. For the given puzzle there are 9 types of tiles, meaning there are possible configurations of the tiles. Since the problem is a “Slide” puzzle, fore a given configuration of the board, half of the configurations are impossible to reach (since we are limited to configurations reached by sliding the tiles). Still, the large number of states (configuration of tiles) is impractical to build the graph in advance.

In order to find the optimal plan between a source state and a target state, we can run Dijkstra algorithm without initializing the whole graph in advance. This can be obtained by calculation the possible actions in each time step and building the states reachable by these actions (building the neighbors of the node reached). Since Dijkstra algorithm anyway runs on parts of the graph each time (each time step extends by calculating distances of only the nodes bordering to ), its functionality will not be harmed by not initializing the whole graph in advance.

* 1. **Solving 8-Puzzle with A\***

1. Let us define the Manhattan distance between two states as follows-

For and

Or, alternately- set vectors and :

And then-

By this definition of the Manhattan distance we sum differences in both row indexes and column indexes of each tile. The differences are between the index of the wanted location to the index of the current location.

Distances in the 8-puzzle problem are equivalent to the amount of actions taken between one step to another. For estimating the cost of reaching a goal state, we shall calculate both the lower bound of distance from a current state to the goal state (using the admissible heuristic) and the minimum distance from the initial state to the current state. The sum of both will be the evaluation distance which our algorithm will prioritize by.

The admissible heuristic will be-

And the new distance is defined –

is admissible-

Fixing a tile location depends on the location of the ‘0’ tile- if ‘0’ is not next to the wrong tile we need to bring it next to it in order to move the wrong tile. Also, in the process of fixing one tile other tiles might move further from their correct location. Meaning that the number of actions needed is at least . For the minimal case where , each tile can be moved to its correct place without “harming” other tiles. Other cases will require more actions.

1. The heuristic function does matter. Changing the MD function (as detailed above) to a heuristic function which returns the number of incorrect tiles (implementation in Appendix A) did not change the result – we still got the same plan of optimal actions from an initial state to a goal state. However, the time solving the problem was longer:

For the example given-

With MD heuristic function - 

With num of incorrect tiles heuristic function -

The advantage of the MD heuristic lies in the fact that it distinguishes between states of same wrong tiles and prioritized the state where the wrong indexes are “quicker” to fix. The wrong tiles heuristic will go over states with smaller numbers of wrong tiles, but since the number of actions needed in a correction of a wrong tile depends on the index difference, these states eventually aren’t included in the optimal path.

For the example given (not including initial and goal states)-

With MD heuristic function – **351** states explored in solution process.

With num of incorrect tiles heuristic – **3122** states explored in solution process.

Note – “states explored” refers to the number of states created in the graph. A certain state can be visited more than once, and therefore-

However, the memory complexity matches .

From all stated above, the MD is a tighter lower bound of the distance function. The number of incorrect tiles heuristic is a lower bound, and there for an admissible heuristic, however, it is not as tight as the MD function.

* The Manhattan-Distance heuristic has better time and memory complexity. The MD function is a tighter lower bound of

1. The hard puzzle chosen-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Initial state:   |  |  |  | | --- | --- | --- | | 0 | 1 | 2 | | 3 | 4 | 5 | | 6 | 7 | 8 | | Goal state:   |  |  |  | | --- | --- | --- | | 8 | 7 | 6 | | 5 | 4 | 3 | | 2 | 1 | 0 | |

In both algorithms the shortest plan to solution included 28 actions-

(‘r’,’r’,’d’,’d’,’l’,’l’,’u’,’u’,’r’,’r’,’d’,’d’,’l’,’l’,’u’,’u’,’r’,’r,’d’,’d’,’l’,’l’,’u’,’u’,’r’,’r’,’d’,’d’)

In Dijkstra algorithm- **181,217** states were explored in solution process, and the run time was-

In A\* algorithm- **1,085** states were explored in solution process, and the run time was- 

* Time complexity was improved (more than 100 times faster) and memory complexity was improved (~180 times less states in graph) using A\* over Dijkstra.

1. The parameter effects the A\* algorithm by changing the impact of each of the following parameters on the total distance the algorithm sorts and searches by:

* The distance between initial state to goal state ( – the smaller is, the stronger ’s part is in the total distance function.
* The heuristic function – the larger is, the stronger ’s part is in the total distance function.

For the over-all distance function for A\* is same as the Dijkstra distance function- simply the min number of actions from initial state to goal state.

For we get the same heuristic function for A\*, meaning the distance from the initial state to current state( has similar impact(in general size order) on the total distance as the heuristic function(.

For we expect the search be more directional in the graph toward the goal state. When reaching the goal state, the solution won’t be necessarily the shortest path possible in terms of least number of actions, since the heuristic function weighs the distances differently. We expect the search to reach the goal state faster since it is more oriented to the goal direction (more ‘depth’ and less ‘breadth’).

Also, for the heuristic function is **not admissible:**

For the distance from the initial state to current state ( has zero effect of the total distance since the heuristic function is much larger. Meaning that the A\* algorithm will search for shortest path solely by the heuristic function.

Note -We refer to as a very large number and not as .

For, all distances will remain and there won’t be any updating of shorter distances (missing the whole concept of the algorithm).

For the given puzzle –

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Plan of solution** | **Num of states explored in solution process** | **Run-time** |
|  | 19 actions - d,r,d,l,u,r,r,d,l,u,r,u,l,d,r,d,l,u,u | 43,021 | 0:00:11.292374 |
|  | 19 actions - d,r,d,l,u,r,r,d,l,u,r,u,l,d,r,d,l,u,u | 351 | 0:00:00.086736 |
| **(We defined )** | 21 actions-  d,r,d,l,u,r,d,r,u,l,d,r,u,u,l,d,r,d,l,u,u | 92 | 0:00:00.016953 |

We received expected results. For we got the correct solution (since the search is the same as Dijkstra) however it was longer and spanned on more states (more memory) than when . For we got better performance and best solution. For the solution was not the shortest path in terms of least number of actions (since the algorithm sorted by the heuristic function only). The run time was shorter for and the graph spanned on less states. So, if the optimality of the solution (number of actions in solution) is less important then reducing time and memory, is the better approach.

**3.3 Solving Cart-Pole with LQR**

1. As shown in Tutorial 4, the motion equations of an inverted pendulum on a cart-

(1)

(2)

After linearization around : (

After setting the control- and the state- :

We get-

1. The parameters chosen for and :

Since the value function for this LQR problem (as described in class) is-

By defining and Q as above we get –

Meaning that minimizing the value function is equivalent to minimizing mainly the angle-

and the angle speed in which it stabilizes- (in order to prevent overshoot and passing of the 0 angle too fast). The cost on and its derivative (the speed) is less important for us. The action (force in our case) is also costly in magnitude as .

The advantages of minimizing the three parameters stated above

are:

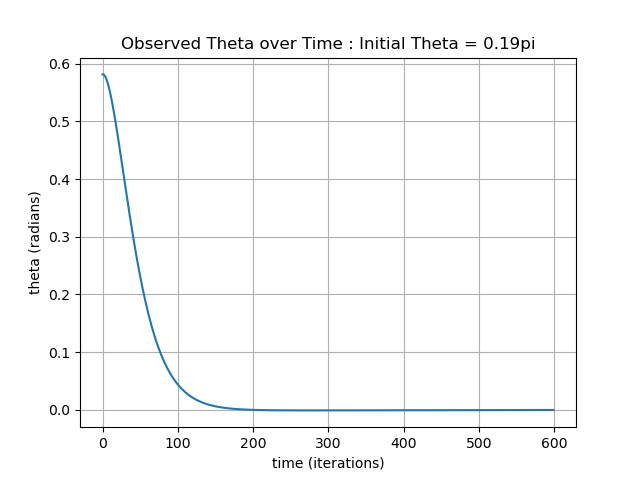
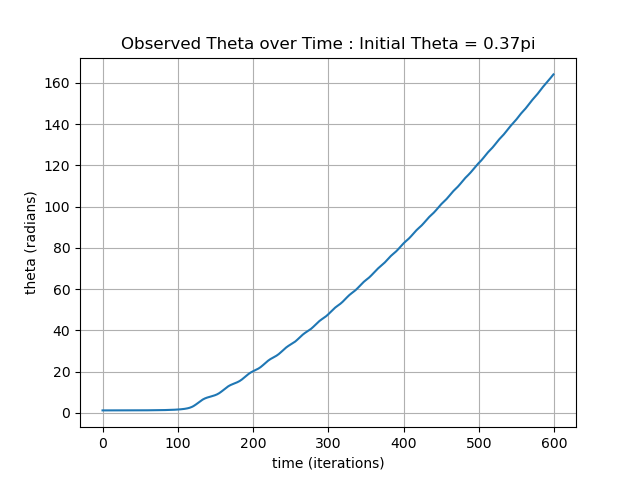
* For getting the pole to the stable angle as fast as we can with a high cost on a

value we want .

* For avoiding overshoot on stabilizing due to too fast , we try to minimize as well.
* For limiting the amount of force used to stable the cart we want

1. The initial angle for which our LQR solution does not stabilize the system –

The results of the observed angle over time for are displayed next.



A close up of a piece of paper

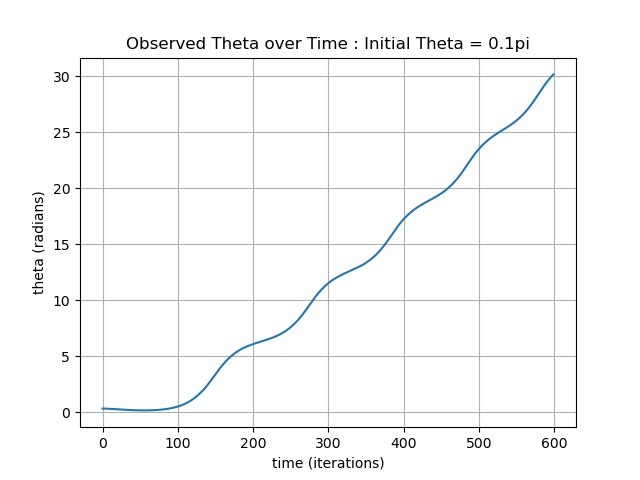
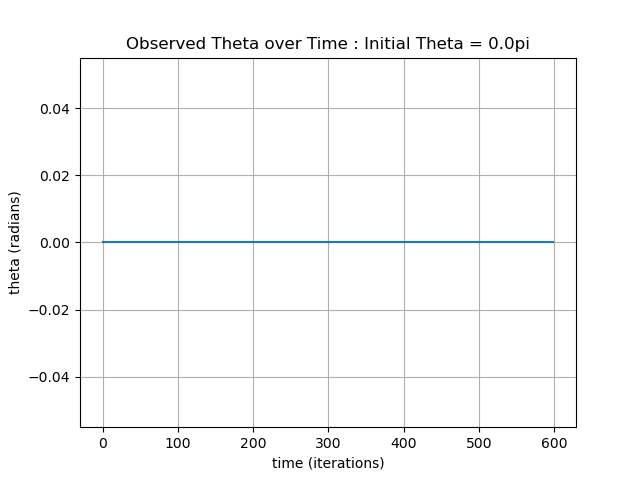
Description automatically generated

We notice that for both and the angle converges to . Both initial angles are within the bound of states for which the LQR solution can stabilize the system. However, for we see that the angle does not converge. Our solution took in consideration many assumptions, one was (as we recall from the linearization of the problem). For this assumption is not accurate therefore the LQR does not stabilize the cart.

1. After repeating the last section only now using the LQR predicted actions (instead of using the calculated actions based on current state), we got that the cart stabilizes for only.

Clearly the method based on observed states is better than the LQR predicting method. Since the LQR solution linearizes the state dynamic, it is a good approximation only close to the state on which the linearization was applied. By calculating the actions based on the actual states, we get greater accuracy.

Results of the only stable initial angle and other requested initial angles are displayed bellow.



A close up of a piece of paper

Description automatically generatedFor all the angle did not converge, so we chose

The implementation of the feedforward control law predicted by LQR is in Appendix B.

1. After limiting the force available in the problem, we still got stability for . Since we initially defied the cost function to take in account the amount of force used(we set ), limiting the force did not have a critical effect for .

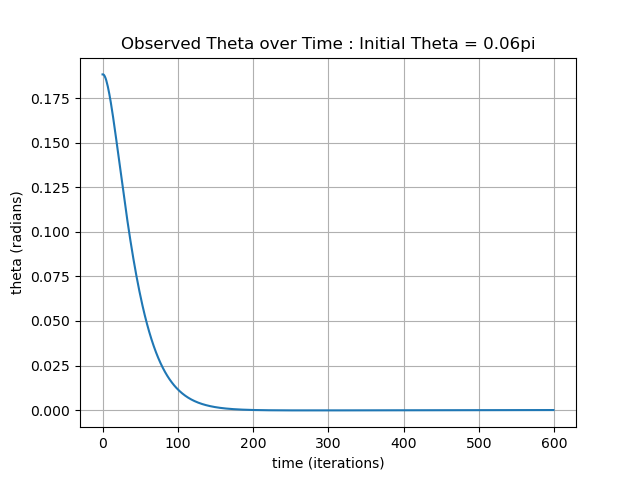
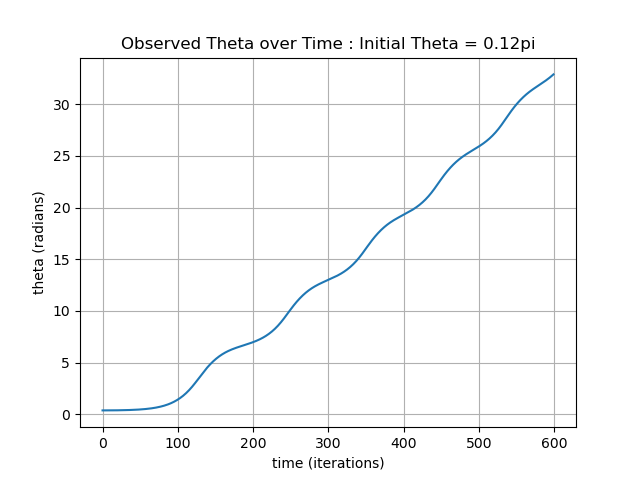
However, we did get worse results for the maximum initial angle that is still stable. For limited force - . For the pole did not stabilize.

In order to improve , we tried to change the cost parameters --

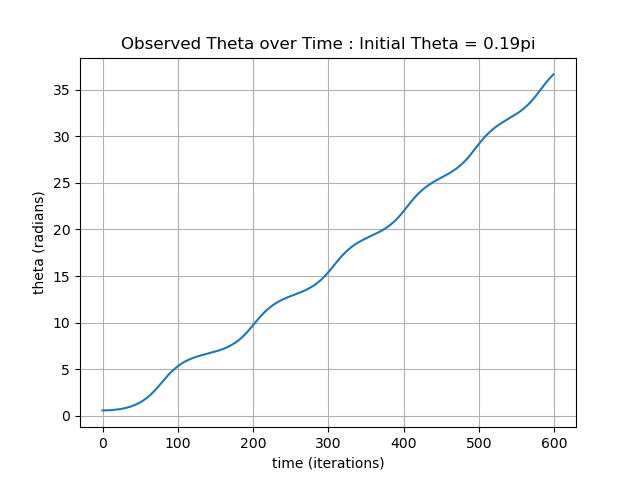
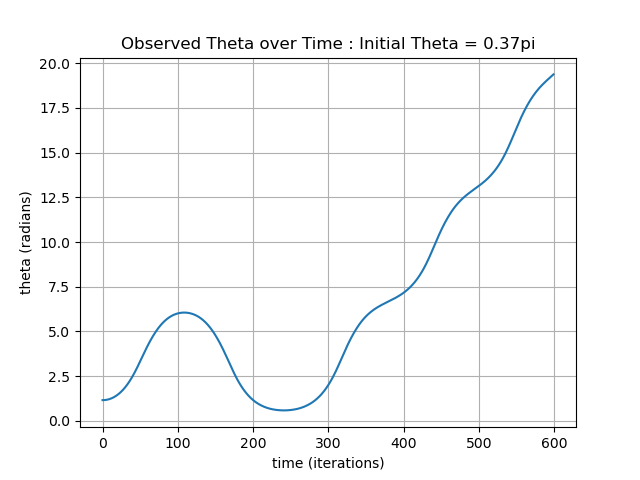
So that the force used will have more impact on the cost function ().

The initial cost function we chose (see) got about same results as different parameters we tried ( ).

The results of the observed angle over time for are displayed below. didn’t change vs previous graph.



The results of the observed angle over time for same angles from 3.3.3 are displayed below.



**3.3 Bonus– Cart-Pole Swing-Up with iLQR**

1. For given problem- initial matrixes for the state’s dynamic are same as in .

The initial state is (starting from the bottom position at zero velocity)-

Iterative LQR (iLQR) solution for stabilizing the pole at an upright position:

1. Set reference controller - (for example: )
2. Calculate states for controller -

(forward simulate- is original nonlinear system).

1. Linearize :
2. Calculate –
3. Get new matrixes (from the gradient on f) for variables -:
4. Quadratize :
5. Calculate-

1. Get new matrixes (from the gradient and hessian on C as described in the general case in the lecture). The in our case is 0 as we calculate the delta (pass in the above equation the components to one side).
2. Solve LQR for and :
3. Init
4. Calculate recursively.
5. Calculate and –
6. Obtain new : . A linear/binary search can be performed on for obtaining an optimal such that the cost using this trajectory is still less than the previous one. If the cost calculated using the new trajectory is bigger, we reduce the alpha, if its still less we can try to increase it for a better linearization point.
7. Return to step #2 until we can’t reduce the cost between trajectories using some threshold on cost improvement.

**Appendix A-**

Implementation of a heuristic function which returns the number of incorrect tiles –

**def** get\_wrongTiles\_distance**(**self**,** other**):**

total\_distance **=** 0

**for** i **in** **range(**3**):**

**for** j **in** **range(**3**):**

**if** self**.**\_array**[**i**][**j**]** **!=** other**.**\_array**[**i**][**j**]:**

total\_distance **+=** 1

**return** total\_distance

**Appendix B-**

Implementation of feedforward control law predicted by LQR (edited rows marked) –

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

env **=** CartPoleContEnv**(**initial\_theta**=**np**.**pi **\***0.1**)**

# print the matrices used in LQR

**print(**'A: {}'**.format(**get\_A**(**env**)))**

**print(**'B: {}'**.format(**get\_B**(**env**)))**

# start a new episode

actual\_state **=** env**.**reset**()**

env**.**render**()**

# use LQR to plan controls

xs**,** us**,** Ks **=** find\_lqr\_control\_input**(**env**)**

# run the episode until termination, and print the difference between planned and actual

is\_done **=** **False**

iteration **=** 0

is\_stable\_all **=** **[]**

ObservedTheta **=** **[]** # list for plotting Observed Theta over time

**while** **not** is\_done**:**

# print the differences between planning and execution time

predicted\_theta **=** xs**[**iteration**].**item**(**2**)**

actual\_theta **=** actual\_state**[**2**]**

ObservedTheta**.**append**(**actual\_theta**)**

predicted\_action **=** us**[**iteration**].**item**(**0**)**

actual\_action **=** **(**Ks**[**iteration**]** **\*** np**.**expand\_dims**(**actual\_state**,** 1**)).**item**(**0**)**

print\_diff**(**iteration**,** predicted\_theta**,** actual\_theta**,** predicted\_action**,** actual\_action**)**

# apply action according LQR predictment

# make action in range

predicted\_action **=** **max(**env**.**action\_space**.**low**.**item**(**0**),** **min(**env**.**action\_space**.**high**.**item**(**0**),** predicted\_action**))**

predicted\_action **=** np**.**array**([**predicted\_action**])**

actual\_state**,** reward**,** is\_done**,** \_ **=** env**.**step**(**predicted\_action**)**

is\_stable **=** reward **==** 1.0

is\_stable\_all**.**append**(**is\_stable**)**

env**.**render**()**

iteration **+=** 1

env**.**close**()**

# we assume a valid episode is an episode where the agent managed to stabilize the pole for the last 100 time-steps

valid\_episode **=** np**.all(**is\_stable\_all**[-**100**:])**

# print if LQR succeeded

**print(**'valid episode: {}'**.format(**valid\_episode**))**