046203 – Computer Exercise 1
Deterministic Planning Algorithms

April 12 2020

1 General Instructions

1. Exercise is due on May 7th.
2. The code should be written in Python 2.7.
3. We provide templates for the code. The templates include code for the domain and evaluation of the results, while you need to fill in the algorithm within the template. You are required to use the templates.
4. You are required to submit both code and answers to the questions. Submit a single ZIP file that includes the code and a PDF for the answers and figures.
5. Code will be checked by running your code using a testing script on various test cases.
6. Instructions for setting up a Python development environment are provided in Section 2. We encourage following these instructions to make sure your results match the testing script.

2 Initial setup

In this section we are going to setup your development environment. We start by installing the programming language used in this course - Python. Next we install the (recommended) development environment (IDE) - Pycharm. Finally, we install required packages used in the exercises.

1. **Install Python:** If you have python **version 2.7** installed (verify by running command `python` in terminal or command line and observing the is output starts with `Python 2.7` or greater) you can skip to the next item. Otherwise, please follow the instructions in this link:

   `https://www.python.org/downloads/`

   that match your OS for installing **python 2.7**.

2. **Install PyCharm (recommended)**: PyCharm is a free IDE for python that allows for convenient debugging. If you prefer working with a different IDE you are welcome to do so, but the examples provided here will assume PyCharm. Download and install PyCharm Community Edition (free) from the following url

   `https://www.jetbrains.com/pycharm/download/`.

3. **Configure interpreter for project:** if you have multiple python versions (for example both python 2.7 and python 3.7) you need to configure PyCharm (or your selected IDE) which version to use (it is recommended to have PyCharm create a virtual environment specific to the project, but it is up to you). For PyCharm, please see:

   `https://www.jetbrains.com/help/pycharm-edu/configuring-local-python-interpreters.html`

4. **Install required packages:** Please install the following packages: `numpy, gym`, and `matplotlib`. NumPy is the fundamental package for scientific computing with Python; OpenAI's gym is a commonly used toolkit and simulator for developing and comparing reinforcement learning algorithms; and Matplotlib is a 2D plotting library that is easy to use. In PyCharm you can follow the instructions for installing packages here:

   `https://www.jetbrains.com/help/pycharm/installing-uninstalling-and-upgrading-packages.html`

   For non-PyCharm users, you can use the `pip` command in terminal or command line, for example: `pip install numpy`.

5. Clone the code from `https://github.com/tomjur/rl_hw_1_public.git` by executing the command `git clone https://github.com/tomjur/rl_hw_1_public.git` in the local directory you want to download the files to (requires git to be installed on your machine). Alternatively, you can use the "clone or download" button on GitHub itself.

6. **Python and Numpy tutorials:** If you are new to Python, we recommend going over some tutorials before starting the exercise. The following is a short tutorial on Python and the NumPy package.

   `http://cs231n.github.io/python-numpy-tutorial/`.

   Other recommended python tutorials are:

   `https://www.learnpython.org/`

   `https://www.python.org/about/gettingstarted/`.

3 Exercises

3.1 Solving 8-Puzzle with Dijkstra's Algorithm:

We consider the popular sliding tile puzzles (e.g., the 15-Puzzle in Figure 1).

Figure 1: Slide Puzzle with 15 tiles

In the following two questions you are going to plan solutions for the 8-puzzle. In this puzzle you have 8 numbered tiles (1-8) and one blank space (denoted by 0). The state of the system encodes the positions of all the tiles in the game. For instance, in the state below the blank is in the top left position and the rest of the tiles are ordered by number.

$$\begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{matrix}$$

The actions are 'up', 'down', 'left' and 'right', indicating the direction of the tile that is going to switch places with the blank tile (you can also think of that as the direction the blank is going to move towards). For instance, starting from the previous example, the available actions are 'down' and 'right', if we apply the 'down' action we obtain:

$$\begin{matrix} 3 & 1 & 2 \\ 0 & 4 & 5 \\ 6 & 7 & 8 \end{matrix}$$

The goal of this puzzle is to plan the shortest sequence of actions starting from a given starting state and ending in a given goal state. In this exercise we will solve the 8-Puzzle using Dijkstra's algorithm, and in the next exercise we will use A^*.

This exercise contains the following files:

- `state.py` and `puzzle.py` contain the classes that describe the state of the world (nodes in the graph) and the puzzle itself. Before starting to code, it could be helpful to go over these files.
- `dijkstra.py`, `a_star.py` and `planning_utils.py` are files with missing functionality that you should implement (see detailed instructions below).

In this question we are going to implement the single pair version of Dijkstra's algorithm to find the optimal plan between a source state and a target state.

1. (5 points) Consider the size of the graph - how many possible configurations of the tiles are there?[1] Building the graph in advance is clearly impractical. Explain how to run Dijkstra without building or initializing the whole graph in advance.

2. (15 points) We separate shortest path planning to two stages. In the first stage we map nodes in the graph to the previous node on the shortest path. In the second part we take the above mapping and transform it to a path starting at the initial state and ending in the goal state. The first part is achieved by Dijkstra or A^*, while the second part is easier; start from the goal state, find the previous state according to the results of the planning algorithm and repeat until initial state was reached. Implement the `dijkstra` method in `dijkstra.py`, and the traversal method in `planning_utils.py` method `traverse`. The `dijkstra` method should return a mapping between all the visited nodes in the graph to the node's previous node, and the `traverse` method should take this mapping and output a list of state-actions that describe the optimal path (see code for examples).

   - To avoid explicit iteration over the nodes in the graph, **do not create the graph in advance**. Instead when a specific node (state) is reached, expand all its neighbours (by going over all valid actions).
   - In the lecture you saw that Dijkstra uses a Priority Queue; this is a data structure that contains pairs of priority and item. In theory, if n is the number of nodes, an item should be inserted in $O(\log(n))$ time, the priority of an item should also be updated in $O(\log(n))$, and the minimal priority item should be popped with $O(1)$ complexity.

     You are not required to implement a priority queue. Instead, use the `heapq` package for priority queue. Because this implementation does not support the priority update method, instead of an update, insert a new duplicate item with the new priority and make sure to ignore repeating (or "completed") items when popping from the queue. Please review the helper script in `queue_ignore_example.py` that provides an example how to ignore items.
   - The code will be tested with an auto-tester. The tester has several test cases (puzzles) and it calls your implementation of the `dijkstra` and `traverse` methods. The `main` function provided in `dijkstra.py` presents an example for one such test case - so make sure this runs correctly without any errors or exceptions (of course you will not be tested for this specific puzzle).

3.2 Solving 8-Puzzle with A*:

In this exercise we are going to implement A^* for the same 8-puzzle problem and compare it with Dijkstra.

1. (5 points) For two vector v, w of dimension D, the Manhattan distance is defined as:

$$MD(v, w) = \sum_{i=1}^{D} |v_i - w_i|$$

   Using the Manhattan distance, suggest an admissible heuristic for the 8-Puzzle. Explain your answer.

2. (10 points) We'll now implement the A^* algorithm - complete method `a_star` in `a_star.py`. This will be graded similarly to the previous question (about Dijkstra).

3. **Does the heuristic function matter?** (5 points) Change the heuristic function such that given a state it returns the number of incorrect tiles. Is this an admissible heuristic? How many states does

[1] Actually, we only need to consider the number of tiles reachable by sliding the tiles, which turns out to be half of that `https://en.wikipedia.org/wiki/15_puzzle`.

it expand in your implementation compared to the Manhattan-distance heuristic? Which heuristic is better? Explain.

Note: do not submit the code of this question, please provide textual answers only and keep the submitted code compatible with the the Manhattan-distance heuristic.

4. **Comparison between A^* and Dijkstra:** (5 points) Design a hard puzzle such that the optimal solution takes at least 25 actions to complete, and solve it using both Dijkstra and A^*. Measure the running times and state visitations it takes each algorithm to reach the goal.

5. **Heuristic function analysis:** In this question we are going to investigate the dependence of A* on the heuristic function $h(s)$.

   We define a new heuristic function parameterized by $\alpha \in [0, \infty)$: $h_\alpha(s) = \alpha \cdot h(s)$.

   - (5 points) Describe how the parameter α affects the heuristic function. In particular, relate the following limiting cases to graph search algorithms: $\alpha = 0, 1$ and when α goes to infinity. What results do we expect for $\alpha > 1$?
   - (5 points) For $\alpha = 0$, 1 and ∞, compare the length of the planned paths and the planning time and explored nodes. Explain your results.

3.3 Solving Cart-Pole with LQR:

In this exercise you will implement the LQR control algorithm for the cart-pole problem discussed in Class Tutorial 4 (Figure 2). We will program a controller that maintains the pole in a stable upright position.

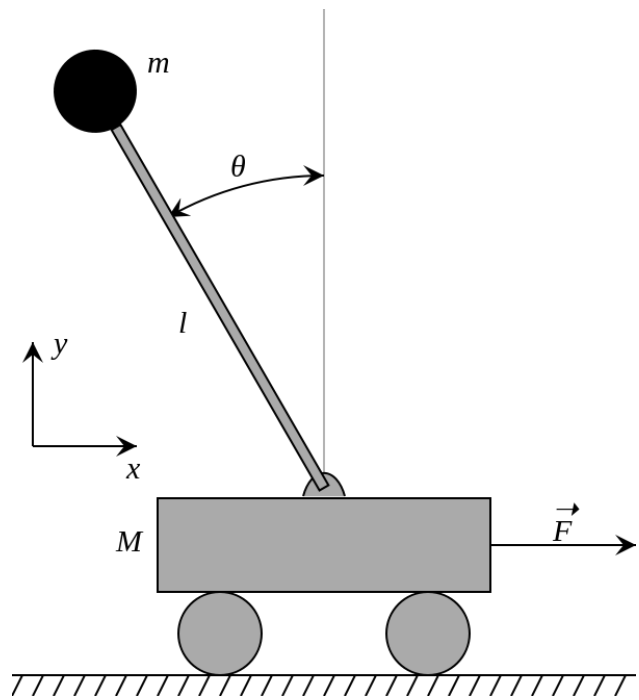

Figure 2: Cart pole system

This exercise contains the following files:

- `cartpole_cont.py` is the simulator of the cart-pole scenario adapted from the OpenAI gym implementation. Note that this file also contains all the constants needed for LQR as you will see later on. Please run main function and observe what happens without control and with random control signals.

- `lqr.py` contains the missing functionality that you should implement (see detailed instructions below).

1. (10 points) As you recall, we first need to linearize the system. Linearize and discretize the system around $\theta = 0$ to obtain the A and B matrices shown in class. Fill-in the code for the methods `get_A` (10 points) and `get_B`. All the required constants appear in the code skeleton.

2. (15 points) Implement the LQR method in `find_lqr_control_input` in `lqr.py`. The time horizon is given by `cart_pole_env.planning_steps`. Find cost function parameters that result in stabilizing the pole from initial position in the range $\theta \in [-\frac{\pi}{10}, \frac{\pi}{10}]$. Explain your choice of parameters. Make sure the `main` method in `lqr.py` shows that the pole is balanced and it prints `valid episode: true`.

3. (5 points) Under your above selection of cost function parameters, find the value of θ for which LQR no longer stabilizes the system (your solution should be a numeric approximation up to 2 digits to the right of the decimal point, no need for analytical solution here). Let this angle be $\theta_{unstable}$. Using the `matplotlib` package, make a plot of the value of θ over time. Plot on the same graph the results for the following initial values of θ: $\frac{\pi}{10}$, $\theta_{unstable}$ and $0.5 \cdot \theta_{unstable}$ (the other elements of the initial state, such as $\dot{\theta}$ are already set to zero by the simulator). Explain the differences.

   Note about plotting: use the script: `https://matplotlib.org/gallery/lines_bars_and_markers/simple_plot.html` as a baseline for generating your plot. For the X-axis, instead of `t` use time units, and for the Y-axis, instead of `s` use the values of the observed θ.

4. (5 points) Repeat the instructions from (3), but instead of using LQR feedback control law (i.e., computing the action based on the observed state), use the feedforward control law predicted by LQR (i.e., execute the `us` list that LQR outputs). Produce the same plots. Which control method is better? Explain.

5. **Force limitation:** (10 points) Limit the allowed force F in the simulation from 100 to 4.0 by changing `high=np.array([100.0])` in `cartpole_cont.py` to `high=np.array([4.0])`. Set initial $\theta = \frac{\pi}{10}$, and re-calibrate LQR (i.e., recompute the cost parameters). Describe what needs to be done in order to allow this. Plot the same figures from (3) for this scenario.

3.4 Bonus – Cart-Pole Swing-Up with iLQR

(bonus question - 20 points):

In this question we will use iterative LQR (iLQR) to stabilize the pole at an upright position, starting *from the bottom position at zero velocity.*

1. Derive the iLQR algorithm for this case. Write your algorithm explicitly (i.e., write down the dynamic programming solution and the equation for the optimal linear controller at each time step).

2. Implement your iLQR solution, and stabilize the pole at an upright position, starting *from the bottom position at zero velocity.* Plot θ vs. time for each iteration of iLQR. Explain what you did to make this work.
   Hint 1: make sure to linearize at every time step (i.e., A and B and the controller should be time dependent).
   Hint 2: you might need to fiddle with the cost function to get this to work.

Note: You cannot use any external packages other than the three specified packages.