COGROB HOMEWORK I: OBI-WAN PLAN-OBI

Classical & Temporal Planning with the Unified Planning Framework

Due by 23:59 on December 15th, 2022

Author: Yotam Granov
E-mail: yotam.g@campus.technion.ac.il

Semester: Winter 2022/23
Course Number: 097244

Guidelines

In this assignment, you will be asked to represent and solve a number of classical, numeric, and temporal planning problems using the Unified Planning Framework (UPF) library for Python. You will receive a folder called *HW#1* containing three sub-folders (*Drones*, *Hanoi*, and *Jedi*) as well as a python file called `HW1.py` and a `.yml` file called `cogrob_environment.yml` (the latter is for you to create a Conda virtual environment for this assignment if you'd like - it's the same one that's on the course Moodle). The `HW1.py` file just runs your code solutions for each of the questions in this assignment - do not modify this file! I will use it to check your solutions.

Inside each of the sub-folders (*Drones*, *Hanoi*, and *Jedi*) in the *HW#1* folder, you will find one or more python (`.py`) files inside which you will write your code solutions. I have already added the necessary lines for you to produce the PDDL (`.pddl`) files from the problem definitions you provide in each file, be sure to use them to debug if needed and make sure to submit the final files along with your code. Feel free to use whatever libraries and/or helper functions you'd like!

You can submit the assignment alone or in pairs, and you will need to submit the following files on Moodle in a single **zip file**:

- The `HW1.py` file (no need to modify this) as well as the *Drones*, *Hanoi*, and *Jedi* folders I have provided you with, each containing your solution `.py` files as well as the final `.pddl` files you produced
- One PDF document (`.pdf`) containing your written solutions to the relevant questions (preferably LaTeX)
- One meme related to AI & robotics in either `.png` or `.jpg` format (this will be **5** points of your grade) - the best meme will receive 5 extra bonus points on this assignment (but everyone must submit one!)

Good luck and have fun!

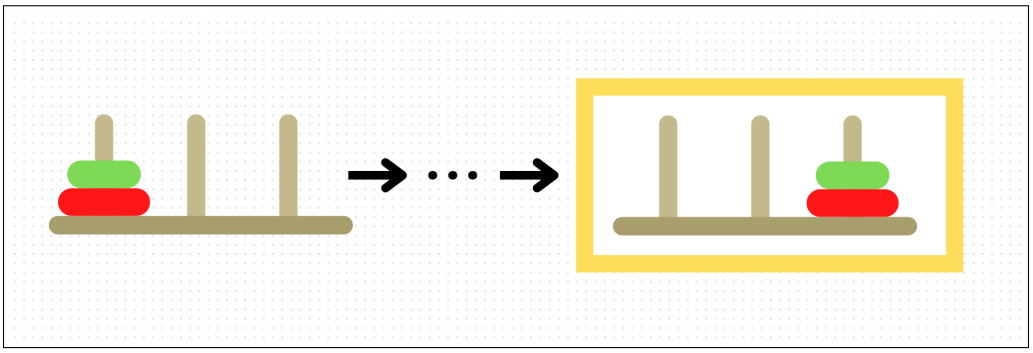
Question 1: The Tower of Hanoi (20 points)

Let's begin this assignment by revisiting the classic Tower of Hanoi problem. The problem statement (as we saw in Tutorial 3) is as follows:

Definition: We have a board with 3 evenly-spaced pegs connected to it and N disks of monotonically-varying sizes that can be placed on the pegs and moved between them. Initially, all of the disks are stacked in decreasing value of diameter on the first peg (the smallest disk is placed on the top). The objective of the puzzle is to move the entire stack of disks to the third peg using the least number of moves while obeying the following rules:

- Only one disk can be moved at a time
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack (so a disk can only be moved if it's the uppermost disk on a stack)
- No disk may be placed on top of a smaller disk

The problem for $N = 2$ looks as follows:



Part 1.1: Initial Challenge

Go to the website `https://www.mathsisfun.com/games/towerofhanoi.html` (which emulates the Tower of Hanoi problem for you), and try to solve the Tower of Hanoi problem with 3 disks ($N = 3$). Record how much time it takes you to solve it, as well as the minimum number of moves you used.

Part 1.2: Coding Assignment (15 points)

Represent the Tower of Hanoi problem for the $N = 3$ case using the UPF in the `hanoi3.py` file, located inside the `Hanoi` folder. Pick an AI solver (you're free to choose any solver you'd like, not only the ones we've seen in class) and have it solve the problem. Additionally, record the amount of time it takes the AI solver to obtain a solution (use any libraries you need to do so).

Part 1.3: Written Questions (5 points)

(a) Return to the website: `https://www.mathsisfun.com/games/towerofhanoi.html`. Use the AI solver's plan to play out the solution. Attach a screenshot of the final image (after the problem is solved) of the website.

(b) How long did it take you to solve the problem? How long did it take the AI solver? Did you both achieve the minimal number of steps?

(c) Estimate the size of the state space for this problem (i.e. the number of nodes in the *state-space graph*). If we run BFS on the *state-space graph*, how many nodes must we expand (not including goal nodes) until we find a

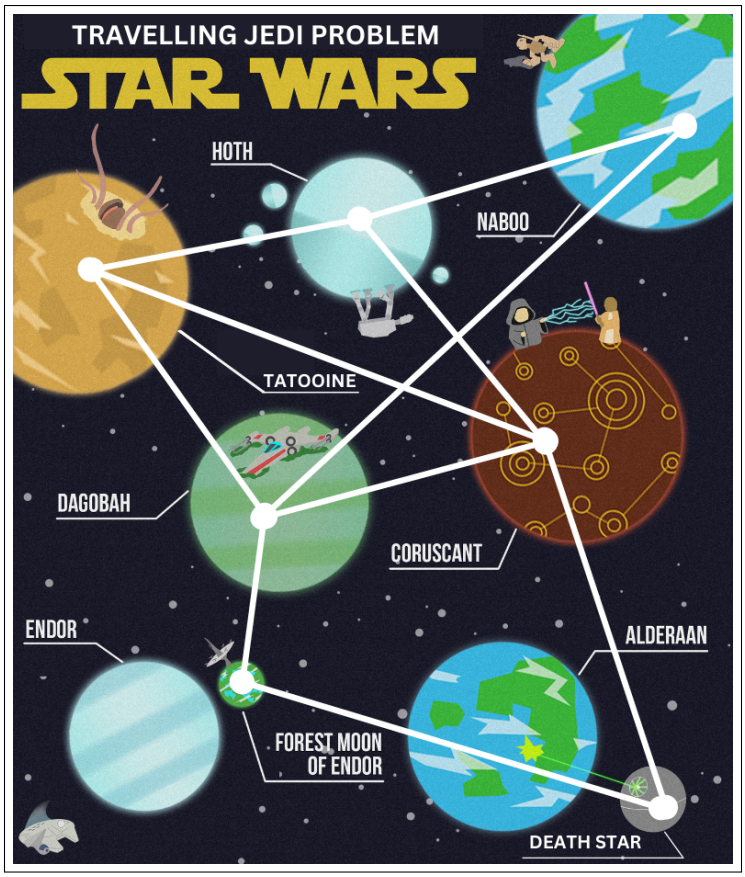
solution, in the worst case? How about for DFS?

(d) Would UCS expand more, less, or the same number of nodes as BFS in this case?

Question 2: The Travelling Jedi Problem (50 points)

Now that we're warmed up, let's tackle a slightly more complex classical planning problem - the Travelling Jedi Problem (TJP), which is based on the Travelling Salesman Problem. The problem statement (as we saw in Tutorial 3) is as follows:

Definition: Luke is taking a road trip with his pals aboard the Millennium Falcon, and he wants to visit a bunch of different planets, moons, and space stations across the galaxy. He will start from Tatooine, and he wants to visit every location exactly once before returning to Tatooine again. He asks the Falcon's computer (known as the Millennium Collective) to plan such a route for him, without regard to the actual geometric distance between the locations (for now). The following figure shows the locations Luke would like to visit, as well as the routes that connect them to each other:



Part 2.1: Coding Assignment [Classic] (15 points)

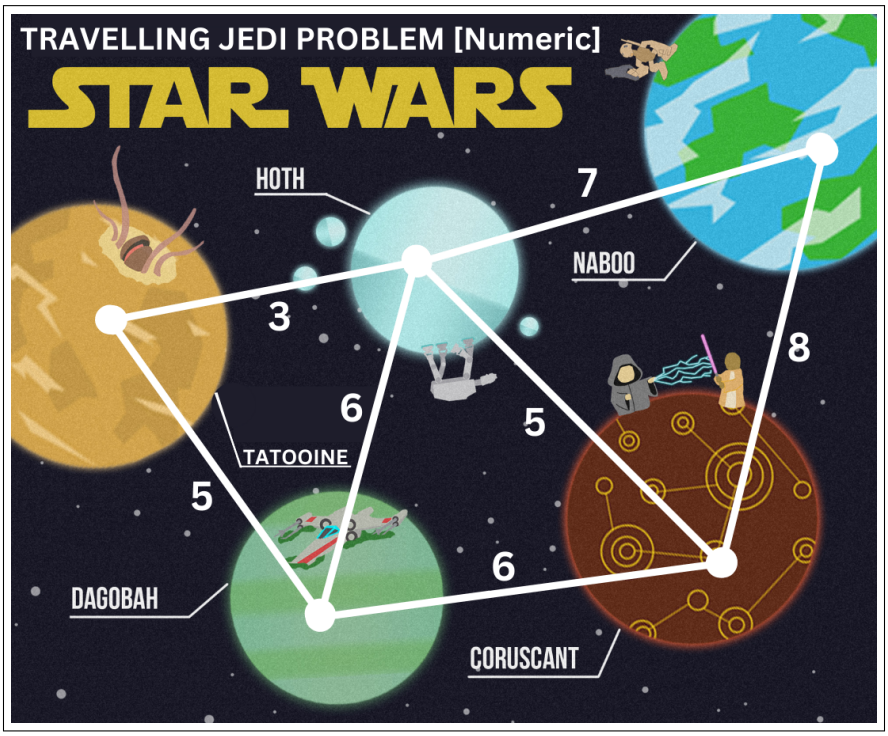
Represent the classic Travelling Jedi Problem using the UPF in the `tjp.py` file, located inside the `Jedi` folder. Pick an AI solver (you're free to choose any solver you'd like, not only the ones we've seen in class) and have it solve the problem.

Part 2.2: Written Questions (1 point)

(a) Is your planner guaranteed to find the optimal solution for this specific problem? Why or why not?

Part 2.3: Coding Assignment [Numeric] (15 points)

Now let's explore a numeric variation of the TJP. Luke and his crew can no longer travel to the Death Star (since they've now successfully blown it up) and they no longer want to travel to the Forest Moon of Endor (the Ewoks have figured out that C3PO was gaslighting them, and have declared war against the whole crew). The Millennium Collective has recalculated all the routes accordingly, and was able to find even better paths between certain locations. On top of that, a new software update has allowed the Collective to determine the geometric length of each route (in parsecs). It produces the following new map of the galaxy:



The number next to each edge in the graph represents the length of that route, in parsecs. We would once again like to find the shortest path for the Falcon to travel along in order to visit every single location *at least* once (we no longer limit each location to a single visit), starting and terminating at Tatooine once again. Represent the numeric variant of the Travelling Jedi Problem using the UPF in the `tjp_num.py` file, located inside the `Jedi` folder. Pick an AI solver (you're free to choose any solver you'd like, not only the ones we've seen in class) and have it solve the problem.

Part 2.4: Written Questions (4 points)

(a) Draw the search tree for the numeric TJP, and show the steps that the UCS algorithm would take in order to reach a solution on it. Is this solution optimal?

Note: The search tree is actually infinite (since we don't limit the number of times Luke can visit each planet, i.e. we're not doing any *cycle-checking*). Thus, you are only required to draw the paths on the search tree that include valid solutions (i.e. they terminate at Tatooine after visiting every planet at least once), and you only need to draw the first seven tiers of the tree (including Tatooine as the root node tier). This should make your lives much easier.

(b) Did your planner find the optimal solution for this specific problem? Why or why not?

(c) Unfortunately, the UPF in its current state does not have great support for optimization metrics when it comes to numeric planning problems represented in this way - there is ongoing work to rectify this. Let's just assume that the UPF (and more specifically, the planner you actually chose) now has better support for search

metrics. What line of code could we add to our python file that would guarantee an optimal solution according to a relevant metric (to see metric options, be sure to check the UPF's documentation)?

Part 2.5: Coding Assignment [Optimal Numeric] (15 points)

It turns out that there is a way (albeit not the most elegant) to reframe the numeric TJP such that we can declare an optimality metric that will allow the Optimal Fast Downward solver to obtain the optimal solution. To do so, we can remove the numeric fluents we defined in the last coding section and instead associate *action costs* to the actions defined in our problem.

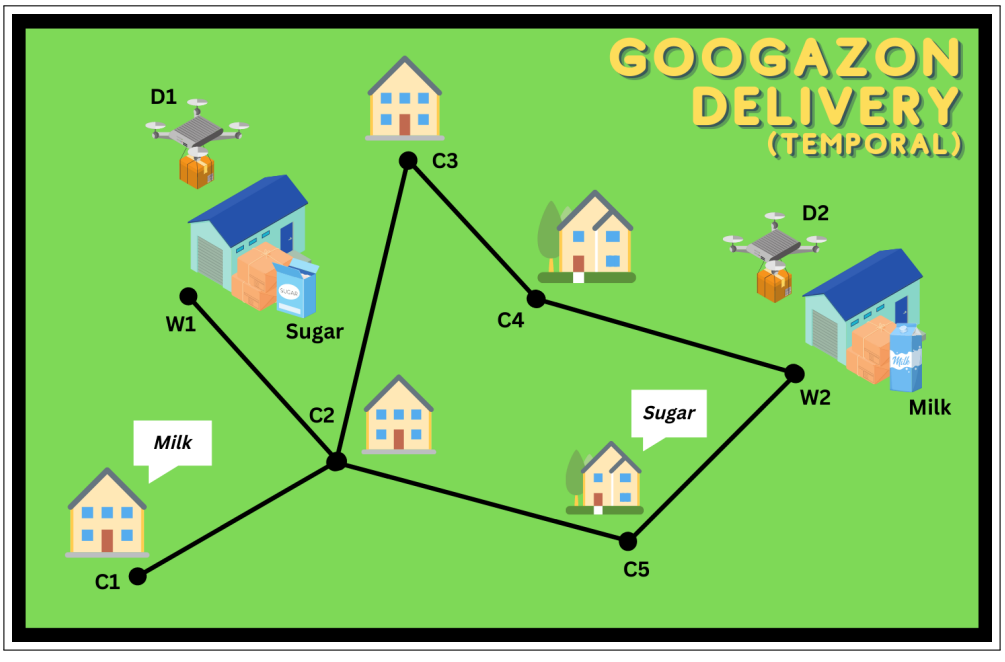
In the current state of the UPF, we must define the action costs in a weird way (we must somehow specify the cost for every single possible action individually; we can't define it using another fluent), and this will allow us to create a dictionary to store the relevant actions (as keys) and their costs (as values of type `Int`, not `int`!). We can then pass this dictionary to a certain metric (hint: check the documentation) in order to optimize our solution. Represent the optimal numeric variant of the Travelling Jedi Problem using the UPF in the `tjp_num_opt.py` file, located inside the `Jedi` folder. Use the Optimal Fast Downward planner here!

Note: I have added a helper function called `Get_Total_Cost` to the `tjp_num_opt.py` file for you, which will automatically calculate the total cost of the path in your solution. Ensure that the number it outputs is in fact the optimal cost!

Question 3: The Googazon Drone Delivery Service (25 points + 10 bonus points)

For the final section of this assignment, we'll venture into temporal planning problems by reframing a classical problem that we've already seen - the Googazon Drone Delivery Service problem. The problem statement (similar to the one from Tutorial 4) is as follows:

Definition: In Lecture 3, you were introduced to Googazon, the largest (fictitious) internet company in the world, who have just launched a new drone delivery service. Googazon currently has *two* drones and *two* warehouses, and each warehouse stores either milk or sugar. The delivery service currently serves *five* customers, who from time to time want either milk or sugar. At a certain point in time, two customers make orders through Googazon's service, and so the company's logistics division must plan the most efficient way (least amount of total time) to get the goods from their warehouses to their customers using the drones (who can only carry one good each at a time). We can visualize the setting of the problem as follows:



It takes each drone 3 minutes to load a good, 5 minutes to unload a good, and 10 minutes to travel between any pair of *adjacent* waypoints (represented as nodes on the graph).

Part 3.1: Coding Assignment [Temporal] (20 points)

Represent the temporal Googazon Drones problem using the UPF in the `drones.py` file, located inside the `Drones` folder. Pick an AI solver (you're free to choose any solver you'd like, not only the ones we've seen in class) and have it solve the problem.

Part 3.2: Written Questions (5 points)

(a) What is the minimum amount of time it could take us to deliver the necessary goods to all of the relevant customers?

(b) Did your solver obtain the optimal solution here? Why or why not?

(c) As we saw before in the case of numerical planning, the UPF in its current state does not have great support for metrics when it comes to temporal planning problems either. Let's just assume that the UPF (and more specifically, the planner you actually chose) now has better support for temporal optimization metrics. What line of code could we add to our python file that would guarantee an optimal solution according to the relevant metric (to see metric options, be sure to check the UPF's documentation)?

BONUS - Part 3.3: Coding Assignment [Optimal Temporal] (10 points)

Given that our problem is simple enough, we can implement our own optimization scheme in order to obtain the optimal solution for this problem. We can do so by breaking up our problem into smaller sub-problems, solving them, and using their results to reach the optimal solution somehow.

Implement an optimization scheme for the temporal Googazon Drones problem using the UPF in the `drones_opt.py` file, located inside the `Drones` folder. Pick an AI solver (you're free to choose any solver you'd like, not only the ones we've seen in class) and have it solve the problem. Ensure that the minimal time you obtain here agrees with your solution for question 3.2(a)!

I hope you enjoyed!