CS802 Assignment 01

Outline

This assignment is worth 25% of the mark for this class. It has two parts:

1. **Temporal Planning** will require you to extend a propositional logistics domain written in PDDL to include new objects and actions (boats and planes), to include durative actions, and deadlines. [20 marks]
2. **Temporal Networks (Python)** will require you to implement algorithms in Python for finding All-Pairs Shortest Paths in temporal networks, checking temporal network consistency, and creating minimally dispatchable temporal networks. [20 marks]

Part 1 Temporal Planning

This part requires you to build a temporal and numeric planning problem, generate plans using a temporal planner, and experiment with different features of PDDL2.2.

This part is worth 12.5% of the mark for this class.

1. Write a PDDL2.2 domain and problem for the Temporal Logistics Scenario.
2. Use a temporal planner to find a plan and extend the domain and problem to include delivery deadlines.
3. Using the planner and problem file, find a plan that optimises the alternative quality metric described below.

On MyPlace you will have to an archive containing 4 files (*part1.zip*):

- Your final PDDL domain and problem files (including deadlines).
- A text file with the optimal plan from part 1.2 (*plan_part2.txt*).
- A text file with the optimal plan from part 1.3 (*plan_part3.txt*).

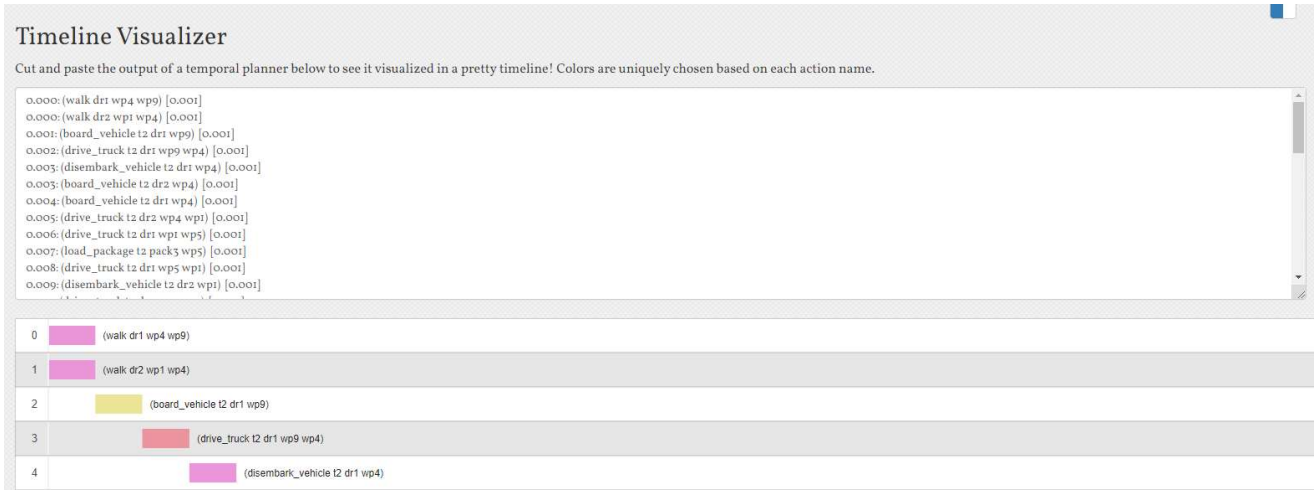
1.1 Temporal Logistics Scenario

The first part of the assignment is to write a PDDL domain and problem for the temporal logistics scenario. *Feel free to use the propositional logistics scenario from CS814 as a base, and modify it.*

The following link includes temporal plugins for the online editor. Either start from this template, or open this link and replace the domain and problem with your own solution from Part 3.

http://editor.planning.domains/#read_session=JAlHoWLhP2

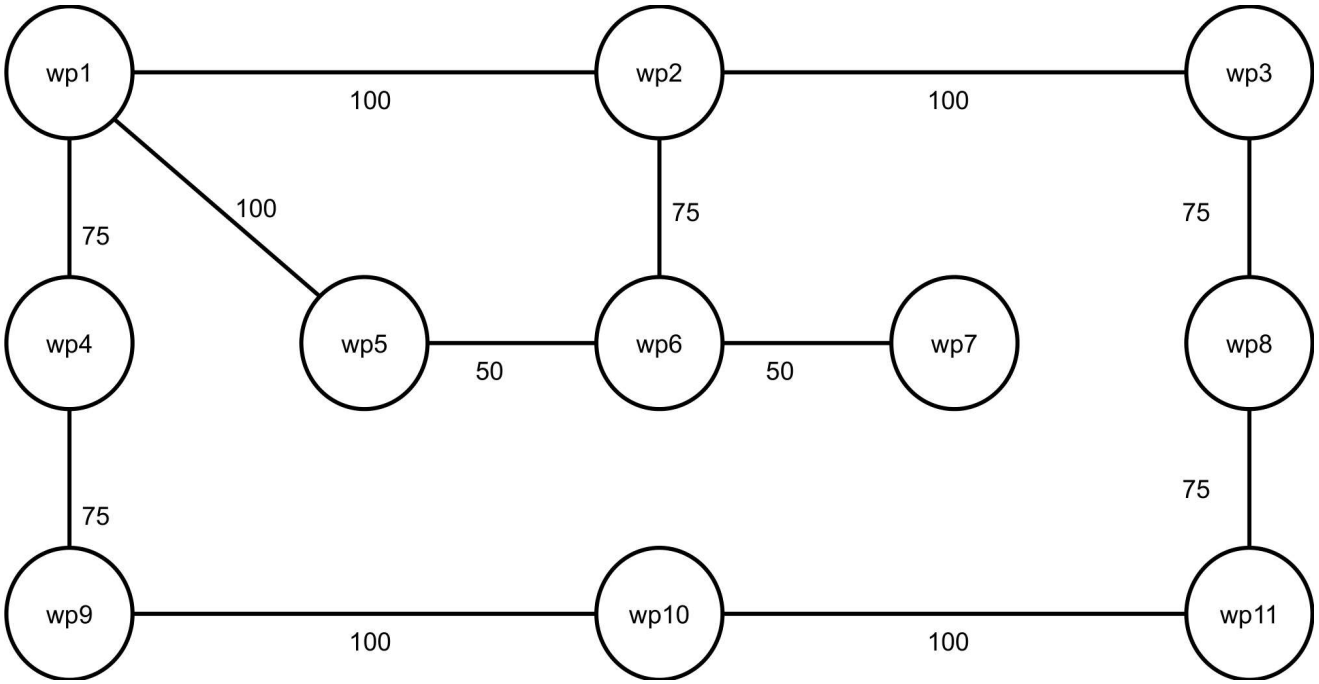
The temporal plugins include a temporal planner to solve the problems, and a timeline visualiser:



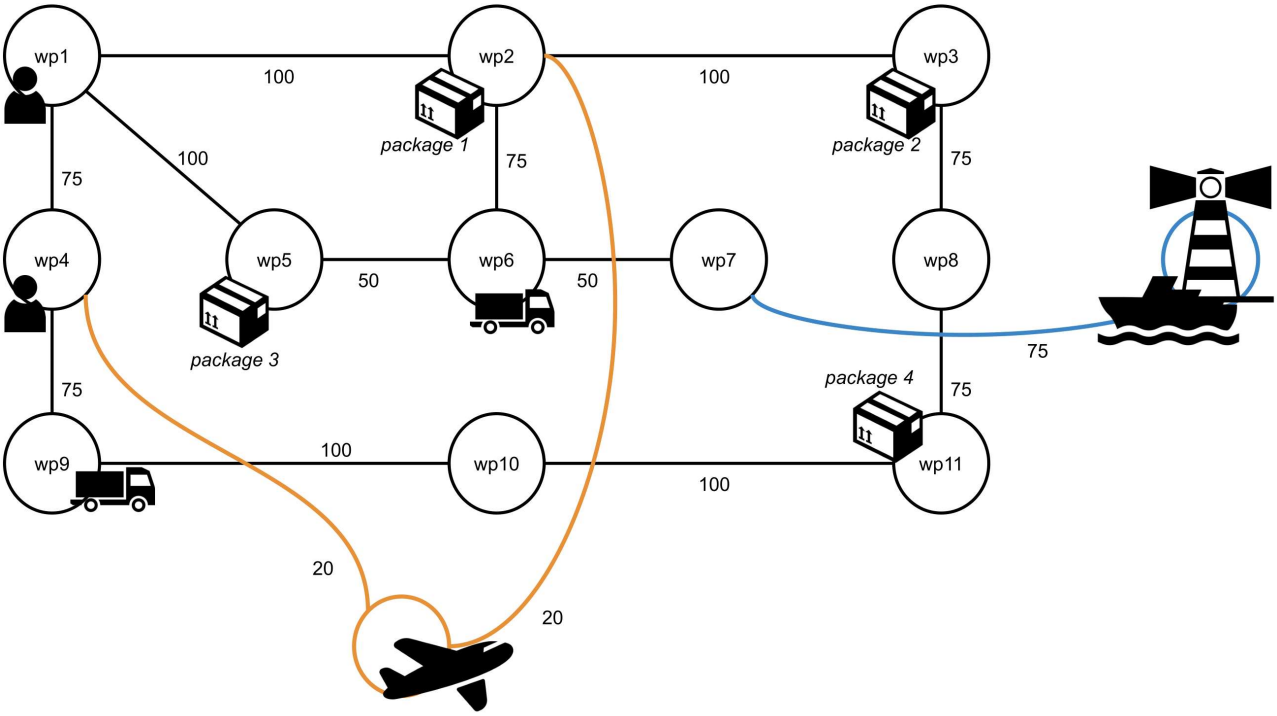
The logistics scenario has the following objects:

- 11 ground waypoints.
- 1 lighthouse and one sky waypoint.
- 4 packages.
- 2 drivers, 2 trucks, 1 plane, and 1 ship.

The distances of each connection are shown below:



The initial state shows the sky and lighthouse waypoints, and the starting locations of all of the objects:



Tip: if the planner is unable to find solutions to the problem, try using a much smaller problem instance until you are confident the domain is correct.

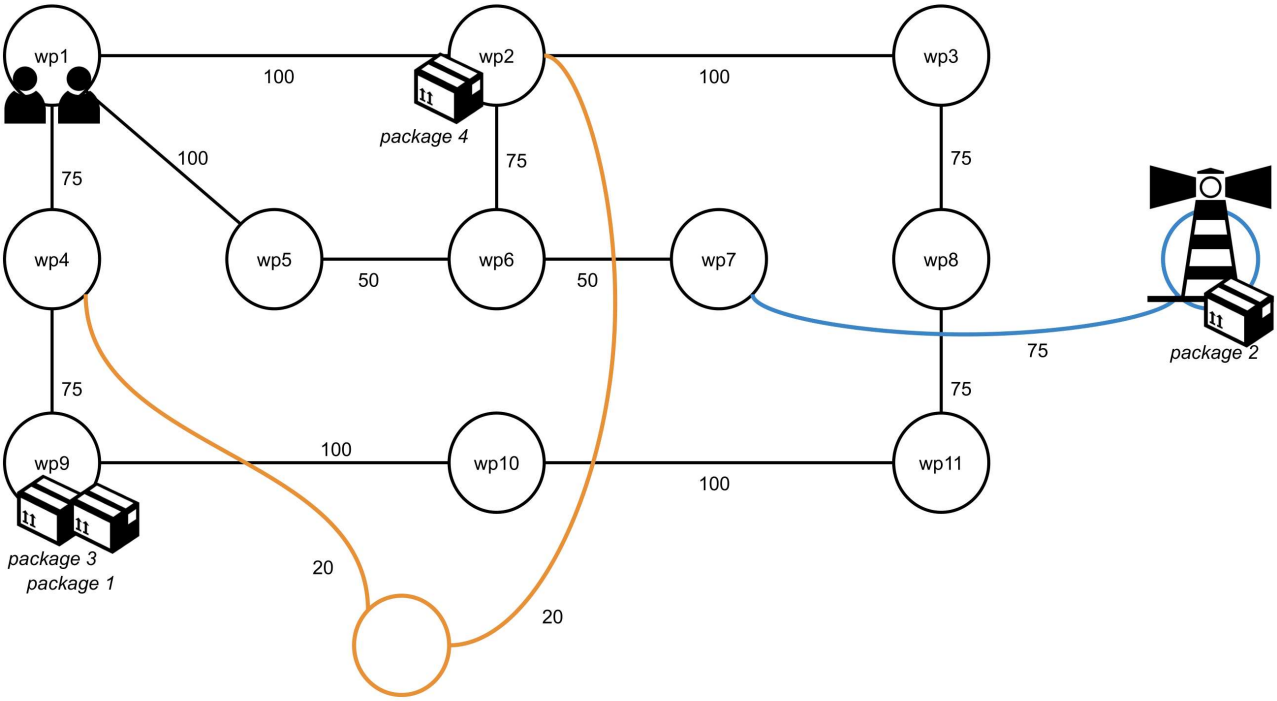
The durative actions in the domain should model the following details:

- Packages can be loaded into and unloaded from trucks (10 time units).
- Drivers can walk between connected waypoints (at a speed of 0.5).
- Drivers can get into and out of trucks (10 time units).
- Trucks with drivers can drive between connected waypoints (at a speed of 1).
- The boat and the plane don't need drivers to move. Packages don't need drivers to load and unload them.

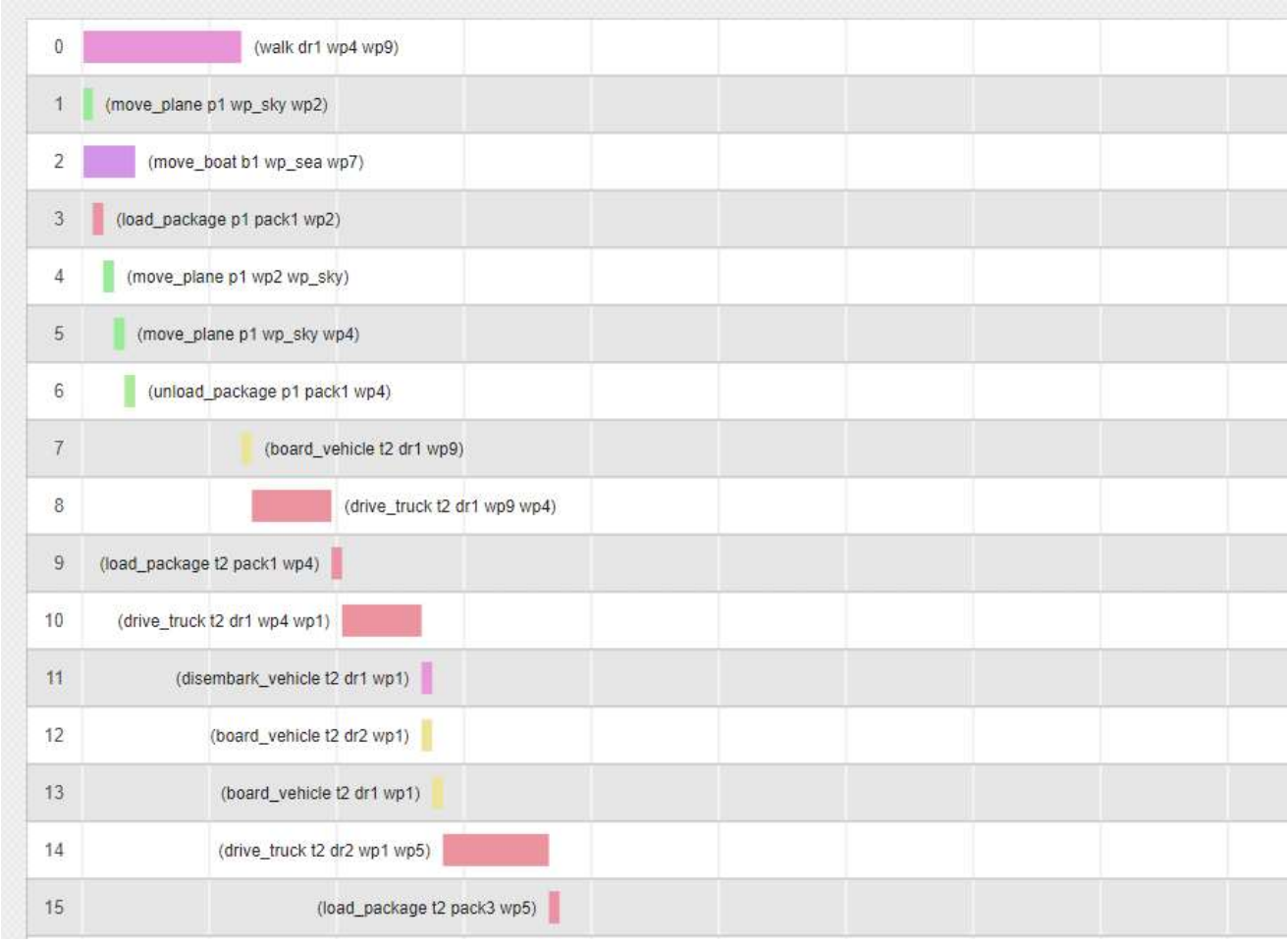
- Boats and Planes can only travel over the blue and yellow edges (connected to the lighthouse and the sky). Trucks can only travel on roads.
- The boat travels at a speed of 1.5.
- The plane travels at a speed of 2.

The goal condition:

- Packages 1 and 3 are at waypoint wp9.
- Packages 2 is at the lighthouse.
- Package 4 is at wp2.
- Both drivers are at wp1.



Once it is working, you should see a timeline like this:



1.2 Include Delivery Deadlines

Task: Add deadlines to the delivery time for each package.

A deadline can be set using a **Timed Initial Literal** (TIL). A TIL can be used to specify that a fact becomes true or false a set time after the initial state. In the example below, the proposition `(building-open)` becomes true at time 1000, and false again at time 2000.

```
(:init
    (at 1000 (building-open))
    (at 2000 (not (building-open)))
```

- The problem (and domain) will need to be updated so that the package delivery goals must be achieved before 2500 time units.
- Experiment with different TILs to find the minimum deadline for your solution.

1.3 Alternative quality metric

Task: In this exercise we will use the planner to produce the **solution that I prefer**:

1. My solution should deliver package 1 in the fastest possible time, and
2. Given (1), the solution should deliver the remaining packages within the shortest possible time.

With some planners it is possible to specify an optimisation metric. For example:

```
(:metric minimise (delivery-time package1))
```

However, the temporal planner connected to the online editor will only attempt to minimise total plan duration. In order to find the solution that I prefer, you will need to find a different approach and use TILs (as deadlines).

2.1 Parsing an STN from file

The first part of the coursework is to parse an STN from a DOT file. On MyPlace there is a template Python file to work from: *stn_check.ipynb*. This file has template code to get you started and *TODO* comments, which indicate code that must be completed.

The template file includes:

- A class for STN graphs.
- Code to load a file from command line arguments.
- Code to call the methods of an STN.

These lines are explained in detail below.

Note: You can use your own implementation and different data structures, the only requirement is that sufficient data is stored to correctly perform parts 2.2-2.4

There are also 7 dot graphs to use for testing.

First the variables that are suggested to store the STN are a list of node names (or IDs) and an adjacency list.

```
class STN:
    ## useful maximum upper bound
    inf = sys.float_info.max

    ## list of nodes and adjacency list
    def __init__(self):
        nodes = []
        adj_list = {}
```

An adjacency list can be stored as a 2D dictionary. For example:

```
graph = {'Step0': {'Z': -0.001, 'Step3': 8.000, },
         'Step1': {'Z': -0.001, 'Step0': 0.000, }}
```

You can read more about how to use python dictionaries here:
https://www.w3schools.com/python/python_dictionaries.asp

After this there are three empty methods that must be completed in parts 2.2-2.4:

```
## TODO print STN in DOT form
def print_stn(self):
    dot = Digraph()
    ## add nodes
    ## add edges
    return dot

## TODO check the consistency of the STN
def check_consistency(self):
    return False

## TODO remove dominated edges from the STN
def make_minimal(self):
    return True
```

The main method for the file is below. Its purpose is to:

1. Open a DOT file and read its contents to create an STN object.
2. Call the consistency check, and
3. if it is consistent, call the make minimal and print methods.

```
## open the DOT file
filename = "ADD FILE NAME"
with open(filename, "r") as dotfile:

    stn = STN()

    for line in dotfile:
        ## read edge
        if "->" in line:
            ## TODO save nodes in list stn.nodes
            ## TODO initialise adjacency list stn.adj_list
            continue

    ## TODO set all self distances to 0
    ## TODO set missing distances to infinity
```

```
## print original STN
stn.print_stn()
```

```
### check consistency and print APSP STN (if consistent)
print(stn.check_consistency())
stn.print_stn()
```

```
### make the STN dispatchable and print the minimal STN
stn.make_minimal()
stn.print_stn()
```

Given the DOT file *tiny_stn.dot:*

```
digraph plan {
    A [label="A: (action_a):0[0]"];
    B [label="B: (action_a):2[2]"];

    A -> B [ label="5.000" ];
    B -> A [ label="-5.000" ];
}
```

The node and adjacency lists should contain:

```
stn.nodes = ['A', 'B']
stn.adj_list = {'A': {'B': 5.000}, 'B': {'A': -5.000}}
```

2.2 Implement Floyd-Warshall

The second part of the assignment is to complete the *check_consistency* method. This method should be an implementation of the Floyd-Warshall algorithm, and do two things:

- Return false if the STN is inconsistent (i.e. there is a negative cycle), otherwise return true.
- Complete the graph so that it is an all-pairs shortest path graph.

2.3 Print in DOT form.

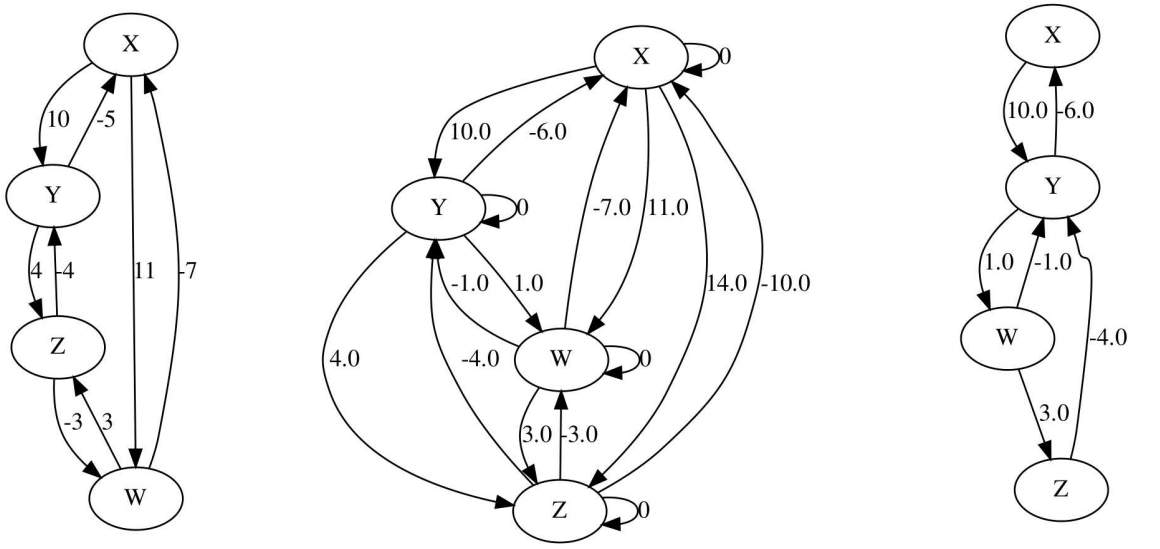
The third part of the assignment is to complete the *print_stn* method. This method should draw the STN in DOT form to the screen.

To learn how to use the graphviz library, you should use the guide here:
https://graphviz.readthedocs.io/en/stable/manual.html

2.4 Removing Dominated Edges

The final part of the assignment is to implement the *make_minimal* method. This method is used to remove redundant edges from the STN so that it can be efficiently executed. An edge is redundant if it is *dominated* by another edge (as described in the lectures).

Example:



The original input (left); all-pairs shortest path STN (middle); minimal STN (right). Note that there are potentially multiple networks with a minimal number of edges.