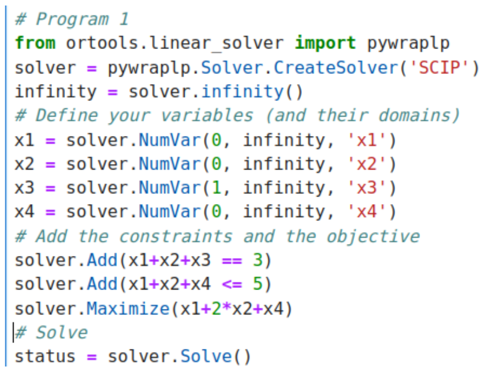
Mini Project 2 – Constraint-Based Optimization and PDDL

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. Constraint-based Optimization

A. Implementing Some Example Programs

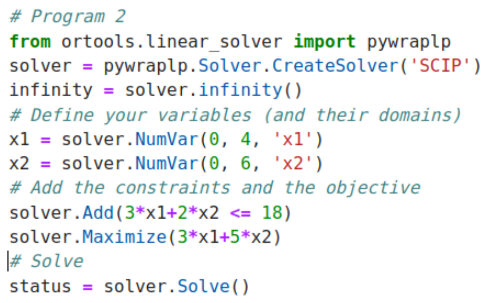
Implement the following two mixed-integer linear programs.



```
# Program 1
from ortools.linear_solver import pywraplp
solver = pywraplp.Solver.CreateSolver('SCIP')
infinity = solver.infinity()
# Define your variables (and their domains)
x1 = solver.NumVar(0, infinity, 'x1')
x2 = solver.NumVar(0, infinity, 'x2')
x3 = solver.NumVar(1, infinity, 'x3')
x4 = solver.NumVar(0, infinity, 'x4')
# Add the constraints and the objective
solver.Add(x1+x2+x3 == 3)
solver.Add(x1+x2+x4 <= 5)
solver.Maximize(x1+2*x2+x4)
# Solve
status = solver.Solve()
```

Fig. 1: Variables definition and constraints of program 1

The optimal solution: $x_1 = 0$, $x_2 = 2$, $x_2 = 1$, $x_4 = 3$, and the objective value is 7.0.



```
# Program 2
from ortools.linear_solver import pywraplp
solver = pywraplp.Solver.CreateSolver('SCIP')
infinity = solver.infinity()
# Define your variables (and their domains)
x1 = solver.NumVar(0, 4, 'x1')
x2 = solver.NumVar(0, 6, 'x2')
# Add the constraints and the objective
solver.Add(3*x1+2*x2 <= 18)
solver.Maximize(3*x1+5*x2)
# Solve
status = solver.Solve()
```

Fig. 2: Variables definition and constraints of program 2

The optimal solution: $x_1 = 2$, $x_2 = 6$, and the objective value is 36.0.

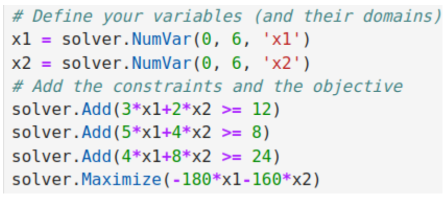
B. A word problem

$$\begin{aligned} \text{min} \;\; & z = 180x_1 + 160x_2 \\ \text{s.t.} \;\; & 3x_1 + 2x_2 \geq 12, \\ & 5x_1 + 4x_2 \geq 8, \\ & 4x_1 + 8x_2 \geq 24, \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \leq 6 \end{aligned}$$

Fig. 3: Mine - Problem

In this problem, we need two variables: the working day at Heigh Ho mine x_1 and at Kessel mine x_2. The problem is defined by the equations in Fig. 3.

We have 3 constraints for the minimum of three grade materials, and two constraints for the number of working day in a week.



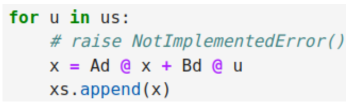
```
# Define your variables (and their domains)
x1 = solver.NumVar(0, 6, 'x1')
x2 = solver.NumVar(0, 6, 'x2')
# Add the constraints and the objective
solver.Add(3*x1+2*x2 >= 12)
solver.Add(5*x1+4*x2 >= 8)
solver.Add(4*x1+8*x2 >= 24)
solver.Maximize(-180*x1-160*x2)
```

Fig. 4: Mine - Code

Because we can take fraction of a working day, so we can find an optimal solution: $x_1 = 3$ days for Heigh Ho mine and $x_2 = 1.5$ days for Kessel mine with the optimal cost is \$780.

II. Trajectory Optimization with MILPs

A. Computing a Trajectory Given Thrusts



```
for u in us:
    # raise NotImplementedError()
    x = Ad @ x + Bd @ u
    xs.append(x)
```

Fig. 5: Given Thrusts – code to determine the state series

We need to define a series of states for the agent with predefined control input values. It is nothing more than a sum of two matrices multiplication as shown in Fig. 5. With the state series, we can determine the positions of velocity of the agent, which are shown in Fig. 6.

B. Trajectory Optimization

In this task, we need to determine the control thrusts to all four actuators, which manipulate the movement of the agent in 2D. Due to $or-tool$, we need to use two external functions *vec_add* and *matmul* to calculate the sum of two vector and multiplication of two matrices. The states calculation and max thrust control constraints are shown in Fig. 7.

The found trajectories with varying time steps are shown in Fig. 8.

As increasing the number of sample points, the trajectories are smoother and the objective values are reduced gradually, which is shown in Fig. 9. It seems with small time-steps, we can calculate more accurate the control to respond to the agent movement, and save energy for that. Particularly, the control values for 10, 50, and 100 time steps are: 3.554, 3.50, and 3.497, respectively.

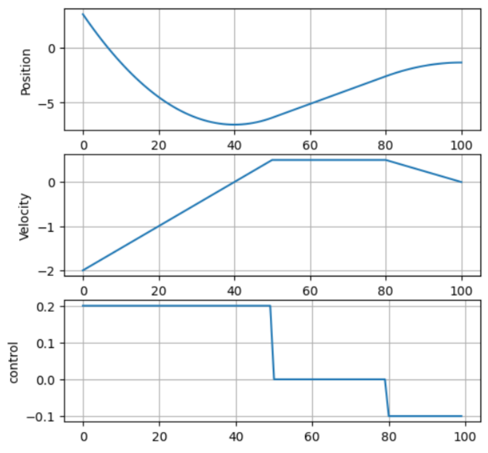
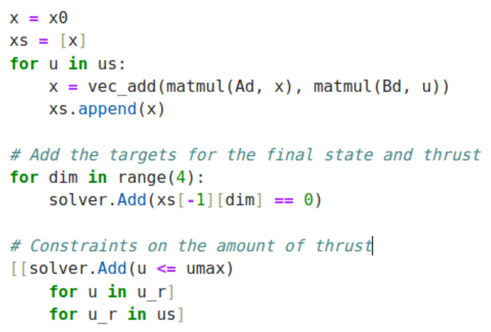


Fig. 6: Given Thrusts: Position, Velocity and Control


```
x = x0
xs = [x]
for u in us:
    x = vec_add(matmul(Ad, x), matmul(Bd, u))
    xs.append(x)

# Add the targets for the final state and thrust
for dim in range(4):
    solver.Add(xs[-1][dim] == 0)

# Constraints on the amount of thrust
[[solver.Add(u <= umax)
    for u in u_r]
    for u_r in us]
```
Fig. 7: Trajectory Optimization: the code to calculate the states, and adding thrust constraints

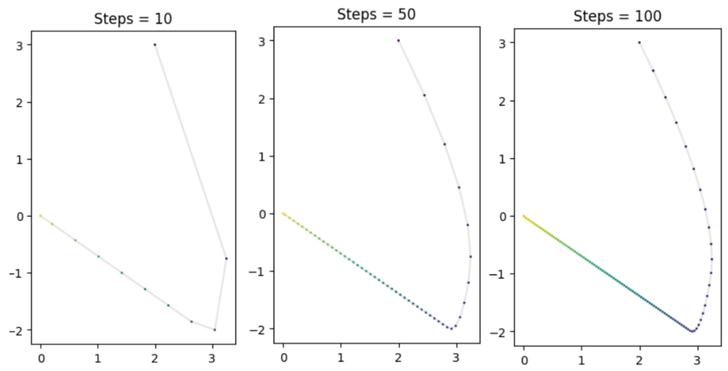


Fig. 8: Trajectories with 3 time steps 10, 50, and 100 – No obstacles

C. Adding an obstacle

As an obstacle appears in the environment, we would like the agent avoiding it. We need to define some ”slack” variables, and add some constraints with them. The slack variables are binary one, so we define them as $integer$ with range from 0 to 1. The ”slack” variables are defined as in Fig. 10.

To ensure all the states in the trajectories are on not in the obstacles, we add constraints for all states along the trajectories. The found trajectories with ”non-zero” initial velocity and zero initial velocity are shown in Fig. 11. The objective value for non-zero initial velocity (3.846) are much larger than the zero one (0.533), it means we need to consume more energy to handle the moving one, and it makes sense.

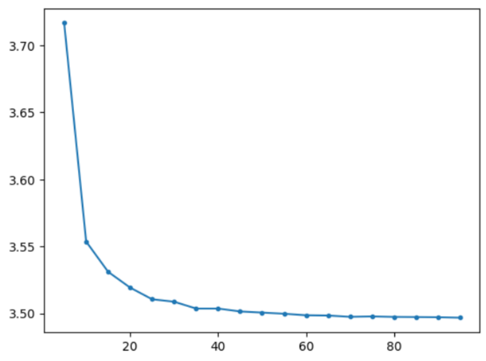
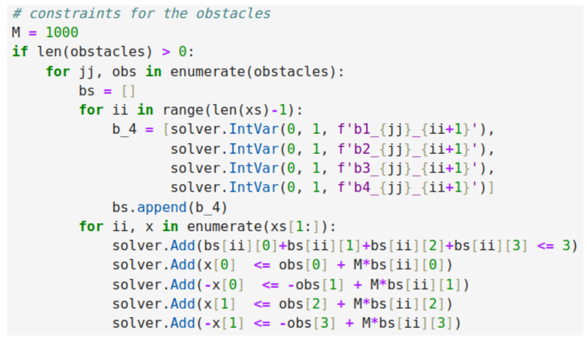


Fig. 9: Objective values with varying time steps – No obstacles


```
# constraints for the obstacles
M = 1000
if len(obstacles) > 0:
    for jj, obs in enumerate(obstacles):
        bs = []
        for ii in range(len(xs)-1):
            b_4 = [solver.IntVar(0, 1, f'b1_{jj}_{ii+1}'),
                   solver.IntVar(0, 1, f'b2_{jj}_{ii+1}'),
                   solver.IntVar(0, 1, f'b3_{jj}_{ii+1}'),
                   solver.IntVar(0, 1, f'b4_{jj}_{ii+1}')]
            bs.append(b_4)
        for ii, x in enumerate(xs[1:]):
            solver.Add(bs[ii][0]+bs[ii][1]+bs[ii][2]+bs[ii][3] <= 3)
            solver.Add(x[0]  <= obs[0] + M*bs[ii][0])
            solver.Add(-x[0]  <= -obs[1] + M*bs[ii][1])
            solver.Add(x[1]  <= obs[2] + M*bs[ii][2])
            solver.Add(-x[1] <= -obs[3] + M*bs[ii][3])
```
Fig. 10: Obstacles constraints

Along the trajectories, we can see that there are collisions at the corner of the obstacles. The reason is, we discretize the continuous time into time steps. The problem can be fixed by inflating the obstacles.

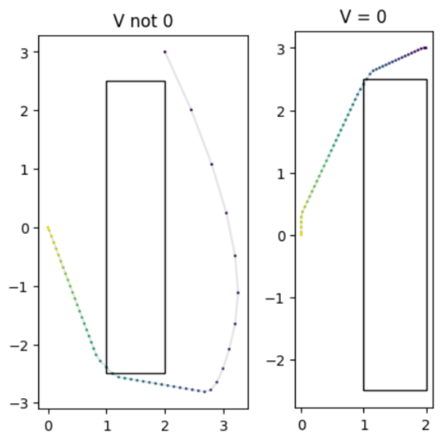


Fig. 11: Trajectories with obstacles

The similar things are also true as we remove the obstacle. With non-zero initial velocity, the objective is 3.501. If no obstacle, and zero initial velocity, the agent just moves in a straight line directly to its goal (Fig. 12) with little effort (objective value is 0.408).

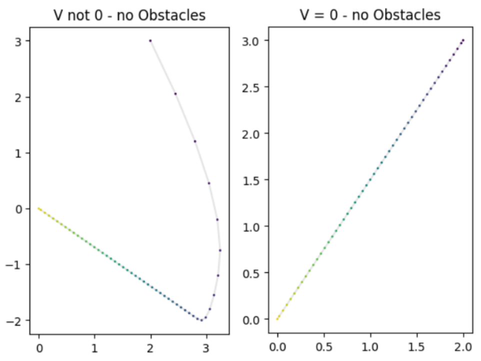
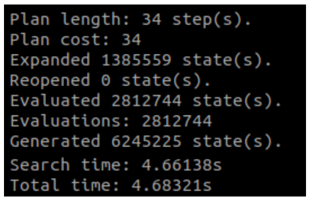


Fig. 12: Trajectories with no obstacle

III. PDDL

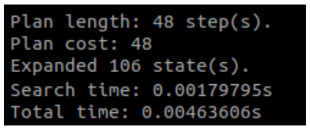
A. Running Fast Downward

The results as running the problem with two different heuristics are shown in Fig. 13 and Fig. 14, respectively.



```
Plan length: 34 step(s).
Plan cost: 34
Expanded 1385559 state(s).
Reopened 0 state(s).
Evaluated 2812744 state(s).
Evaluations: 2812744
Generated 6245225 state(s).
Search time: 4.66138s
Total time: 4.68321s
```

Fig. 13: Plan length, Plan cost, Expanded states, Total Run time with A^* heuritic



```
Plan length: 48 step(s).
Plan cost: 48
Expanded 106 state(s).
Search time: 0.00179795s
Total time: 0.00463606s
```

Fig. 14: Plan length, Plan cost, Expanded states, Total Run time with FF heuritic

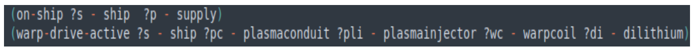
As we can see, A^* heuristic provides an optimal solution with shorter plan length (34 steps), meanwhile FF heuristic plan length is 48 steps. However, to have optimality, the cost is the run time (4.68 sec) is much longer than 0.0046 seconds of FF heuristic.

In my opinion, in applications where planning is implemented offline, we can use A^* heuristic to find an optimal solution. However, in real-time scenarios or online planning, we prefer FF heuristic because runtime is more critical than optimal solutions in the applications.

B. An Example PDDL Problem

To solve the problem, we define some new predicates and actions. Moreover, we all need to add some new initial conditions for the problem.

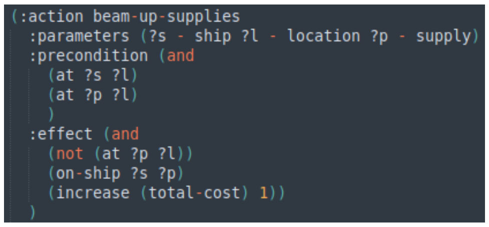
The two new predicates are: *on-ship* and *warp-drive-active* (Fig. 15). The first one check whether a supply is on the ship. The second will activate the warp-drive if there is all the need components.



```
(on-ship ?s - ship  ?p - supply)
(warp-drive-active ?s - ship ?pc - plasmaconduit ?pli - plasmainjector ?wc - warpcoil ?di - dilithium)
```

Fig. 15: Two new predicates

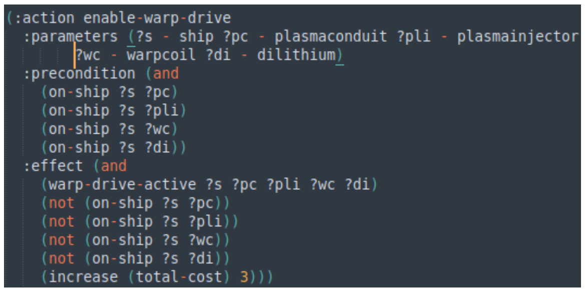
There are three new actions: *travel-warp-speed*, *beam-up-supplies*, and *enable-warp-drive* are shown in Fig. 18, 16, and 17, respectively. In action *beam-up-supplies*, the ship will travel to several planets to collect the necessary supplies. As the ship is at a planet which process a supply, we will perform this action to get the supply on the ship.



```
(:action beam-up-supplies
  :parameters (?s - ship ?l - location ?p - supply)
  :precondition (and
    (at ?s ?l)
    (at ?p ?l)
    )
  :effect (and
    (not (at ?p ?l))
    (on-ship ?s ?p)
    (increase (total-cost) 1))
  )
```

Fig. 16: Beam-up-supplies action

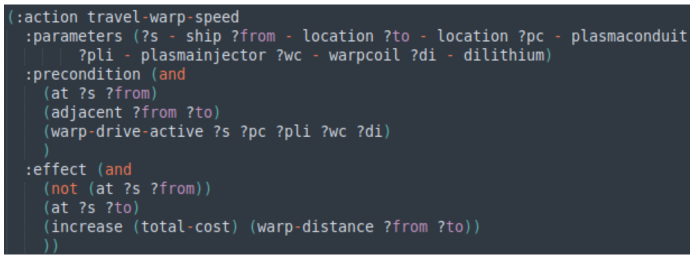
Action *enable-warp-drive* (Fig. 17) will assembly all the necessary components (if they are on the ship) into a *warp-drive*. From this time, the *warp-drive* is ready to use, and the separate components are not available anymore.



```
(:action enable-warp-drive
  :parameters (?s - ship ?pc - plasmaconduit ?pli - plasmainjector
        ?wc - warpcoil ?di - dilithium)
  :precondition (and
    (on-ship ?s ?pc)
    (on-ship ?s ?pli)
    (on-ship ?s ?wc)
    (on-ship ?s ?di))
  :effect (and
    (warp-drive-active ?s ?pc ?pli ?wc ?di)
    (not (on-ship ?s ?pc))
    (not (on-ship ?s ?pli))
    (not (on-ship ?s ?wc))
    (not (on-ship ?s ?di))
    (increase (total-cost) 3)))
```

Fig. 17: enable-warp-drive

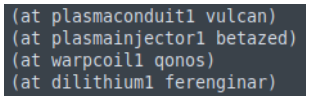
As the *warp-drive* is ready to use, the ship can use this actuator to speed up the travel, and save-up a lot of time. It is performed by action *fig:warp-speed* (Fig. 18).



```
(:action travel-warp-speed
  :parameters (?s - ship ?from - location ?to - location ?pc - plasmaconduit
        ?pli - plasmainjector ?wc - warpcoil ?di - dilithium)
  :precondition (and
    (at ?s ?from)
    (adjacent ?from ?to)
    (warp-drive-active ?s ?pc ?pli ?wc ?di)
    )
  :effect (and
    (not (at ?s ?from))
    (at ?s ?to)
    (increase (total-cost) (warp-distance ?from ?to))
    ))
```

Fig. 18: Warp-drive action

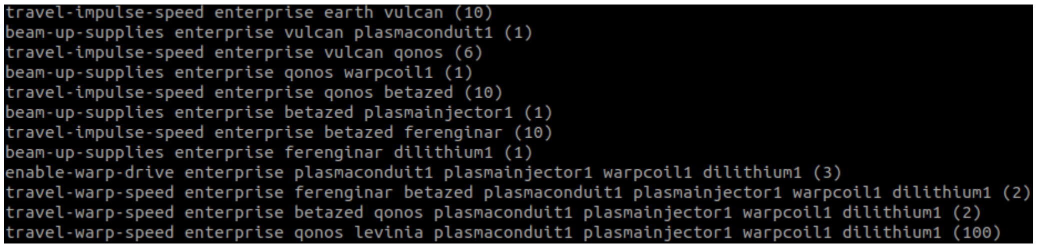
Because we use some supplies to build a *warp-drive*, we will initialize the supplies on the planets as in Fig. 19.



```
(at plasmaconduit1 vulcan)
(at plasmainjector1 betazed)
(at warpcoil1 qonos)
(at dilithium1 ferenginar)
```

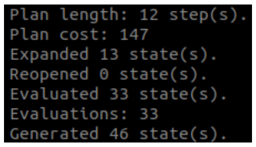
Fig. 19: Initialize the supplies

Now, we can plan a trajectory for the ship. The action sequence is shown in Fig. 20. First, the ship travels around to collect the warp-drive supplies. As the supplies are ready, we build warp-drive and then travel with warp-drive power.



```
travel-impulse-speed enterprise earth vulcan (10)
beam-up-supplies enterprise vulcan plasmaconduit1 (1)
travel-impulse-speed enterprise vulcan qonos (6)
beam-up-supplies enterprise qonos warpcoil1 (1)
travel-impulse-speed enterprise qonos betazed (10)
beam-up-supplies enterprise betazed plasmainjector1 (1)
travel-impulse-speed enterprise betazed ferenginar (10)
beam-up-supplies enterprise ferenginar dilithium1 (1)
enable-warp-drive enterprise plasmaconduit1 plasmainjector1 warpcoil1 dilithium1 (3)
travel-warp-speed enterprise ferenginar betazed plasmaconduit1 plasmainjector1 warpcoil1 dilithium1 (2)
travel-warp-speed enterprise betazed qonos plasmaconduit1 plasmainjector1 warpcoil1 dilithium1 (2)
travel-warp-speed enterprise qonos levinia plasmaconduit1 plasmainjector1 warpcoil1 dilithium1 (100)
```

Fig. 20: Move state



```
Plan length: 12 step(s).
Plan cost: 147
Expanded 13 state(s).
Reopened 0 state(s).
Evaluated 33 state(s).
Evaluations: 33
Generated 46 state(s).
```

Fig. 21: Plan length, Plan Cost, and Expanded states

The total path length is 12, with planning cost of 147. The planner expands 13 states, and generates 46 state (Fig. 21). With this impressive runtime, the ship is on time to implement the rescue mission.