Motion Planning Lab 3

Daniel Gershkovich - 209088723
Barakat Gadban - 200384436
Roman Spektor - 329066526

Introduction

For this lab, the goal was to write the first letter of each of our names using 6 cubes, moved into place by the robot manipulator from a known starting position.

We first implemented the RRT-Star algorithm for the UR5e robot manipulator in the lab.

We then devised an algorithm to improve plans generated by RRT*, created a framework to optimise the plans using parallelism and implemented code to generate a .gif of the whole plan.

Implementation Details

Outline

1) Load existing paths (if possible)
2) Run k iterations of RRT* on each part of the path using multiprocessing
3) Apply smoothing to each resulting path part
4) If a new path part is better than the existing one, save it

Overall Plan and path parts

We split the plans into three folders, one for each of the team members.
Inside each folder we have two .npy files detailing the plan for each of the six cubes: one for gripping the cube with URE5e, and one for placing the cube in the correct location.

The target locations for each cube were pre-computed using the inverse kinematics functions provided in the initial code.
For each cube, we selected the target robot configuration (from which the cube was then dropped unto the table) which was the closest, in terms of edge cost, to the approach configuration for that cube.
We then saved the configurations to a numpy file to avoid re-calculating them and to use the same configs for each run of the algorithm, which was required for the next part (optimization).

RRT vs RRT* and the plan optimization framework

After implementing the basic algorithm, we decided to create a framework to optimise the plans for each cube, putting an emphasis on two parameters:

1) Plan cost in terms of the edge_cost function
2) Plan simplicity, in this case measured by the number of configurations in it

To optimise the cost parameter, we created a class which uses multiprocessing to run the RRT-Star algorithm many times in parallel, updating the existing plan part (saved as a .npy file) if the new plan is cheaper.
We noticed that running RRT-Star results in much cheaper plans compared to normal RRT, but also in much much larger (= more complex) plans, depending on the maximum number of iterations passed to the RRT* algorithm.

To allow for many iterations of the RRT* algorithm, we decided to use python's multiprocessing library, running 16 iterations of RRT* in parallel.
This was done on the faculty's lambda cluster, in 3 separate jobs (one for each team member).

We noticed that generating plans for gripping cubes (especially cubes 3 and 6) was much harder for the algorithm compared to generating other plans. Therefore, when trying to find a plan to grip a cube, we decided to find the reverse plan (from the cube approach to the previous arm's position, instead of the other way around) and then reversing the order of configurations in the resulting plan. This proved to have a large effect on the success rate of RRT* using a relatively low number of iterations, and allowed for much more overall iterations of the whole algorithm to be run.

Path smoothing

In order to fix this issue, we implemented a path smoothing algorithm:

```
while length(plan) > 2:
        find index for which, if removed:
                Results in a legal plan (no collisions)
                Results in the cheapest plan
                Results in a plan that is cheaper than the original plan
        If no such index found:
                Break loop
        Remove the point at the index from the plan
Return plan
```

This algorithm, while very inefficient computationally, turned out to be very effective - shortening plans from >200 configurations down to just 2 - 5, without sacrificing on plan cost.
Additionally, to avoid re-calculating if removing a config between two other configs is legal for each iteration of the while loop (as in each iteration, one point gets removed, leaving the others unchanged), we used a caching mechanism. This could be implemented using a more efficient loop instead, but caching was much simpler to implement and was 'good enough'.
(Note: this much path shortening might indicate a problem with the RRT* algorithm, but could also be attributed to numerical errors in floating point computations somewhere in the RRT* or the smoothing algorithm, or both)

Generating a .gif of the plan

We created a class which saves a whole plan (i.e. the whole process of placing each of the cubes in the new positions) to a single .gif file, allowing much easier debugging of the process.

To draw such a plan, two arguments are required:

1) The parts of the plan, i.e. a list of numpy arrays, each containing the configurations for that part of the plan.
2) A list of the positions of the cubes at each part of the plan

Below is an example of the usage of this class

```
   from gif_visualizer import VisualizeGif
   plans = []
   for cube_idx in range(num_cubes):
       grip_plan = np.load(f"grip_cube_{cube_idx+1}.npy")
       place_plan = np.load(f"place_cube_{cube_idx+1}.npy")
       plans.extend([grip_plan, place_plan])
   # generate list of current cube positions for each plan in
plans
   cube_positions = []
   cubes = init_cube_coords.copy()
   for cube_idx in range(6):
       cube_positions.append(cubes.copy())
       cubes.pop(cube_idx)
       cube_positions.append(cubes.copy())
       cubes.insert(cube_idx, target_cube_coords[cube_idx])

   gif_vis = VisualizeGif(ur_params, env, transform, bb)
   gif_vis.save_paths_to_gif(plans, cube_positions,
"cube_plan.gif")
```

The code for this is contained in gif_visualizer.py

We think this code may prove very useful for future iterations of this lab project.