Don’t Drop the Package - Programmer Guide

Guy Eisenberg, Avigail Ben-Eliyahu and Inbar Strull (partially contribution)

8-Sep-2022

Dependencies:

Follow the user guide.

Here we will go over the flow of the program, and explain more thoroughly about the logic.
An explanation about the car-like API is covered at the bottom of this document.

Flow of main():

- First of all, Make sure you are connected to the robot, otherwise it won't be able to run and it is needed to re-run the program after connecting.
- Running **main.py** will create an instance of **EnvironmentConfiguration** that will initiate the start of the program.
- Its creation creates and holds as a parameter an instance of **SolverViewerGUI(Ui_MainWindow)**, that we have changed to our needs. Most of the code there is taken from discopygal/tools/solver_viewer_main.py.
- During **SolverViewerGUI __inti__()**, it will call **run_default_scene()** that will simulate the flow of choosing the scene and the solver, and hitting **“Solve”** button, with our default parameters:
    - **DEFAULT_SCENE=”demo3_scene.json”**
    - **DEFAULT_SOLVER=”prm.py”**

  They can be found and be set differently in **solver_viewer_main.py** upper part.
- The flow for the **“Solve”** button will be discussed in a later section of this document.
- The flow for **PRM.load_scene()** will be discussed in a later section of this document.
- The **Discopygal GUI** (handled within **solver_view_main.py**) will appear on screen showing the scene and the routes that were calculated.
- If no path is found - it will notify the user, and it is needed to click on **“Solve”** again in order to try again. This result of not finding a path may happen and this is ok since the creation of the configuration graph is random (PRM), and points may not be able to form a connected graph. To increase the chance of finding a path, it is recommended to set the Number of Landmarks to a higher number.
- The default parameters for the prm can be found and changed at the top of **prm.py:**
    - **NUM_OF_LANDMARKS**
    - **NEAREST_NEIGHBOUR**
- Those parameters can be changed through GUI as well, but those will not be saved.

- Second step, in order to actually run the robot, hit the **“Play”** button on the top bar of the screen.
- Flow of **“Play”** will be discussed later in this document.

Flow for **“Solve”** button:

- Assumption: scene and solver are loaded. (when running main, we load them by default)
- Flow is identical to the discopygal flow of **“Solve”** but we added and altered few things which we will discuss here:
- The pressing of the button calls the method **SolverViewerGUI.solve()** in **solver_viewer_main.py**, which sets the worker (type **Worker**, defined in discopygal/gui/Worker.py) function to be **SolverViewerGUI.solve_thread()**, and when this is done, a finished signal is sent to call **SolverViewerGUI.solver_done()** (discussed next).
- During **SolverViewerGUI.solve_thread(),** solver.solve() is called, and in our case, solver is loaded to be **PRM(Solver),** so this is **PRM.solve().** Which returns to us both paths fields of SolverViewerGUI (both from type **PathCollection** defined at discopygal/solvers/__init__.py:)
    - **SolverViewerGUI.paths -** contains the shortest path
    - **SolverViewerGUI.paths_optimized** - contains the optimized path we calculated

- If **SolverViewerGUI.paths_optimized.paths** is not an empty dictionary, **SolverViewerGUI.paths_created** will be set to True. False otherwise.
- **PRM.solve()** logic will be discussed later in this document.

Flow for **SolverViewerGUI.solver_done()**

- This method is called when the worker started at **SolverViewerGUI.solve()** finished its run.
- That means, at this point we can assume that all the prm calculations are done, and we need to check **SolverViewerGUI.paths_created** content (True or False).
- If we do have a path, we then start optimizing it :
    - send it to **douglas_peuker(path)** located at **path_optimizations.py**
    - Send the result of douglas to **get_smooth_path()** located at **smooth_path.py**
    - Set the result to **SolverViewerGUI.smooth_path**
- Final thing is to call **finished_solving(SolverViewerGUI.paths_created, SolverViewerGUI.writer)** located in main.py to notify the user that the path is ready for the robot to run on.

Flow for **“Play”** button:

- Identical for the flow from discopygal, we only added functionality to run the robot after the animation is finished. Aka - this is the flow of **SolverViewerGUI.anim_finished():**
- if **SolverViewerGUI.paths_created,** call **parse_and_run(SolverViewerGUI.smooth_path)** located in **main.py**.
- **parse_and_run(SolverViewerGUI.smooth_path)** calls for **parse_path2** located in **path_optimizations.py** which returns a list of PathSection that holds everything the robot needs to know to run properly.
- Once we parsed the path into list of **PathSection,** we can send it to the robot, by calling **RobotControl.run_path(path_for_robot),** that basically go over this list and execute the needed commands to the robot, using our developed API for non-holonomic movement, located at robot_control.py

Logic for get_smooth_path():

- This function receives a **SolverViewerGUI** instance and assumes it contains in its attributes a robot, optimized path and solver of type **prm** from the project (that includes our modifications).
- It goes through the path points, and for each turn finds a suitable circle that smooths it without colliding with the scene’s obstacles. The output is an alternating list of segments and circles (Ker.Segment_2 and Ker.Circle_2 of CGAL) such that each circle is tangent to its adjacent segments at their midpoint, or at a point closer to the turn.
- For each turn, the function fits a circle that is tangent to both segments and goes through the midpoint that is closer to the turn out of the two segments. It does so by calling the function **get_circle()** that finds the circle center, radius and orientation using geometric considerations. It then calculates the arc starting point and end point using the function **get_arc_source_and_target().**
- Next, the arc is checked using the function **is_arc_valid_approximated()** of the collision detector (discussed below). If the arc is valid it will be added to the output list with the segment that leads to it. Otherwise, a binary search will be performed on the distance between the turn and the closer midpoint, to find a tangent arc with the largest radius that does not collide. From efficiency considerations, the binary search is bounded to 10 iterations. If it does not find a non-colliding arc within 10 iterations, it returns a circle of radius 0.0001.
- The function has a parameter **use_cd** that is set by default to “True”. In this setting, the above collision detection functionality is active. If it is set to “False”, all the circles that the function returns will be tangent to the closer midpoint, even if they collide with obstacles.

Logic for is_arc_valid_approximated():

- This function verifies that an arc does not collide with an obstacle (valid) by approximation: it does so by dividing the arc into equal angle sub-arcs, connects the starting point and end point of each sub-arc with a linear segment, and checks collision of the segment using **is_edge_valid()**.

- Division into sub-arcs is done in the following way: the length of each segment is calculated such that the longest distance between its midpoint and the sub-arc is the radius of the robot (which is approximated as a circle). All but the last segment are of equal length; these constraints determine the length of all segments while the last one is simply the remainder.

Logic for parse_path2():

- This function takes the output of **get_smooth_path()** as its input and returns a list of **PathSection** objects with all attributes filled. The function contains four loops performed successively.
- The first loop scans the smooth path and locates sequences of circular arcs with length-zero linear segments in between. It calculates the maximum tangent speed for each circular arc such that the load does not slip, then outputs a list of **PathSection** objects such that all circular arcs in a sequence have the same speed (the lowest of the calculated speeds of the circles in the sequence). Linear segments of nonzero length are given as initial speed the speed of the previous sequence and as final speed the speed of the next one.
- The second loop scans the linear segments in the **PathSection** list and verifies that none exceed their maximum allowed acceleration. If one does, it reduces its end speed (and by doing so, decreases the acceleration) and then cascades the change forward along the path.
- The third loop acts the same as the second for decelerating paths, going backwards along the path and reducing initial speeds in order to fix decelerations so they do not go below the minimum allowed value.
- The fourth loop calculates the maximum allowed midpoint speed of each nonzero linear segment such that the acceleration and deceleration from starting point to the end point of the segment do not exceed their bounds.

Flow of PRM.load_scene():

- The handling of the regular graph has remained exactly the same as in PRM in discopigal/solvers/prm.py. We added the new graph with the optimized weights to the new edges.
- We made a new class OptimizedGraph(nx.Graph) only for visibility difference. It has no actual purpose beside that.
- We will describe here the flow for creating the optimized graph:
- The idea is to take the original graph that was created, and to give a weight to all angles, such that higher angles will get lower weight.
- We do that by expanding each vertex of the graph into a bunch of vertices, according to its degree. For example - if the original point, p, in the original graph has 15 neighbors, we create 15 new vertices to the optimized graph, such that every point is of type

PointForOptimization, that holds inside it the original vertex, p, as field **point**, and the neighbor vertex it is connected to, as field **connected_to.**

- Each original vertex is being transformed into a small and condensed **mini_cluster**, with each point (of type PointForOptimization) inside it is connected to the other in the **mini_cluster**.
- This way, we can go over this mini_cluster and give the proper weight to that edge, based on the angle it creates.
- A corner case here is the start point and the end point which are present in the nearest neighbors, they are addressed as weight zero. So that we will be able to include them in the beginning and ending of the path.
- The rest of the edges are getting the same metric weight as in the original graph.

Flow of PRM.solve():

- First of all we check if there exists a path in the calculated graph. We use the **networkx** library here which contains all the implementations of the graph and has algorithms for checking if path exists (nx.algorithms.has_path(roadmap, start, end))
- If there is no path - we declare it and return a tuple of two empty **PathCollection**.
- The default **PathCollection** has field **paths**, which is initialized as an empty dictionary. We use that fact to verify that paths were created, in a later phase.
- Note that there is a small bug/feature that **PathCollection** has a mutable default argument, causing it to be shared across all instances of PathCollection. https://stackoverflow.com/questions/1132941/least-astonishment-and-the-mutable-default-argument
- As we need two different **PathCollection** instances, we worked around it by passing it a "reset ={}" as the paths argument.
- We let Michael know about this.
- We call the names of the two shortest paths: **tensor_path** and **tensor_path_optimized**
- The flow is exactly as it is in discopygal/solvers/prm.py, only we added the same logic for the optimized path - **tensor_path_optimized.**
- We go over all the points of **tensor_path_optimized** and create Path instance out of them.
- We use a temporary dictionary in order to make sure there is no duplicate points
- We add those paths to the PathCollection instances that we created and return them.

Logic of douglas_peuker():

- We had to implement this algorithm ourselves in order to make it consider collision detection. The constraint is that we can't just "erase" points because this can cause collisions from the new segments that were created.
- Original algorithm is done recursively, by creating a segment between the leftmost and the rightmost points, finding the middle point that is the farest from them ,and if its distance is smaller than epsilon, we eliminate all the points in between. If not, we are splitting the list into 2 parts, and going recursively on each one.

- In order to consider collision detection, each time we are getting a list of only 2 points, we perform collision detection of this segment. If it is illegal, we are rolling back and performing it without the leftmost\rightmost point.

About robot_control.py:
The basic functionality we created based on the original API of the robot is all implemented in this script.

The original API that we use is ep_chassis.drive_wheels, which gets as input the rpms for each wheel, and the time that it should run in those rpms.
Important note: all the speeds and the radiuses we use are in m/s and in meters.
Be careful not to give high numbers

def move_straight_exact(self, distance, speed=0.5):

- Moves the robot for the given distance in the given speed.
- **IMPORTANT NOTE**: speed is in m/s. Be very careful not to give it high numbers, as the robot **WILL TRY** to run them. 0.5 m/s is fairly ok speed, 0.3m/s is quite slow. Anything beyond 1.5 m/s is quite fast and must be dealt with caution.
- This function supports giving it negative distances as well. The robot will just drive that distance in reverse direction.

def glide_smoothly(self, start_speed, end_speed, distance, func:Any = lambda x:x, oposite_func:Any = lambda x:x ,should_stop=False, proportion=0.8):

- Simulating smooth acceleration/deceleration, in a general way, while moving from start_speed to end_speed within the given distance.
- Func and opposite_func must be opposites. E.g: func = math.exp, and opposite_func=math.log
- We found that linear function works best for out needs
- This method will use move_straight_exact for every section it is calculating
- The proportion argument is not used or implemented - the idea was to make the robot move in the glide motion for the proportion of the distance that was given, and the rest will drive at constant end_speed.

def move_circle_Husband(self, speed=0.3, R=0.6, theta=math.pi/2, should_stop=False, circle_orient=None):

- This function will make the robot drive at the given radius in the given speed (in m/s)
- Circle_orient should be the orientation of the circle (aka either Ker.CLOCKWISE or Ker.COUNTERCLOCKWISE)
- The speed for the both set of wheels is calculated in calc_wanted_rpms and is based on a physics equation that we wrote in the presentation.
- The average of the speeds of the sets of wheels (the outer and the inner) will be the wanted speed that was requested.