APC 524 Final Project: Path Finding Algorithm Solver

Release 1.0

K. Andrade, R. Laitner, L.H. Lam, S. Sarwar, M. Zhang

Dec 16, 2021

CONTENTS:

CHAPTER

ONE

ABOUT

Within this project, we hope to be able to build a software package that enables users to explore the tradeoffs of utilizing different path-planning algorithms with different vehicles in both static and dynamic environments. Currently we will be utilizing A*, Djikstra's, and RRTs as each of these algorithms have their own pros and cons.

CHAPTER

TWO

OVERVIEW OF OUR SYSTEM'S FUNCTIONALITY

The system will include the following options:

- Type of Path-Finding Algorithm
- Type of Obstacles
- Dynamic/Static Obstacles
- Type of Vehicle (UAV or Car)

CHAPTER

THREE

NOTES

(To be completed. . .)

CHAPTER

FOUR

CONTENTS

4.1 Algorithm Factory

class algo_factory.**py**

4.1.1 Generates a PathingAlgorithm class to be used by higher level code.

Functions: Returns a PathingAlgorithm object of a specified type:

```
from typing import Type, Dict

from pathingSim.pathing_algorithm import PathingAlgorithm
from pathingSim.a_star import AStar
from pathingSim.RRT import RRT
```

```
def algo_factory(chosen_algo: str) -> PathingAlgorithm
```

Returns an instantiated implementation of a PathingAlgorithm object.

Parameters

chosen_aglo: str A string that specifies the desired pathing algorithm to use.

The possible algorithms currently are:

- A*: The A* search algorithm
- RRT: The rapidly-exploring random tree algorithm

Returns

PathingAlgorithm Implementation of pathing algorithm specified by input parameter

Raises

NotImplementedError Raised when a pathing algorithm is chosen that does not exist in the factor and cannot be provided

```
possible: Dict[str, Type[PathingAlgorithm]]
possible = {
        "A*": AStar,
        "RRT": RRT
}

try:
        return possible[chosen_algo]()
except KeyError:
        raise NotImplementedError(f"The specified algorithm {chosen_algo}" +
                                  " is invalid.")
```

4.2 Project Background

4.2.1 Introduction

In recent decades, there has been a tremendous proliferation of autonomous vehicles such as self-driving cars and self-navigating drones and robots. Although there are many branches of science at work that enable these complex systems to continuously improve, one important component is the path-planning algorithm. Path-planning algorithms are critical when it comes to deploying autonomous vehicles because they enable the vehicles to be able to go from one point to another while navigating around obstacles. Such autonomous vehicles are able to leverage path-planning algorithms by collecting information through cameras and use the gathered information in clever ways. Generally, it is desired for the robot to reach a specific goal from wherever its origin point may be within a 2D or 3D space. In a realistic setting, there are often obstacles these vehicles must be able to circumvent while on their way to the destination. The existence of such obstacles makes it difficult for vehicles to go from their origin to their desired destination, thus requiring careful selection of an algorithm. Ideally, the algorithm is able to carve out a path for the vehicle to follow, spanning from its current location to its desired destination. In real systems, resources come at a premium. Within this project, we hope to be able to build a software package that enables users to explore the trade-offs of utilizing different path-planning algorithms with different vehicles in both static and dynamic environments. For now, we will utilize A* and RRTs given that each of these algorithms have their own pros and cons.

4.2.2 Overview of Functionality

The system will include the following options:

- Type of Path-Finding Algorithm
- Type of Obstacles
- Dynamic and Static Obstacles
- Type of Vehicle

4.2.3 Path-Finding Algorithms

An important aspect of this project is to provide the user with options on the kind of path-finding algorithm used for the given autonomous vehicle and set of obstacles. The following path-finding algorithms will be implemented as options to the user.

Rapidly-Exploring Random Tree (RRT)

The RRT algorithm is centered around randomly sampling points. Points are randomly sampled across the map, where if they do not collide with an obstacle, then the vehicle moves towards that randomly sampled point by a determined amount. This process is done iteratively until the vehicle reaches the destination, if one exists. If a path to its destination exists, under mild assumptions, the RRT algorithm is guaranteed to find a path to it, no matter how long it takes. On the other hand, if no such path exists, the algorithm will run forever until it is forced to be terminated. One of the advantages of using this method is that it is able to operate in a continuous domain and does not require explicit prior knowledge of the environment for it to be able to navigate around the obstacles as long as some collision condition can be defined.

A* Algorithm

The A* algorithm is a dynamic programming graph traversal algorithm that can return the optimal path between two points in a graph with light assumptions. These assumptions are mainly made upon the choice of heuristic used to estimate the value of each location or node in a graph. In a path optimization context, A* is generally implemented by first discretizing the space in which pathing must occur and then assigning costs to each discretized point. The optimal path is then found by minimizing the overall cost between the desired start and end point. The benefit of A* search is that the optimality can be guaranteed if used correctly; however, the choice of heuristic can greatly impact both the performance, even leading to some non-optimal trajectories, and the time complexity of the algorithm, impact the run time from exponential to polynomial. Another downside is A* memory complexity is linear in the size of the search space which means in practice, certain iterative approximations are made to the actual algorithm for large search spaces.

Dijkstra's Algorithm

The D* algorithm is an incremental heuristic graph search algorithm that is used to search dynamic and time-variant graphs. Specifically in this context, D* Lite is used to solve goal-directed navigation in unknown environments by repeatedly determining the shortest paths between the position of the robot and the goal as the edge costs of a graph change while the robot moves towards the goal. This algorithm, like A*, also relies on a discretized search space and can be thought of as an extension of A* with memory. This memory allows the algorithm to learn dynamic values of the individual location costs as time progresses. Because the algorithm is based upon an iterative version of A*, it can be shown for some certain choices of heuristic to be more computationally efficient than A* and better suited for dynamic environments while not being significantly more complicated to implement.

Obstacles

An important aspect of this project is to provide the user with options on the kind of shapes that are used on the description of the physical space that will be traversed.

- **Quadrangle**: A quadrangular obstacle can be represented with 4 coordinate values for each of the corners of the shape.
- **Circle**: A circular obstacle can be represented by a pair of coordinates that represents the center of the circle and an accompanying radius value.
- **Triangle**: A triangular obstacle can be represented with a pair of coordinates that represents the centroid and then three other pairs of coordinates that each represent a vertex.

For our project, we are looking to give the user the option to provide both static and dynamic obstacles for which to optimize a path around.

Static Obstacles

In the case of static obstacles, a standard algorithm is run for finding a path. This is the typical case where we have a space that contains obstacles which remain in the same location as for any given time (t).

Dynamic Obstacles

The more complex case involves using dynamic obstacles which would be considered obstacles that are actively moving over time. These would have to be specified with functions that are directly dependent on time, as this will allow us to track them while the path-finding algorithm is running. These trajectory functions should also be cyclical, so that they do not exit the space where the route is being calculated.

Vehicles

The type of vehicle is crucial for understanding the size of the vehicle when finding the trajectory. Since we are not considering the 3-D case, and are assuming identical movement possible for each vehicle type, this is simply for the purposes of determining the size of the vehicle for the appropriate path planning. The different vehicle kinds we will allow are the following:

- **Car**
- **Unmanned Aerial Vehicle (UAV)**

4.3 Installing the Library

This library can be installed with `pip install -e` from the top directory if it is already cloned.

You can also clone the repository using `git clone https://github.com/rlaitner/APC-Final-Project`

4.4 Setting up the Interface

4.5 Algorithms

4.5.1 RRT

4.5.2 A*

4.6 Obstacles

4.7 Vehicles

4.7.1 Car

4.7.2 Tricycle

4.7.3 UAV

4.8 Pathing Simulator

4.9 Visualizer

4.10 Using Sphinx for Documentation

4.10.1 Overview

Documentation for this project can be built using Sphinx.

The files are provided in the `main/source` directory and can build both html documentation and PDF files (through LaTeX).

Sphinx-autodoc has provided the Makefile and make.bat in the `main` repository. Use the follow commands when located in `main` for quick build:

- `make html`
- `make latexpdf`

To manually build the documentation, use `sphinx-build`.

See sphinx-build for more detailed explanation.

4.10.2 Other Resources

4.11 Indices and tables

- genindex
- modindex
- search

INDEX

A