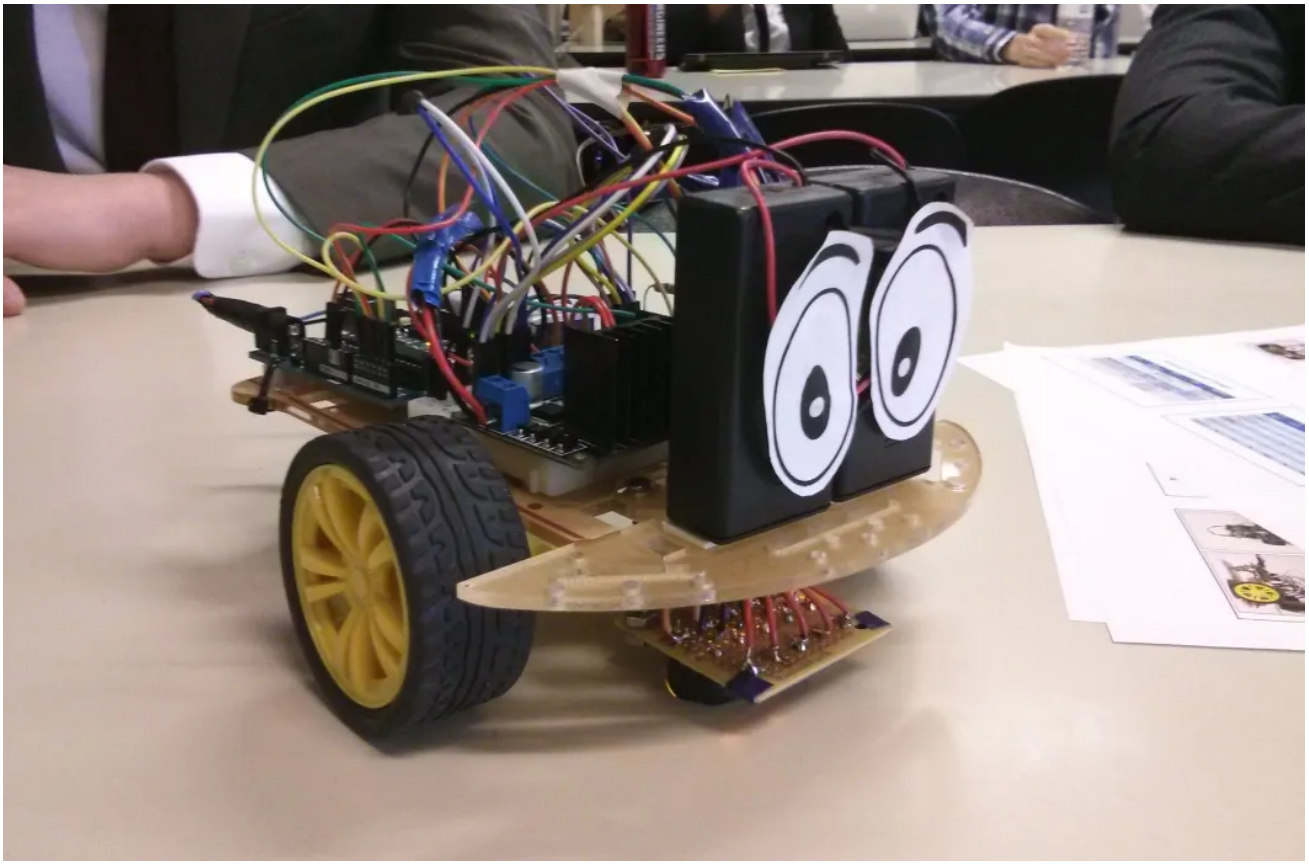


The eyes help navigate

Robotic Path Planning: RRT and RRT*

Exploring the optimized version of a orthodox path planning algorithm

Tim Chinenov · Follow

6 min read · Feb 14, 2019

The robotic path planning problem is a classic. A robot, with certain dimensions, is attempting to navigate between point A and point B while avoiding the set of all obstacles, C_{obs}. The robot is able to move through the open area, C_{free}, which is not necessarily discretized.

Unlike most path planning algorithms, there are two main challenges that are

imposed by this problem. First, the robot does not have existing nodes to travel between. If one considers the famous Dijkstra's algorithm, that problem includes a graph. With a free continuous space, a graph of edges and vertices needs to be created. This fosters questions on the characteristics a graph should have for such a problem. Secondly, one must determine how a shortest path will be determined. The problem of building a graph and navigating are not necessarily solved by the same algorithm.

There exist numerous path planning algorithms that address the navigation problem. Rapidly-exploring random trees (RRT) is a common option that both creates a graph and finds a path. The path will not necessarily be optimal. RRT*, popularized by Dr. Karaman and Dr. Frazzoli, is an optimized modified algorithm that aims to achieve a shortest path, whether by distance or other metrics. Both are implemented in python and observed in this article. To demonstrate the idea, the algorithms will be implemented in a 2D space with bounds. However, both algorithms can be built into any real continuous dimensional space. A video demonstration of these algorithms, as well as additional algorithms, is at the end of this article.

RRT

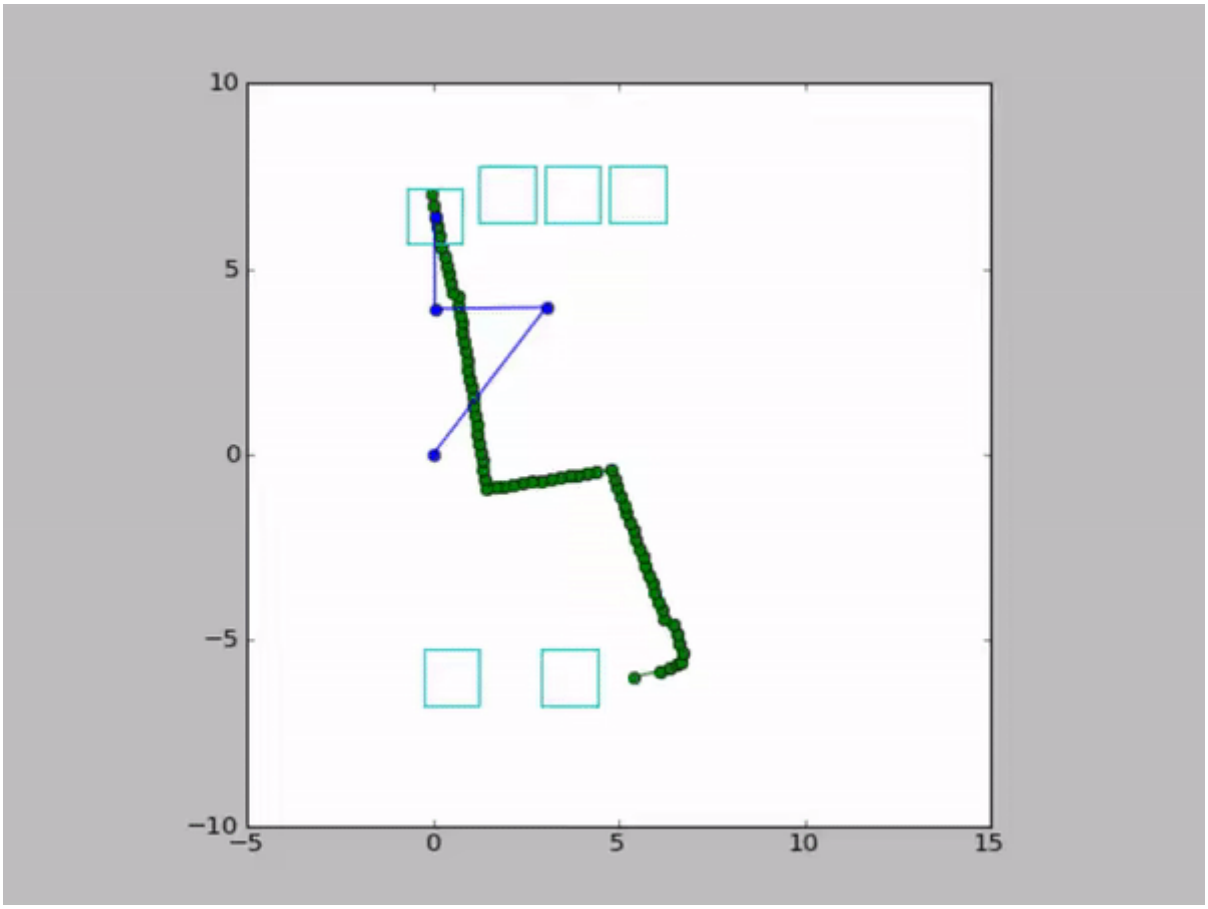
The premise of RRT is actually quite straight forward. Points are randomly generated and connected to the closest available node. Each time a vertex is created, a check must be made that the vertex lies outside of an obstacle. Furthermore, chaining the vertex to its closest neighbor must also avoid obstacles. The algorithm ends when a node is generated within the goal region, or a limit is hit.

RRT Pseudo Code

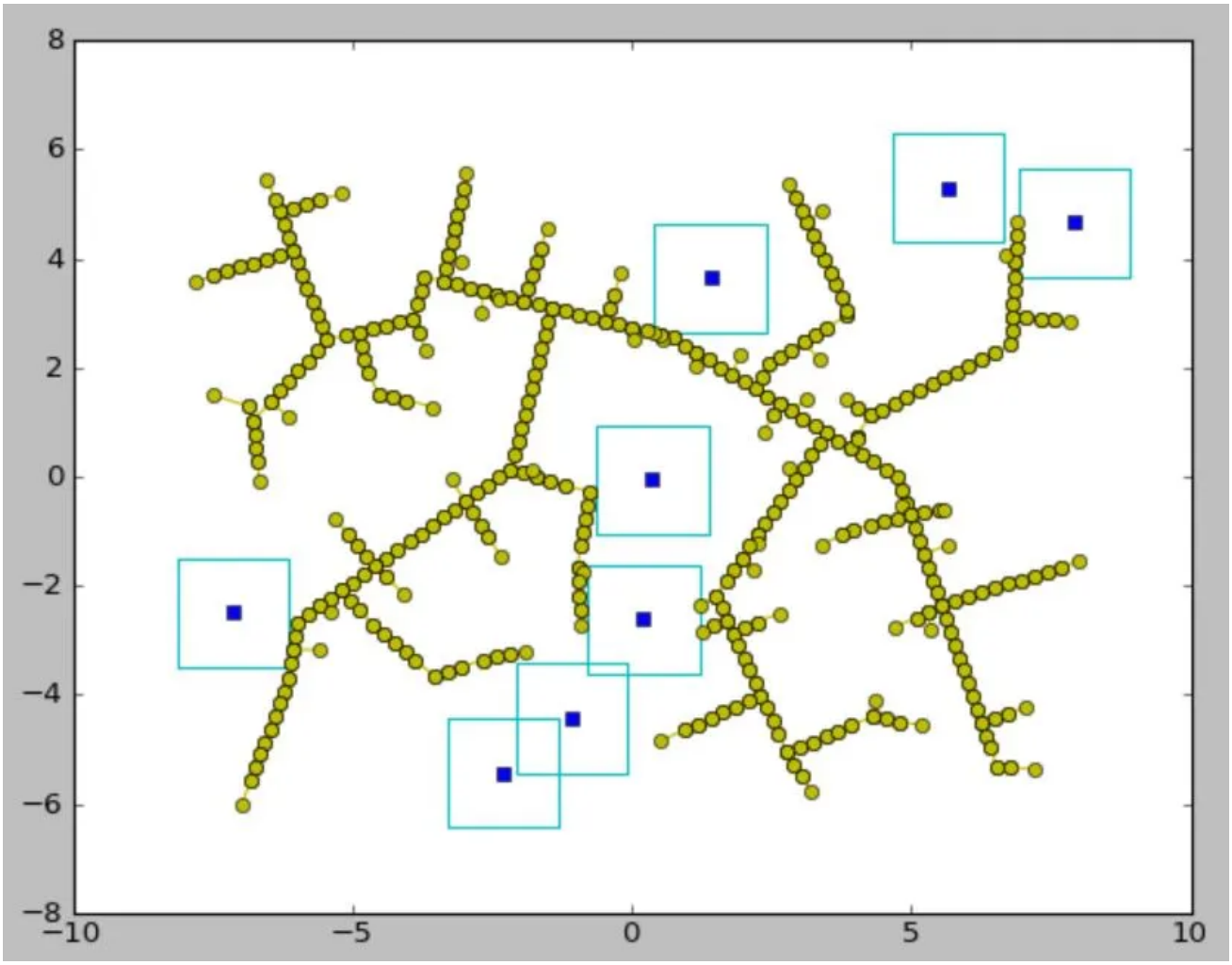
```
Qgoal //region that identifies success
Counter = 0 //keeps track of iterations
lim = n //number of iterations algorithm should run for
G(V,E) //Graph containing edges and vertices, initialized as empty
While counter < lim:
    Xnew  = RandomPosition()
    if IsInObstacle(Xnew) == True:
        continue
    Xnearest = Nearest(G(V,E),Xnew) //find nearest vertex
    Link = Chain(Xnew,Xnearest)
    G.append(Link)
    if Xnew in Qgoal:
```

```
        Return G
Return G
```

The method of determining a random position is a design decision. Simple methods, such as using built in random number generators can be used. For basic applications, such approaches suffice. Yet, developers should understand that random number generators are not truly random and do contain a degree of bias. An entire topic of study, known as Sampling Theory (La Valle, 195) , exists for the curious.



An Example of an RRT path



The graph generated from RRT algorithm

Additionally, the method of chaining the randomly generated vertex is customizable. One method involves calculating the vector that forms the shortest distance between the new vertex and the closest edge. At the point of intersection, a new node is added to the edge and connected to the randomly generated vertex. Alternatively, the vertex can be attached to the closest node by chaining a link of discretized nodes to it. This method requires less computation and is simpler to implement, but requires more points to be stored.

RRT produces very cubic graphs. This is expected as nodes are attached to their nearest neighbor. The structural nature of these graphs hinders the probability of finding an optimal path. Instead of taking the hypotenuse between two points, the two legs of a triangle are navigated across. This is evidently a longer distance. The cubic nature and irregular paths generated by RRT are addressed by RRT*.

The benefit of the algorithm is its speed and implementation. Compared to other

path planning algorithms, RRT is fairly quick. The costliest part of the algorithm is finding its closest neighbor as this process grows depending on the number of vertices that have been generated.

RRT*

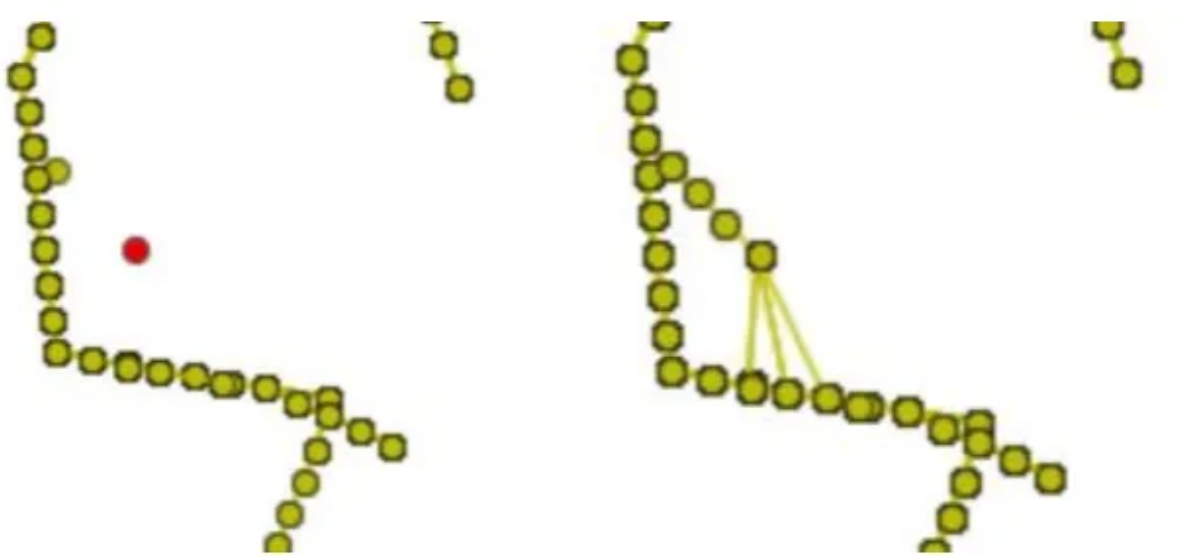
RRT* is an optimized version of RRT. When the number of nodes approaches infinity, the RRT* algorithm will deliver the shortest possible path to the goal. While realistically unfeasible, this statement suggests that the algorithm does work to develop a shortest path. The basic principle of RRT* is the same as RRT, but two key additions to the algorithm result in significantly different results.

Fan shaped structure

First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the `cost()` of the vertex. After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper `cost()` than the proximal node is

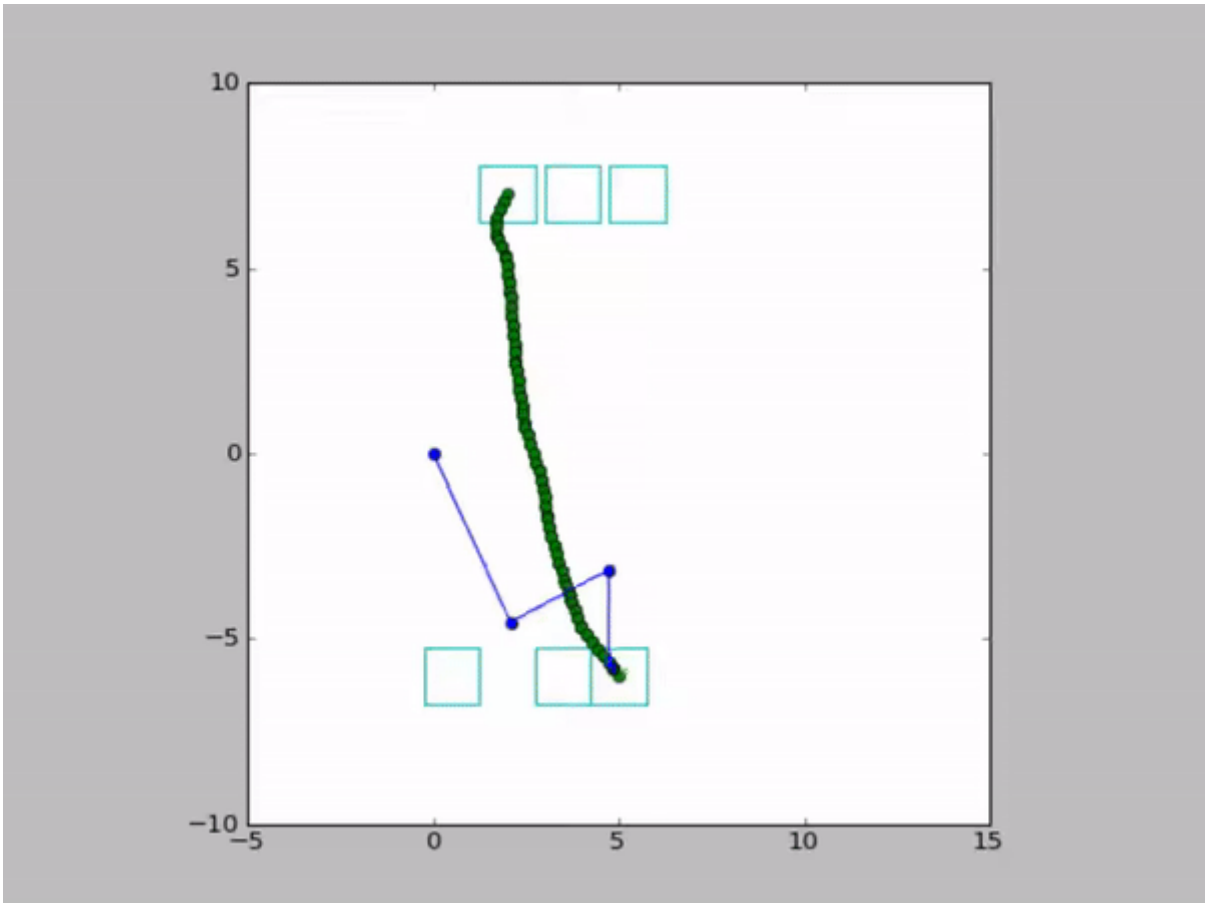
found, the cheaper node replaces the proximal node. The effect of this feature can be seen with the addition of fan shaped twigs in the tree structure. The cubic structure of RRT is eliminated.

The second difference RRT* adds is the rewiring of the tree. After a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path more smooth.

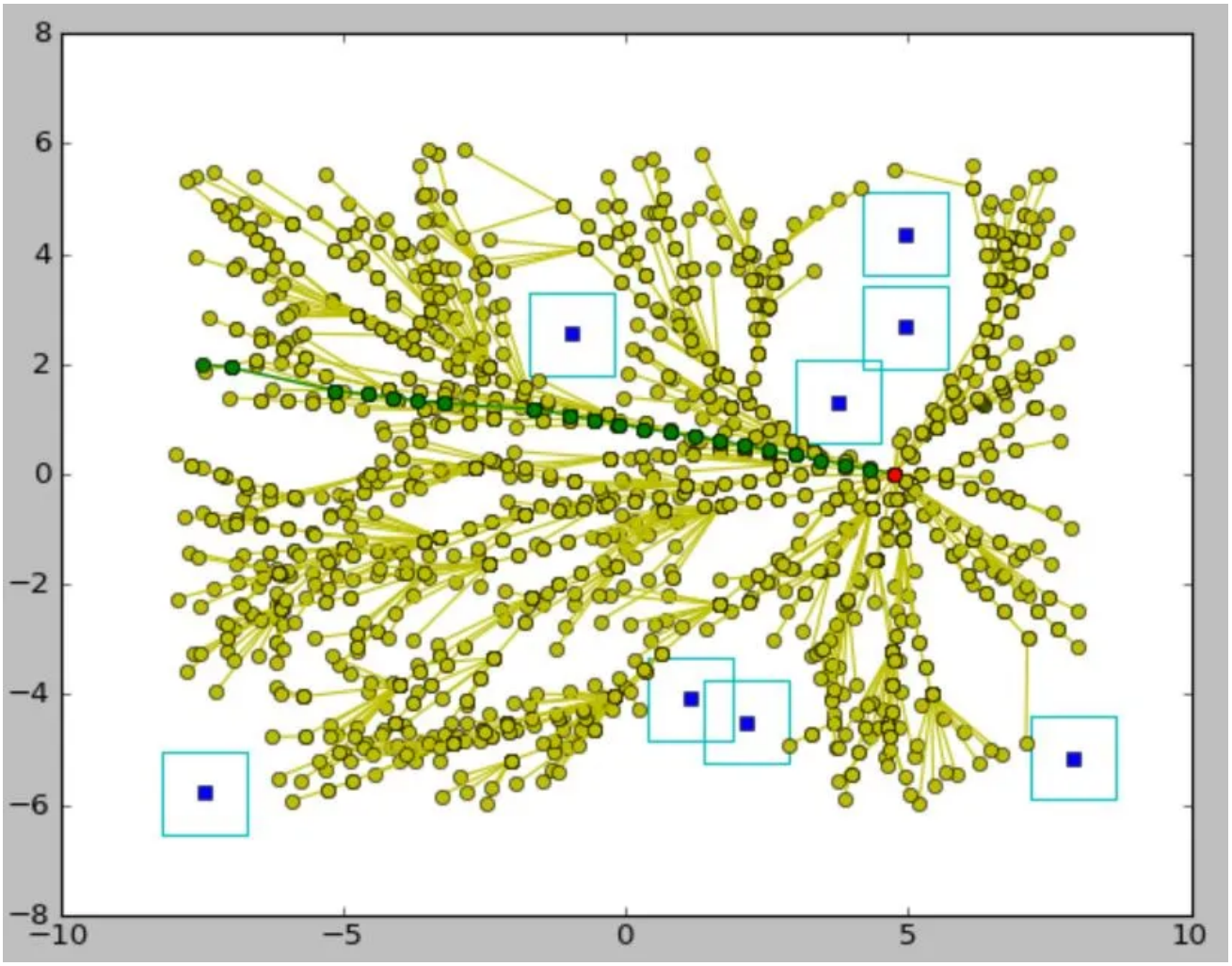


Example of rewiring of the graph

RRT* creates incredibly straight paths. Additionally, its graphs are characteristically different from those of RRT. For finding an optimal path, especially in a dense field of obstacles, the structure of RRT* is incredibly useful. The graph vines around objects, finding shorter paths in comparison to RRT. If the destination were to change, the original graph can still be used as it represents the quickest path to most locations in the region.



Path generated by RRT*



Two examples of RRT*

While not apparent from the images, RRT* suffers from a reduction in performance. Due to examining neighboring nodes and rewiring the graph, my implementation of RRT* took nearly eight times longer to complete a single path on average than the default version.

The majority of computing effort came from obstacle avoidance. Obstacle avoidance must be checked when a node is placed, when a node is connected to its neighbor, and for each node that is to be rewired. This is a considerable number of checks to make. Yet, one cannot deny the success of the generated paths.

A final note to make is on the simplifications made in this article. I aimed to filter out much of the mathematics and details in this article, while retaining the key points. The attempt at brevity should help those who are purely interested in path planning algorithms. I greatly encourage for the curious to read more about these

two algorithms and other path planning algorithms from the original source.

RRT* Pseudo Code

```
Rad = r
G(V,E) //Graph containing edges and vertices
For itr in range(0…n)
    Xnew = RandomPosition()
    If Obstacle(Xnew) == True, try again
    Xnearest = Nearest(G(V,E),Xnew)
    Cost(Xnew) = Distance(Xnew,Xnearest)
    Xbest,Xneighbors = findNeighbors(G(V,E),Xnew,Rad)
    Link = Chain(Xnew,Xbest)
    For x' in Xneighbors
        If Cost(Xnew) + Distance(Xnew,x') < Cost(x')
            Cost(x') = Cost(Xnew)+Distance(Xnew,x')
            Parent(x') = Xnew
            G += {Xnew,x'}
    G += Link
Return G
```

I discuss an alternative method of path planning, PRM and PRM here!*

Robotics Computer Science Programming Python Algorithms

Follow

Written by Tim Chinenov

821 Followers

A SpaceX software engineer. Im an equal opportunity critic that writes about tech and policy. instagram: @classy.tim.writes

More from Tim Chinenov

Tim Chinenov

The Realities of an Engineering Degree

An attempt to dissolve false preconcieved notions relating to the field.

Tim Chinenov

Using Image Processing to Detect Text

A demonstration of how easy it is to find words and letters in an image

6 min read · Jan 24, 2019

-- 1

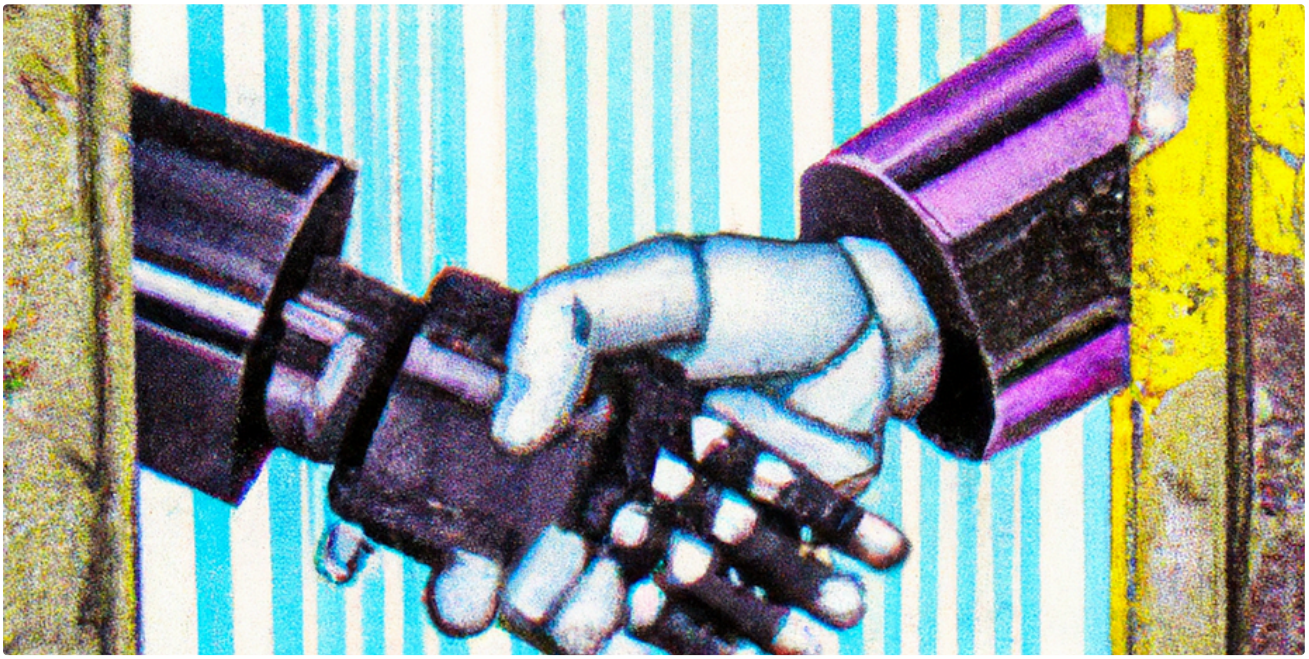
Tim Chinenov

Robotic Path Planning: PRM & PRM*

Building a infrastructure for path planning

· 7 min read · Feb 19, 2019

-- 1



Tim Chinenov

How to Mitigate AI Art Theft

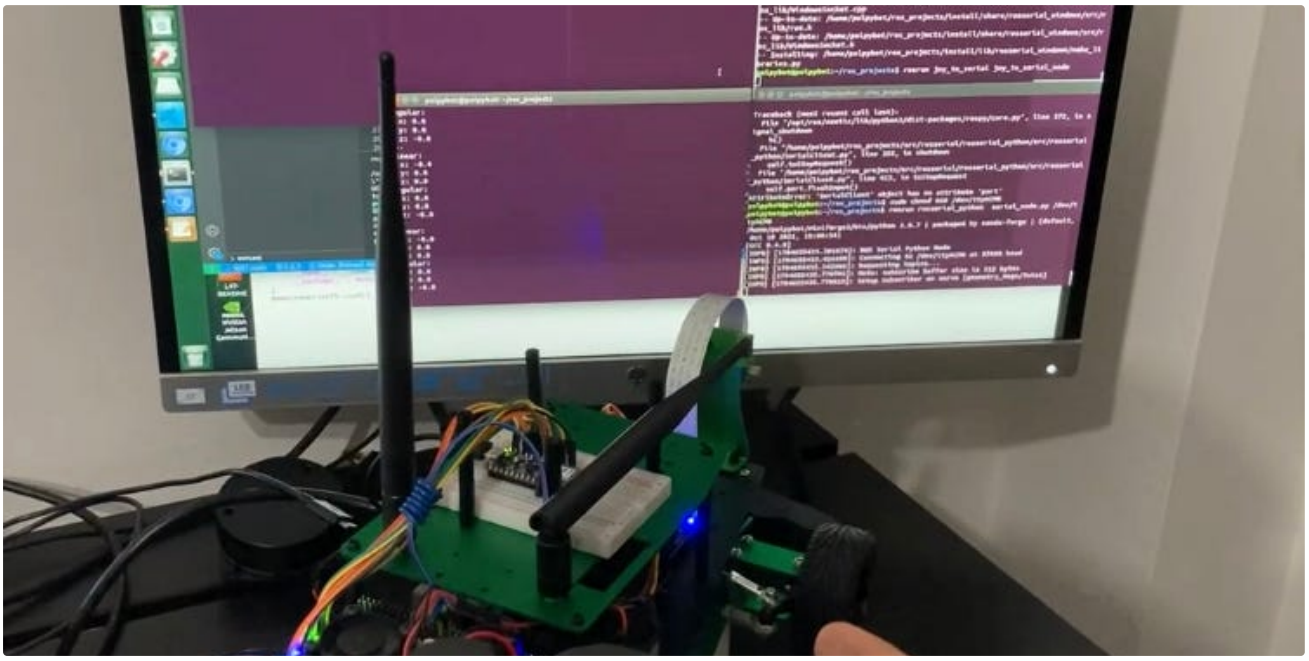
Instead of abolishment, let's consider new solutions for new technologies

· 7 min read · Jan 6, 2023

-- 1

See all from Tim Chinenov

Recommended from Medium



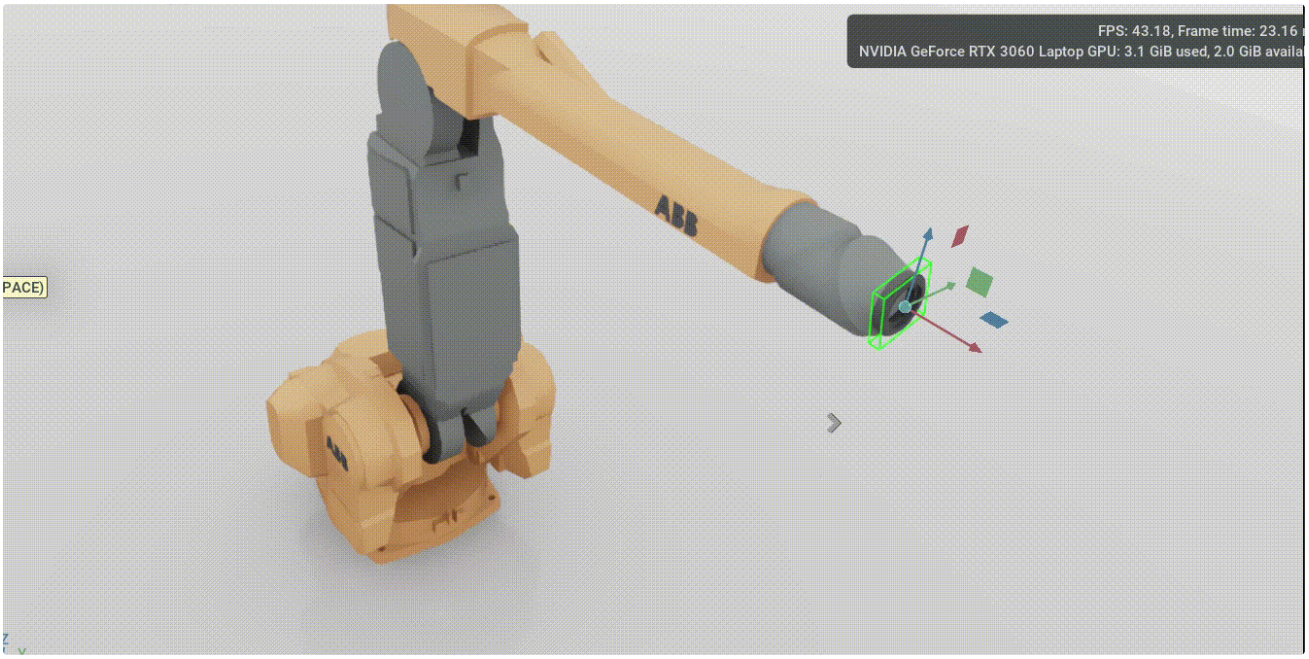
Stefano C

A self driving car project. Pt 2.

Serial communication between Jetson and Arduino.

5 min read · Jan 10, 2024

--



Ake Zavala JM

A Practical Approach to the NVIDIA Omniverse Robotics Simulation Toolkit, Part 1:

Creating a robotic arm via the Python OpenUSD API

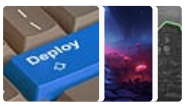
8 min read · Nov 29, 2023

--

Lists

Coding & Development

11 stories · 615 saves



Predictive Modeling w/ Python

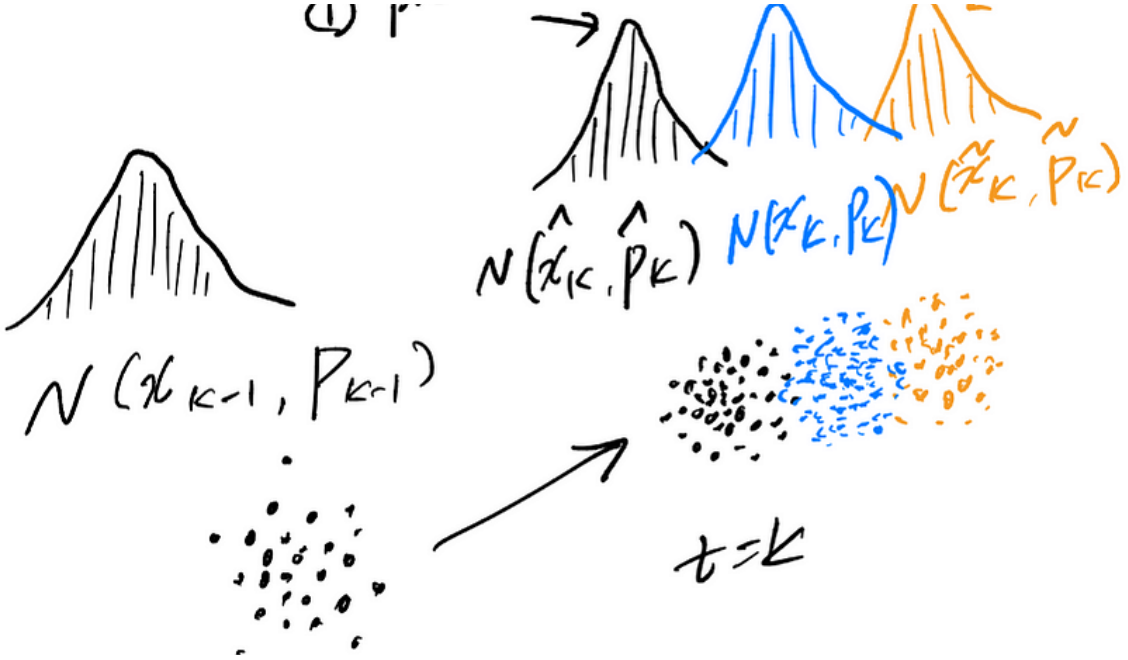
20 stories · 1197 saves

Practical Guides to Machine Learning

10 stories · 1447 saves

ChatGPT

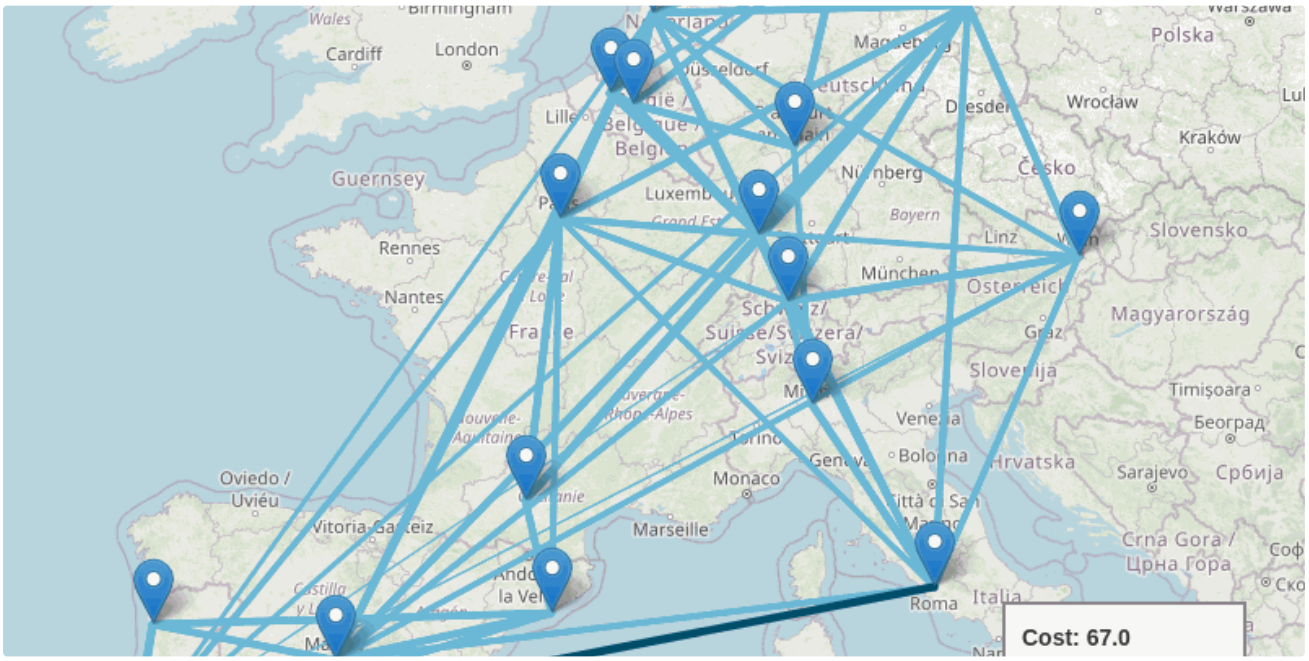
21 stories · 637 saves



Hahnsang Kim

Use Case: A Linear Battery Model using The Kalman Filter

9 min read · May 9, 2024

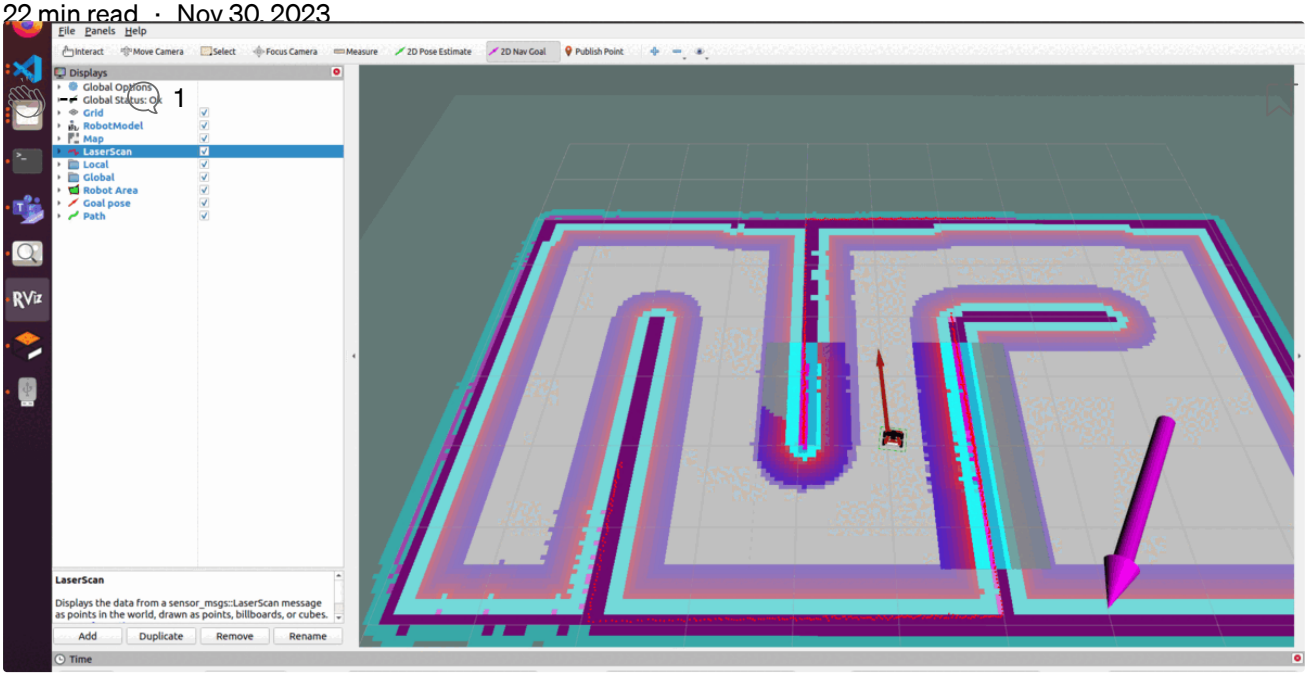


Cristina Fernandes in Data And Beyond

How to solve a Routing Problem with a Genetic Algorithm: A Practical Guide

In the fast-paced world of logistics and transportation, the efficiency of routing from one city to another is a cornerstone of economic...

22 min read · Nov 30, 2023

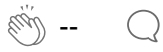


Mansoor Alam

Path Planning Mobile Robots ROS

Path planning is a crucial aspect of robotics and autonomous systems, where the goal is to find a suitable path for a robot or vehicle to...

4 min read · Jan 21, 2024



Ignitarium

3D LiDAR SLAM — Scan Matching Explained

Author: Srichitra S

9 min read · Mar 5, 2024