Using RRTs to Navigate Gibson Environments

Paul McKerley
George Mason University
pmckerle@masonlive.gmu.edu

Abstract

Produce efficient maps to navigate two-dimensional floorplans extracted from three-dimensional Gibson Environment buildings.

1. Introduction

The goal of this project was to produce road maps for Gibson Environment buildings that would allow robots to navigate from one arbitrary point in the map to another. This navigation should find reasonable straight-line paths. My approach was to use Rapidly-Exploring Random Trees, developed by Lavalle and Kuffner [1], which provided maps in a matter of seconds. A* search can traverse these maps, and the results paths refined to be straighter. Many samples paths can be found and refined, with the resulting paths added to the map. These refined paths give highly efficient paths between rooms.

2. Approach

There are several steps requirement to produce a roadmap of a Gibson Environment structure.

2.1. Extract Two-Dimensional Floorplan

The Gibson buildings come as three-dimensional graphs from the GibsonEnv database [4]. To extract two-dimensional floor plans I used a Python module called meshcut.py [2] (kindly provided by Yimeng Li.) Figure 1 shows a 3D representation of the Allensville apartment. When a 2D images is extracted at a height of 0.5 meters the result is found in 2. This picture is an image saved from the Python module matplotlib. But it can be converted to an OpenCV-compatible numpy array and mantipulated in memory or saved to file.

2.2. Preparing Floorplan for Geometry Checks

The size of the buildings are not large compared to the size of robot we would expect to operate in them. There also need to be many checks of lines for collisions with solid objects (walls and other items in the rooms.) An efficient way to do these checks is by rasterizing the maps and using Bresenham's line-drawing algorithm to find obstructions on the image. Solid areas are represented as black pixels, and free areas as white pixels. To do this I used use opencv.floodFill() starting at a point in the free space to fill the freespace with white pixels.

Figure 1. 3D Image of Allensville

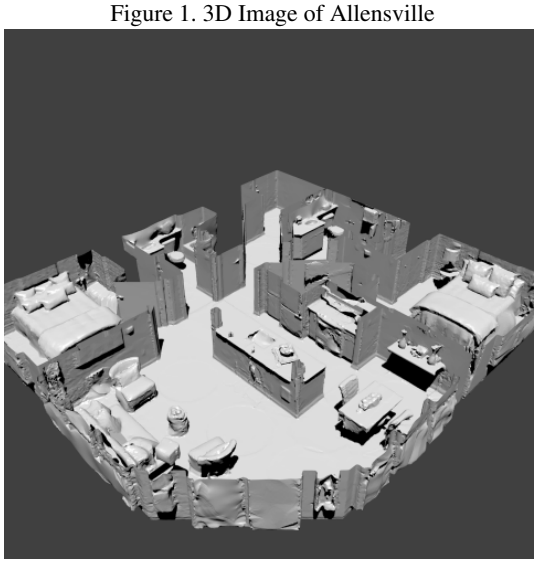


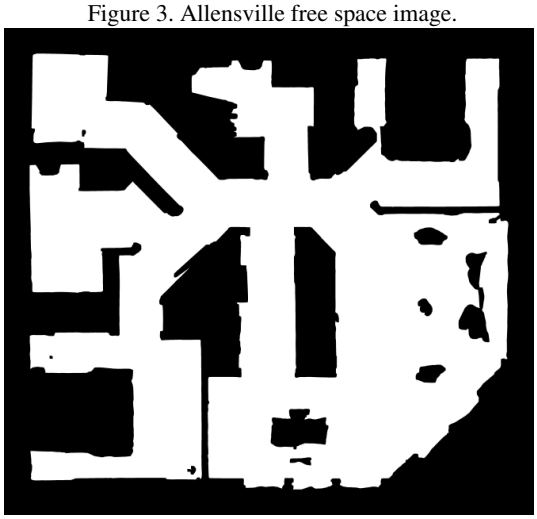
Figure 2. 2D Image of Allensville at 0.5 m

Even though the dataset documentation say most of the Gibson maps have the holes filled, it is still the case that the 2D cross-sections have visible holes in the walls. This makes the floodFill() operation fail. These gaps can be filled "manually", by examining the cross-sections and adding appropriate lines, but this is tedious. So a function finds the end-points to all lines, and connects them to the nearest endpoint of another line. This can be done efficiently with a numpy representations of the line arrays. Not all maps work, but enough to have a decent set to work with.

Finally, There are still small gaps between objects that are evidently too small for a robot to fit through, but which the RRT lines would traverse. To get rid of these, opencv.erode is applied to the image. This also has the effect of providing some space from the walls that represents the thickness of a robot. Frankly, I used only a few pixels of erosion, so the spacing effect is not realistically large. However, more erosion could certainly be used to get the correct effect.

Figure 3 shows a rasterized free space map for Allensville after gaps have been filled and erosion applied.

Figure 3. Allensville free space image.



In summary, the algorithm to convert a set of numpy arrays, each representing a line in the map, to the rasterized free space image, is:

1. Find minimum and maximum x and y positions of lines.
2. Pad lines out with configurable quanity (typically 0.5 m in examples.)
3. Draw lines with opencv.polylines() on free image initialized to black.
4. Calculate lines to fill gaps and draw free image.
5. Floodfill image with white starting at a fixed point in image (this should be made configurable.)
6. Erode image with a 5x5 mask and a configurable number of times (5 by default.)

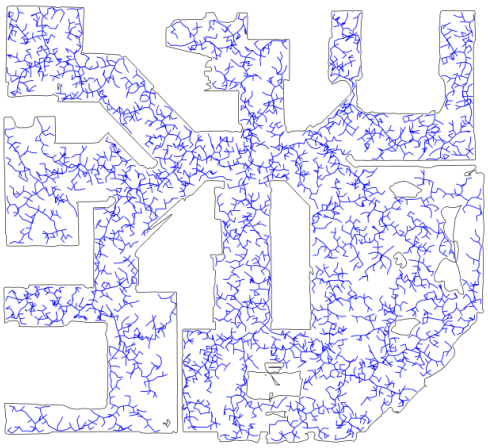
2.3. Creating the RRT

The algorithm to produce the RRT is fairly simple:

1. Find random starting point in free space.
2. Perform N times.
   (a) Find random point s in anywhere the image.
   (b) Find existing point t in tree closest to s.
   (c) If line of length d can be drawn from s to t without hitting an obstruction (or, optionally, another line), then draw it, and add end of line as a new point in the RRT.

The check for an obstruction is made by using Bresenham line-drawing to trace the line on the free-space image. If it hits a black pixel then the line is not used. If the option not to cross tree lines is specified, then a copy of the free space image is made, and each successful new edge is drawn in black on it. This prevents future edges from crossing it. A sample RRT is shown in 4. The number of nodes to draw in the tree is configurable. For Allensville 5000 is sufficient to fill the graph. Larger spaces require more nodes.

Figure 4. Allensville RRT



Two programs can be used to produce an RRT: make_rrt.py, and rrtgui.py. Both use the library file rrt.py. make_rrt.py creates RRTs with several command line options and saves the floormap image, with the RRT, the free image, and the parameters to files. rrtgui.py is a demonstration program that displays an animation of the RRT as rrt.py creates it. I did not get to the point of saving the products of rrtgui.py to file.

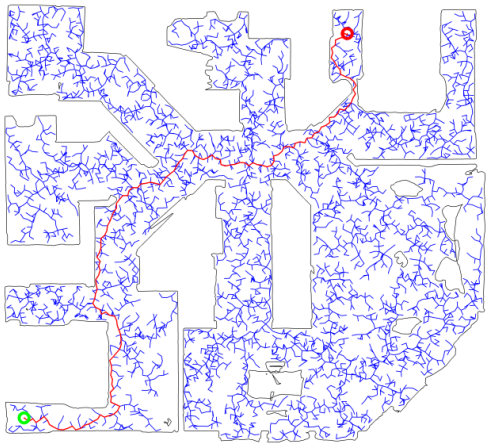
Before evaluating the usefulness of the RRT, we'll examine the method to find paths between arbitrary points.

2.4. Finding a Path Using A* Search

Given a starting point, and an ending point, both in real world coordinate measured in meters, A* is used to find a path on the tree between them. This algorithm is basically a direct implementation of the one found in Russell and Norvig [3]. The only difficulty was finding a heapq with replacement. I didn't find one, but subclassed the standard Python module as AStarHeap to work with special objects that have a deleted flag to support replacement. The AStarHeap has a dictionary of nodes to track when a node value already exists. This allows it to implement a method to see if a node already exists with a worse cost.

The algorithm uses the free image to find the closest node to each of the start and end points, and then find the path. This is very fast. A sample path is shown in 5. The path starts at the green circle and ends at the red one. This path looks reasonable, but because it has to follow the meandering RRT path, it doesn't really look as good as it could be.

Figure 5. Allensville Sample Path 1



There are other inefficiencies besides the meandering RRT path. If we look carefully at the circled area in 6 we see that the leaves of the RRT do not meet up behind the object (a sofa.) This is because an RRT really is a tree, not a graph, so the edges never connect to existing nodes. This leads to a path from one side of the sofa to the other that goes around in front of is, as shown in 7.

2.5. Refining the Path

In order to make the path more efficient we do the following:

1. Create a new graph with just the nodes and edges of the path.

2. Repeatedly add random edges between pairs of unconnected nodes.

Figure 6. Allensville RRT Gap

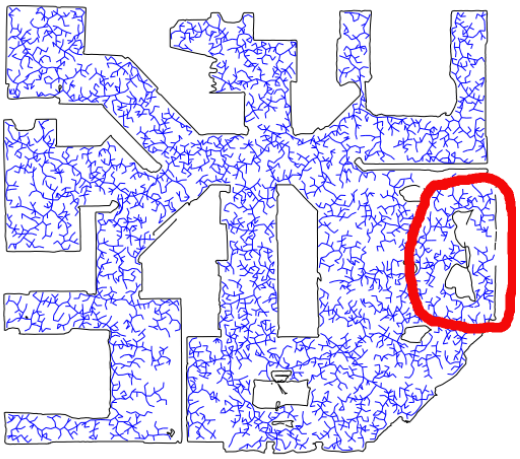
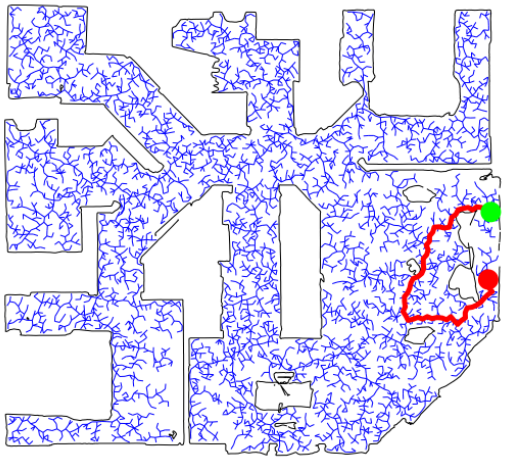


Figure 7. Allensville Sample Path 2



3. Check cost of A* search for new graph (since it is no longer a tree.)

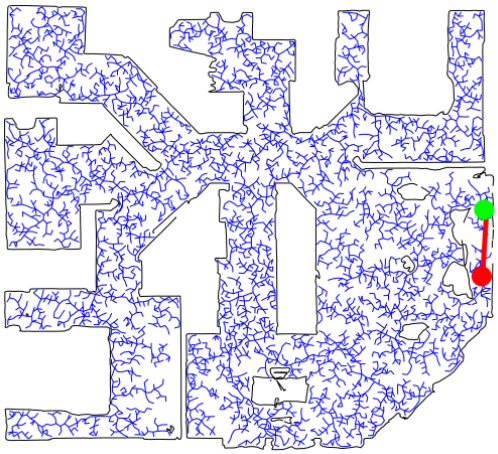
4. When the cost appears to converge (no more improvements are being made to the cost of A*) then return the new path.

We can see the effect in 8 of refining the path from 7. The path now goes directly behind the sofa, instead of going around the front. But rather than refine the path every time, we'd rather be able to have a static map and calculate the path one time.

2.6. Refining the Map

To make a fixed map that is more efficient, we perform the the steps in the previous section a configurable number of times on random start and end points. After each iteration, we add the new path the the map graph. Once this is

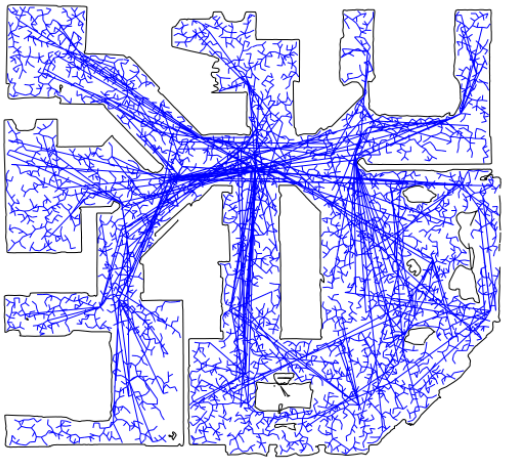
Figure 8. Refined Path



done we have a graph where the most common paths between major areas of the building have more direct straight-line routes.

Figure 9 shows a map with 100 added paths. Figures 10 and 11 show paths created using the refined map.

Figure 9. Refined Map



3. Results

Several results have already been shown. To demonstrate the scalability of this approach, figure 12 shows a map for the Brinnon environment. Brinnon is a much larger environment than Allensville. The latter is about 8m by 8m, and has about seven rooms. Brinnon is 20m by 25m, and has about 14 seperate rooms. Fewer larger environments can be made free of holes, and this is the largest one I found. It took approximately 60 seconds to create the 2D view of the Brinnon house; 23 seconds to create the RRT; and another 130 seconds to create the refined map. Once these are done, creating paths is nearly instantaneous. Figure 13 shows one

Figure 10. Refined Path 2

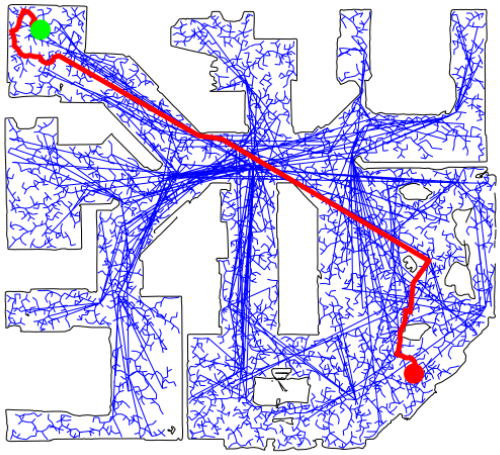
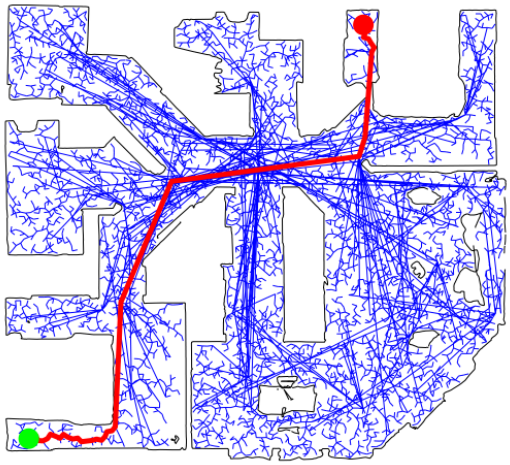


Figure 11. Refined Path 3



such path. If this were being used by a robot in this environment, the time to create the map would be small, and subsequent searches for efficient paths would be negligable.

Figure 12. Brinnon Map

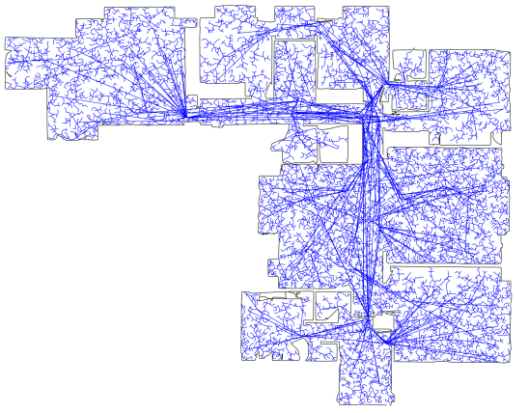
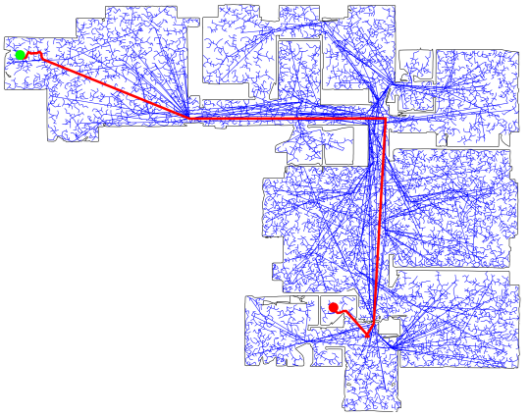


Figure 13. Brinnon Path



I admit I didn't spent much time comparing my approach to other techniques. I spent most of my time working on my program and improving it. I wrote the graphical program because I thought the animation would be pretty neat, and it would be easier to test various environments with it. These things were true, but writing the UI took quite a bit of time because I've never done much graphical programming in Python.

4. Conclusion

Using RRTs proved a promising approach for creating static maps for navigating Gibson environment. In a matter of a few minutes detailed maps based on RRTs can be produced that will allow paths to be generated using A* search.

One limitation of the approach is that the refining relies on line-of-sight between vertexes. In these fairly simple environments this does not appear to be a problem. If there were complex, zig-zagging corriders, however, this limitations might be more apparent in the results.

There are command line programs (make_rrt.py and find_path.py) which have many command-line parameters available to control the creation of the maps. They also save the generated maps and paths to files. The GUI tool, rrtgui.py, has only hard-coded values (except for choosing the environment), and it cannot save the maps and paths it produces. Further work would be to enhance rrtgui.py to allow setting of parameters and saving work.py. Also, the command-line tools do not do the refining. They would need to have these capabilities added to be useful.

The code I wrote is available in a public Github repository. The primary code artifacts are:

1. rrt.py: Library of functions for extracting a cross-section from a 3D represenation of a building, creating the free-space image, generating RRTs, and finding paths with A* search.

2. bresenham.py: My implementation of the Bresenham line-drawing algorithm.

3. find_path.py: Command-line program to find a path given start and end points and an RRT file.

4. make_rrt.py: Command-line program use rrt.py to create an RRT from a Gibson Environment building.

5. rrtgui.py: A graphical program to read Gibson Environment files, create RRTs, find paths, refine paths, and perform multi-path refining.

There are some other Python programs in there, but they are mostly cruft.

References

[1] S. M. Lavalle and J. James J. Kuffner. Rapidly-exploring random trees: Progress and prospects. 1

[2] J. Rebetez. Meshcut. https://github.com/julienr/meshcut, 2018. 1

[3] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. 3

[4] F. Xia, A. R. Zamir, Z. He, A. Sax, J. Malik, and S. Savarese. Gibson Env: real-world perception for embodied agents. In *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*. IEEE, 2018. 1