Algorithmic Robotics and Motion Planning
School of Computer Science, Tel Aviv University

Instructor: Dan Halperin
Teaching Assistant: Michal Kleinbort
Scribe: Tomer Epshtein
Date: February 23, 2022

Final Project

1 Abstract

During the course, we focused a lot on motion planning algorithms. We learned different types of algorithms for motion of one or more robots within a room with obstacles.
We learned different approaches to measure the quality of a path which represents the motion of each robot to its destination. One of these approaches is to find the shortest path, which is calculated by the sum of each path for every robot.
In the following paper, I tried to give a new algorithm (extension of an existing one); in order to produce the best path, each scene includes 2 disc robots within a room, containing polygonal obstacles and disc obstacles.
I will describe my implementations[1], the experiments I did and the intent to choose the best value for a parameter, s.t running my algorithm with that value should yield the best results.

2 Introduction

In the project I implemented[2] an extension to the PRM discs algorithm which we were given by the course's staff in the CGAL package. In addition, I implemented the RRT* algorithm in order to compare its results with my algorithm.
The comparison was made on 100 scenes which I created[3].
Furthermore, on the base of these 100 scenes I created a data-science model which predicts (for each different scene with its features) one of the parameters in my extended algorithm, such that running my algorithm with the predicted value should yield the best results.

[1] Implemented in python, https://github.com/TomerEpshtein/robotics_project.
[2] Full documentation about the implementations is given at the docs.md file in the repository.
[3] Example for some of the scenes is given in Figure 1

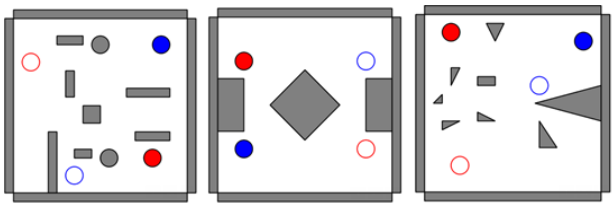

Figure 1: Example for scenes which were used in the project

3 RRT*

RRT*[1] is an optimized variation of the RRT[2] algorithm. It will return shorter paths than RRT, and the more iterations it runs, the shorter the path it will return. The optimization is based on a technique such that after adding a new point to the tree, a search for a small neighborhood of the new point is done (we find all the points in the tree with small distance from the new point). Then, for each such point RRT* checks whether it is better to add a new edge from the new point to the current point and if it's better – RRT* adds this edge and removes the old edge from the current point to its parent in the tree (in order to preserve the tree structure so they won't be many edges).

4 My Extension to PRM

My extension to the PRM[3] algorithm follows the following steps:

1. Run PRM discs algorithm and use the output path.
2. Traverse each 3 vertices in the path and if we have an edge between the first and last vertices, delete the vertical in the middle.
3. Pass over the output path of step 2, by chunk_size[4]: For each sub path of size chunk_size, we try to find a faster path between the first and last vertices (very similar to step 2 in case chunk_size = 3).
    - 3.1. We first check if there is an edge between them, if yes we remove all the edges inside, and add to our path only the first and last vertices and continue to the next sub path. Otherwise:
    - 3.2. We run a Local PRM[5] with the attempt of finding a better path. If one is found, we add it to our path.
    - 3.3. If a sub path isn't found or it's found but it's worse than the origin chunk sub path (longer), we add to our path the original sub-path.

[4] A parameter to the algorithm which is given by the user.

[5] Local PRM is a change I made to PRM - it acts almost exactly like PRM, except it chooses randomly the whole points from a ball defined by a given point as its center and a radius I fixed in the implementation (and not from the whole configuration space).

4. Repeat step 2

5. Return the path

5 Experiments

In order to compare my extended algorithm with RRT*, the following steps were done for each scene out of the 100 scenes I created:

1. I ran the RRT* algorithm 10 times, then I calculated for each run the length of the output path and averaged it.

2. For each chunk_size $\in [3, 9]$ I ran my extended algorithm 10 times and calculated the averaged length of the output path.

5.1 Results

I implemented a script[6] which by running it all the results of the experiments are written to two csv files[7]. In the following table (Table 1) an example to some of the results is given.

scene name	RRT*	arg=3	arg=4	arg=5	arg=6	arg=7	arg=8	arg=9
1.json	38.95	38.10	38.90	38.78	39.17	37.90	39.29	38.60
17.json	48.80	35.96	35.90	35.92	36.83	34.09	35.85	37.07
32.json	46.20	41.23	45.19	43.83	43.02	44.73	43.43	43.83
64.json	61.80	66.71	64.39	66.06	64.81	65.66	65.95	65.32
85.json	47.21	44.51	50.30	47.71	47.75	46.28	45.86	45.44

Table 1: Results of some of the scenes

For example, the average length of the RRT* path for the scene 1.json is 38.95 where for my extended algorithm with chunk_size equals to 7 the average length is 37.90.

5.2 Conclusions

1. Out of 100 scenes, the extended algorithm "won" the RRT* algorithm on 64 scenes. On these scenes, the extended algorithm (with some chunk_size $\in [3, 9]$) outputted a shorter path than the RRT* algorithm.

2. The average output length improvement of the extended algorithm regards to RRT* is 5.36%. In the best case, the improvement was 30.15%.

[6]The script is given at the experiment_main.py file in the repository.

[7]For each scene and algorithm the average path length and the average running time are calculated. All the lengths were written to results.csv and the respectively running times were written to times.csv.

3. The running time of the extended algorithm on 44% of the scenes is even better than the running time of RRT*.

4. As we can observe (Figure 2), there is no a specific value of chunk_size that works best with all scenes. For different scenes, it seems that there are different values of chunk_size which it is better to run with.

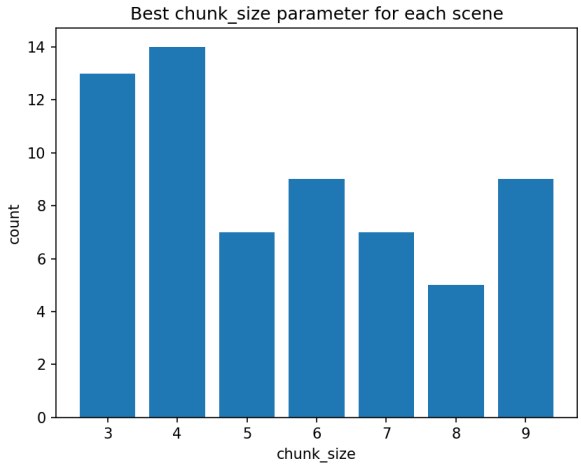

Figure 2: Bar plot of the chunk_size distribution

For example, there were exactly 14 scenes which running the extended algorithm on them with chunk_size=4 gave us the best results.

6 Optimization

In this section I will try to optimize the extended algorithm in order to determine for each scene which value of chunk_size is the best to run the algorithm with.

6.1 Features

I will describe the features I calculated[8] in order to use them to determine the best chunk_size to choose for the running of each scene. The features are:

1. total_area – The sum of the areas of the whole obstacles in the room.

   Note: All the scenes I created share the same bounding box of the room.

[8]The calculation was made both manually and automatically by the scripts features.py and prm_features.py.

2. robots_distances – The sum of the distances of each robot from its starting location to its destination.

3. maximum_distance_required – This is the biggest length between two obstacles, which one of the robots must pass between these two obstacles in order to arrive its destination.

4. is_linear – 0 if both robots can't get linearly from their starting point to their destination, 1 if only one of the robots can get linearly from its starting to its destination and 2 if both can do that.

5. prm_length - The average length (number of points) of the path which PRM outputs on the scene.

6. prm_quality - The average quality of the path which PRM outputs on the scene.

scene_name	F1	F2	F3	F4	F5	F6	chunk_size
1.json	128	33.97	4	0	6.1	39.89	6
11.json	122	37.56	3	1	8	47.98	5
22.json	101.28	29.66	3	2	2	29.66	3
64.json	200.28	34.4	3	0	11.9	67.83	4
90.json	111.56	21.04	2.2	0	6.5	37.18	8

Table 2: Example for the tabular data I've worked with. Full data is located at data.csv.

6.2 Base model

I tried to find the best model to predict the chunk_size, and each model was trained on 80% of the data, which we call the training set. The remaining 20% of the data was used in order to test the average error of the model.
Firstly, I ran 3 different types of models with the most basic configurations which sklearn gives us: Linear regression, Random Forest and XGBoost.

Out of these three models, the lowest average error (out of 100 runs) was for the XGBoost model.

6.2.1 Results

The average error for the base XGBoost model is 2%: For each prediction on a scene, I checked what would be the path length if we would choose the predicted chunk_size and then I compared it with the lowest length from the whole chunk sizes and measured the error (0% if doesn't exist). I did it 100 times and finally averaged the whole data.

6.3 Improving the model

I played with the parameters of XGBoost and the best ones I found are:

```
import xgboost as xgb
xgb.XGBRegressor(n_estimators=10, max_depth=50, learnin_rate=0.1,
↪  colsample_bytree=0.4)
```

6.3.1 Results

After optimizing the XGBoost and choosing the best parameters, we got an average error of 1.8%.
The baseline in this case is to choose always the chunk_size which appears the most (out all of the scenes). The average error for it is 1.96%. Therefore, we observe that we got better results than the baseline.

6.4 Conclusions and Further Research

1. As we can see, the data science attitude gave us better results rather than running the algorithm with some fixed value of chunk_size.

2. A similar research for more parameters and more algorithms could be helpful in order to fit for each scene its best parameters.

3. An automatic program which creates scenes and calculates its features could be very useful.

    3.1. For this research I created 100 scenes, but the data science models would be able to learn better with more scenes.

    3.2. I created scripts which automatically calculate some of the features I created, but there are features which writing such script is more difficult and I calculated them manually in the project.

References

[1] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning . In *Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, USA*, 2011.

[2] LaValle and Steven M. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report. Computer Science Department, Iowa State University*, 1998.

[3] J.-C. Overmars M. H. Kavraki; L. E. Svestka, P. Latombe. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE Transactions on Robotics and Automation*, 1996.