The CR RRT-Connect

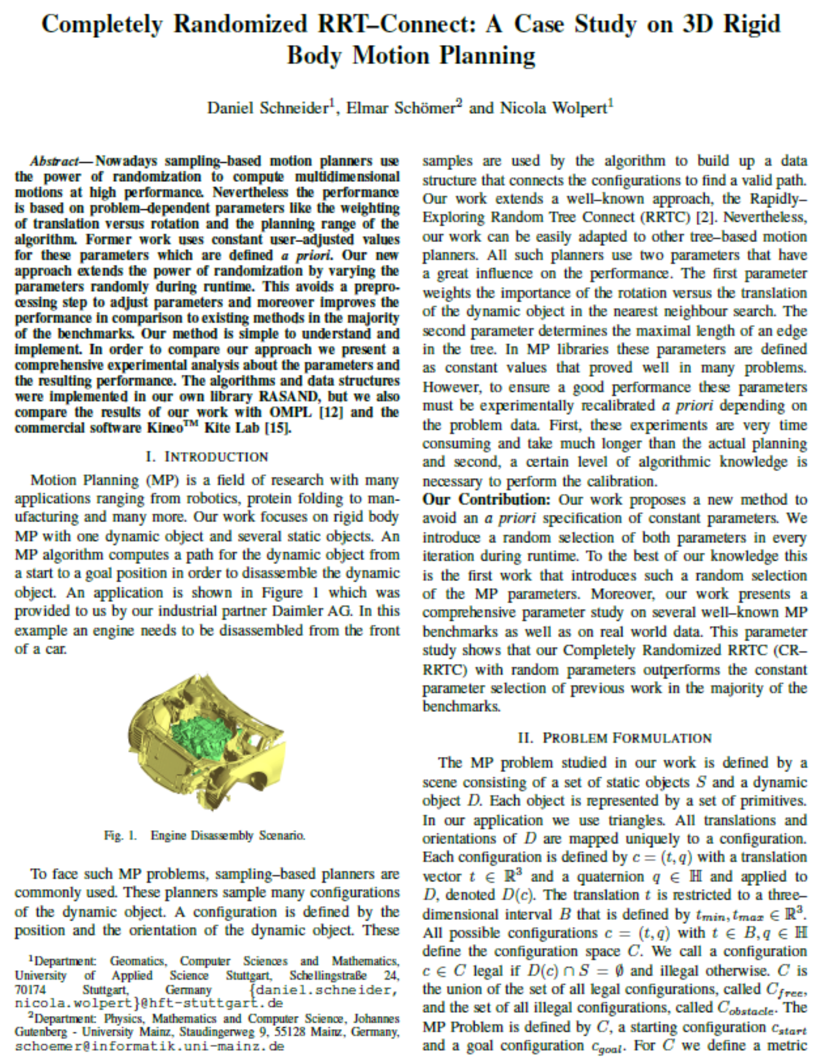
Algorithms and Data Structures 2 – Motion Planning and its applications

University of Applied Sciences Stuttgart

Dr. Daniel Schneider

The **C**ompletely **R**andomized **RRT-C**onnect

“A algorithm that addresses an issue that you will face with motion planner on 3D rigid bodies”



Completely Randomized RRT–Connect: A Case Study on 3D Rigid Body Motion Planning

Daniel Schneider[1], Elmar Schömer[2] and Nicola Wolpert[1]

***Abstract*— Nowadays sampling–based motion planners use the power of randomization to compute multidimensional motions at high performance. Nevertheless the performance is based on problem–dependent parameters like the weighting of translation versus rotation and the planning range of the algorithm. Former work uses constant user–adjusted values for these parameters which are defined *a priori*. Our new approach extends the power of randomization by varying the parameters randomly during runtime. This avoids a preprocessing step to adjust parameters and moreover improves the performance in comparison to existing methods in the majority of the benchmarks. Our method is simple to understand and implement. In order to compare our approach we present a comprehensive experimental analysis about the parameters and the resulting performance. The algorithms and data structures were implemented in our own library RASAND, but we also compare the results of our work with OMPL [12] and the commercial software Kineo^{TM} Kite Lab [15].**

I. Introduction

Motion Planning (MP) is a field of research with many applications ranging from robotics, protein folding to manufacturing and many more. Our work focuses on rigid body MP with one dynamic object and several static objects. An MP algorithm computes a path for the dynamic object from a start to a goal position in order to disassemble the dynamic object. An application is shown in Figure 1 which was provided to us by our industrial partner Daimler AG. In this example an engine needs to be disassembled from the front of a car.

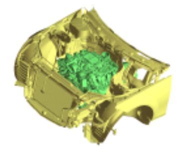


Fig. 1. Engine Disassembly Scenario.

To face such MP problems, sampling–based planners are commonly used. These planners sample many configurations of the dynamic object. A configuration is defined by the position and the orientation of the dynamic object. These samples are used by the algorithm to build up a data structure that connects the configurations to find a valid path. Our work extends a well–known approach, the Rapidly–Exploring Random Tree Connect (RRTC) [2]. Nevertheless, our work can be easily adapted to other tree–based motion planners. All such planners use two parameters that have a great influence on the performance. The first parameter weights the importance of the rotation versus the translation of the dynamic object in the nearest neighbour search. The second parameter determines the maximal length of an edge in the tree. In MP libraries these parameters are defined as constant values that proved well in many problems. However, to ensure a good performance these parameters must be experimentally recalibrated *a priori* depending on the problem data. First, these experiments are very time consuming and take much longer than the actual planning and second, a certain level of algorithmic knowledge is necessary to perform the calibration.

Our Contribution: Our work proposes a new method to avoid an *a priori* specification of constant parameters. We introduce a random selection of both parameters in every iteration during runtime. To the best of our knowledge this is the first work that introduces such a random selection of the MP parameters. Moreover, our work presents a comprehensive parameter study on several well–known MP benchmarks as well as on real world data. This parameter study shows that our Completely Randomized RRTC (CR–RRTC) with random parameters outperforms the constant parameter selection of previous work in the majority of the benchmarks.

II. Problem Formulation

The MP problem studied in our work is defined by a scene consisting of a set of static objects S and a dynamic object D. Each object is represented by a set of primitives. In our application we use triangles. All translations and orientations of D are mapped uniquely to a configuration. Each configuration is defined by $c = (t, q)$ with a translation vector $t \in \mathbb{R}^3$ and a quaternion $q \in \mathbb{H}$ and applied to D, denoted $D(c)$. The translation t is restricted to a three–dimensional interval B that is defined by $t_{min}, t_{max} \in \mathbb{R}^3$. All possible configurations $c = (t, q)$ with $t \in B, q \in \mathbb{H}$ define the configuration space C. We call a configuration $c \in C$ legal if $D(c) \cap S = \emptyset$ and illegal otherwise. C is the union of the set of all legal configurations, called C_{free}, and the set of all illegal configurations, called $C_{obstacle}$. The MP Problem is defined by C, a starting configuration c_{start} and a goal configuration c_{goal}. For C we define a metric

[1]Department: Geomatics, Computer Sciences and Mathematics, University of Applied Science Stuttgart, Schellingstraße 24, 70174 Stuttgart, Germany {daniel.schneider, nicola.wolpert}@hft-stuttgart.de

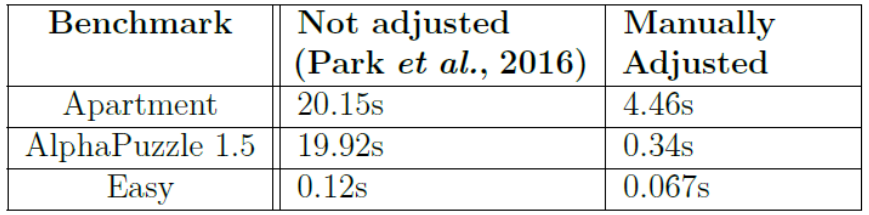
[2]Department: Physics, Mathematics and Computer Science, Johannes Gutenberg - University Mainz, Staudingerweg 9, 55128 Mainz, Germany, schoemer@informatik.uni-mainz.de

What is the problem?

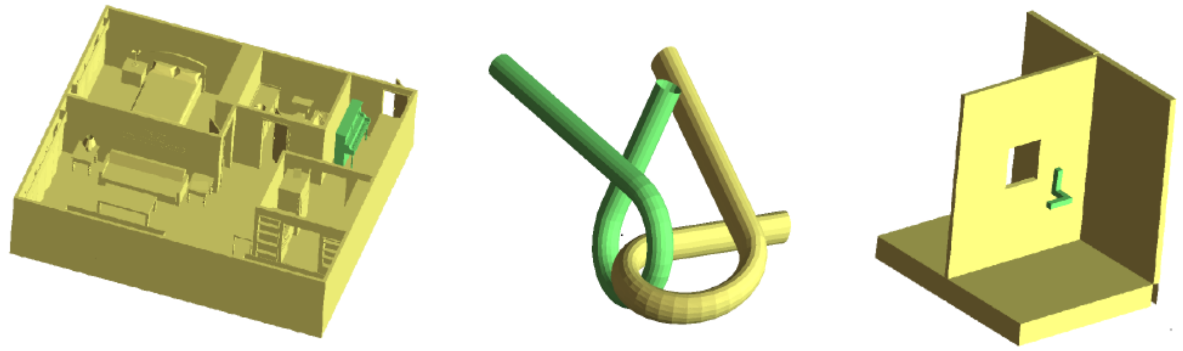
In motion planning there are quite a few parameters for the algorithm. These parameters have a high influence on the performance.

→ Here in this paper we discuss the two main parameters

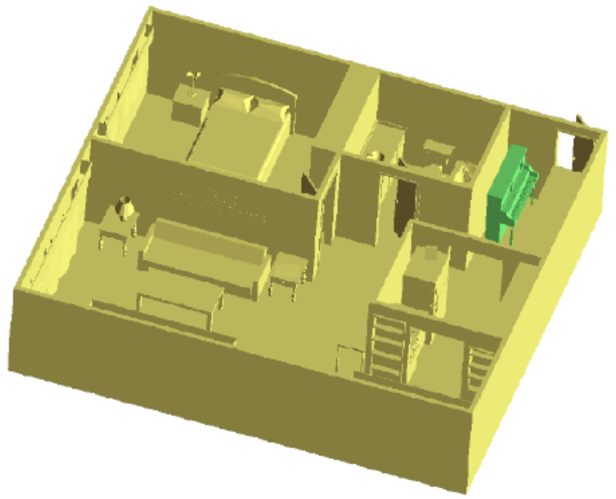
Table 1: Performance comparison OMPL, Parameters.



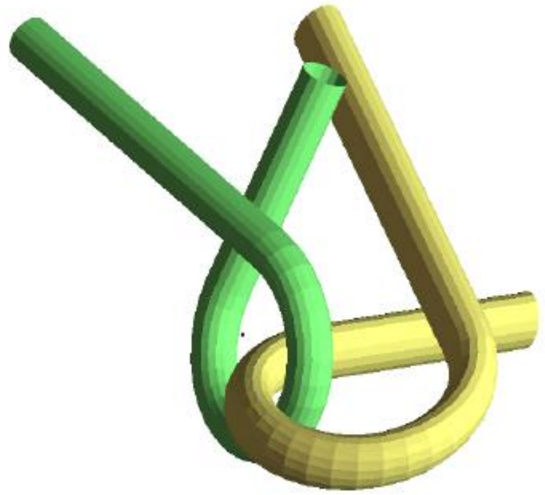
Benchmark	Not adjusted (Park *et al.*, 2016)	Manually Adjusted
Apartment	20.15s	4.46s
AlphaPuzzle 1.5	19.92s	0.34s
Easy	0.12s	0.067s



Parameter 1 - What is the difference?



For solving the piano movers problem, you have to mainly translate the robot and on some parts you have to rotate the robot.



For solving the alpha puzzle, you have to mainly rotate the robot and on some positions you have to translate the object.

Parameter 1 - How to measure distance in 3D?

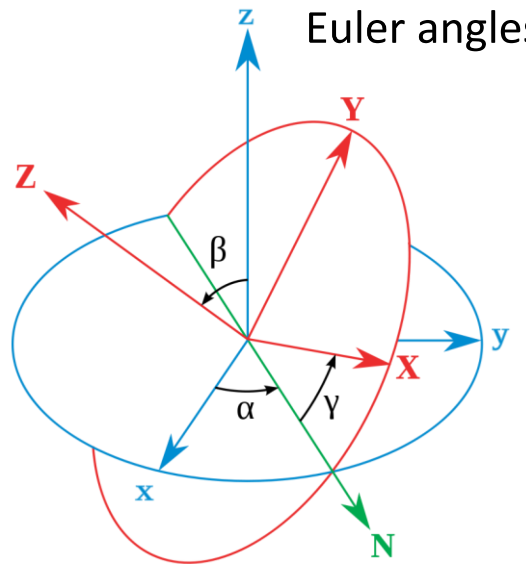
The common metric for Motion planning:

$$d_c(c_1, c_2, u_1, u_2) = u_1 \cdot d_t(t_1, t_2) + u_2 \cdot d_q(q_1, q_2)$$

$$d_t(t_1, t_2) = ||t_1 - t_2||_2$$

$$d_q(q_1, q_2) = min\{acos(q_1 \cdot q_2), acos(q_1 \cdot (-q_2))\}$$

Euler angles



Easier alternative

These are quaterions

Parameter 1 - How to measure distance in 3D?

An alternative metric for Motion planning:

$$d'_c(c_1, c_2, u) = u \cdot d'_t(t_1, t_2) + (1 - u) \cdot d'_q(q_1, q_2); \quad u \in (0, 1)$$

$$d'_t(t_1, t_2) = \nu^{-1} \cdot ||t_1 - t_2||_2$$

$$\nu \text{ is the maximum extension } ||t_{max} - t_{min}||_2$$

$$d'_q(q_1, q_2) = \pi^{-1} \cdot min\{acos(q_1 \cdot q_2), acos(q_1 \cdot (-q_2))\}$$

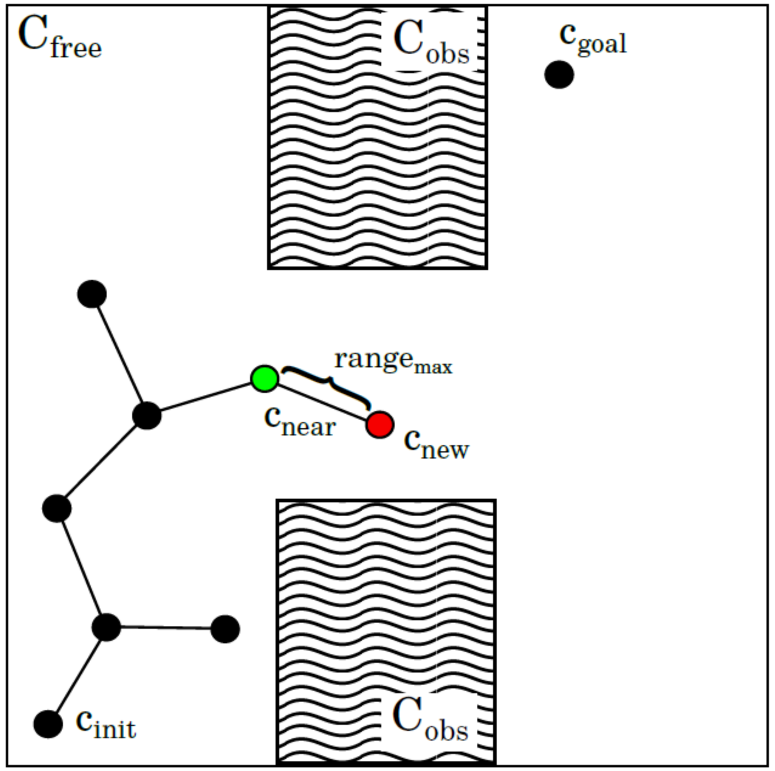
Advantages:

- Only one parameter
- Scaling/definition of freespace no longer a problem
- Distance in [0,1]

Parameter 2 – Edge length?

Algorithm 1: RRT (c_{init}, c_{goal}, $range_{max}$, $time_{max}$)

```
T.init($c_{init}$)
while (! $TimeElapsed(T_{max})$) do
    $c_{rand}$ ← RandomState()
    $c_{near}$ ← NearestN($c_{rand}$, $T$)
    $c_{new}$ ← getValidEdge($c_{near}$, $c_{rand}$, $range_{max}$)
    if (! $Trapped$) then
        T.addVertex($c_{new}$)
        T.addEdge($c_{near}$, $c_{new}$)
        if $goalReached(c_{new}, c_{goal})$ then
            return $Path(T)$
return $ApproxPath(T)$
```



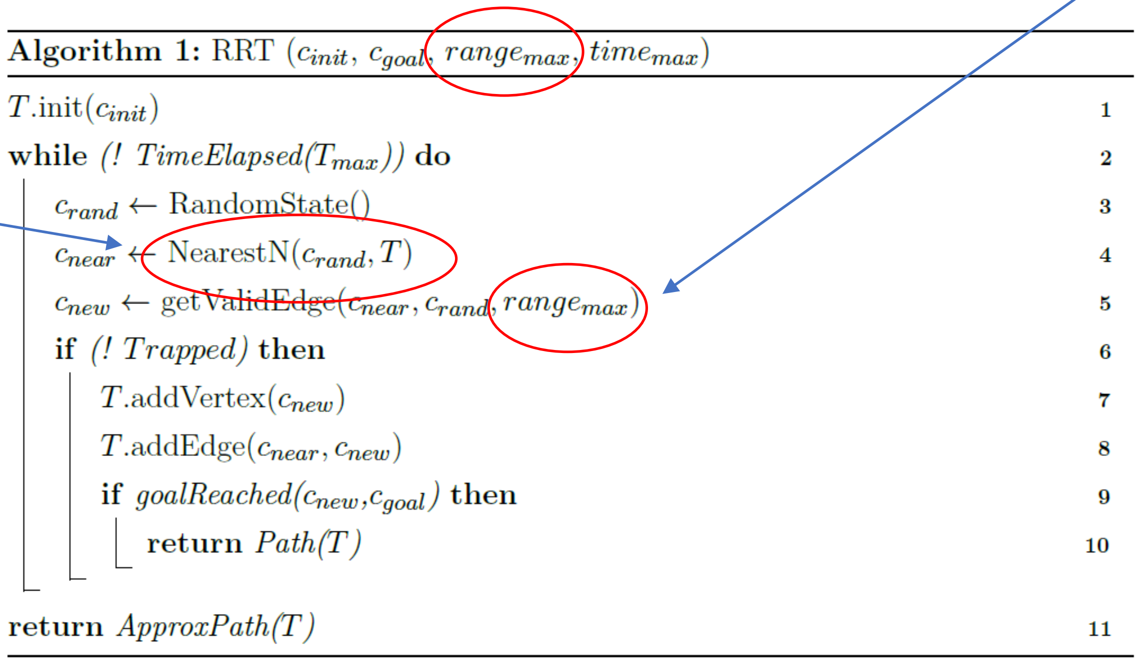
$$\boxed{range(c_i, c_j) := max\{d'_t(t_i, t_j), d'_q(q_i, q_j)\}}$$

The $range_{max}$ determines how long the edges in our tree can be. For some motion planning problems it is better to use short edges and for some examples it is better to use long edges.

Where are the parameter used?

How long do we draw the edges?

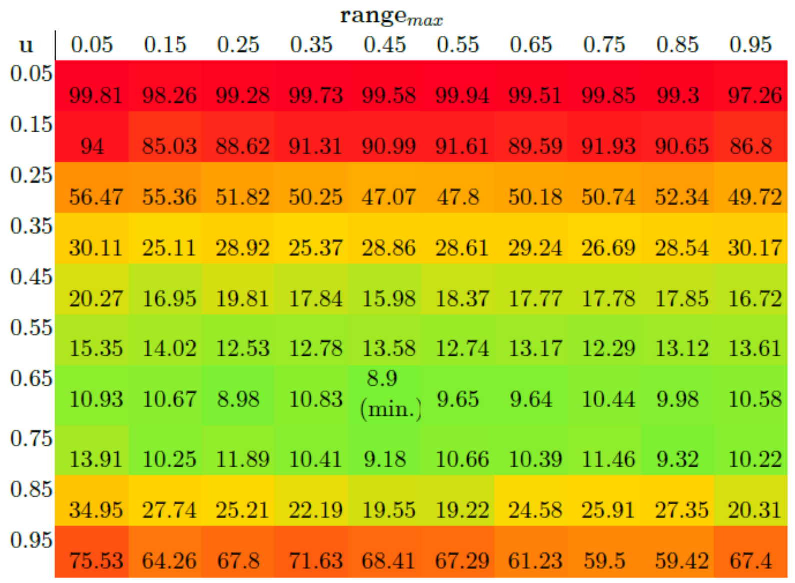
What is the nearest neighbour? → determined by the defined metric.



```
Algorithm 1: RRT ($c_{init}$, $c_{goal}$, $range_{max}$, $time_{max}$)
$T$.init($c_{init}$)
while (! $TimeElapsed(T_{max})$) do
    $c_{rand} \leftarrow$ RandomState()
    $c_{near} \leftarrow$ NearestN($c_{rand}$, $T$)
    $c_{new} \leftarrow$ getValidEdge($c_{near}$, $c_{rand}$, $range_{max}$)
    if (! $Trapped$) then
        $T$.addVertex($c_{new}$)
        $T$.addEdge($c_{near}$, $c_{new}$)
        if $goalReached(c_{new}, c_{goal})$ then
            return $Path(T)$
return $ApproxPath(T)$
```

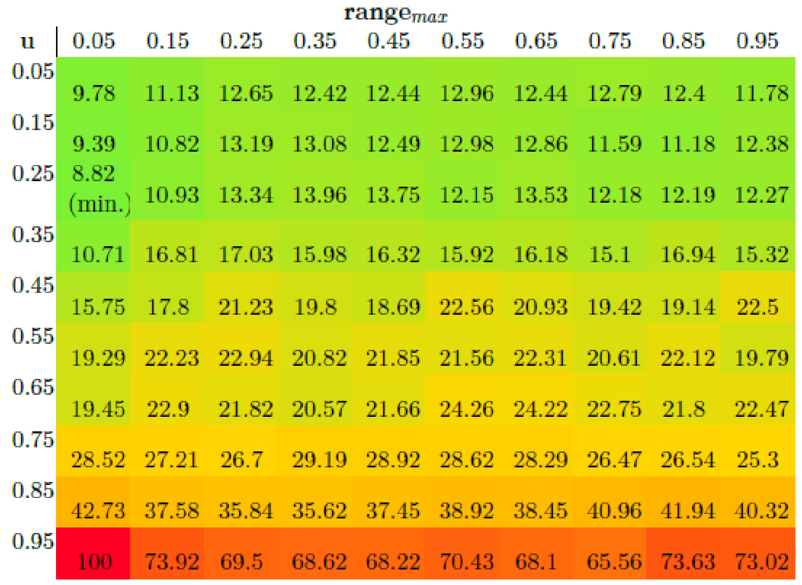
Are there “good” parameters? Best practices?

Table 2: Heatmap of Alpha 1.2, Values in %, $T_{max} = 35s$.



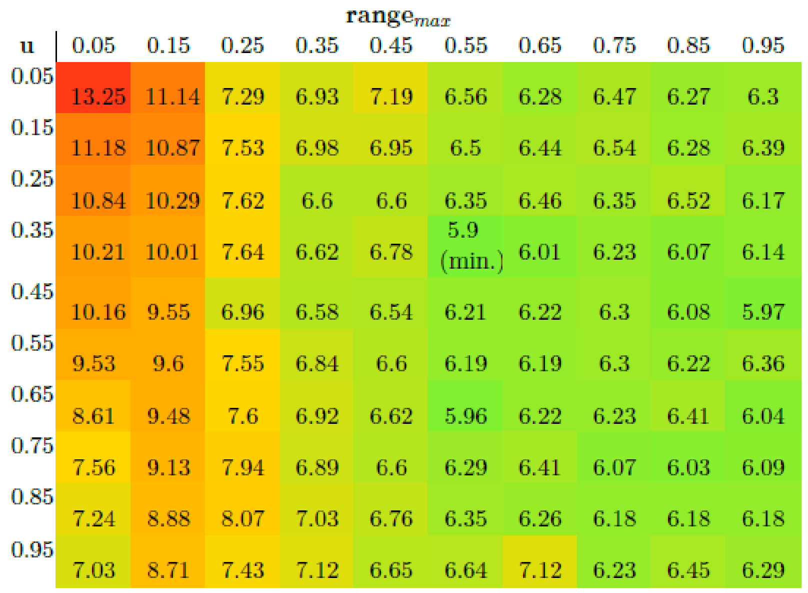
	range$_{max}$									
u	0.05	0.15	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95
0.05	99.81	98.26	99.28	99.73	99.58	99.94	99.51	99.85	99.3	97.26
0.15	94	85.03	88.62	91.31	90.99	91.61	89.59	91.93	90.65	86.8
0.25	56.47	55.36	51.82	50.25	47.07	47.8	50.18	50.74	52.34	49.72
0.35	30.11	25.11	28.92	25.37	28.86	28.61	29.24	26.69	28.54	30.17
0.45	20.27	16.95	19.81	17.84	15.98	18.37	17.77	17.78	17.85	16.72
0.55	15.35	14.02	12.53	12.78	13.58	12.74	13.17	12.29	13.12	13.61
0.65	10.93	10.67	8.98	10.83	8.9 (min.)	9.65	9.64	10.44	9.98	10.58
0.75	13.91	10.25	11.89	10.41	9.18	10.66	10.39	11.46	9.32	10.22
0.85	34.95	27.74	25.21	22.19	19.55	19.22	24.58	25.91	27.35	20.31
0.95	75.53	64.26	67.8	71.63	68.41	67.29	61.23	59.5	59.42	67.4

Table 3: Heatmap of Apartment, Values in %, $T_{max} = 2.5s$.



	range$_{max}$									
u	0.05	0.15	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95
0.05	9.78	11.13	12.65	12.42	12.44	12.96	12.44	12.79	12.4	11.78
0.15	9.39	10.82	13.19	13.08	12.49	12.98	12.86	11.59	11.18	12.38
0.25	8.82 (min.)	10.93	13.34	13.96	13.75	12.15	13.53	12.18	12.19	12.27
0.35	10.71	16.81	17.03	15.98	16.32	15.92	16.18	15.1	16.94	15.32
0.45	15.75	17.8	21.23	19.8	18.69	22.56	20.93	19.42	19.14	22.5
0.55	19.29	22.23	22.94	20.82	21.85	21.56	22.31	20.61	22.12	19.79
0.65	19.45	22.9	21.82	20.57	21.66	24.26	24.22	22.75	21.8	22.47
0.75	28.52	27.21	26.7	29.19	28.92	28.62	28.29	26.47	26.54	25.3
0.85	42.73	37.58	35.84	35.62	37.45	38.92	38.45	40.96	41.94	40.32
0.95	100	73.92	69.5	68.62	68.22	70.43	68.1	65.56	73.63	73.02

Table 6: Heatmap of Easy, Values in %, $T_{max} = 1s$.



	range$_{max}$									
u	0.05	0.15	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95
0.05	13.25	11.14	7.29	6.93	7.19	6.56	6.28	6.47	6.27	6.3
0.15	11.18	10.87	7.53	6.98	6.95	6.5	6.44	6.54	6.28	6.39
0.25	10.84	10.29	7.62	6.6	6.6	6.35	6.46	6.35	6.52	6.17
0.35	10.21	10.01	7.64	6.62	6.78	5.9 (min.)	6.01	6.23	6.07	6.14
0.45	10.16	9.55	6.96	6.58	6.54	6.21	6.22	6.3	6.08	5.97
0.55	9.53	9.6	7.55	6.84	6.6	6.19	6.19	6.3	6.22	6.36
0.65	8.61	9.48	7.6	6.92	6.62	5.96	6.22	6.23	6.41	6.04
0.75	7.56	9.13	7.94	6.89	6.6	6.29	6.41	6.07	6.03	6.09
0.85	7.24	8.88	8.07	7.03	6.76	6.35	6.26	6.18	6.18	6.18
0.95	7.03	8.71	7.43	7.12	6.65	6.64	7.12	6.23	6.45	6.29

There are no “good” values. For each problem there is a different pair of parameters best. Also not the structure (color change) is completely different.

Some notes

- It is not possible to know the parameters beforehand.
- You can only get good values by experiments.
- These experiments take a lot of time, even on modern computers (at least 25 runs per parameter pair)
- Doing a parameter study before running the algorithm is not practical.
- There is not way of guessing the parameter based on the geometry.

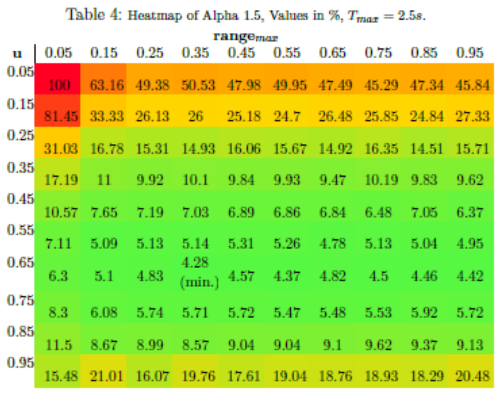


Table 4: Heatmap of Alpha 1.5, Values in %, $T_{max} = 2.5s$.

	range$_{max}$									
u	0.05	0.15	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95
0.05	100	63.16	49.38	50.53	47.98	49.95	47.49	45.29	47.34	45.84
0.15	81.45	33.33	26.13	26	25.18	24.7	26.48	25.85	24.84	27.33
0.25	31.03	16.78	15.31	14.93	16.06	15.67	14.92	16.35	14.51	15.71
0.35	17.19	11	9.92	10.1	9.84	9.93	9.47	10.19	9.83	9.62
0.45	10.57	7.65	7.19	7.03	6.89	6.86	6.84	6.48	7.05	6.37
0.55	7.11	5.09	5.13	5.14	5.31	5.26	4.78	5.13	5.04	4.95
0.65	6.3	5.1	4.83	4.28 (min.)	4.57	4.37	4.82	4.5	4.46	4.42
0.75	8.3	6.08	5.74	5.71	5.72	5.47	5.48	5.53	5.92	5.72
0.85	11.5	8.67	8.99	8.57	9.04	9.04	9.1	9.62	9.37	9.13
0.95	15.48	21.01	16.07	19.76	17.61	19.04	18.76	18.93	18.29	20.48

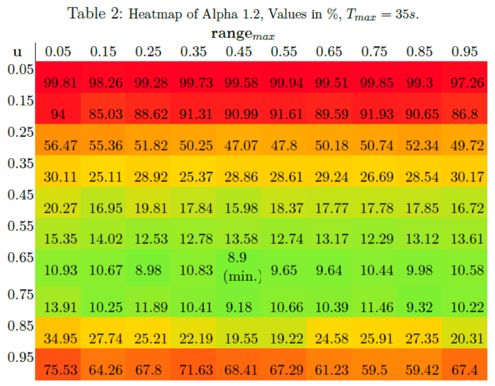


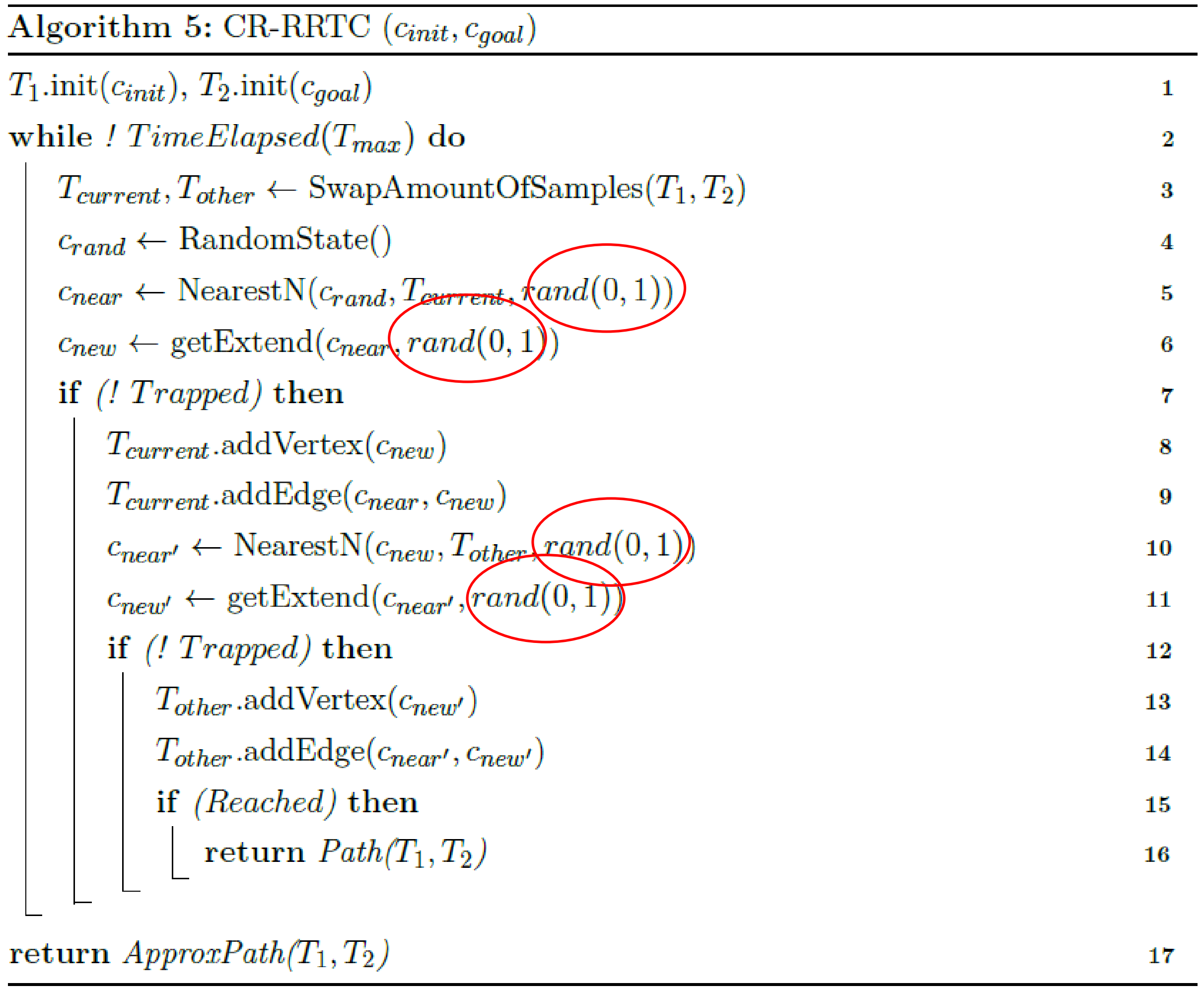
Table 2: Heatmap of Alpha 1.2, Values in %, $T_{max} = 35s$.

	range$_{max}$									
u	0.05	0.15	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95
0.05	99.81	98.26	99.28	99.73	99.58	99.94	99.51	99.85	99.3	97.26
0.15	94	85.03	88.62	91.31	90.99	91.61	89.59	91.93	90.65	86.8
0.25	56.47	55.36	51.82	50.25	47.07	47.8	50.18	50.74	52.34	49.72
0.35	30.11	25.11	28.92	25.37	28.86	28.61	29.24	26.69	28.54	30.17
0.45	20.27	16.95	19.81	17.84	15.98	18.37	17.77	17.78	17.85	16.72
0.55	15.35	14.02	12.53	12.78	13.58	12.74	13.17	12.29	13.12	13.61
0.65	10.93	10.67	8.98	10.83	8.9 (min.)	9.65	9.64	10.44	9.98	10.58
0.75	13.91	10.25	11.89	10.41	9.18	10.66	10.39	11.46	9.32	10.22
0.85	34.95	27.74	25.21	22.19	19.55	19.22	24.58	25.91	27.35	20.31
0.95	75.53	64.26	67.8	71.63	68.41	67.29	61.23	59.5	59.42	67.4

- People with experiences in running such motion planner could guess them – not practical as well.

The approach

Motion planners work with randomness. If there is not best parameter, why not choose the parameters also randomly?



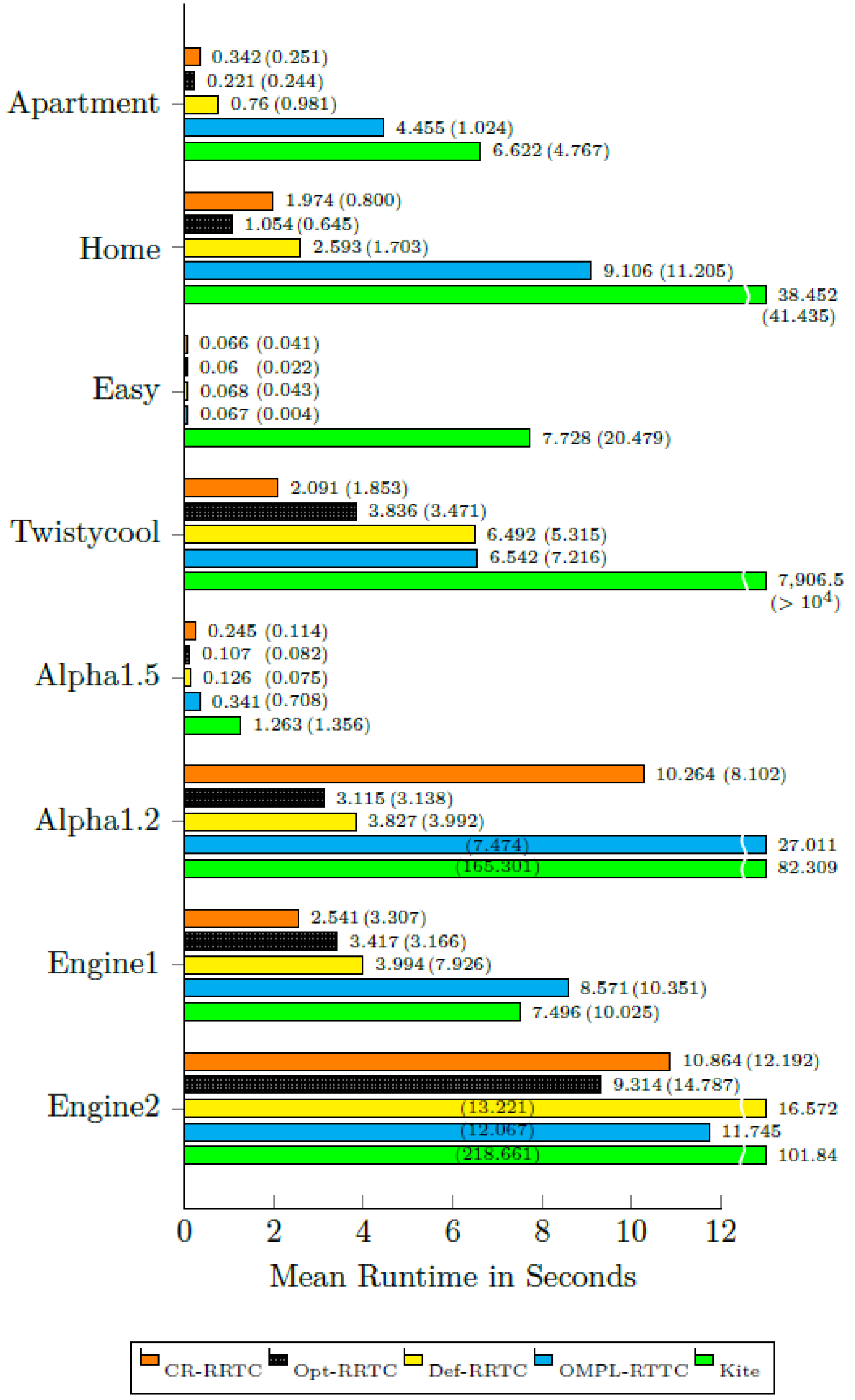
Algorithm 5: CR-RRTC (c_{init}, c_{goal})

```
$T_1$.init($c_{init}$), $T_2$.init($c_{goal}$)
while ! $TimeElapsed(T_{max})$ do
    $T_{current}, T_{other} \leftarrow$ SwapAmountOfSamples($T_1, T_2$)
    $c_{rand} \leftarrow$ RandomState()
    $c_{near} \leftarrow$ NearestN($c_{rand}, T_{current}, rand(0,1)$)
    $c_{new} \leftarrow$ getExtend($c_{near}, rand(0,1)$)
    if (! $Trapped$) then
        $T_{current}$.addVertex($c_{new}$)
        $T_{current}$.addEdge($c_{near}, c_{new}$)
        $c_{near'} \leftarrow$ NearestN($c_{new}, T_{other}, rand(0,1)$)
        $c_{new'} \leftarrow$ getExtend($c_{near'}, rand(0,1)$)
        if (! $Trapped$) then
            $T_{other}$.addVertex($c_{new'}$)
            $T_{other}$.addEdge($c_{near'}, c_{new'}$)
            if ($Reached$) then
                return $Path(T_1, T_2)$
return $ApproxPath(T_1, T_2)$
```

Does it work?

- The performance often better than with default values.
- The performance is better than with an OpenSource Library
- The performance is also better than a commercial software tool

→ In some cases, the random approach does show a better performance than the optimal “fixed” values.


Apartment
0.342 (0.251)
0.221 (0.244)
0.76 (0.981)
4.455 (1.024)
6.622 (4.767)
Home
1.974 (0.800)
1.054 (0.645)
2.593 (1.703)
9.106 (11.205)
38.452 (41.435)
Easy
0.066 (0.041)
0.06 (0.022)
0.068 (0.043)
0.067 (0.004)
7.728 (20.479)
Twistycool
2.091 (1.853)
3.836 (3.471)
6.492 (5.315)
6.542 (7.216)
7,906.5 (> 10^4)
Alpha1.5
0.245 (0.114)
0.107 (0.082)
0.126 (0.075)
0.341 (0.708)
1.263 (1.356)
Alpha1.2
10.264 (8.102)
3.115 (3.138)
3.827 (3.992)
(7.474) 27.011
(165.301) 82.309
Engine1
2.541 (3.307)
3.417 (3.166)
3.994 (7.926)
8.571 (10.351)
7.496 (10.025)
Engine2
10.864 (12.192)
9.314 (14.787)
(13.221) 16.572
(12.067) 11.745
(218.661) 101.84
Mean Runtime in Seconds
CR-RRTC
Opt-RRTC
Def-RRTC
OMPL-RTTC
Kite