Technische Universität Berlin

Robotics

Thursday Group A

Lab Assignment 5

Authors:

Oliver Horst
Gábor Lajkó
Matteo Mainenti
Aditya Mohan

February 9, 2020

1 RRT Extensions

The task is to develop and implement variants of an RRT algorithm. The variants should improve the performance, i.e. the time until a solution is found, without being overly tailored to the given problem. The path optimization method needs not be modified.

1.1 Defined macros

For better customization of the runtime behaviour of our extensions, a set of macro expressions have been defined. Their values are dependent on the given configuration of the given test always, so the ones below are just examples. We only list them in order to avoid confusion about our code snippets.

1.1.1 Planner macros

Listing 1: Runtime macros in Planner

```
#define EXHAUSTED 15
#define COLLTIMES 20
#define EXHAUST false
#define HEUR false
#define RANDOMPCT 100
#define STEPSIZE 5
#define STEPLIMIT false
#define CONNECTTREESPCT 0
#define SWAPSTEPS 5000
#define LIMITSWAP false
#define M_OPT 0.1
#define BASIC 1
#define NEARESTCHECK 0
#define CONTROLSWAP 1
#define CONTROLCHOSEN 0
```

1.1.2 Sampler macros

Listing 2: Runtime macros in Sampler

```
#define PASSAGEPCT 0
#define BORDERPCT 0
#define NEIGHBOURNUM 5
#define BORDERSTEP 0.02
```

1.2 Implementation 1:Swap limiter

In the solve() function, the basic implementation swaps between the two trees at the end of each cycle, but within the connect() function the "while(!reached)" loop calls for additional extension steps from vertices that have bigger unexplored space around them. The swap in solve() only happens after this loop also came to a halt, which means that multiple extensions take place on one tree, if the node from that tree can be extended towards the goal without collision. Our modification takes count of the actual steps taken, and therefore balances this extension bias between the two trees.

The basic code extends one node per tree by switching between the active trees and calling one connect function onto each. This however leads to extensive exploration of open spaces, where a connect loop will take many steps (= collision checks) until the configuration actually collides, while only relatively little time is being spent on exploring narrow passages in C-space, where little steps already lead to collisions.
Instead, we decided to even out the steps being taken within each tree by introducing a stepcounter, which regulates the switching between trees. This results in a less biased exploration of all spaces and therefore a quicker passing through narrow passages.

1.2.1 Code changes

Listing 3: The modified code segments

```
//#define SWAPSTEPS 5000

//global variable int currSteps

//In connect() a counter incrementing within the "until reached" loop
    (currSteps)

//From solve():
   if(currSteps>=SWAPSTEPS){
             currSteps = 0;
             using ::std::swap;
             swap(a, b);
          }
```

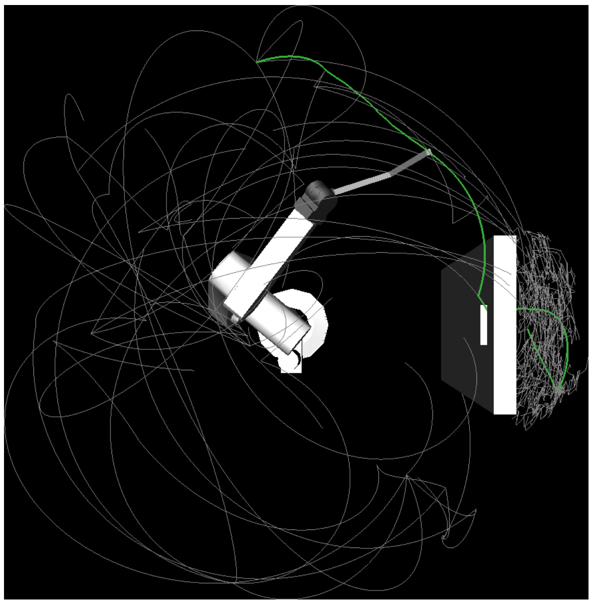

Figure 1: An execution of the modified code, where the balanced extension of the two trees is visible

1.2.2 Performance evaluation

Comparison of the extension and the base implementation based on 10 runs.

	RRTConCon	RRTConCon (reversed)	Your Planner	Your Planner (reversed)
avgT	39.37s	33.37s	3.73s	4.76
stdT	24.8s	22.4s	2.50s	2.96s
avgNodes	11660	10047	1866	1685
avgQueries	579757	510213	45541	39541

1.3 Implementation 2: CollisionSwap

This implementation modifies the way in which we choose to swap or not the two trees. In fact, as we previously stated, swapping tree is a high cost for our algorithm, this is why we came up with a second version for swapping (which also enhance the performances if used with the previous one, as we will following demonstrate). This time we just use a global boolean value which becomes true only if we detect a collision during the connect() function. This reduces the amount of swap per each run of the program, slightly improving the performance.

1.3.1 Implementation

Listing 4: Code snippet

```
#define CONTROLSWAP 1
bool collision=FALSE;

// in the connect() function
       if (this->model->isColliding())
       {
#if CONTROLSWAP
          collision=TRUE;
#endif
          break;
       }
// in the solve() function
   using ::std::swap;
#if CONTROLSWAP
     if(collision){
        swap(a, b);
        collision=FALSE;
     }
#else
```

1.3.2 Performance evaluation

Comparison of the extension and the base implementation based on 10 runs.

	RRTConCon	RRTConCon (reversed)	Your Planner	Your Planner (reversed)
avgT	39.37s	33.37s	21.56s	20.4
stdT	24.8s	22.4s	13.72s	13.9
avgNodes	11660	10047	10731.6	10176
avgQueries	579757	510213	536918.6	513427

1.4 Implementation 3: Exhaustion based heuristic approach

This approach sought to improve the algorithm and build upon the exhausted nodes approach. Instead of using a simple step function to determine the exhaustion of a node, this modification changed the ”perceived” distance of a node based on the number of failed connect attempts. One important addition here is that, as the robot needs to move through a narrow passage in C-space, which is harder to explore, nodes with a certain number (COLLTIMES) of failed connect attempts are favoured, while nodes with less or more fails are linearly penalized. The rationale behind this is that a node that is chosen often, while continuing to fail connects might be (up to a certain time) considered to be in a narrow passage and be worth exploring.

1.4.1 Implementation

Listing 5: Heuristic in the NN selection

```
RrtConConBase::Neighbor
YourPlanner::nearest(const Tree& tree, const ::rl::math::Vector& chosen)
{
    //create an empty pair <Vertex, distance> to return
    bool own = true;
    bool heur = false;
    if(HEUR || EXHAUST){
    /* .... */
        if(HEUR) {
            double failval = std::abs(COLLTIMES - fails);
            double heurMod = 0.9 + (0.1 / COLLTIMES * failval);
            if (d * heurMod < p.second) {
                p.first = *i.first;
                p.second = d;
            }
        }
        /*...*/
    }
}
```

1.4.2 Performance evaluation

Comparison of the extension and the base implementation based on 10 runs.

	RRTConCon	RRTConCon (reversed)	Your Planner	Your Planner (reversed)
avgT	39.37s	33.37s	31.29s	29.40s
stdT	24.8s	22.4s	17.23s	16.64s
avgNodes	11660	10047	9700	8688
avgQueries	579757	510213	478535	446662

be

in a narrow passage and be worth exploring.

1.5 Implementation 4: NeighborCost

We implemented the following cost function to evaluate which couple of sample and neighbor would suit best the algorithm while it runs. We basically choose it based on the distance between the neighbor itself and the goal of the robot. m_quality represent the quality of the neighbor chosen for the current iteration.

$$m_{quality} = 1 -$$

$$\frac{(C_{vertex} - C_{opt})}{(C_{max} - C_{opt})} \qquad (1)$$

1

We calculate m_quality based on different factors present in our tree:

- C_opt: Distance between start and goal
- C_vertex: Distance between the vertex and the goal plus the distance between the vertex and the start
- C_max: the vertex inside the tree with the longest distance from the goal and the beginning
- R: a random value between 0 and 1

Then we compare the value of R with the value of m_quality, if the value of m_quality is lower then we will sample another point and find a new neighbor for it. Filtering the choice of the neighbor will make the program calculate more efficient paths, improving its performance.

1.5.1 Implementation

Listing 6: C++ code using listings

```
#define NEARESTCHECK 1


#elif NEARESTCHECK

        double r;
        double m_quality;
        Neighbor aNearest;

        if(first) {
            do {

                this->choose(chosen);
                aNearest = this->nearest(*a, chosen);


                for (VertexIteratorPair i = ::boost::vertices(*a);
                     i.first != i.second; ++i.first) {
                    ::rl::math::Real d =
                        this->model->transformedDistance(*(*a)[*i.first].q,
                        *this->start)
                            +
                                this->model->transformedDistance(*(*a)[*i.first].q,
                                *this->goal);
                    if (d > C_max) {
```

[1]Ri.cmu.edu. (2020). [online] Available at: https://www.ri.cmu.edu/pub_files/pub4/urmson_christopher_2003_1/urmson_christopher_2003_1.pdf[Accessed8Feb.2020].

```
                C_max = d;
            }
        }
        if (m_quality > M_OPT) m_quality = M_OPT;
        C_opt = this->model->transformedDistance(*this->goal,
            *this->start);
        C_vertex =
            this->model->transformedDistance(*(*a)[aNearest.first].q,
            *this->start) +
                this->model->transformedDistance(*(*a)[aNearest.first].q,
                    *this->goal);
        m_quality = 1.0 - (double) ((C_vertex - C_opt) / (C_max -
            C_opt));
        r = (double)((rand()%1000))/1000;

    } while (r >= m_quality);
}else{
    this->choose(chosen);
    aNearest = this->nearest(*a, chosen);
}

first=TRUE;
#endif
```

1.5.2 Performance evaluation

Comparison of the extension and the base implementation based on 10 runs.

	RRTConCon	RRTConCon (reversed)	Your Planner	Your Planner (reversed)
avgT	39.37s	33.37s	29.6s	22.26
stdT	24.8s	22.4s	22.9s	5.64
avgNodes	11660	10047	13371	8395.3
avgQueries	579757	510213	658073	436358.4

1.6 Implementation 5: Gaussian bridge

This extension adapted the sampler in a way to act similar to a bridge sampler. For this, random uniform samples are drawn, around which in turn a number (5) of Gaussian samples are placed. A uniform sample will then be chosen, if some, but not all of these samples collide, in order to find a narrow section in the configuration space.

1.6.1 Implementation

Listing 7: generatePassage() function

```
::rl::math::Vector YourSampler::generatePassage() {
            int collCount = 0;
           ::rl::math::Vector pos;
            while(collCount<2 || collCount>8) {
                collCount = 0;
                ::rl::math::Vector rand(this->model->getDof());
                for (::std::size_t i = 0; i < this->model->getDof(); ++i)
                {
                    rand(i) = this->rand();
                }
                pos = this->model->generatePositionUniform(rand);
                ::rl::math::Vector sigma(this->model->getDof());
                for (::std::size_t i = 0; i < this->model->getDof(); ++i)
                {
                    sigma(i) = 0.05;
                }
                for (int j = 0; j < NEIGHBOURNUM; j++) {
                    for (::std::size_t i = 0; i < this->model->getDof();
                         ++i) {
                        rand(i) = this->rand();
                    }
                    ::rl::math::Vector neigh =
                         this->model->generatePositionGaussian(rand, pos,
                         sigma);
                    this->model->setPosition(neigh);
                    this->model->updateFrames();
                    if (this->model->isColliding()) {
                        collCount++;
                    }
                }
            }
           return pos;
       }
```

1.6.2 Performance evaluation

Comparison of the extension and the base implementation based on 10 runs.

	RRTConCon	RRTConCon (reversed)	Your Planner	Your Planner (reversed)
avgT	39.37s	33.37s	29.0s	42.1s
stdT	24.8s	22.4s	19.0s	35.8s
avgNodes	11660	10047	9419	11871
avgQueries	579757	510213	481298	587028

1.7 Implementation 6: ControlChosen

In this extension we modified the way in which we choose the sample, by implementing a sampling procedure that gets closer to the goal instead of going further from it. To do this we just calculated the distance from the start and the goal of our system, and then compare this value with the distance between our sampled value and the goal. If distance between our new point and the goal is higher than the start and the end points then we re-sample it. This mechanism goes on until we will find a suitable point in each iteration.

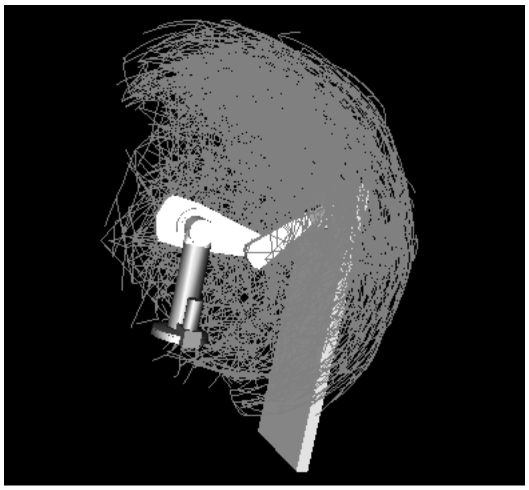

Figure 2: Visualization of the algorithm to choose the sampled point.

1.7.1 Implementation

Listing 8: C++ code using listings

```
//Into the solve() function
::rl::math::Real MaxDistance=
    this->model->transformedDistance(*this->start, *this->goal);
// ....
//During this->choose(chosen) execution
#if CONTROLCHOSEN
        ::rl::math::Real distancechosen =
            this->model->transformedDistance(chosen,*this->goal);
        while(distancechosen>MaxDistance){
            this->choose(chosen);
            distancechosen =
                this->model->transformedDistance(chosen,*this->goal);
        }
#endif
```

Finally, to the above code we added the collision swap condition which allowed the tree to swap only when a collision is detected. This is the key ingredient that made the code converge to a solution, on a n average faster than usual.

1.7.2 Performance evaluation

Comparison of the extension and the base implementation based on 10 runs.

	RRTConCon	RRTConCon (reversed)	Your Planner	Your Planner (reversed)
avgT	39.37s	33.37s	27.19s	20.95s
stdT	24.8s	22.4s	24.35s	13.2s
avgNodes	11660	10047	14085	12657.4
avgQueries	579757	510213	305672	306073

1.8 Implementation 7: Goal bias exploitation

Using a random number to choose at what points in time during execution it should change the extension strategy from exploratory to an exploitation-based one. Basically it only chooses a randomly sampled node below a certain threshold, and otherwise it sets the goal as this cycle's chosen sample.

1.8.1 Implementation

Listing 9: Goal bias in the choose() function

```
YourPlanner::choose(::rl::math::Vector& chosen, bool origin)
{
    int pctRand = RANDOMPCT;
    int rand = std::rand()%100;
    if(rand<pctRand) {
        chosen = this->sampler->generate();
    }
    else if(origin){
        chosen = *this->goal;
    }
    else{
        chosen = *this->start;
    }
}
```

1.8.2 Performance evaluation

Comparison of the extension and the base implementation based on 10 runs.

	RRTConCon	RRTConCon (reversed)	Your Planner	Your Planner (reversed)
avgT	39.37s	33.37s	29.5s	30.32
stdT	24.8s	22.4s	16.3s	15.14
avgNodes	11660	10047	8309	8161
avgQueries	579757	510213	457102	451164

1.9 Comparing every extension side-by-side

A general comparison table of the implementations. With the base algorithm being RRTConCon, that is built on two trees, it does little to no difference in what order we initiate the search. Therefore we only gathered the statistics from "forward" executions, i.e.: from start to goal.

	RRTConCon	SwapLimit	Heuristic	Exhaust	Bias	Gauss
avgT	39.37s	3.73s	31.29	54.22s	29.5s	29.0s
stdT	24.8s	2.50s	17.23s	57.1s	16.3s	19.0s
avgNodes	11660	1866	9700	12813	8309	9419
avgQueries	579757	45541	478535	604321	457102	481298

	CollisionSwap	NeighborCost	ControlChosen
avgT	21.56s	29.69s	23.29s
stdT	13.72s	22.9s	22.04s
avgNodes	10731.6	13371.7	12862.2
avgQueries	536918	658073	282697

2 Task distribution

Student	Impl.1.	Doc.1	Impl.2	Doc.2	Impl.3	Doc.3	Impl.4	Doc.4	Impl.5	Doc.5
Oliver H.	X	X			X	X			X	X
Gábor L.	X	X				X				X
Aditya M.			X	X			X	X		
Matteo M.			X	X			X	X		

Student	Impl.6	Doc.6	Impl.7	Doc.7
Oliver H.			X	X
Gábor L.				X
Aditya M.	X	X		
Matteo M.	X	X		