

# RRT-blossom: RRT with a local flood-fill behavior

Maciej Kalisiak

Dept. of Computer Science, University of Toronto  
mac@dgp.toronto.edu

Michiel van de Panne

Dept. of Computer Science, University of British Columbia  
van@cs.ubc.ca

**Abstract**—This paper proposes a new variation of the RRT planner which demonstrates good performance on both loosely-constrained and highly-constrained environments. The key to the planner is an implicit flood-fill-like mechanism, a technique that is well suited to escaping local minima in highly constrained problems. We show sample results for a variety of problems and environments, and discuss future improvements.

## I. INTRODUCTION

Path and motion planning has been a much studied problem over the past two decades, whose appeal stems from its applicability to many diverse areas, spanning industrial robot locomotion, object manipulation, autonomous actors in computer animation, and even protein folding and drug design. In the past decade two approaches have become eminently popular: Probabilistic Roadmaps (PRMs) [1] and Rapidly-expanding Random Trees (RRTs) [2], both of which are based on stochastic search strategies.

Despite their robustness, these methods can perform poorly in environments which have narrow passages or are otherwise highly constrained, such as the problem shown in Figure 7(c). This is an incidental characteristic since highly constrained environments should in practice present a simpler problem, given how adding constraints generally reduces the search-space. The most difficult motion planning problems should be the ones that are neither highly-constrained, nor highly under-constrained, but rather somewhere in the middle of these two extremes. A loose parallel can be drawn with observations about the satisfiability (SAT) problem and other NP-hard problems, where many such problems can be summarized by at least one order parameter and that the hard problems occur at a critical value of such a parameter [3], [4], resulting in an “easy-hard-easy” difficulty curve.

This paper presents a planner that performs well in highly constrained environments, while retaining RRT’s robust performance in highly under-constrained problems. The proposed modifications to the RRT algorithm give rise to a local, “on demand” flood-fill behavior. Only the key, bottleneck areas of free-space are explored exhaustively, rather than all of the free-space, from the starting point outwards, as would be the case in a naive flood-fill method.

The remainder of the paper is organized as follows. Section II discusses prior work in RRTs. The bulk of the paper, section III frames RRTs as potential field planners, and presents our algorithm. The remaining sections lay out our results (§IV), discuss them (§V), and identify future directions (§VI).

## II. PREVIOUS WORK

The seminal form [5] of RRT grows a single tree from the initial configuration,  $x_{init}$ , until one of its branches encounters  $x_{goal}$ , the goal state. Due to the method of its construction, this tree possesses a very useful property which accounts for its “rapidly-exploring” trait: as the tree grows, newly added edges never regress into already explored space. Algorithm 1 presents pseudocode for this most basic variant. Because “stumbling” onto  $x_{goal}$  is not a very efficient way to solve the query, a common extension is to bias RRT’s growth. That is, a certain portion of iterations (5%–10% is common), use  $x_{goal}$  as the target towards which the tree is grown, rather than  $x_{rand}$ , the usual randomly-chosen point. This produces better results because a tree branch can quickly close the distance to goal anytime the immediate, direct path to it is locally unobstructed.

A further extension is given by RRT-Connect [6]. Here, once the most target-bearing direction is selected for the iteration, it is applied for as many time steps as possible, stopping only on a collision, or when the target is reached. This greedy approach frequently performs better since any relatively open and unobstructed regions are traversed in a single iteration.

Another important idea, as explored in [2], [6], [7], is to use *two* trees, rooted at  $x_{init}$  and  $x_{goal}$ . The most common subvariant, outlined in Algorithm 2, grows one tree toward a random state, as before, while the other tree is grown towards any such new growth of its counterpart, with the trees switching roles between iterations. The  $x_{goal}$  tree needs to be grown using reverse-time simulation in systems where direction of time matters (e.g., kinodynamic systems). As with the single-tree RRT algorithm, edges are created using either the Extend (single time step) or the Connect (maximal time steps) operation. With two trees there are 4 possible growth combinations: ExtExt, ExtCon, ConExt, and ConCon. In practice ExtCon is often favored, especially in mildly constrained terrains, due to its greedy nature in searching for a tree connection (and hence a solution), but frugal approach while exploring, resulting in a conservative escape mechanism for local minima, as we will see later.

A useful extension proposed in [8], which we will refer to as *RRT-CT* (RRT with Collision Tendency), advocates keeping track of unsuccessful edge expansions and then exploiting this information. First, this is used to prevent further (redundant) expansion attempts of failed edges, a significant weakness of the original RRT algorithms. Second, this information is

---

**Algorithm 1** single-tree RRT (without biasing)

---

```
1: function QUERY( $x_{init}, x_{goal}$ )
2:    $\tau \leftarrow \text{tree}(x_{init})$ 
3:   while time_elapsed() < MAX.TIME do
4:      $x_{rand} \leftarrow \text{random\_state}()$ 
5:      $x_{new} \leftarrow \text{grow\_tree}(\tau, x_{rand})$ 
6:     if  $x_{new} \wedge \rho(x_{new}, x_{goal}) < \epsilon$  then
7:       return extract_soln( $x_{new}$ )
8:   return failure

9: function GROW_TREE( $\tau, x_{target}$ )
10:   $x_{near} \leftarrow \text{nearest\_neighbor}(\tau, x_{target})$ 
11:   $u_{best} \leftarrow \text{pick\_ctrl}(x_{near}, x_{target})$ 
12:  if  $u_{best}$  then
13:     $\tau \leftarrow \tau + \text{new\_edge}(x_{near}, u_{best})$ 
14:  return  $x_{new}$ 

15: function PICK_CTRL( $x, x_{target}$ )
16:   $d_{min}, u_{best} \leftarrow \rho(x, x_{target}), \emptyset$ 
17:  for  $u \in \mathcal{U}$  do
18:     $x_{new} \leftarrow \text{sim}(x, u)$ 
19:    if failure( $x, u, x_{new}$ ) then
20:      next  $u$ 
21:     $d \leftarrow \rho(x_{new}, x_{target})$ 
22:    if  $d < d_{min}$  then
23:       $d_{min}, u_{best} \leftarrow d, u$ 
24:  return  $u_{best}$ 
```

where

- $\rho(x_1, x_2)$ : distance metric
  - $\text{extract\_soln}(\dots)$ : constructs solution by travelling up the tree, from given node(s) to corresponding root.
  - $\text{new\_edge}(x, u)$ : create new edge from state  $x$  using control input  $u$  for a single (Extend) or maximal (Connect) number of time steps
  - $\text{failure}(x_1, u, x_2)$ : test whether the transition from  $x_1$  to  $x_2$ , using control input  $u$ , incurs a collision or violates other global constraints
  - $\text{sim}(x, u)$ : compute state of agent after application of control input  $u$  from starting state  $x$  (paper assumes a constant time step)
- 

further leveraged to bias the selection of the next node to expand, favoring nodes with lower “collision tendency”, the approximate probability of failure<sup>1</sup> in the progeny of the node. For details of RRT-CT please refer to Algorithm 3.

More recently, Strandberg [9] addressed the constrained environment problem by proposing the addition of “local trees” to the primal ones of the base RRT algorithm. With this approach, anytime the planner encounters a target state towards which extant trees cannot be grown, it uses the state to seed a new, *local* tree, one which then takes equal part in the usual process of growth toward random targets and neighboring trees. Whenever two branches meet, their trees are merged, while a connection between the primal trees signals discovery of a solution. The ideas presented in our work are orthogonal to the idea of using local trees; it is likely that they could be combined for maximum benefit.

Other variations of RRT have also been explored. Carpin & Pagello [10] show how the typically high variance of RRT query times can be exploited through parallel execution on

<sup>1</sup>We will often use the terms *failure* and *collision* interchangeably to mean the violation of global constraints; the avoidance of collision with the terrain is the canonical global constraint, but often there may be others, such as the subject remaining upright and controllable.

---

**Algorithm 2** dual-tree RRT (RRTextExt, etc.)

---

```
1: function QUERY( $x_{init}, x_{goal}$ )
2:   $\tau_a, \tau_b \leftarrow \text{tree}(x_{init}), \text{tree}(x_{goal})$ 
3:  while time_elapsed() < MAX.TIME do
4:     $x_{rand} \leftarrow \text{random\_state}()$ 
5:     $x_a \leftarrow \text{grow\_tree}(\tau_a, x_{rand})$ 
6:    if  $x_a$  then
7:       $x_b \leftarrow \text{grow\_tree}(\tau_b, x_a)$ 
8:      if  $x_b$  then
9:        if  $\rho(x_a, x_b) < \epsilon$  then
10:         return extract_soln( $x_a, x_b$ )
11:     $\tau_b, \tau_a \leftarrow \tau_a, \tau_b$ 
12:  return failure
```

(inherits  $\text{grow\_tree}()$  &  $\text{pick\_ctrl}()$  from single-tree RRT)

---

multiple processors. Urmson & Simmons [11] look at heuristically biasing RRT growth to obtain better quality solutions, or less costly ones in variable cost domains. Lindemann & LaValle [12] look at biasing tree growth by favoring expansions that reduce its dispersion. Finally, Yershova, Jaillet, *et al.* [13], [14] address the equally important RRT problem of handling environments with hard-to-find gaps and openings.

### III. RRT-BLOSSOM

The ideas behind RRT-blossom are best exposed by first framing RRT as a potential field planner. The central mechanism of such planners is the use of gradient descent over a “potential field”, typically an approximation of the distance to goal that has been in some way modulated by the presence of obstacles. Unfortunately such methods are frequently susceptible to getting trapped in local minima, thus requiring inclusion of some form of an escape mechanism. A typical example of this approach is RPP [15]. Its potential field is computed by performing a discrete, obstacle-aware flood-fill from  $x_{goal}$ , in which the potential value of a discrete location is the iteration number on which it was reached, while a simple random-walk serves as the escape mechanism.

RRT shares some basic ideas with potential field planners when its operation is viewed on a macro scale. The  $x_{goal}$ -biased iterations effect a descent down a potential field, viz.  $\rho(x, x_{goal})$ , while the tree growth towards random points in state-space acts as a local minima escape mechanism.

The dual-tree variant of RRT can be likewise analyzed by viewing it in terms of a simple finite state machine, as shown in Figure 1. In Algorithm 2, lines 4–5 make up the EXPLORE mode, which behaves like an escape mechanism, while line 7 constitutes the SEEK mode, which corresponds to gradient descent. The overall potential field planner parallel here is similar to the single-tree case, with the difference that the

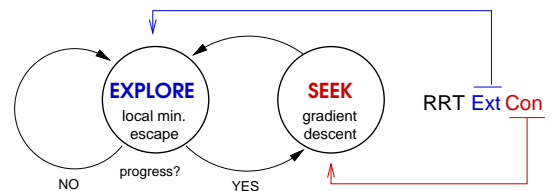


Fig. 1. dual-tree RRT as a simple finite state machine (FSM)

---

**Algorithm 3** RRT w/“Collision Tendency” (RRT-CT)

---

(inherits `query()` & `grow_tree()` from single- or dual-tree RRT)

```
1: function NEAREST_NEIGHBOR( $\tau, x$ )
2:    $d_{min}, n_{best} \leftarrow \infty, \emptyset$ 
3:   for  $n \in \tau$  do
4:     if  $\exists$  unexpanded input out of node  $n$  then
5:        $r \leftarrow \text{random}() \mid r \in [0, 1]$ 
6:       if  $r > \sigma(n)$  then
7:          $d \leftarrow \rho(n, x)$ 
8:         if  $d < d_{min}$  then
9:            $d_{min}, n_{best} \leftarrow d, n$ 
10:  return  $n_{best}$ 

11: function PICK_CTRL( $x, x_{target}$ )
12:   $d_{min} \leftarrow \infty$ 
13:  for  $u \in \mathcal{U}$  do
14:    if  $u$  has not been expanded for  $x$  then
15:       $x_{new} \leftarrow \text{sim}(x, u)$ 
16:      if failure( $x, x_{new}$ ) then
17:        mark  $u$  as expanded
18:        update_treeinfo( $x, \tau$ )
19:      else
20:         $d \leftarrow \rho(x, x_{new})$ 
21:        if  $d < d_{min}$  then
22:           $d_{min}, u_{best} \leftarrow d, u$ 
23:  mark  $u_{best}$  as expanded
24:  return  $u_{best}$ 

25: function UPDATE_TREEINFO( $x, \tau$ )
26:   $p \leftarrow 1$ 
27:  while  $x$  do
28:     $p \leftarrow p / |\mathcal{U}|$ 
29:     $\sigma(x) \leftarrow \sigma(x) + p$ 
30:     $x \leftarrow \text{parent}(x)$ 
```

where

- $\sigma(n)$ : collision tendency of node  $n$
- 

potential fields encourages the trees to grow towards each other rather than growing towards a fixed goal state.

Finally, we can think of RRT-CT in the potential-field planner context, where it introduces changes that result in a behavior akin to a flood-fill of a local minimum. A significant part of RRT-CT’s power comes from a modification to the control input picking mechanism, viz. initializing<sup>2</sup>  $d_{min}$  to  $\infty$ , rather than  $\rho(n_{parent}, x_{goal})$ . The main consequence of this change is that it allows the creation of edges which *recede* from their appointed target. At first this may seem undesirable, as many such edges will regress into space already explored by the tree, but this is not always the case (e.g., Figure 2). Receding edges which do not regress are highly beneficial since they provide tree growth in iterations which would otherwise be wasted, thus generating a more sustained rate of node creation, an important characteristic for a flood-fill. Unfortunately RRT-CT has no mechanism to separate out the non-regressing edges, and accepts them all equally, leading to poor performance in regression-prone cases (e.g., kinematic systems). RRT-CT’s other, more explicit changes, namely the tracking of failed edge expansions and use of

<sup>2</sup>cf. line 12 in Algorithm 3 vs. line 16 in Algorithm 1

collision tendency, do not address this problem, although they do improve the planner’s node generation efficiency, which too is a required trait of a successful flood-fill mechanism.

#### A. RRT-blossom & regression avoidance

The main contribution of RRT-blossom is to introduce a robust, local, non-regressing flood-fill mechanism into RRT. This is done without systematically visiting the entire search space around the originating state, as might be the case with a typical level-set method. Algorithm 4 gives the pseudocode for RRT-blossom. It too allows for tree expansions that recede from the target, but these expansions are subject to a constraint that prevents regression. A secondary improvement we add is the instantiation of *all* eligible edges when expanding a node<sup>3</sup> (subject to collision and regression constraints), not just the single best one. Although RRT-CT handles unsuccessful expansions efficiently, by ensuring they are never re-attempted, it is wasteful with regards to the others. Specifically, after it surveys the expansions out of a node and instantiates one of them, it then discards useful information for the remaining *collision-free* expansions, such as their end-states and collision-free status. Instantiating all eligible expansions mitigates this, and has little negative cost: in constrained regions these expansions would eventually be instantiated anyhow, while expansive spaces are traversed quickly with few node expansions, thus generating only negligible overhead. Also, such blossoming is consistent with RRT’s “rapidly-exploring” spirit.

Implementing a robust regression constraint is challenging and is another contribution of this paper. Computing an explicit model of the portion of state-space that is considered already explored is unwieldy because of issues of dimensionality, and it is furthermore difficult to define the volume of state-space that is “occupied” by a branch of a search tree. We sidestep these difficulties by using an implicit approximation that captures the desired spirit of the term, an approximation which we have found to be highly effective: a “leaf” edge ( $n_{parent}, n_{leaf}$ ) is considered to be regressing if a node *other* than  $n_{parent}$  is closest to  $n_{leaf}$ :

$$\text{regression} : \exists n \in \tau \mid \rho(n, n_{leaf}) < \rho(n_{parent}, n_{leaf})$$

Figure 3 further illustrates the concept.

<sup>3</sup>Hence the “blossom” moniker.

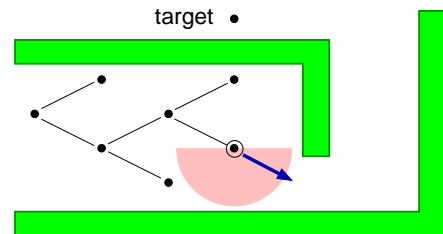


Fig. 2. An example of a receding-yet-useful expansion. The blue arrow indicates the useful expansion, while the pink half-disc shows the receding-from-target directions for the corresponding node.

---

**Algorithm 4** RRT-blossom
 

---

```

(inherits query() from dual-tree RRT)

1: function GROW_TREE( $\tau, x_{target}$ )
2:    $x_{near} \leftarrow \text{nearest\_neighbor}(\tau, x_{target})$ 
3:    $x_{new} \leftarrow \text{node\_blossom}(x_{near}, x_{target}, \tau)$ 
4:   return  $x_{new}$ 

5: function NODE_BLOSSOM( $x, x_{target}, \tau$ )
6:   for  $u \in \mathcal{U}$  do
7:      $x_{new} \leftarrow \text{sim}(x, u)$ 
8:     if  $\text{failure}(x, u, x_{new})$  then
9:       next  $u$ 
10:    if  $\text{regression}(x, x_{new}, \tau)$  then
11:      next  $u$ 
12:     $\tau \leftarrow \tau + \text{new\_edge}(x, u)$ 
13:  return the new node closest to  $x_{target}$ 

14: function REGRESSION( $x_{parent}, x_{new}, \tau$ )
15:  for node  $n \in \tau$  do
16:    if  $\rho(n, x_{new}) < \rho(x_{parent}, x_{new})$  then
17:      return True
18:  return False
  
```

---

### B. Interplay of viability & regression constraint

The above definition works well for simple kinematic systems, but it becomes problematic for nonholonomic or kinodynamic systems. Figure 4 illustrates one such case. Here, the leftward path of the car is expanded first, but all its follow-on paths lead to collisions. The ‘straight forward’, middle path, on the other hand, would be feasible, but unfortunately it is disallowed because instantiating it would form a “regression” into space already explore by the leftward path.

The problem, and the solution, are best framed in terms of *viability* [16], [17], although in the planning context it is necessary to first widen the concept. Specifically, a *viable* state is taken to mean one from which the system can evolve indefinitely, *or* one that can reach  $x_{goal}$  prior to failure. In this framework then, the problem is that it is possible for a nonviable edge, one that by definition cannot be part of any solution trajectory, to block a neighboring, viable expansion. This is particularly detrimental when the blocked expansion lies on the critical path, as this then eliminates all chance of finding a solution, as shown in Figure 4.

The solution then is to disregard nonviable nodes and edges

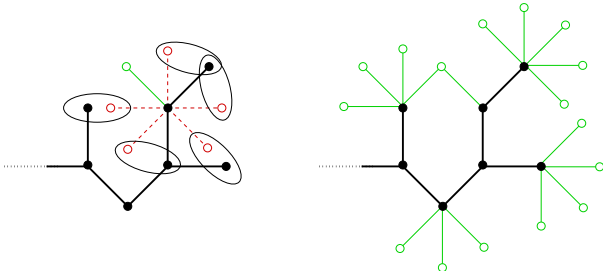


Fig. 3. “Regression”; The left subfigure shows the possible expansions for a particular node; all the red expansions (dashed) are *regressing* since a foreign node is closer than the parent (indicated with loops). Only the single edge in green is suitable for instantiation. The right subfigure shows all the expansions in the tree that do *not* regress for the depicted tree state.

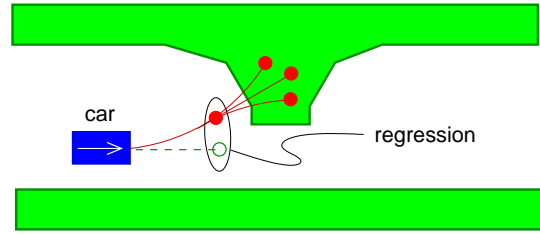


Fig. 4. Interplay of viability and the regression constraint: the green (dashed) expansion is blocked by an extant nonviable edge, since instantiating it would constitute a regression. As the green edge is essential to any solution, it is now impossible for the planner to succeed.

in the regression test. The definition is thus extended as follows:

$$\begin{aligned}
 \text{regression} : \exists n \in \tau \quad & | \quad \rho(n, n_{new}) < \rho(n_{parent}, n_{new}) \\
 & \wedge n \notin \text{NonViable}
 \end{aligned}$$

where *NonViable* is the set of nodes found to be nonviable.

Unfortunately the viability of edges is not known ahead of time, and thus must be incorporated retroactively as it is discovered. This is achieved through a method reminiscent of reinforcement learning or dynamic programming. Each edge, instantiated or potential, carries a viability status, one of: *untried*, *live*, *dormant*, or *dead*. Edges that have not yet been considered for expansion are marked *untried*. Upon instantiation they become *live*. If the expansion is disallowed due to regression, it is marked *dormant*. Finally, *dead* edges are ones that have been found to have left viable space. Figure 5 gives a fuller description of the transitions.

Since changing the status of an edge *may* precipitate a status change in the parent, status changes must be propagated. This is done by traveling up the parent hierarchy towards the root node, re-evaluating the status of each edge passed. The process stops when root node is reached, or when a re-evaluation results in no change of status.

### C. dormant deadlock

Despite these measures it is still possible for the planner to become unduly stuck. This occurs when all paths towards the goal, usually narrow chokepoints, have been cut off by older, lengthier branches, ones which cannot explore into the passages even though they pass the closest to them. Figure 6 shows an example.

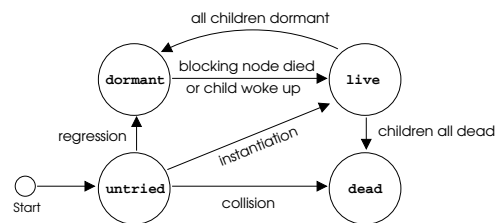


Fig. 5. Progression of the viability status of an edge

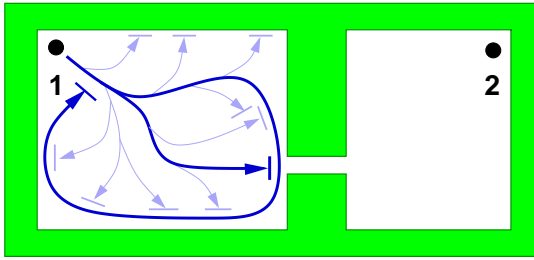


Fig. 6. *Dormant deadlock*: a viable branch may cutoff access to a critical passage without being able to explore it itself. This limits the planner’s exploration to the “fenced off” area, and once this is exhausted, the remaining non-dead branches are locked in a cycle, mutually blocking each other’s way.

Fortunately these occasions are easily identified and mitigated. When the remaining accessible free-space has been exhausted, the `dormant` condition starts “backing up” the tree towards the root node, eventually reaching it once all other lines of exploration have been exhausted. This signals the deadlock to the planner, which then allows the very next expansion attempt to disregard the regression constraint, thereby breaking the deadlock. In practice this works well, and more importantly, it has no impact on queries unaffected by this issue. Nonetheless there is room for improvement, since in particularly unfavourable cases, where the cycle or blockade occurs early on while much unexplored free-space remains, the planner will spend much time unnecessarily exhausting the free-space before invoking the special countermeasures.

#### IV. RESULTS

Figures 7 & 9 illustrate some of the problem environments in which queries were executed using a holonomic point agent. These environments were also used in queries for a nonholonomic car and a kinodynamic bike, although with slight alterations. In particular the “door jams” were removed from the gaps in “rooms”, while “tunnel” was widened and scaled up to accommodate the turning radii of the subjects. In the diagrams, the points labelled “1” and “2” mark  $x_{init}$  and  $x_{goal}$ , respectively; for car and bike queries the vehicle’s orientation at both endpoints is “facing right”. The bike, which is the most complex agent we have used, has a 5D state vector:  $(x, y, \theta, \psi, \psi')$ , where  $\theta$  is the bike’s bearing, and  $\psi$  is the bike’s lateral lean. Its control input is the steering angle (forward velocity is fixed); steering is simultaneously used to maintain balance as well as effect progress.

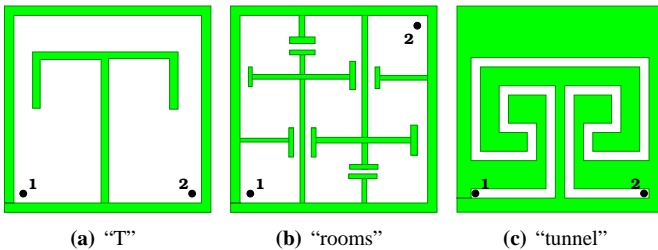


Fig. 7. Some of the problem environments used

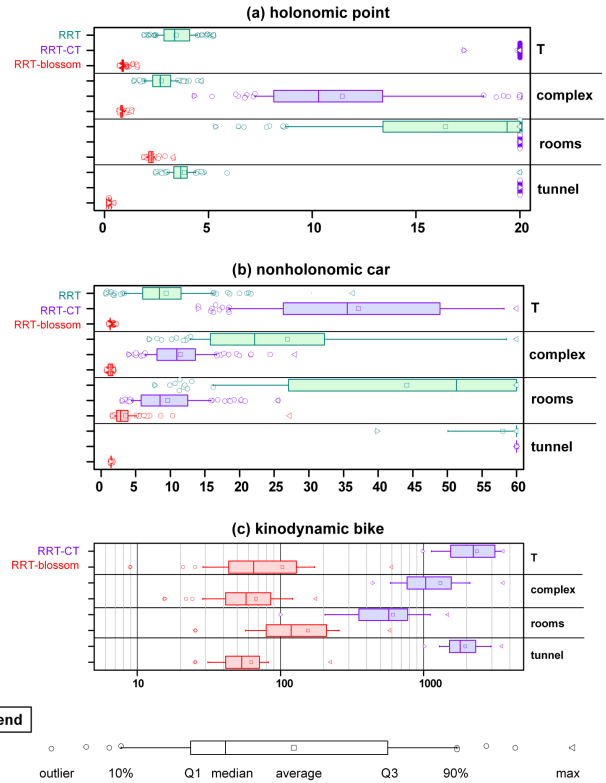


Fig. 8. Algorithm runtimes, in seconds: (a) holonomic point ( $|U| = 8$ ); (b) nonholonomic car ( $|U| = 3$ ); (c) kinodynamic bike ( $|U| = 5$ ). Samples per boxplot: point & car = 100, bike = 40.

The boxplots in Figure 8 illustrate the timing results for various agents and terrains. To further ground the comparisons made, Tables I, II and III give average runtimes, number of collision checks, nearest neighbor checks, and number of nodes created; last column gives the number of (time-limited) runs which failed to find a solution. Each datum is the average over the indicated number of runs performed. It should be noted that the data in “NN” columns does *not* include evaluations of the regression constraint, since the latter is computed using a cheaper method. Rows shown in *italics* indicate test cases for which a significant portion of the runs did not find a solution within the specified time limit (these samples were wholly excluded from the averages); thus those averages represent significant underestimates of the true costs.

Figure 9, on the other hand, shows sample evolved tree structures for the three algorithms. Of particular note is how RRT-blossom’s structure is strikingly regular in the kinematic case, and how RRT-CT tends to spend a lot of time filling the minimums with many redundant edges. As discussed later, RRT is unable to make any progress with a bike.

All algorithms were written in Python 2.3, running on Linux (Debian “sid”, kernel 2.6), using Psyco (a JIT-like optimization for Python), on a Pentium IV 2.4 GHz machine. In the following “RRT” refers to RRTextCon, while “RRT-CT” refers to RRTextExt w/CT. All implementations share the same component functions where feasible.

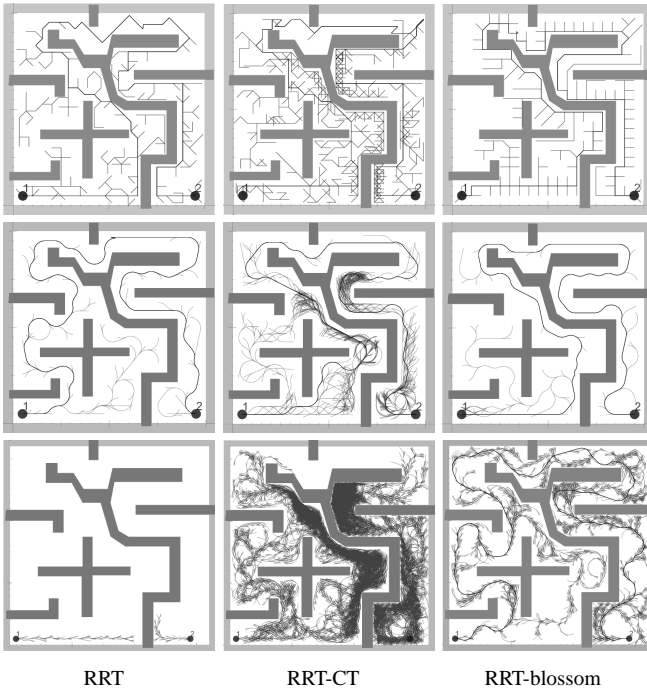


Fig. 9. Comparison of evolved tree structure in “complex” environment; **top**: holonomic point; **middle**: nonholonomic car; **bottom**: kinodynamic bike.

## V. DISCUSSION

The problem environments were chosen to present deep local minima (“T”), to be highly constrained (“tunnel”), or to offer mixes of these qualities (“complex” and “rooms”). RRT-blossom outperforms both RRT and RRT-CT in all of these scenarios, often by an order of magnitude. In the holonomic case RRT-CT’s poor performance stems from the ease with which self-negating expansions are made ( $\mathcal{U}$  often contains complementary pairs of control inputs, where one undoes the displacement of the other). Without regression-prevention RRT-CT succumbs to a back-and-forth chase around local minima, and only a lucky but rare  $x_{rand}$  can pull it out. In the nonholonomic case the deep local minima of “T” prove again to be a problem for RRT-CT for similar reasons, but otherwise it outperforms RRT as expected. “Tunnel” proves particularly difficult for both RRT and RRT-CT, as their EXPLORE modes are hampered by random-target distributions that are often directionally-biased<sup>4</sup>. The bike queries are nigh impossible for RRT, since it tends to quickly evolve the trees such that the most prominent nodes, the ones most often chosen for expansion, are already nonviable, and the usual  $L_2$  distance metric tends to only pull tree growth towards other such nodes. Since not a single RRT query ever made any significant progress, let alone complete, we have not included it in the kinodynamic table and plot. RRT-CT fares better, but it still carries an exorbitant cost in time and number of nodes required. RRT-blossom, on the other hand, performs well in all cases, and appears to follow the hypothesized ascending-

<sup>4</sup> $x_{rand}$  is chosen uniformly from the state-space, but for off-center nodes, especially ones close to the edges of the terrain, this translates to a skewed distribution of growth directions, from the node’s point of view.

HOLONOMIC POINT:  $|\mathcal{U}| = 8$ , averages over 100 runs,  $\max\_time = 20s$

terrain	algorithm	time	failure()	NN	nodes	time-outs
T	RRT	3.45	21,100	2628	410	—
	RRT-CT	19.06	13,250	2870	2870	97
	RRT-blossom	0.90	2246	280	316	—
complex	RRT	2.75	10,048	1247	281	—
	RRT-CT	10.90	8858	1889	1889	6
	RRT-blossom	0.85	1767	221	266	—
rooms	RRT	13.10	39,398	4911	621	48
	RRT-CT	N/A	N/A	N/A	N/A	100
	RRT-blossom	2.25	3276	409	499	—
tunnel	RRT	3.68	22,080	2754	122	1
	RRT-CT	N/A	N/A	N/A	N/A	100
	RRT-blossom	0.21	944	118	118	—

TABLE I

NONHOLONOMIC CAR:  $|\mathcal{U}| = 3$ , averages over 100 runs,  $\max\_time = 60s$

terrain	algorithm	time	failure()	NN	nodes	time-outs
T	RRT	9.39	13,317	4407	486	—
	RRT-CT	35.13	8890	3848	3585	8
	RRT-blossom	1.36	1343	451	448	—
complex	RRT	23.62	13,656	4542	294	9
	RRT-CT	11.42	4049	1677	1465	—
	RRT-blossom	1.39	811	295	267	—
rooms	RRT	32.62	27,119	9018	724	42
	RRT-CT	9.59	4071	1717	1507	—
	RRT-blossom	3.53	1967	644	649	—
tunnel	RRT	51.27	24,917	8281	408	77
	RRT-CT	N/A	N/A	N/A	N/A	100
	RRT-blossom	1.43	806	277	266	—

TABLE II

KINODYNAMIC BIKE:  $|\mathcal{U}| = 5$ , averages over 40 runs,  $\max\_time = 3600s$

terrain	algorithm	time	failure()	NN	nodes	time-outs
T	RRT-CT	2359.45	163,632	43,440	29,963	7
	RRT-blossom	103.02	34,054	8538	5808	—
complex	RRT-CT	1310.84	129,546	34,140	22,604	1
	RRT-blossom	67.49	27,368	7088	4675	—
rooms	RRT-CT	609.06	83,549	22,094	14,949	—
	RRT-blossom	154.90	43,822	11,049	7461	—
tunnel	RRT-CT	1957.93	194,767	51,322	31,952	—
	RRT-blossom	62.65	28,744	7476	4903	—

TABLE III

then-descending difficulty curve. It is interesting to note that it also generally tends to have a much smaller runtime variance, which is desirable [18].

Although not shown for lack of space, we have found RRT-blossom to perform well in easier, less constrained environments, on par with whichever of RRT or RRT-CT is faster in a given case.

In the current implementation of RRT-blossom performance starts to suffer as the number of nodes in the tree gets large, as may happen with more expansive yet still difficult environments. This is due to the naive implementation of `regression()`, which is  $O(n)$  in the number of nodes. We expect significant improvement from re-implementing it using [19]. An even cheaper approximation we are considering is to exploit the already collected information in the `nearest_neighbor()` call in `grow_tree()`: one could collect the  $k$  nearest neighbors of  $x_{target}$ , and then assess regression by checking only against this set; aside from  $n_{parent}$ , the set will also contain most of its closest neighbor nodes, meaning that in case of a regression there is a good chance the offending node will be detected.

## VI. CONCLUSION

In this paper we presented a novel variation of the RRT algorithm, one that performs well in constrained environments. Its core ideas are to allow creation of receding edges, the addition of a regression-prevention mechanism, and the instantiation of all allowable control actions out of the active node.

In the future we plan to replace the naive implementation of the regression check in order to enhance the algorithm's performance for large trees. Performance in higher dimensional problems also deserves further investigation, although it is expected that this will merely result in larger trees, in proportion to the attendant increase of free-space volume. The filling of deep local minima could likely be improved by further borrowing the "collision tendency" concept from RRT-CT. Finally, it would be interesting to gauge the combined performance of RRT-blossom with local trees.

## REFERENCES

- [1] L. Kavraki, P. Švestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration space," in *IEEE Trans. on Robotics & Automation*, vol. 12(4), 1996, pp. 566–580.
- [2] S. M. LaValle and J. J. Kuffner, Jr., "Rapidly-exploring random trees: Progress and prospects," in *Workshop on Algorithmic Foundations of Robotics*, 2000.
- [3] P. Cheeseman, B. Kanefsky, and W. M. Taylor, "Where the really hard problems are," in *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence*, 1991, pp. 331–337.
- [4] B. Selman, D. G. Mitchell, and H. J. Levesque, "Generating hard satisfiability problems," *Artificial Intelligence*, vol. 81, no. 1–2, pp. 17–29, 1996.
- [5] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Computer Science Dept., Iowa State University, technical report TR 98-11, 1998.
- [6] J. J. Kuffner, Jr. and S. M. LaValle, "RRT-Connect: An efficient approach to single-query path planning," in *IEEE Int. Conf. on Robotics & Automation*, 2000.
- [7] S. M. LaValle and J. J. Kuffner, Jr., "Randomized kinodynamic planning," in *IEEE Int. Conf. on Robotics & Automation*, 1999.
- [8] P. Cheng and S. M. LaValle, "Reducing metric sensitivity in randomized trajectory design," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2001.
- [9] M. Strandberg, "Augmenting RRT-planners with local trees," in *IEEE Int. Conf. on Robotics & Automation*, 2004, pp. 3258–3262.
- [10] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," *Proc. 8th Conf. Italian Association for Artificial Intelligence*, pp. 834–841, 2002.
- [11] C. Urmson and R. Simmons, "Approaches for heuristically biasing RRT growth," *Proc. IEEE Intl. Conf. on Intelligent Robots and Systems*, pp. 1178–1183, 2003.
- [12] S. R. Lindemann and S. M. LaValle, "Incrementally reducing dispersion by increasing Voronoi bias in RRTs," *Proc. of IEEE Intl. Conf. on Robotics & Automation*, pp. 3251–3257, 2004.
- [13] A. Yershova, L. Jaillet, T. Siméon, and S. M. LaValle, "Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain," *Proc. IEEE Intl. Conf. on Robotics & Automation*, 2005.
- [14] L. Jaillet, A. Yershova, S. M. LaValle, and T. Siméon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs," *Proc. of the IEEE Int. Conf. on Robots and Systems*, 2005.
- [15] J. Barraquand and J.-C. Latombe, "Robot motion planning: A distributed representation approach," *The International Journal of Robotics Research*, vol. 10(6), pp. 628–649, 1991.
- [16] J.-P. Aubin, *Viability Theory*, ser. Systems & Control: Foundations & Applications, C. I. Byrnes, Ed. Birkhäuser, 1991.
- [17] M. Kalisiak and M. van de Panne, "Approximate safety enforcement using computed viability envelopes," in *IEEE Int. Conf. on Robotics & Automation*, vol. 5, 2004, pp. 4289–4294.
- [18] P. Isto, M. Mäntylä, and J. Tuominen, "On addressing the run-cost variance in randomized motion planners," *Proc. IEEE Int. Conf. on Robotics & Automation*, 2003.
- [19] A. Atramentov and S. M. LaValle, "Efficient nearest neighbor searching for motion planning," *Proc. IEEE Int. Conf. on Robotics & Automation*, 2002.