To appear in: proceedings of International Conference on Robotics & Automation 2006, Orlando, Florida

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 risen to the forefront: 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 underconstrained, 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 underconstrained problems. The proposed modifications to the RRT algorithm give rise to a local, "on demand" flood-fill behavior. Only the key, bottleneck areas of freespace are explored exhaustively, rather than all of the freespace 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 posesses 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 timesteps 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 timestep) or the Connect (maximal timesteps) 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 further leveraged to bias the selection of the next node to

Algorithm 1 single-tree RRT (without biasing)

```
1:  function QUERY(x_init, x_goal)
2:      τ ← tree(x_init)
3:      while time_elapsed() < MAX_TIME do
4:          x_rand ← random_state()
5:          x_new ← grow_tree(τ, x_rand)
6:          if x_new ∧ ρ(x_new, x_goal) < ε then
7:              return extract_soln(x_new)
8:      return failure

9:  function GROW_TREE(τ, x_target)
10:     x_near ← nearest_neighbor(τ, x_target)
11:     u_best ← pick_ctrl(x_near, x_target)
12:     if u_best then
13:         τ ← τ + new_edge(x_near, u_best)
14:     return x_new

15: function PICK_CTRL(x, x_target)
16:     d_min, u_best ← ρ(x, x_target), ∅
17:     for u ∈ U do
18:         x_new ← sim(x, u)
19:         if failure(x, u, x_new) then
20:             next u
21:         d ← ρ(x_new, x_target)
22:         if d < d_min then
23:             d_min, u_best ← d, u
24:     return u_best
```

where

- $\rho(x_1, x_2)$: distance metric
- `extract_soln(...)`: constructs solution by travelling up the tree, from given node(s) to corresponding root.
- `new_edge`(x, u): create new edge from state x using control input u for a single (Extend) or maximal (Connect) number of timesteps
- `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
- `sim`(x, u): compute state of agent after application of control input u from starting state x (paper assumes a constant timestep)

expand, favouring nodes with lower "collision tendency", the approximate probability of failure[1] 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 random target state towards which extant trees cannot be grown, a *local* tree is created, which then takes equal part in the usual process of growth toward random targets and neighbouring 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 maximal benefit.

Other variations of RRT have also been explored. Carpin & Pagello [10] show how the typically high variance of RRT query times can be tempered through parallel execution on multiple processors. Urmson & Simmons [11] look at

[1]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 falling over, or becoming otherwise uncontrollable.

Algorithm 2 dual-tree RRT (RRTExtExt, etc.)

```
1:  function QUERY(x_init, x_goal)
2:      τ_a, τ_b ← tree(x_init), tree(x_goal)
3:      while time_elapsed() < MAX_TIME do
4:          x_rand ← random_state()
5:          x_a ← grow_tree(τ_a, x_rand)
6:          if x_a then
7:              x_b ← grow_tree(τ_b, x_a)
8:              if x_b then
9:                  if ρ(x_a, x_b) < ε then
10:                     return extract_soln(x_a, x_b)
11:         τ_b, τ_a ← τ_a, τ_b
12:     return failure
```

(inherits `grow_tree()` & `pick_ctrl()` from single-tree RRT)

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 favouring 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} biasing serves the function of a simple potential field, computed as $\rho(x, x_{goal})$, while the tree growth towards random points in state space constitutes a form of exploration for escaping local minima.

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

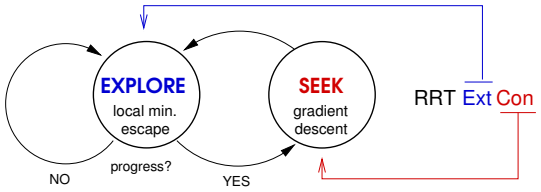


Fig. 1. dual-tree RRT as a simple finite state machine (FSM)

Algorithm 3 RRT w/"Collision Tendency" (RRT-CT)

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

1: **function** NEAREST_NEIGHBOR(τ, x)
2: $d_{min}, n_{best} \leftarrow \infty, \varnothing$
3: **for** $n \in \tau$ **do**
4: **if** $\exists$ unexpanded input out of node n **then**
5: $r \leftarrow$ `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: $x_{new} \leftarrow$ `sim(`x`,`u`)`
16: **if** `failure(`x`,`x_{new}`)` **then**
17: mark u as expanded
18: `update_treeinfo(`x`,`τ`)`
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,τ)
26: $p \leftarrow 1$
27: **while** x **do**
28: $p \leftarrow p/|\mathcal{U}|$
29: $\sigma(x) \leftarrow \sigma(x) + p$
30: $x \leftarrow$ `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[2] d_{min} to ∞, 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 growith 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

[2]cf. line 12 in Algorithm 3 vs. line 16 in Algorithm 1

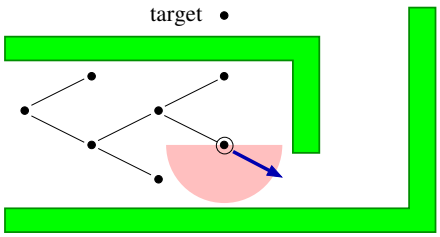


Fig. 2. An example of a receding-yet-useful expansion. The blue arrow indicates the useful expansion, while the pink half-disc shows the receding-from-target directions for the corresponding node.

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 visiting the entire search space, as would happen in the case of a level-set potential field method applied to the full state-space. Algorithm 4 gives the psuedocode 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[3] (subject to collision and regression constraints), not just the single best one. Although RRT-CT handles unsuccessful expansions efficiently by ensuring they are never reattempted, 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 giving only negligible overhead. Also, such blossoming is consistent with RRT's "rapidly-exploring" spirit.

Implementing a robust regression constraint is challenging and is an another contribution of this paper. Computing an explicit model of the portion of state-space that has been deemed to be previously-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}:

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

Figure 3 further illustrates the concept.

[3]Hence the "blossom" moniker.

Algorithm 4 RRT-blossom

(inherits `query()` from dual-tree RRT)

function GROW_TREE(τ,x_{target})
  $x_{near} \leftarrow$ `nearest_neighbor(`τ, x_{target}`)`
  $x_{new} \leftarrow$ `node_blossom(`$x_{near}, x_{target}, \tau$`)`
  return x_{new}

function NODE_BLOSSOM(x,x_{target},τ)
  for $u \in \mathcal{U}$ **do**
    $x_{new} \leftarrow$ `sim(`x, u`)`
    if `failure(`x, u, x_{new}`)` **then**
      next u
    if `regression(`x, x_{new}, τ`)` **then**
      next u
    $\tau \leftarrow \tau +$ `new_edge(`x, u`)`
  return the new node closest to x_{target}

function REGRESSION(x_{parent}, x_{new}, τ)
  for node $n \in \tau$ **do**
    if $\rho(n, x_{new}) < \rho(x_{parent}, x_{new})$ **then**
      return `True`
  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 attempted 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

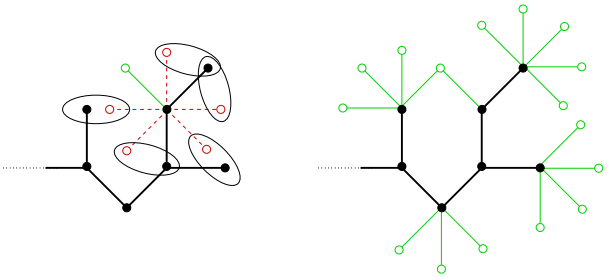


Fig. 3. "Regression"; The left subfigure shows the possible expansions for a particular node; all the red expansions (dashed) are *regressing* since a foreign node is closer than the parent (these occurrences are indicated with ellipses). Only the single edge in green is suitable for instantiation. The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

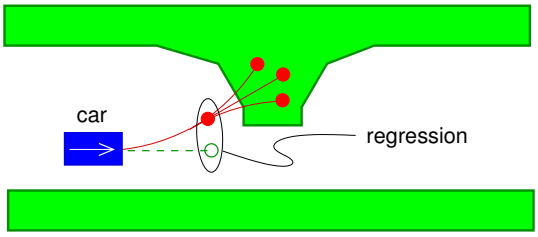


Fig. 4. Interplay of viability and the regression constraint: the green (dashed) expansion is blocked by an extant nonviable edge, since instantiating it would constitute a regression. As the green edge is essential to any solution, it is now impossible for the planner to succeed.

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

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

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 may be either due to a `dormant` dependency cycle, or due to an ancestor edge blocking a critical pathway which it itself cannot traverse. Figure 6 illustrates both cases.

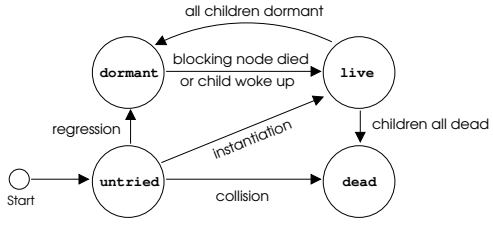


Fig. 5. Progression of the viability status of an edge

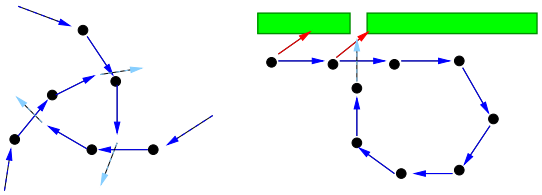


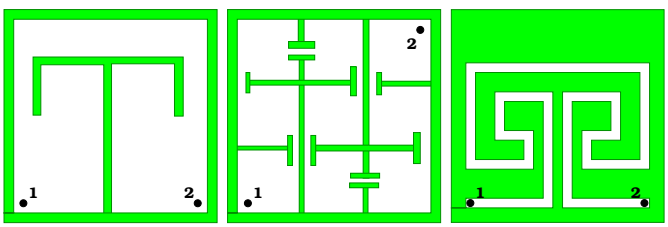
Fig. 6. Examples of where viability-tracking alone is insufficient to prevent planner from stalling. On the left, a dormant edge dead-lock; on the right, a critical expansion is blocked by an ancestor edge.

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 & 10 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 non-holonomic car and a kinodynamic bike, although with slight modifications. In particular the "door jambs" were removed from the gaps in "rooms", while "tunnel" was widened and scaled up to accomodate 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 θ is the bike's bearing, and ψ is the bike's lateral lean. Its control input is the steering angle (forward velocity is fixed). The bicycle needs to steer to maintain balance as well as to move in a desired direction.

All algorithms were written in Python 2.3, running on Linux 2.6 (Debian "sid"), 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 RRT-ExtExt w/CT. All implementations share the same component functions where feasible.



(a) "T" (b) "rooms" (c) "tunnel"

Fig. 7. Some of the problem environments used

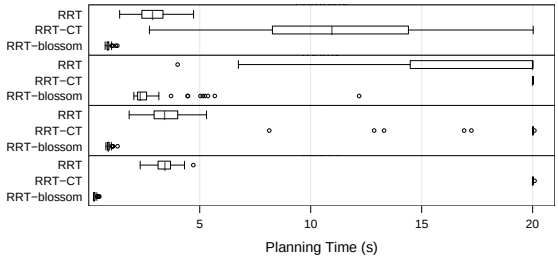


Fig. 8. Holonomic point, $|\mathcal{U}| = 8$; the boxplots, from top to bottom, are for: T, complex, rooms, tunnel.

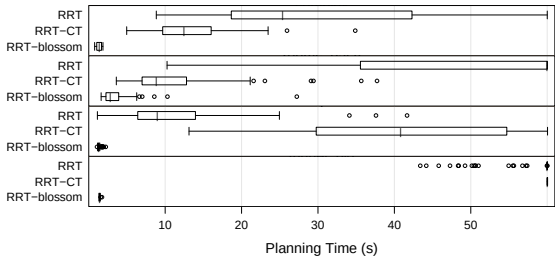


Fig. 9. Nonholonomic car, $|\mathcal{U}| = 3$; the boxplots, from top to bottom, are for: T, complex, rooms, tunnel.

The boxplots in Figures 8 and 9 illustrate the timing results obtained for the point and car agents, respectively. A boxplot for the bike was deemed of little value due to the sparse nature of the data. 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-limitted) 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; thus those averages represent significant underestimates of the true costs.

V. Discussion

The problem environments were chosen to present deep local minima ("T"), to be highly constrained ("tunnel"), as well as 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, or more. 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.

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*	*9014*	*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

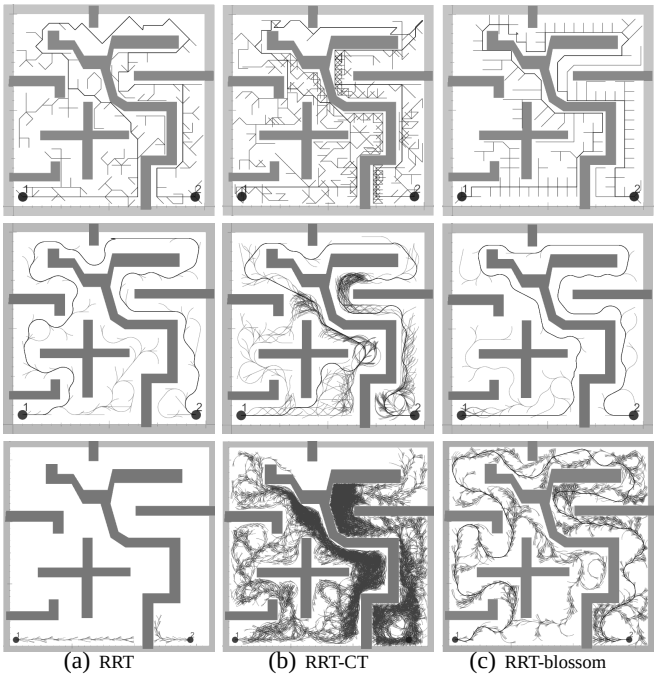
KINODYNAMIC BIKE: $|\mathcal{U}| = 5$, # runs=40 (RRT-blossom), =3 (RRT-CT)

terrain	algorithm	time	failure()	NN	nodes	time-outs
T	RRT-CT	1666.81	139,488	37,032	25556	—
	RRT-blossom	103.02	34,054	8538	5808	—
complex	RRT-CT	1215.39	128,572	33,894	22,424	—
	RRT-blossom	67.49	27,368	7088	4675	—
rooms	RRT-CT	345.85	64,570	17,034	11493	—
	RRT-blossom	154.90	43,822	11,049	7461	—
tunnel	RRT-CT	1536.77	182,082	47,973	29,879	—
	RRT-blossom	62.65	28,744	7476	4903	—

TABLE III

"Tunnel" proves particularly difficult here for RRT and RRT-CT as their EXPLORE modes are hampered by random target distributions that are often directionally-biased[4]. 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 tried for expansion, are already nonviable, and the naive distance metric tends to only pull tree growth further towards nonviable space. Since not a single RRT query made any significant headway, we have not included it in the results table. 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 quite well in all these cases, and does appear to follow the ascending-then-descending difficulty curve proposed earlier. It is interesting to note that it also generally tends to have a much smaller runtime variance, which is desirable [18].

[4] x_{rand} is chosen uniformly from the state-space, but for off-center nodes, especially ones closer to the edges of the terrain, this translates to a skewed distribution of growth directions, from the node's point of view.



(a) RRT (b) RRT-CT (c) RRT-blossom

Fig. 10. Comparison of evolved tree structure in "complex" environment; **top:** holonomic point query; **middle:** nonholonomic car query; **bottom:** kinodynamic bike query.

Although not shown for lack of space, we have found RRT-blossom to perform well in easier environments, on par with whichever of RRT or RRT-CT is faster in a given case, although it does become marginally slower for trivially constrained queries.

RRT-blossom's 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 solution we are considering is to exploit the information that can be cheaply collected during the `nearest_neighbor()` call in `grow_tree()`; for example one could collect the k nearest neighbors of x_{target}, and then assess regression by checking only against these. Naturally this is a weaker approximation of `regression()` presented earlier, but we expect the trade-off to be positive. A more general alternative would also be to investigate other methods of detecting and preventing regression.

Finally, in Figure 10 we show some 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 mentioned earlier, RRT has an extremely difficult time making any headway in the bike scenario.

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 an active node.

In the future we plan to replace the naive implementation of the regression check, to enhance the algorithm's performance for large trees. We also plan to examine the algorithm in higher dimensional problems. The filling of deep local minima could likely be improved by further borrowing the "collision tendency" concept from RRT-CT. Finally, it would also be interesting to see 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 and 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. [Online]. Available: citeseer.ist.psu.edu/cheeseman91where.html

[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. [Online]. Available: citeseer.ist.psu.edu/selman96generating.html

[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 and Automation*, 2000.

[7] S. M. LaValle and J. J. Kuffner, Jr., "Randomized kinodynamic planning," in *IEEE Int. Conf. on Robotics and Automation*, 1999.

[8] P. Cheng and S. M. LaValle, "Reducing metric sensitivity in randomized trajectory design," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2001.

[9] M. Strandberg, "Augmenting RRT-planners with local trees," in *IEEE Int. Conf. on Robotics and 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 and 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.

[20] J.-C. Latombe, *Robot Motion Planning*. Kluwer Academic Press, 1991.