2014 IEEE International Conference on Robotics & Automation (ICRA)
Hong Kong Convention and Exhibition Center
May 31 - June 7, 2014. Hong Kong, China

Any-Com Collision Checking: Sharing Certificates in Decentralized Multi-Robot Teams

Michael Otte, Joshua Bialkowski, and Emilio Frazzoli

***Abstract*— We present an any-com algorithm that enables a decentralized team of robots to share the work of collision checking while each robot independently calculates its own motion plan. In our method "safety-certificates" (i.e., bounds on the collision-free subspace around each collision-checked point [1]), are shared among the team so that all robots can benefit from their encoded knowledge. Future points drawn from within a certificate are guaranteed to be safe; therefore, sharing certificates among team members reduces collision checking for all robots. Experiments demonstrate that our algorithm scales well vs. both team size and vs. communication quality.**

I. Introduction

Achieving faster collision checking is a key hurdle to enabling more sophisticated real-time motion planning [3]. Robots that operate in a shared environment face similar collision checking problems. In general, it makes sense for all robots facing a mutual and computationally complex problem to pool their computational resources in order to solve it. *Any-com*[1] algorithms [4], [5] provide a robust framework for facilitating this type of computational collaboration given realistic communication constraints between robots, e.g., when communication quality is imperfect, unknown *a priori*, and/or changing with time. **In this paper we present an any-com algorithm that extends the collision checking certificate idea from [1] to work with *decentralized*[2] multi-robot teams.**

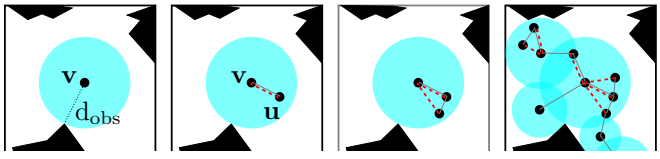
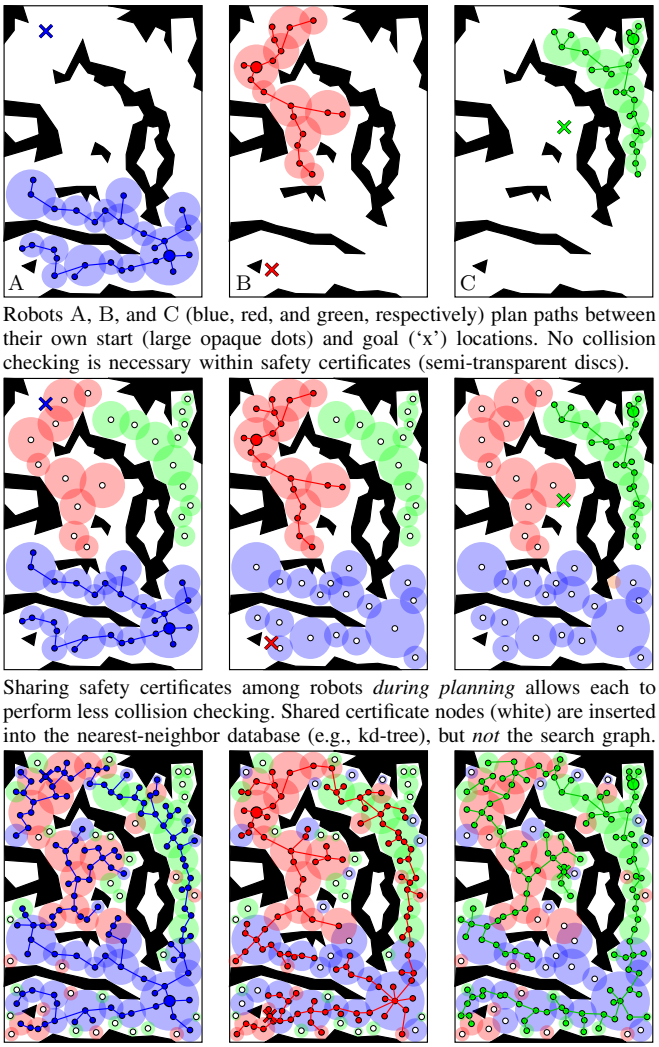


Fig. 1. Our single-robot collision certificate method from [1]. A: Collision checked nodes $\mathbf{v}$ store "safety certificates" (blue) defined by d_{obs} the distance to the nearest obstacle (black). B: Future nodes $\mathbf{u}$ within a certificate can forgo collision checking. C: Pointers (red-dotted lines) are maintained to certifying nodes. D: The ratio of collision checks vs. all nodes approaches zero in the limit vs. graph size. Note, this particular graphic originally appeared in [2].

M. Otte, J. Bialkowski, and E. Frazzoli, Massachusetts Institute of Technology, Cambridge, MA. ottemw@mit.edu.

[1]An *any-com* algorithm is a distributed algorithm that gracefully tolerates communication failure between computational nodes. Better/quicker solutions are found as communication quality increases, but communication failure does not prevent a solution from being found.

[2]In *decentralized* planning each robot computes its own motion without full knowledge of the other robots' plans. Decentralized methods are *incomplete* (may fail to find a solution when one exists) but are significantly faster than centralized methods and often used in practice.



Robots A, B, and C (blue, red, and green, respectively) plan paths between their own start (large opaque dots) and goal ('x') locations. No collision checking is necessary within safety certificates (semi-transparent discs).

Sharing safety certificates among robots *during planning* allows each to perform less collision checking. Shared certificate nodes (white) are inserted into the nearest-neighbor database (e.g., kd-tree), but *not* the search graph.

As the search continues, shared certificate nodes are added to the search graph if they certify new graph-nodes; new certificates continue to be shared.

Fig. 2. Overview of the any-com distributed multi-robot collision checking algorithm presented in the current paper.

In [1] we show that collision checking operations can be significantly reduced for *single*-robot planning in a metric space by using "safety certificates" that record d_{obs}, the distance of (explicitly collision-checked) point $\mathbf{v}$ to the nearest obstacle, see Figure 1. If a new node $\mathbf{u}$ is drawn from within an existing certificate (i.e., $\|\mathbf{v}-\mathbf{u}\| < d_{obs}$), then $\mathbf{u}$ cannot possibly be in collision and a new check (for $\mathbf{u}$) is *unnecessary*. $\mathbf{u}$ then stores a pointer to $\mathbf{v}$ so that future nodes drawn near $\mathbf{u}$ can also check their status vs. the

certificate stored at $\mathbf{v}$. In practice, certificates can be stored within a kd-tree (which is already a common subroutine in motion planning algorithms, e.g., [6], [7], [8]), and so our method can be used without increasing the run-time complexity of many common motion planning algorithms. Moreover, the expected proportion of collision checks vs. all samples approaches zero as the number of samples increases to infinity (see [1] for details).

In the current paper we investigate an any-com algorithm that allows a decentralized team of robots to share the work of collision checking their shared environment, see Figure 2. There are three robots labeled A, B, and C. If a particular robot $R \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ adds a point $\mathbf{v}$ to its search graph such that R calculated $\mathbf{v}.\mathrm{d}_{\mathrm{obs}}$ (the safety certificate of $\mathbf{v}$) then R broadcasts $\mathbf{v}$ (including $\mathbf{v}.\mathrm{d}_{\mathrm{obs}}$) to $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\} \setminus \{R\}$ so that the other robots can add it to their graph and thus avoid collision checks for all points $\mathbf{u}$ such that $\|\mathbf{v} - \mathbf{u}\| < \mathbf{v}.\mathrm{d}_{\mathrm{obs}}$. Full details are provided in Section III.

The any-com properties of the algorithm are due to the fact that dropped messages do not prohibit any robot from eventually finding a solution, while successful messages enable the receiving robot to perform less collision checking.

The rest of this paper is organized as follows: Section II contains related work. The details of our algorithm are presented in Section III. Run-time analysis appears in Section IV. Section V contains a series of experiments designed to test both the any-com properties of our algorithm and its performance vs. team size. Section VI contains a discussion of our results, and conclusions appear in Section VII.

II. Related Work

Sampling-based algorithms[3] are often used for solving high-dimensional motion planning problems in robotics, computer graphics, and synthetic biology [9], [10], [11]. The main idea is to construct a graph of collision-free trajectories through the configuration space, thereby avoiding an explicit (and often computationally prohibitive) representation of the obstacles in that space. However, building the graph in this way requires collision checking in order to test which new trajectories can safely be attached to the existing graph.

The idea of using "safety-certificates" to reduce collision checking first appeared in [1], which focused on collision checking in the context of a single robot. In [2] we extend the idea from [1] to *centralized*[4] multi-robot teams; however, [2] is significantly different from the current paper because it focuses on the *centralized*-specific problem of planning in a high-dimensional Cartesian product of nearly-identical individual-robot configuration spaces. Further, it does *not* attempt to leverage the combined computational resources available in a multi-robot team. In contrast, the current paper focuses on using certificates with a *decentralized* team; it *does* attempt to leverage the team's combined computational resources; and it also contains an investigation into the any-com properties of the certificate method.

When (explicit) collision checking must be done, a variety of approaches exist to minimize its computational burden, such as those in [12], [13] or modern techniques taking advantage of parallel architectures such as [14], [15]. However, the main difference between these and certificate methods is that the former seek to perform collision checking efficiently, while the latter attempt to avoid collision checking as much as possible. It is important to note that the ideas are complementary and should not be considered mutually exclusive. Indeed, we recommend using safety certificates in conjunction with an efficient collision checking data-structure so that the number of explicit checks is minimized, while those that must happen do so as efficiently as possible.

The study of any-com algorithms was introduced in [4] and then extended in [5] and [16]. The main difference between [4], [5], [16] and the current work is that the former focus on any-com multi-robot path planning while the latter focus on multi-robot collision checking. In particular, [4], [5], [16] investigate an any-com algorithm in which all robots simultaneously plan between the *same* start and goal locations in a *centralized* problem, while sharing the best paths found by any robot in order help the team (as a whole) find better paths more quickly. The current work differs because it considers a *decentralized* problem in which robots plan between *different* start and goal locations, while sharing safety certificates in order to help the team avoid collision checks. Moreover, [4], [5], [16] assumed a single-query shortest-path planning problem, while the current work can be applied to either feasible- or shortest-path planning and either single- or multi-query problems.

III. Algorithm

Our algorithm can be summarized as follows: a set of robots, each individually planning a different path through the same environment, share the computational burden of collision checking the configuration space of that environment by broadcasting safety certificates as they plan. This idea can be applied to all sampling-based motion planning problems that assume a metric space, and in conjunction with all popular sampling-based algorithms that require collision checking. However, in the current paper we will concern ourselves with the particular implementation of our certificate method in the context of RRT [6] and RRT* [8]. Extensions to PRM [7], PRM* [8], RRT-sharp [17], etc. are straightforward, requiring only the usual modifications, but are omitted due to space limitations.

We assume all robots are identical, but note that our algorithm can be modified for use by a homogeneous set

[3] In general, sampling-based algorithms are either *single-* or *multi-query*, depending on if they solve problems requiring a single path or multiple different paths through a particular configuration space, respectively (e.g., the Rapidly-exploring Random Tree (RRT) [6] vs. Probabilistic RoadMap (PRM) [7] algorithms, respectively). *Feasible-path* algorithms are concerned with returning a valid solution, whereas *shortest-path* algorithms search for the "best" solution with respect to a distance-metric (e.g., RRT [6] vs. RRT* [8], respectively).

[4] In a *centralized* implementation a single computational entity searches for a collision-free path through the Cartesian product of all individual robots' configuration spaces. While this approach is *probabilistically complete* (i.e., if a solution exists, then the probability of finding it approaches 1 as the computation time approaches infinity), the search complexity of centralized motion planning algorithms scales exponentially vs. team size, making them impractical whenever team size is not small.

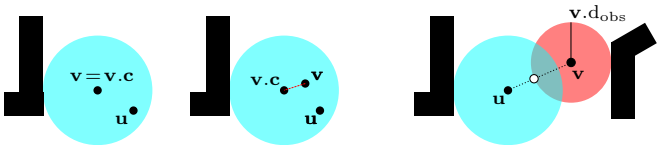


Fig. 3. Left: Depiction of when $\mathbf{certifies}(\mathbf{v},\mathbf{u})$ returns $\mathbf{true}$ (two possibilities). Right: when $\mathbf{kiss}(\mathbf{v},\mathbf{u})$ returns $\mathbf{true}$.

$\mathbf{planningAlgorithm}()$
1: **while not** $\mathbf{stoppingCriteriaMet}()$ **do**
2: **while** $\mathbf{top}(\mathbf{V}_{received}) \neq \emptyset$ **do**
3: $\mathbf{insertSharedNode}(\hat{\mathbf{G}}, \mathbf{pop}(\mathbf{V}_{received}))$
4: $\mathbf{insertNewNode}(\hat{\mathbf{G}})$

Fig. 4.

$\mathbf{certifies}(\mathbf{v},\mathbf{u})$
1: **return** $\|\mathbf{v.c} - \mathbf{u}\| < \mathbf{v.c}.\mathrm{d}_{\mathrm{obs}}$

Fig. 5.

of robots by calculating and sending safety certificates for point-robots, and then having each robot calculate $\mathrm{d}_{\mathrm{obs}}$ by subtracting its own size (ignoring any certificate for which $\mathrm{d}_{\mathrm{obs}} \leq 0$).

We make a distinction between the set of all nodes in the search graph $\mathbf{G}$ and all nodes in the kd-tree $\hat{\mathbf{G}}$, where $\mathbf{G} \subset \hat{\mathbf{G}}$, because we allow nodes containing shared certificates to be inserted into the kd-tree *even if* they cannot be connected to the current search graph. This is done so that the knowledge represented in a safety certificate can be stored until it is eventually needed, i.e., when the search graph expands into regions of the configuration space that have only been explored by other robots. Note that we call a node $\mathbf{v}$ an orphan if $\mathbf{v} \in \hat{\mathbf{G}} \setminus \mathbf{G}$. In practice, we find that it is easier to store only $\hat{\mathbf{G}}$. Nodes $\mathbf{v} \in \hat{\mathbf{G}}$ can be extracted using standard kd-tree interface functions, while $\mathbf{v} \in \mathbf{G}$ are obtained using modified versions of the kd-functions that ignore $\mathbf{v} \in \hat{\mathbf{G}} \setminus \mathbf{G}$ (See Appendix).

The major subroutines of our algorithm are presented in Figures 4-10, and the rest of this section contains a line-by-line description of each. The main planning loop is located in $\mathbf{planningAlgorithm}()$ (Figure 4). This is a stand-in for either RRT or RRT*. Planning continues until the stopping criteria are met (i.e., $\mathbf{stoppingCriteriaMet}()$ returns $\mathbf{true}$, line 1). Examples of possible stopping criteria include 'a valid path has been found' or 'no more planning time remains' depending on if RRT or RRT* is being used, respectively. $\mathbf{V}_{received}$ is the set of all nodes that have recently been passed to the current robot from other robots. In practice $\mathbf{V}_{received}$ should be a first-in-first-out queue, in order to preserve the mutual visibility of shared certificates. Shared nodes (i.e., the nodes that contain the shared certificate data) are inserted into the current robot's data structures using $\mathbf{insertSharedNode}()$, line 2-3, until $\mathbf{V}_{received}$ is empty. Note that $\mathbf{top}()$ and $\mathbf{pop}()$ are the usual queue functions of those names.

The subroutine $\mathbf{certifies}(\mathbf{v},\mathbf{u})$, Figure 5, returns $\mathbf{true}$ if the safety of $\mathbf{u}$ can be guaranteed by $\mathbf{v}$. Figure 3-Left illustrates the two possible ways this can happen (either $\mathbf{v}$ certifies $\mathbf{u}$ itself, or the node that certifies $\mathbf{v}$ also certifies $\mathbf{u}$).

$\mathbf{kiss}(\mathbf{v},\mathbf{u})$
1: **return** $(\mathbf{v}.\mathrm{d}_{\mathrm{obs}} > 0$ **and** $\mathbf{u}.\mathrm{d}_{\mathrm{obs}} > 0$ **and** $\mathbf{certifies}(\mathbf{u}, \mathbf{saturate}(\mathbf{u},\mathbf{v},\mathbf{v}.\mathrm{d}_{\mathrm{obs}})))$

Fig. 6.

$\mathbf{insertNewNode}(\hat{\mathbf{G}})$
1: $\mathbf{v} = \mathbf{randomNode}()$
2: $\mathbf{v}.\mathrm{d}_{\mathrm{obs}} = -1$
3: $\mathbf{v}_{near} = \mathbf{nearestGraphNode}(\hat{\mathbf{G}}, \mathbf{v})$
4: $\mathbf{v} = \mathbf{saturate}(\mathbf{v}, \mathbf{v}_{near}, \delta)$
5: **if not** $\mathbf{certifies}(\mathbf{v}_{near}, \mathbf{v})$ **then**
6: $\mathbf{v}.\mathrm{d}_{\mathrm{obs}} = \mathbf{explicitNodeCert}(\mathbf{v})$
7: **if** $\mathbf{v}.\mathrm{d}_{\mathrm{obs}} \leq 0$ **then**
8: **return**
9: $\mathbf{v.c} = \mathbf{v}$
10: $\mathbf{insert}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$

Fig. 7.

$\mathbf{insertSharedNode}(\hat{\mathbf{G}}, \mathbf{v})$
1: $\mathbf{v}.parent = \emptyset$
2: $\mathbf{v}.g = \infty$ /* RRT* only */
3: $\mathbf{v}_{near} = \mathbf{nearestGraphNode}(\hat{\mathbf{G}}, \mathbf{v})$
4: **if** $\mathbf{certifies}(\mathbf{v}_{near}, \mathbf{v})$ **or** $\mathbf{certifies}(\mathbf{v}, \mathbf{v}_{near})$ **then**
5: $\mathbf{insert}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$
6: **else if** $\|\mathbf{v}_{near} - \mathbf{v}\| > \mathbf{v}.\mathrm{d}_{\mathrm{obs}}$ **then**
7: $\mathbf{addOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$
8: **else if** $\mathbf{kiss}(\mathbf{v}_{near}, \mathbf{v})$ **then**
9: $\mathbf{insert}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$
10: **else**
11: $\mathbf{addOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$

Fig. 8.

The subroutine $\mathbf{kiss}(\mathbf{v},\mathbf{u})$, Figure 6, returns $\mathbf{true}$ if $\mathbf{v}$ and $\mathbf{u}$ have overlapping certificates. This is accomplished by checking if the particular point between $\mathbf{v}$ and $\mathbf{u}$ that is on the edge of $\mathbf{v}$'s certificate is also in $\mathbf{u}$'s certificate, see Figure 3-Right.

New nodes $\mathbf{v}$ are evaluated for insertion into the graph using $\mathbf{insertNewNode}(\hat{\mathbf{G}})$, Figure 7. $\mathbf{randomNode}()$ returns a random node from the configuration space, line 1. The certificate radius of the new node $\mathbf{v}.\mathrm{d}_{\mathrm{obs}}$ is initialized to -1 (on line 2). $\mathbf{v}_{near}$ the nearest node to $\mathbf{v}$ that is already in the search graph $\mathbf{G}$ is found (line 3). Following the convention started in RRT and continued in RRT* the new node is repositioned a distance δ away from $\mathbf{v}_{near}$ in the direction of the original random sample using $\mathbf{saturate}(\mathbf{v}, \mathbf{v}_{near}, \delta)$, line 4. If $\mathbf{v}$ is certified safe by $\mathbf{v}_{near}$, then it is inserted into the graph (according to the convention of whatever algorithm is being used, e.g., RRT) with $\mathbf{insert}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$, line 10. Otherwise, we must perform an explicit point-collision check to determine if $\mathbf{v}$ is safe (and calculate its certificate in the process), line 6. If $\mathbf{v}$ is not safe then it is ignored, lines 7-8. If $\mathbf{v}$ is safe then we recording that it certifies itself, line 9, and insert it into the graph, line 10.

Nodes $\mathbf{v}$ that have been shared (and received) are eval-

$\mathbf{insert_{RRT}}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$

1: $\hat{\mathbf{v}}_{near} = \mathbf{nearest}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$
2: **if** $\mathbf{certifies}(\mathbf{v}_{near}, \mathbf{v})$ **or** $\mathbf{certifies}(\mathbf{v}, \mathbf{v}_{near})$ **or** $(\hat{\mathbf{v}}_{near} \neq \mathbf{v}_{near}$ **and** $\mathbf{certifies}(\hat{\mathbf{v}}_{near}, \mathbf{v}_{near})$ **and** $\mathbf{certifies}(\hat{\mathbf{v}}_{near}, \mathbf{v}))$ **or** $\mathbf{kiss}(\mathbf{v}_{near}, \mathbf{v})$ **then**
3: $\quad \mathbf{C} = \bigcup \mathbf{c} : \mathbf{c} \in \{\mathbf{v}.\mathbf{c}, \mathbf{v}_{near}.\mathbf{c}, \hat{\mathbf{v}}_{near}.\mathbf{c}\} \wedge \|\mathbf{v} - \mathbf{c}\| < \mathbf{c}.\mathrm{d}_{\mathrm{obs}}$
4: $\quad \mathbf{v}.\mathbf{c} = \arg\max_{\mathbf{c} \in \mathbf{C}}(\mathbf{c}.\mathrm{d}_{\mathrm{obs}})$
5: **else if** $\mathbf{explicitEdgeCollision}(\mathbf{v}_{near}, \mathbf{v})$ **then**
6: $\quad$ **return**
7: $\mathbf{v}.parent = \mathbf{v}_{near}$
8: $\mathbf{addKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$
9: **if** $\mathbf{v}.\mathbf{c} = \mathbf{v}$ **then**
10: $\quad \mathbf{broadcast}(\mathbf{v})$
11: **if** $\hat{\mathbf{v}}_{near}.parent = \emptyset$ **and** $(\mathbf{certifies}(\mathbf{v}, \hat{\mathbf{v}}_{near})$ **or** $\mathbf{certifies}(\hat{\mathbf{v}}_{near}, \mathbf{v}))$ **then**
12: $\quad \mathbf{C} = \bigcup \mathbf{c} : \mathbf{c} \in \{\mathbf{v}.\mathbf{c}, \hat{\mathbf{v}}_{near}.\mathbf{c}\} \wedge \|\hat{\mathbf{v}}_{near} - \mathbf{c}\| < \mathbf{c}.\mathrm{d}_{\mathrm{obs}}$
13: $\quad \hat{\mathbf{v}}_{near}.\mathbf{c} = \arg\max_{\mathbf{c} \in \mathbf{C}}(\mathbf{c}.\mathrm{d}_{\mathrm{obs}})$
14: $\quad \hat{\mathbf{v}}_{near}.parent = \mathbf{v}$
15: $\quad \mathbf{deOrphanKD}(\hat{\mathbf{G}}, \hat{\mathbf{v}}_{near})$

Fig. 9.

uated for insertion into the graph and/or kd-tree using $\mathbf{insertSharedNode}(\hat{\mathbf{G}}, \mathbf{v})$, Figure 8. The parent of $\mathbf{v}$ is set as undefined, and (assuming RRT*) its graph cost is set to ∞, line 2. The nearest node in the search graph is found, line 3; note that we *do not* modify the position of the shared node to be closer to $\mathbf{v}_{near}$ as in the standard implementation of RRT. This is because we are provided with its certificate—which would be invalidated by any movement of the node itself. Instead, we perform a series of tests to determine if the edge between $\mathbf{v}$ and $\mathbf{v}_{near}$ is guaranteed to be safe given the certificates known to $\mathbf{v}$ and $\mathbf{v}_{near}$. If either $\mathbf{v}$ or $\mathbf{v}_{near}$ certifies the other, then $\mathbf{v}$ is inserted into the graph (according to the convention of whatever algorithm is being used), lines 4-5. If not, then $\mathbf{kiss}(\mathbf{v}_{near}, \mathbf{v})$ checks if the edge between $\mathbf{v}$ and $\mathbf{v}_{near}$ can be certified as safe using a combination of their certificates, lines 8-9; if so, then $\mathbf{v}_{near}$ is also inserted into the search graph. Otherwise, $\mathbf{v}$ is inserted as an orphan (inserted into the kd-tree, but not the search-graph), lines 10-11.

The RRT version of insert, i.e., $\mathbf{insert_{RRT}}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$, appears in Figure 9. In addition to the original $\mathbf{v}_{near}$ (the potential parent of $\mathbf{v}$ from the search graph) we also query the kd-tree for the nearest node $\hat{\mathbf{v}}_{near}$ (i.e., that may not be in the search-graph, and thus may be an orphan node containing a shared certificate), line 1. If some combination of certificates among $\mathbf{v}$, $\mathbf{v}_{near}$, and $\hat{\mathbf{v}}_{near}$ can certify the edge between $\mathbf{v}$ and $\mathbf{v}_{near}$, then an explicit edge collision check can be avoided—and the certifier of $\mathbf{v}$ is determined to be the member of $\{\mathbf{v}.\mathbf{c}, \hat{\mathbf{v}}_{near}.\mathbf{c}\}$ with the largest certificate, lines 2-3. Note that the largest certificate is used because, in expectation, it will allow for the most future nodes near $\mathbf{v}$ to be certified as safe. If the edge between $\mathbf{v}$ and $\mathbf{v}_{near}$ cannot be implicitly certified as safe, then it must be explicitly checked, line 5. If a collision is found then $\mathbf{v}$ is ignored, line 6; otherwise, $\mathbf{v}_{near}$ is made the parent of $\mathbf{v}$, line 7, and

$\mathbf{insert_{RRT*}}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$

1: $r_{ball} = \mathbf{neighborBallRad}(|\mathbf{G}|, \eta)$
2: $r_{safe} = \mathbf{v}.\mathrm{d}_{\mathrm{obs}}$; $\mathbf{v}.parent = \emptyset$; $\mathbf{v}.g = \infty$, $\mathbf{V}_{safe} = \emptyset$
3: $\mathbf{V}_{near} = \mathbf{near}(\hat{\mathbf{G}}, \mathbf{v}, r_{ball}, \mathbf{v}_{near})$
4: **for all** $\mathbf{v}_i \in \mathbf{V}_{near}$ **do**
5: $\quad$ **if** $\mathbf{certifies}(\mathbf{v}, \mathbf{v}_i)$ **or** $\mathbf{certifies}(\mathbf{v}_i, \mathbf{v})$ **then**
6: $\quad\quad \mathbf{V}_{safe} = \mathbf{V}_{safe} \cup \{\mathbf{v}_i\}$
7: $\quad\quad r_{safe} = \max(r_{safe}, \mathbf{v}_i.\mathbf{c}.\mathrm{d}_{\mathrm{obs}} - \|\mathbf{v} - \mathbf{v}_i.\mathbf{c}\|)$
8: $\quad\quad$ **if** $r_{safe} > r_{ball}$ **then**
9: $\quad\quad\quad$ **break**
10: **for all** $\mathbf{v}_i \in \mathbf{V}_{near}$ **do**
11: $\quad b_{safe} =$ **true**
12: $\quad$ **if** $r_{safe} > r_{ball}$ **or** $\mathbf{v}_i \in \mathbf{V}_{safe}$ **then**
13: $\quad\quad \mathbf{V}_{safe} = \mathbf{V}_{safe} \cup \{\mathbf{v}_i\}$
14: $\quad$ **else if** $\mathbf{v}.\mathrm{d}_{\mathrm{obs}} < 0$ **then**
15: $\quad\quad \mathbf{v}.\mathrm{d}_{\mathrm{obs}} = \mathbf{explicitNodeCert}(\mathbf{v})$
16: $\quad\quad r_{safe} = \max(r_{safe}, \mathbf{v}.\mathrm{d}_{\mathrm{obs}})$
17: $\quad\quad$ **if not** $\mathbf{certifies}(\mathbf{v}, \mathbf{v}_i)$ **then**
18: $\quad\quad\quad b_{safe} =$ **false**
19: $\quad$ **else**
20: $\quad\quad b_{safe} =$ **false**
21: $\quad$ **if not** b_{safe} **then**
22: $\quad\quad$ **if** $\mathbf{explicitEdgeCollision}(\mathbf{v}_i, \mathbf{v})$ **then**
23: $\quad\quad\quad$ **continue**
24: $\quad\quad \mathbf{C} = \bigcup \mathbf{c} : \mathbf{c} \in \{\mathbf{v}.\mathbf{c}, \mathbf{v}_i.\mathbf{c}\} \wedge \|\mathbf{v} - \mathbf{c}\| < \mathbf{c}.\mathrm{d}_{\mathrm{obs}}$
25: $\quad\quad \mathbf{v}.\mathbf{c} = \arg\max_{\mathbf{c} \in \mathbf{C}}(\mathbf{c}.\mathrm{d}_{\mathrm{obs}})$
26: $\quad\quad$ **if** $\mathbf{v}.g > \|\mathbf{v}_i, \mathbf{v}\| + \mathbf{v}_i.g$ **then**
27: $\quad\quad\quad \mathbf{v}.g = \|\mathbf{v}_i, \mathbf{v}\| + \mathbf{v}_i.g$
28: $\quad\quad\quad \mathbf{v}.parent = \mathbf{v}_i$
29: **if** $\mathbf{v}.parent \neq \emptyset$ **then**
30: $\quad \mathbf{addKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$
31: $\quad$ **if** $\mathbf{v}.\mathbf{c} = \mathbf{v}$ **then**
32: $\quad\quad \mathbf{broadcast}(\mathbf{v})$
33: $\quad$ **for all** $\mathbf{v}_i \in \mathbf{V}_{safe} \setminus \{\mathbf{v}.parent\}$ **do**
34: $\quad\quad$ **if** $\mathbf{v}_i.g > \|\mathbf{v}_i, \mathbf{v}\| + \mathbf{v}.g$ **then**
35: $\quad\quad\quad$ **if** $\mathbf{v}_i.parent = \emptyset$ **then**
36: $\quad\quad\quad\quad \mathbf{deOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}_i)$
37: $\quad\quad\quad \mathbf{v}_i.g = \|\mathbf{v}_i, \mathbf{v}\| + \mathbf{v}.g$
38: $\quad\quad\quad \mathbf{v}_i.parent = \mathbf{v}$

Fig. 10.

$\mathbf{v}$ is added to the kd-tree[5], line 8. If $\mathbf{v}$ certifies itself, then it is broadcast to the other robots using $\mathbf{broadcast}(\mathbf{v})$, lines 9-10. If $\hat{\mathbf{v}}_{near}$ has no parent (i.e., it is an orphan node) and the edge $(\hat{\mathbf{v}}_{near}, \mathbf{v})$ is implicitly safe, then $\hat{\mathbf{v}}_{near}$ is inserted into the search-graph using $\mathbf{v}$ as its parent, line 11-15 (note that we never explicitly check an orphan node). On line 13 we reset $\hat{\mathbf{v}}_{near}$ be certified by $\mathbf{v}.\mathbf{c}$ if the latter has a larger certificate—because $\hat{\mathbf{v}}_{near}$ is an orphan it has not yet certified any other nodes, and therefore doing this does not affect the status of any nodes in the search graph. Finally, we change the kd-tree status of $\hat{\mathbf{v}}_{near}$ to reflect the fact that it is now in $\hat{\mathbf{G}}$ and not $\hat{\mathbf{G}} \setminus \mathbf{G}$ using $\mathbf{deOrphanKD}(\hat{\mathbf{G}}, \hat{\mathbf{v}}_{near})$, line 15.

The RRT* version of insert appears in Figure 10. The radius r_{ball} of the shrinking hyper-ball is calculated on line 1 and then used to find $\mathbf{V}_{near}$ the set of $\mathbf{v}$'s potential neighbors on line 3, see [8] for more information. Note that we allow

[5]Note that $\mathbf{addKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$ seeds its search for the (kd-tree) insertion location of $\mathbf{v}$ using $\mathbf{v}_{near}$; we find that this can provide considerable speedup in practice

all nodes (even orphaned nodes) to be part of $\mathbf{V}_{near}$. The variable r_{safe} keeps track of the largest safe ball around $\mathbf{v}$ that we can calculate given all of the neighbors in the near set, and is initialized to $\mathbf{v}.\mathrm{d}_{\mathrm{obs}}$, line 2. The parent and graph-cost of $\mathbf{v}$ are initialized to undefined and ∞, line 2. $\mathbf{V}_{safe}$ will be populated with those members of $\mathbf{V}_{near}$ that can be connected to $\mathbf{v}$, it is initialized to the empty set on, line 2.

Lines 4-9 contain an initial iteration through the members of $\mathbf{V}_{near}$ to determine if any $\mathbf{v}_i \in \mathbf{V}_{near}$ certifies $\mathbf{v}$, line 5, as well as to get the largest "cheap" estimate of r_{safe} (i.e., while doing no explicit checks), line 7. r_{safe} is the largest ball centered at $\mathbf{v}$ that is completely contained in a certificate. If r_{safe} ever gets larger than r_{ball} then we know that all remaining nodes in $\mathbf{V}_{near}$ have implicitly safe edges to $\mathbf{v}$, lines 8-9, 12.

Lines 10-28 contain a second iteration through the members of $\mathbf{V}_{near}$. The Boolean b_{safe} is used to remember if the current $\mathbf{v}_i$ can safely be connected to $\mathbf{v}$, lines 11, 18, 20, 21. If a node $\mathbf{v}_i$ exists for which the edge $(\mathbf{v}, \mathbf{v}_i)$ is not implicitly safe (line 14–and note the second half of the check on line 12), then $\mathbf{v}$ performs an explicit node check to calculate its own certificate, line 15. If this certificate does not contain $\mathbf{v}_i$, then a full explicit edge check is required, line 22. Nodes that cannot be connected to $\mathbf{v}$ are *not* added $\mathbf{V}_{safe}$, line 23. Care is taken to remember the best certifying node (i.e., the node with the largest certificate that contains $\mathbf{v}$) of $\mathbf{v}$, lines 24-25. With respect to graph search, we seek to find the best parent of $\mathbf{v}$ that exists in $\mathbf{V}_{safe}$ (i.e., the node that minimizes $\mathbf{v}$ cost of reaching the goal and that can be safely connected to $\mathbf{v}$), lines 26-28. Note that if $\mathbf{v}_i$ is an orphan then $\mathbf{v}_i.g \equiv \infty$ and so it will not be made a parent of $\mathbf{v}$, line 26.

Assuming we can connect $\mathbf{v}$ to the graph, line 29, then it is added to the kd-tree, line 30. If $\mathbf{v}$ certifies itself, then it is broadcast to the other robots, lines 31-31. Finally, any neighbor $\mathbf{v}_i$ with a valid edge to $\mathbf{v}$ (except $\mathbf{v}.parent$) is re-wired to use $\mathbf{v}$ as its parent *if* doing so can reduce $\mathbf{v}_i.g$ (and orphan nodes that are added to the graph are marked as such), lines 33-38.

IV. Run-time Analysis

Let $|\hat{\mathbf{G}}_i|$ be the size of the kd-tree of the i-th robot. By inspection we see that all kd-tree insert and search operations retain their usual run-time complexity vs. $|\hat{\mathbf{G}}_i|$. In particular, our modifications to $\mathbf{addKD}(\hat{\mathbf{G}}_i, \mathbf{v}, \mathbf{v}_{near})$ and $\mathbf{addOrphanKD}(\hat{\mathbf{G}}_i, \mathbf{v}, \mathbf{v}_{near})$ involve running at most twice the usual number of kd-searches, and also going back up the tree in $\mathbf{addOrphanKD}(\hat{\mathbf{G}}_i, \mathbf{v}, \mathbf{v}_{near})$ and again at most once per node in $\mathbf{deOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}_i)$. Similarly, the search functions $\mathbf{nearestGraphNode}(\hat{\mathbf{G}}_i, \mathbf{v})$ and $\mathbf{near}(\hat{\mathbf{G}}_i, \mathbf{v}, r_{ball}, \mathbf{v}_{near})$ retain the usual expected search time vs. $|\hat{\mathbf{G}}_i|$ (ignoring orphan nodes in the former does not affect the run-time vs. $|\hat{\mathbf{G}}_i|$). Both adding a new node and a nearest-node query require the normal expected time $\mathcal{O}(\log(|\hat{\mathbf{G}}_i|))$, while searching for a set of near nodes requires time $\mathcal{O}(\hat{k}\log(|\hat{\mathbf{G}}_i|))$, where $\hat{k}$ is the expected number of nodes within the shrinking hyper-ball used by RRT* when the search is conducted.

Let $\mathbf{C}_i$ be the set of certificates shared by robot i. In the worst case, all robots cover the entire space with certificates, yet this only increases the total expected number of certificates on a single robot by a constant factor. The limiting ratio between all certificates and the total nodes on a single robot follows directly from the analysis in [1]. In particular,

$$\lim_{|\mathbf{G}_i|\to\infty} \frac{|\bigcup_{i\in\{1,\ldots,r\}} \mathbf{C}_i|}{|\mathbf{G}_i|} = 0$$

It is also insightful to evaluate how our algorithm affects kd-tree run-time complexity when $|\hat{\mathbf{G}}_i|$ is small. For this we assume that all robots share approximately the same number of nodes. Formally, we assume $\frac{|\mathbf{C}_i|}{|\mathbf{C}_j|} < \kappa$ for all robots $i, j \in \{1, ..., r\}$ and some 'small' κ, e.g., $\kappa = 10$. It is easy to see that the number of nodes shared by any particular robot i is bounded by that robot's search graph size $|\mathbf{G}_i|$. In a normal implementation of RRT or RRT* $\mathbf{G}_i = \hat{\mathbf{G}}_i$, but in our algorithm $\mathbf{G}_i \subset \hat{\mathbf{G}}_i$. However, given our assumptions $|\hat{\mathbf{G}}_i| < r\kappa|\mathbf{G}_i|$, and so $\log(|\hat{\mathbf{G}}_i|) < \log(r\kappa|\mathbf{G}_i|) = \log(r\kappa) + \log(|\mathbf{G}_i|)$. Thus, when $r\kappa < |\mathbf{G}_i|$ kd-tree insertion retains its usual expected run-time of $\mathcal{O}(\log(|\mathbf{G}_i|))$ and kd-tree search retains its run-time of $\mathcal{O}(\log(|\mathbf{G}_i|))$ or $\mathcal{O}(\hat{k}\log(|\mathbf{G}_i|))$, depending on if the nearest-node or a set of near nodes is desired. By inspection we see that there are no other changes that significantly affect run-time complexity; therefore sharing certificates does not affect per-iteration run-time complexity of the overall motion planning algorithm.

V. Experiments

We perform three experiments to evaluate the performance of our algorithm vs. different team sizes, communication qualities, and in conjunction with both RRT and RRT*.

Experiment 1 uses certificate sharing on five laptop computers (Intel atom with 2GB of RAM) that communicate using UDP messages over an ad-hoc wireless 802.11b network. **This experiment is designed to verify that our algorithm can work in a practical any-com scenario using an unreliable wireless communication protocol**. The environment depicted in Figure 11-Top is used.

We evaluate the performance of a particular robot (the red robot) as team size ranges from 1 to 5. The choice to use the red robot is arbitrary and other robots give similar results. Increasing team sizes are created by adding robots in the order: red, blue, light-green, teal, magenta. Our evaluation metric is (the red robot's) graph-size $|\mathbf{G}|$ vs. time ($\mathbf{G}$ does *not* include orphan nodes). We use this metric because larger graph size correlates with better path quality and higher rate of success for any particular sampling-based shortest-path or feasible-path planning algorithm, respectively. Results appear in Figure 12, which shows normalized values vs. results obtained by (the red robot) using certificates *without* sharing (e.g., a value of 2 means that sharing certificates produced results that were twice as good as using certificates without sharing). Also plotted are the results of (the red robot) using

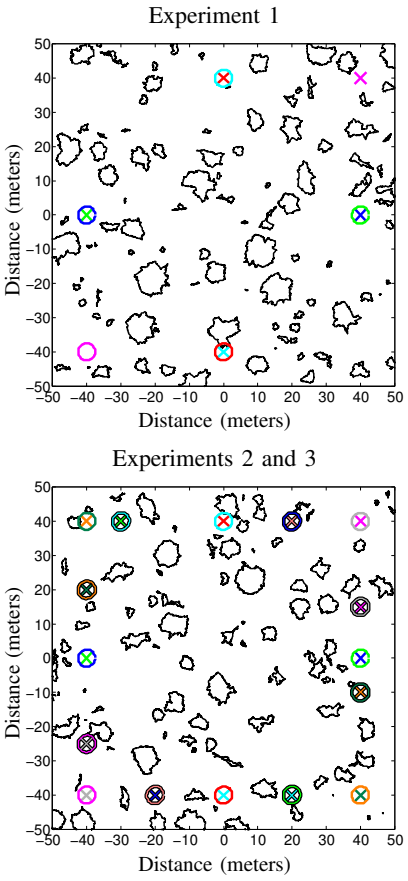


Fig. 11. The randomly generated environments used in our experiments. Obstacles are black, robots are drawn as circles at their starting locations, and goal locations are designated with an 'X' of the same color (note: robots are depicted 5X their actual size). The red robot is the particular robot for which statistics are collected.

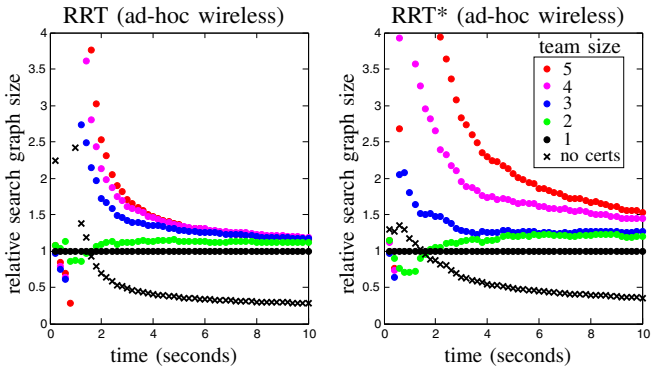


Fig. 12. Search graph size vs. time given different team sizes, for robots communicating over an ad-hoc wireless network. The results are depicted as scaled relative to a single robot using certificates in isolation—higher values are more desirable. Solid markers denote team size, while 'x's show a comparison to a single robot using a standard collision checker *without* certificates (e.g., a value of 2 means that a particular method had twice as many nodes in the tree at a given time, compared to a single robot using the certificate method without sharing). Each point represents the mean value over 10 trials. Higher values are better.

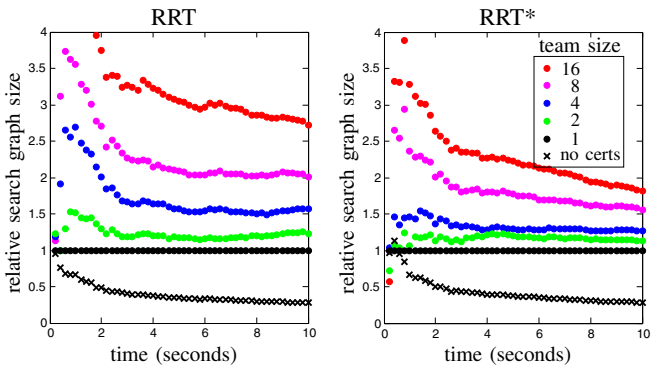


Fig. 13. Search graph size vs. time given different team sizes. The results are depicted as scaled relative to a single robot using certificates in isolation—higher values are more desirable. Solid markers denote team size, while 'x's show a comparison to a single robot using a standard collision checker *without* certificates. Each point represents the mean value over 10 trials. Higher values are better.

standard RRT or RRT* without certificates. Each data-point represents the mean value over 10 trials. Communication quality was above 95% in all trials.

Experiment 2 evaluates performance (of the red-robot) vs. larger team sizes in simulation (up to 16 robots). All 'robots' are simulated on the same computer and the communication protocol is also simulated. **Note that we use simulation because we wish to evaluate the performance of team sizes larger than the number of wireless laptops that we own.** Teams containing up to 16 robots are tested on the environment depicted in Figure 11-Bottom. The obstacles in Figure 11-Bottom have four times as many edges as those in Figure 11-Top in order to strain the relatively large computational power of the simulation computer (Intel i7 chip with 16GB of RAM). Robots are added in the order: red, blue, light-green, teal, magenta, lavender, orange, dark-green—followed by the same colors outlined in black. Communication quality was set to 100% for this experiment.

Experiment 3 evaluates performance (of the red robot) as a function of communication quality when a 16 robot team uses certificate sharing. **This experiment is run in simulation so that we can control communication quality**. Communication quality is calculated as the ratio of received packets vs. sent packets. Packet success/failure is governed by a Bernoulli distribution, and we perform different runs for message-success probability p fixed at 1.0, 0.8, 0.4, 0.2, 0.1, or 0.05. As in the previous experiment results are depicted normalized vs. results obtained by (the red robot) using certificates without sharing, and the results of running standard RRT (or RRT*) without certificates are also shown, 'x's. We ran similar experiments for team sizes between 2 and 15 (omitted due to space constraints) and found that results were exactly what one would expect given the results of Experiment 2, i.e., the red robot's $|\mathbf{G}|$ was smaller when smaller teams were used, but certificate sharing still enabled an increase in $|\mathbf{G}|$ for all $p > 0$.

VI. Discussion

A. Performance vs. team-size

We observe that the number of nodes that each robot is able to add to its own search graph increases significantly vs. the number of robots that are sharing certificates. The increase in $|\mathbf{G}_i|$ appears to be sub-linear vs. team-size, and given the relatively modest size of our teams, we are unable

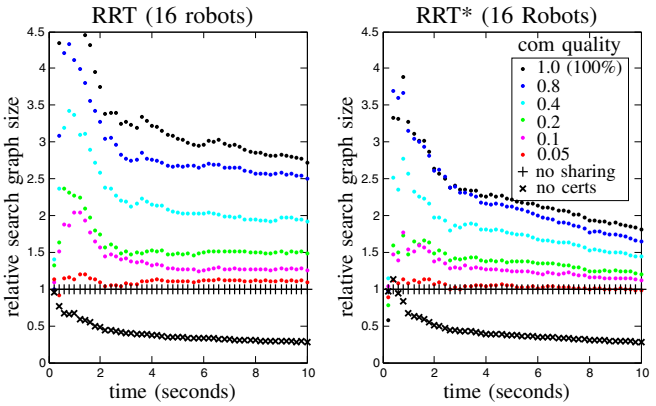


Fig. 14. Search graph size vs. time for a 16 robot team experiencing different communication qualities (color). Results are depicted as scaled relative to a single robot using certificates but *not* communicating ('+'s). Results from a single robot using a standard collision checker *without* certificates are also shown ('x's). Each point represents the mean value over 10 trials. Higher values are better.

to conclude whether or not the practical performance gains of sharing certificates will continue to increase for very large number of robots (e.g., 1000s). That said, our results suggest that certificate sharing can be quite beneficial when used with small to moderately sized teams. The results in Figures 12 and 13 show that sharing certificates among a team can provide a significant advantage vs. using certificates alone without sharing and also vs. not using certificates at all. Note that sharing among an 8 robot team increases average graph size by the same factor as using certificates in the first place.

Another trend that can be seen in Figures 12 and 13 is that sharing certificates appears to provide an extra advantage early in the search—with the notable exception of the two-robot team in the ad-hoc wireless experiment. We attribute the former result to the fact that most of the space is explored (e.g., with respect to sampling dispersion) early-on—which means that the largest and most beneficial certificates tend to be found near the beginning of the search. We believe the latter exception (i.e., that the two-robot team experiences an early dip in performance) is due to the relative expense of wireless communication vs. the the cumulative benefits of certificate sharing—which are small at the beginning of the run but increase vs. time. Larger teams receive enough initial benefits that this is not an issue.

B. Performance vs. communication quality

Experiment 3 indicates that certificate sharing has good any-com properties. Figure 14 shows that performance declines gracefully as communication quality deteriorates. Even when communication quality is very poor (e.g., $p = 0.05$), $|\mathbf{G}_i|$ does not fall below what would be obtained if the robot decided to abstain from sharing. We reiterate that there was an exception to this trend in Experiment 1, but that it was only observed for very small team sizes (≤ 2 robots).

C. Wireless communication vs. simulation

Any comparison between Experiment 1 (wireless communication) and Experiment 2 (simulation) must acknowledge the fact that experiments 1 and 2 were run on different hardware and solved a different planning problem; nonetheless, we are pleased that the overall trends appear qualitatively similar. Besides the aforementioned early performance dip in the 2 robot team in Experiment 1, another difference is an early advantage of *not* using certificates vs. very small teams (≤ 2 robots) in Experiment 1. We are unable to explain why this occurs at this time. A final difference is that certificate sharing provided more advantage to RRT* than to RRT in Experiment 1 (ad-hoc wireless), but the reverse is true in Experiment 2 (simulation).

VII. Summary and Conclusions

Robots that operate in a shared environment must collision check against many of the same obstacles. In [1] it was shown that single-robot motion planning could be expedited by using "safety-certificates" to avoid collision checking in areas already known to be safe. We extend this idea to decentralized multi-robot teams. In particular, we present an any-com algorithm in which a set of robots share safety-certificates as they are found. This enables all robots to increase the rate at which space is certified as "collision-free" and thereby increase the rate of search-graph growth and exploration.

Our method retains the same asymptotic properties of the original certificate method. In particular, the number of certificates vs. graph points is expected to approach 0, in the limit, as the number of graph points approaches infinity. Further, the per-iteration run-time complexity is the same as the original certificate method—and therefore also the "normal" (non-certificate) versions of the motion planning algorithms with which it is used. That said, as with all collision checking certificate methods, the proposed algorithm should only be used when collision checking is relatively time-consuming vs. nearest neighbors search; in other words, our method is not expected to be useful on easy-to-collision-check environments.

Certificate sharing has good any-com properties in the sense that better communication provides increased benefits to all robots (e.g., better communication enables quicker graph growth and exploration), and poor communication does not appear to hinder overall search progress. The algorithm also scales well—at least for the small to moderately sized sets of robots that we investigate. That is, when more robots participate in certificate sharing, then the benefits to all robots increase. For instance, even switching from 15 to 16 robots noticeably improves the rate that each individual robot is able to plan.

Acknowledgments

This work was partially supported by the Office of Naval Research, MURI grant #N00014-09-1-1051, the Army Research Office, MURI grant #W911NF-11-1-0046, and the Air Force Office of Scientific Research, grant #FA-8650-07-2-3744.

$\mathbf{addOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$

1: **while** $\mathbf{u}.\mathbf{p}_{\mathrm{kd}} \neq \emptyset$ **and** $\|\mathbf{v}, \mathbf{u}.\mathbf{p}_{\mathrm{kd}}.\phi_{\mathrm{kd}}\| < \|\mathbf{v}, \mathbf{u}.\phi_{\mathrm{kd}}\|$ **do**
2: $\quad \mathbf{v} = \mathbf{v}.\mathbf{p}_{\mathrm{kd}}$
3: search down the kd-tree to find the appropriate leaf for $\mathbf{v}$
4: link $\mathbf{v}$ to its appropriate parent
5: $\mathbf{v}.b_{orphan} = \mathbf{true}$
6: $\mathbf{v}.b_{oroot} = \mathbf{true}$

Fig. 15. $\mathbf{addOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$ adds $\mathbf{v}$ to the kd-tree $\hat{\mathbf{G}}$ with the understanding that $\mathbf{v} \notin \mathbf{G}$ (that is, $\mathbf{v}$ is an orphan and not in the search graph). The search is seeded with $\mathbf{v}_{near}$ (note, this can be the root of the kd-tree if no better information exists). On lines 1-2 we walk up the kd-tree until we reach the root of the appropriate sub-tree in which to insert $\mathbf{v}$. Next, we proceed exactly as in a standard kd-tree insertion, lines 3-4. Finally, we mark that $\mathbf{v}$ is both an orphan and the root of an orphan sub-tree, lines 5-6.

$\mathbf{addKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{u})$

1: $\mathbf{addOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$
2: $\mathbf{deOrphanKD}(\hat{\mathbf{G}}, \mathbf{v})$

Fig. 16. $\mathbf{addKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$ adds $\mathbf{v}$ to the kd-tree $\hat{\mathbf{G}}$ with the understanding that $\mathbf{v} \in \mathbf{G}$ ($\mathbf{v}$ is *not* an orphan). As with $\mathbf{addOrphanKD}(\hat{\mathbf{G}}, \mathbf{v}, \mathbf{v}_{near})$ the search can be seeded with $\mathbf{v}_{near}$. First we add $\mathbf{v}$ to the kd-tree as if it were an orphan, line 1, then we explicitly mark it as not an orphan using $\mathbf{deOrphanKD}(\hat{\mathbf{G}}, \mathbf{v})$, line 2.

APPENDIX

This appendix contains an overview of the kd-tree modifications necessary to store $\mathbf{G}$ and $\hat{\mathbf{G}}$ in the same kd-tree. This is done by marking orphan nodes, i.e., any $\mathbf{v} \in \hat{\mathbf{G}} \setminus \mathbf{G}$, as such, and also keeping track of which sub-trees contain *only* orphan nodes, so that we can avoid searching them when calling $\mathbf{nearestGraphNode}(\hat{\mathbf{G}}, \mathbf{v})$. Descriptions of the subroutines appear in the captions of the figures containing the subroutines themselves (Figures 15-18). We assume that the reader is familiar with kd-trees (if not, then see [18] for details). For brevity we use high-level language to describe the "standard" parts of the kd-tree implementation. $\mathbf{v}.\mathbf{p}_{\mathrm{kd}}$ and $\mathbf{v}.\mathbf{l}_{\mathrm{kd}}$ and $\mathbf{v}.\mathbf{r}_{\mathrm{kd}}$, are the parent and left- and right-children of $\mathbf{v}$ within the kd-tree. $\|\mathbf{vu}.\phi_{\mathrm{kd}}\|$ is the minimum distance from $\mathbf{v}$ to the splitting plane $\phi_{\mathrm{kd}} \ni \mathbf{u}$ (i.e., the splitting plane stored at, associated with, and containing node $\mathbf{u}$). $\mathbf{v}.b_{orphan}$ is a Boolean value that is $\mathbf{true}$ when $\mathbf{v}$ is an orphan, and $\mathbf{v}.b_{oroot}$ is a Boolean value that is $\mathbf{true}$ if $\mathbf{v}$ and all of its descendants are orphans.

$\mathbf{nearestGraphNode}(\hat{\mathbf{G}}, \mathbf{v})$

1: search the kd-tree for the leaf that would contain $\mathbf{v}$, using the definition $\mathbf{u} \equiv \emptyset$ for all $\mathbf{u} : \mathbf{u}.b_{oroot} = \mathbf{true}$
2: $\mathbf{u}$ would be the parent of $\mathbf{v}$ (if $\mathbf{v}$ were being inserted)
3: $\mathbf{v}_{near} = \mathbf{u}$
4: **while** $\|\mathbf{v}, \mathbf{u}.\mathbf{p}_{\mathrm{kd}}.\phi_{\mathrm{kd}}\| < \|\mathbf{v}, \mathbf{v}_{near}\|$ **do**
5: $\quad$ recursively search down any siblings $\mathbf{u}$ of $\mathbf{v}$ such that $\mathbf{u}.b_{oroot} \neq \mathbf{true}$, remembering the best $\mathbf{v}_{near}$
6: **return** $\mathbf{v}_{near}$

Fig. 17. $\mathbf{nearestGraphNode}(\hat{\mathbf{G}}, \mathbf{v})$ returns the nearest graph node to $\mathbf{v}$. Although $\mathbf{G} \subset \hat{\mathbf{G}}$, in general $\arg\min_{\mathbf{v}_i \in \mathbf{G}}(\|\mathbf{v} - \mathbf{v}_i\|) \neq \arg\min_{\mathbf{v}_i \in \hat{\mathbf{G}}}(\|\mathbf{v} - \mathbf{v}_i\|)$. This is essentially a standard kd-tree search, except that all orphan sub-trees are treated as if they were empty. We locate the would-be leaf of $\mathbf{v}$, lines 1-2, and its would-be parent is the initial guess of $\mathbf{v}_{near}$, line 3. We walk up the tree, recursing down siblings if the splitting plane between siblings is closer to $\mathbf{v}$ than $\mathbf{v}$ is to $\mathbf{v}_{near}$ (ignoring all orphan sub-trees), lines 4-5.

$\mathbf{deOrphanKD}(\hat{\mathbf{G}}, \mathbf{v})$

1: $\mathbf{v}.b_{orphan} = \mathbf{false}$
2: **while** $\mathbf{v}.parent \neq \emptyset$ **and** $\mathbf{v}.parent.b_{oroot}$ **do**
3: $\quad \mathbf{v}.parent.b_{oroot} = \mathbf{false}$
4: $\quad \mathbf{v} = \mathbf{v}.parent$

Fig. 18. $\mathbf{deOrphanKD}(\hat{\mathbf{G}}, \mathbf{v})$ takes a node $\mathbf{v} \in \hat{\mathbf{G}} \setminus \mathbf{G}$ and adds it to $\mathbf{G}$. We mark $\mathbf{v}$ as a non-orphan, lines 1-2, then propagate this information to all ancestors of $\mathbf{v}$ that are no longer the root of an orphan sub-tree.

REFERENCES

[1] J. Bialkowski, S. Karaman, M. Otte, and E. Frazzoli, "Efficient collision checking in sampling-based motion planning," in *Proc. International Workshop on the Algorithmic Foundations of Robotics*, 2012.

[2] J. Bialkowski, M. Otte, and E. Frazzoli, "Fast collision checking: From single robots to multi-robot teams," in *IEEE International Conference on Robotics and Automation: Crossing the Reality Gap - From Single to Multi- to Many Robot Systems*, 2013.

[3] S. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[4] M. Otte and N. Correll, "Any-Com multi-robot path-planning: Maximizing collaboration for variable bandwidth," in *Proc. International Symposium on Distributed Autonomous Robotics Systems*, 2010.

[5] M. Otte and N. Correll, "Any-Com multi-robot path-planning with dynamic teams: Multi-robot coordination under communication constraints," in *Proc. International Symposium on Experimental Robotics*, 2010.

[6] S. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[7] L. Kavraki, P. Svestka, J. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, Jan. 1996.

[8] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, June 2011.

[9] N. Amato and G. Song, "Using motion planning to study protein folding pathways," *Journal of Computational Biology*, vol. 9, no. 2, pp. 149–168, Mar. 2004.

[10] J. Cortes, T. Simeon, V. Ruiz de Angulo, D. Guieysse, M. Remaud-Simeon, and V. Tran, "A path planning approach for computing large-amplitude motions of flexible molecules," *Bioinformatics*, vol. 21, no. 1, pp. 116–125, Jun. 2005.

[11] J. Latombe, "Motion planning: A journey of molecules, digital actors, and other artifacts," *International Journal of Robotics Research*, vol. 18, no. 11, pp. 1119–1128, Sep. 2007.

[12] P. Jimnez, F. Thomas, and C. Torras, "3d collision detection: a survey," *Computers & Graphics*, vol. 25, no. 2, pp. 269 – 285, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849300001308

[13] M. C. Lin and D. Manocha, "Collision and proximity queries," in *Discreate and Computational Geometry, Handbook of*, J. E. Goodman and J. O'Rourke, Eds. Chapman and Hall, 2003, pp. 387 – 808.

[14] J. Pan and D. Manocha, "Gpu-based parallel collision detection for fast motion planning," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 187–200, 2012. [Online]. Available: http://ijr.sagepub.com/content/31/2/187.abstract

[15] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *IEEE Int. Conference on Robotics and Automation*, Minneapolis, Minnesota, 05/2012 2012.

[16] M. Otte, "Any-com multi-robot path planning," Ph.D. dissertation, University of Colorado at Boulder, 2012.

[17] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *Robotics and Automation, 2013. Proceedings. ICRA '13. IEEE International Conference on*, 2013, pp. 2413–2420.

[18] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, September 1975.