The Nearest Polytope Problem: Algorithms and Application to Controlling Hybrid Systems

Albert Wu, Sadra Sadraddini, Russ Tedrake

***Abstract*— We present three algorithms for solving the nearest polytope problem: given a list of polytopes and a distance metric in Euclidean space, find the nearest polytope to a query point. We consider the AH-polytope representation, which generalizes the H-polytope representation and is particularly useful in the context of control. Through preprocessing the polytopes into efficient data structures, we avoid exhaustive search at query time. We discuss the properties of the algorithms and compare their performances using polytopic datasets motivated by control applications, including sampling-based motion planning and model predictive control.**

I. Introduction

Given a finite list of polytopes in n-dimensional space and a query point, the *point location problem* is finding the set of polytopes that contain the point. When this set is empty, a more general and difficult problem, the *nearest polytope problem*, is determining the nearest polytope given a metric. The solution to the point location problem is contained in the solution to the nearest polytope problem - the set of polytopes with zero distance from the point. The naive solution to both problems is an exhaustive check over all polytopes. However, evaluating point membership or computing the distance from a polytope to a query point are expensive operations. More efficient solutions are necessary for practicality.

Both of these problems are purely geometrical but have profound applications in control theory. The point location problem has long been studied in explicit linear model predictive control (MPC) literature, where polytopes represent pre-computed sets that correspond to linear control laws [1]–[4]. Given a "query" state, one needs to locate the corresponding "active" polytope to execute the associated control law. This scheme is typically faster than solving the MPC optimization problem online. In "exact explicit MPC", the polytopes are non-overlapping and are described by their hyperplanes, a representation known as H-polytopes [5]. Infeasibility is returned if the query state is not within any of the polytopes. The nearest polytope problem is relevant in "approximate explicit MPC", where the nearest but not necessarily query point-containing polytopic feasible set is used as a heuristic to compute control inputs [6].

Another application of the nearest polytope problem lies in reachability-guided motion planning [7], [8]. In the R3T algorithm in [8], the reachable sets of states are approximated by (a union of) polytopes, and a rapidly-exploring random tree (RRT) [9], [10] of the polytopes is maintained. To expand the tree, the R3T algorithm requires finding the nearest polytope to samples from the state space [8].

In this paper, we investigate methods to improve query speed in the nearest polytope problem. We consider *AH-polytopes* [11], a more general representation for polytopic objects whose hyperplanes may be difficult to compute. Examples include zonotopes [5], polytopes in lifted spaces, and convex-hulls/ Minkowski sums of H-polytopes. AH-polytopes are highly relevant in the context of control. For instance, they naturally capture the transformation from admissible control space to state space reachable sets in linear systems. As motivated by control applications, our methods are primarily targeted at medium-sized problems (state dimensions around 10). For very large problems, computational bottlenecks from implementing MPC/RRT often overshadow the nearest polytope problem.

This paper is organized based on our four main contributions: three fast algorithms for solving the nearest polytope problem and benchmarking of these algorithms. The contributions of this paper are as follows:

- An algorithm for solving the nearest polytope problem using axis-aligned bounding boxes (AABB) of the polytopes. This differs from previous work (such as [12]) in that through leveraging key points, our algorithm is not limited to point location problems.
- An algorithm for solving the nearest polytope problem by precomputing distances to a finite set of key points. During online querying, the key points help eliminate explicit point-polytope distance queries using triangle inequality for polytopes.
- An algorithm that leverages the well-known binary space partitioning (BSP) trees [13], [14] to solve the nearest polytope problem. Our key contribution here is using polytope containment reasoning [11] to construct BSP-trees for the nearest polytope problem.
- Benchmarking of our algorithms with synthetic datasets and examples from MPC/RRT for (hybrid) control problems. These examples arise in planning and feedback policy design, including those for hybrid systems.

II. Related Work

The nearest polytope problem degenerates to the well known *nearest neighbor problem* [15] when points instead of polytopes are considered. Many efficient algorithms have been proposed for this problem, such as k-d trees [16] and R-trees [17]. Variants of these data structures will be leveraged

The authors are with the computer science and artificial intelligence laboratory (CSAIL) at Massachusetts Institute of Technology, Cambridge, MA 02139. This research was partially funded by Department of the Navy, Office of Naval Research Award No.: N00014-17-1-2699 {wualbert,sadra,russt@mit.edu }

to solve the nearest polytope problem in this paper. The nearest neighbor problem has also been extensively studied in the computer graphics literature. However, such methods are tailored for 2D/3D applications and do not apply well to control applications, which typically have higher dimensions [18].

The point location problem has been widely studied in the control community [1], [4], [19], [20]. Some approaches include using bounding boxes and interval trees for faster search [12] and organizing the space-dividing hyperplanes into a fast binary search tree [3]. Other approaches exploit system dynamics to reduce the search space [21], [22], but they are restricted to simple systems with linear dynamics. Still other approaches leverage a piecewise affine (PWA) form of the value function, which arise from exact explicit MPC with linear cost, to solve the point location problem [23], [24]. Unfortunately, these methods do not generalize to the nearest polytope problem and hybrid systems. We also note that most previous work requires the disjoint space partitioning. The work in [12] is able to handle overlapping polytopes. Surprisingly, to the best of the authors' knowledge, there has been no development on efficient algorithms for the nearest polytope problem.

III. Problem Statement

A. Preliminaries

The set of n-dimensional real values is denoted by $\mathbb{R}^n$. Given $a \in \mathbb{R}^n$, we denote its transpose by a^T. Given two matrices A, B with appropriate dimension, their vertical stacking is $[A, B]$. Given a finite set S, $|S|$ is its cardinality.

Definition 1. *(H-Polytope) [5]. An H-polytope $P \subset \mathbb{R}^n$ is a bounded set described by its hyperplanes:*

$$P = \{x \in \mathbb{R}^n \mid Hx \leq h\},$$

where $H \in \mathbb{R}^{q \times n}$ and $h \in \mathbb{R}^q$, q is the number of hyperplanes, the kernel of H is only the origin, and all the inequalities are interpreted element-wise.

Definition 2. *(AH-Polytope) [11]. An AH-polytope $P \subset \mathbb{R}^n$ is a polytope given by an affine transformation $G \in \mathbb{R}^{n \times m}$, $g \in \mathbb{R}^n$, of polytope $S \subset \mathbb{R}^m$:*

$$P = \{g + Gx \mid Hx \leq h\},$$

where $H \in \mathbb{R}^{q \times m}$ and $h \in \mathbb{R}^q$.

Given an AH-polytope $P \subset \mathbb{R}^n$, a point $q \in \mathbb{R}^n$, a metric $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_{\geq 0}$, the point-polytope distance function $d(P, q)$ and the closest point $p^* \in P$ to q are defined as

$$d(P, q) := \min_{p \in P} d(p, q),\ p^* = \arg\min_{p \in P} d(p, q).$$

This minimization can be cast as the following optimization problem:

$$\begin{aligned} d(P, q) = \quad & \min . && \|\delta\|_\rho \\ & \text{subject to} && Gx + g = q + \delta,\ Hx \leq h, \end{aligned} \tag{1}$$

which is a linear program for $\rho = 1, \infty$. For $\rho = 2$, a quadratic program can be constructed by optimizing for the squared norm $\|\cdot\|_2^2$. Other types of norms such as polytopic norms can be also used. We have $q \in P$ if and only if $d(P, q) = 0$. In this paper, we primarily use $\rho = 2$. Note that this paper focuses on reducing the *number* of $d(P, q)$ evaluations. Hence, while the complexity of $d(P, q)$ scales with the dimension and the shape of the polytope, we consider it to be constant in the subsequent discussion. In practice, computing $d(P, q)$ requires solving a convex optimization problem, which its complexity grows polynomialy with dimensions and the complexity of the representation of the polytope.

B. Problem Formulation

Problem 1. *Given a set of AH-polytopes $\mathbf{P} = \{P_i\}_{i=1,\cdots,N}$, $P_i \subset \mathbb{R}^n$, and a query point $q \in \mathbb{R}^n$, find the nearest $P^* \in \mathbf{P}$ such that*

$$P^*(q) = \arg\min_{P \in \mathbf{P}} d(P, q). \tag{2}$$

If $P^*(q)$ is not unique, return one of the minimizers.

Since $\mathbf{P}$ is often given in advance, the task is further split into "offline" and "online" components. The "offline" component consists of preprocessing $\mathbf{P}$ and constructing data structures. In some applications such as motion planning [8], $\mathbf{P}$ is expanded incrementally. The "online" component is the actual query and is more speed critical. The subsequent algorithms focus on creating a fast data structure offline to allow sub-$\mathcal{O}(|\mathbf{P}|)$ online computation. When the minimizer is not unique, we may use a control theoretic tiebreaker such as the (approximate) value function associated with the polytopes [6]. We assume such a function is available. Otherwise, we pick a random minimizer. In the subsequent sections, we arbitrarily choose a minimizer if multiple exist.

IV. Finding the Nearest Polytope with Axis-Aligned Bounding Boxes (AABB)

A. Preliminaries

For obtaining an approximation of a polytope that supports quick querying and construction, consider the axis-aligned bounding box, or AABB, given in Definition 3:

Definition 3. *(Axis-Aligned Bounding Box, AABB) [25]. An axis-aligned bounding box (AABB) $B \in \mathbb{R}^n$ can be described by its the "lower" and "upper" corner points $(l, u) \in \mathbb{R}^n$:*

$$B(l, u) = \{x \mid l_i \leq x_i \leq u_i, \forall i = 1, 2, ..., n\}. \tag{3}$$

Finding the axis aligned bounding box (l, u) of an AH-polytope $P \in \mathbb{R}^n$ can be formulated as a linear optimization problem. Let $\mathbb{1}_i$ be the column vector with 1 at the ith entry and 0 at all the other entries. The optimization problem is given in Proposition 1.

Proposition 1. *(AABB of an AH-Polytope). The AABB $B(l, u)$ of $P = \{g + Gx \mid Hx \leq h\}$ can be found by solving the following linear programs:*

$$l_i = \min_{x | Hx \leq h} \mathbb{1}_i \cdot (g + Gx),\ u_i = \max_{x | Hx \leq h} \mathbb{1}_i \cdot (g + Gx)$$

Algorithm 1 `AABB-Precompute`

Require: $\mathbf{P}$ ▷ List of polytopes
Require: $K \in \mathbf{P}$ ▷ Key points inside the polytopes
$B \leftarrow \emptyset$
for $P_i \in \mathbf{P}$ **do**
  $B_i \leftarrow \texttt{ComputeAABB}(P_i)$ ▷ Compute AABB
  $B \leftarrow B \cup \{B_i\}$ ▷ Store AABB
$\mathbf{B} \leftarrow \texttt{BuildFastAABBStructure}(B)$
$\mathbf{K} \leftarrow \texttt{BuildFastPointStructure}(K)$

B. AABB Algorithm

We leverage AABBs to create a data structure for fast nearest polytope querying. During the offline phase, for each of the polytopes $P_i \in \mathbf{P}$, we construct AABBs B_i such that $P_i \subseteq B_i$. Through the `BuildFastAABBStructure` routine, the AABBs are stored in a data structure $\mathbf{B}$ such as R-tree [17] that supports fast AABB querying. We also maintain a data structure $\mathbf{K}$ using the `BuildFastPointStructure` routine for a constant number of key points from each polytope. An example of this structure is k-d tree [16]. The only requirement for the key points K is they must each be inside a polytope. In Section VII, arbitrary points that are by construction inside the polytopes are used. The offline precomputation is summarized in Algorithm 1.

Given a query point q online, the closest key point $p^* = \arg\min_{p \in \mathbf{K}} d(p, q)$ is first found. We identify the "pivot" polytope P^* that contains p^*, then compute $d^* = d(P^*, q)$. A "heuristic box" B_h with side length $2d^*$ is constructed, and $\mathbf{B}$ is queried for all overlapping boxes $\mathbf{B}_v$ with B_h. We then randomly choose a polytope P_r from all the polytopes corresponding to $\mathbf{B}_v$, compute the distance $d(P_r, q)$, then compare to P^*. If P_r is closer, we replace P^* with P_r, reconstruct B_h using the new P^*, and update $\mathbf{B}_v$. This is repeated until either $\mathbf{B}_v$ is exhausted, which means the nearest polytope is P^*, or a polytope containing q is found. The online computation is summarized in Algorithm 2.

This algorithm is valid for L1, L2 and L-infinity distance metrics. This is due to the fact that all AABB operations are intersection checks and requires no distance metric. As long as the heuristic box B_h is constructed using a side length of $2d_\rho(\cdot,\cdot), \rho = 1, 2, \infty$, B_h will encompass all polytopes that can be closer to q than the pivot polytope. We also note that this algorithm can potentially be generalized to arbitrary convex objects as long as the AABB of such object can be computed, but pursuing this direction is beyond the scope of the paper.

C. Correctness and Complexity Analysis

The algorithm differ from [12] in that it can handle the case where no polytope contains the query point q. This is done by the construction of the heuristic box. Since the heuristic box must contain at least one polytope and encompasses everywhere that has closer distance to q than the polytope, the true closest polytope is guaranteed to be found through searching over boxes overlapping with the heuristic box. Figure 1 visualizes this process.

Algorithm 2 `AABB-Query`

Require: $\mathbf{B}$, $\mathbf{K}$, q ▷ AABBs, key points, query point
$p^* \leftarrow \arg\min_{p \in \mathbf{K}} d(p, q)$ ▷ Find closest key point
$P^* \leftarrow p^*.P$ ▷ Choose corresponding polytope as pivot
$d^* \leftarrow d(P^*, q)$ ▷ Compute distance to pivot
$B_h \leftarrow \texttt{AABB}(q, 2d^*)$ ▷ Construct AABB centered at q with side length $2d_p$
$\mathbf{B}_v \leftarrow \{B \mid B \in |T|, B \cap B_h \neq \emptyset\}$ ▷ Find AABBs intersecting B_h
$\mathbf{P}_{cand} \leftarrow \mathbf{B}_v.P$ ▷ Get candidate polytopes corresponding to $\mathbf{B}_v$
while $|\mathbf{P}_{cand}| > 0$ **do**
  if $d^* = 0$ **then**
    return P^*, d^* ▷ Found a polytope containing q
  $P_s \leftarrow \texttt{Sample}(\mathbf{P}_{cand} \backslash P^*)$
  $d_s \leftarrow d(P_s, q)$
  if $d_s \geq d^*$ **then**
    $\mathbf{P}_{cand} \leftarrow \mathbf{P}_{cand} \backslash P_s$
  else
    $P^*, d^* \leftarrow P_s, d_s$
    $B_h \leftarrow \texttt{AABB}(q, 2d^*)$
    $\mathbf{B}_v \leftarrow \{B \mid B \in |T|, B \cap B_h \neq \emptyset\}$ ▷ Update the list of candidates
return P^*, d^*

The offline complexity of the algorithm lies in constructing the axis aligned bounding boxes ($\mathcal{O}(|\mathbf{P}|)$) and creating the tree structure for searching (`BuildFastAABBStructure` and `BuildFastPointStructure`). Insertion of a new polytope is $\mathcal{O}(1)$ for constructing a new AABB plus the complexity of updating $\mathbf{B}$ and $\mathbf{K}$. The online complexity depends on the distribution of the polytopes, which will be evaluated empirically in Section VII.

V. Finding the Nearest Polytope with Triangle Inequality

A. Triangle Inequality Algorithm

In this approach, we attempt to lower-bound the distance between a query point q and each of the polytopes $P \in \mathbf{P}$. The bound is used to prune the list of polytopes to evaluate online. This bound is motivated by the triangle inequality, shown in Theorem 1.

Theorem 1. *(Point-Polytope Triangle Inequality). Given points $u, v \in \mathbb{R}^n$, a polytope $P \subset \mathbb{R}^n$, and metric $d(\cdot,\cdot)$, $d(u, v) + d(v, P) \geq d(u, P)$.*

Proof: Suppose the closest point to u in P is $u^* \in P$, the closest point to v in P is $v^* \in P$. By the triangle inequality, $d(u, v) + d(v, v^*) \geq d(u, v^*)$. By definition of v^*, $d(u, v^*) \geq d(u, u^*)$. Therefore,

$$d(u, v) + d(v, P) = d(u, v) + d(v, v^*) \geq d(u, u*) = d(u, P).$$

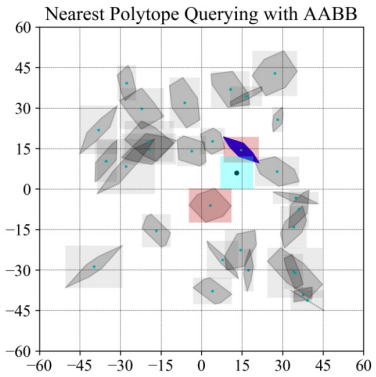


Fig. 1: Nearest polytope query with AABB. $|\mathbf{P}| = 30$. L2 distance was used. The query point q is black, while the key points K are blue. Through constructing a heuristic box B_h (cyan) to prune impossible polytopes and gradually shrinking B_h using Algorithm 2, only a small subset of $d(P, q)$ (red boxes, $|\mathbf{B}_v| = |\mathbf{P}_{cand}| = 2$) is evaluated.

If the polytopes are given in advance, we can use Theorem 1 to perform precomputations that reduce online comoputation. Specifically, we avoid evaluating $d(P, q)$ for polytopes P that cannot possibly be the closest. Given two polytopes P_1, P_2, a key point k, and a query point q, Theorem 2 provides a lower bound on $d(P_2, q)$ given $d(P_1, q)$ and $d(k, q)$.

Theorem 2. *(Point-Polytope Distance Lower Bound). Given two polytopes P_1, P_2, a key point k and a query point q, the following condition implies a lower bound on $d(P_2, q)$ by $d(P_1, q)$:*

$$d(P_2, k) \geq d(k, q) + d(P_1, q) \implies d(P_2, q) \geq d(P_1, q)$$

Proof: From Theorem 1, we have $d(P_2, q) \geq d(P_2, k) - d(k, q)$ Thus if $d(P_1, q) \leq d(P_2, k) - d(k, q)$, $d(P_1, q) \leq d(P_2, q)$.

Algorithm 3 and 4 describe a procedure for finding the closest polytope leveraging Theorem 2. When a new set of polytopes $\mathbf{P}$ is given to the algorithm, `TI-Precompute` is run offline to construct the necessary data structures. When a query point q is given, `TI-Query` is run to compute the closest polytope P^*. This algorithm is applicable to any metric as long as Theorem 2 holds.

B. Correctness and Complexity Analysis

The algorithm is guaranteed to not miss the closest polytope by construction. The reason is the closest distance from the query point q to a polytope is upper bounded by $d(P(k^*), q)$. Thus, only polytopes that can possibly be closer than $P(k^*)$ needs to be checked.

Offline complexity of this algorithm is $O(|K||\mathbf{P}| \log |\mathbf{P}|)$ plus the complexity for `BuildFastPointStructure`. This is a consequence of computing and sorting pairwise distances between all the key points and polytopes. In practice, the number of key points can be chosen arbitrarily to avoid large precomputations, but this will negatively impact online performance. In Section VII, we set $|K||\mathbf{P}| \leq 10^6$ to limit precomputation time. Insertion is $\mathcal{O}(|\mathbf{P}| \log \mathbf{P})$

Algorithm 3 `TI-Precompute`

Require: $\mathbf{P}$ ▷ List of polytopes
Require: $K \in \mathbf{P}$ ▷ Key points inside the polytopes
$\mathbf{D} \leftarrow \emptyset$ ▷ Initialize distance map
for $k_i \in K$ **do**
  for $P_j \in \mathbf{P}, P_j \neq P(k_i)$ **do**
    $\mathbf{D}(k_i, P_j) \leftarrow d(k_i, P_j)$
  `sort`$(\mathbf{D}(k_i, \cdot))$ ▷ Sort key point distances
$\mathbf{K} \leftarrow$ `BuildFastPointStructure`(K)

Algorithm 4 `TI-Query`

Require: q ▷ Query point
Require: $\mathbf{P}, \mathbf{K}$
$k^* \leftarrow \arg\min_{k \in \mathbf{K}} d(k, q)$ ▷ Find closest key point
$d^* \leftarrow d(q, P(k^*))$ ▷ Compute distance to polytope corresponding to k^*
$\mathbf{P}_{cand} \leftarrow \{P | P \in \mathbf{P}, d(k^*, P) < d^* + d(k, q)\}$ ▷ Select polytopes satisfying Theorem 2
$P^* \leftarrow \arg\min_{P \in \mathbf{P}_{cand}} d(q, P)$ ▷ Find closest among candidates
return $P^*, d(P^*, q)$

VI. Finding the Nearest Polytope with Binary Space Partitioning Trees (BSP-Trees)

In this section, we introduce our method to use BSP-trees for the nearest polytope problem.

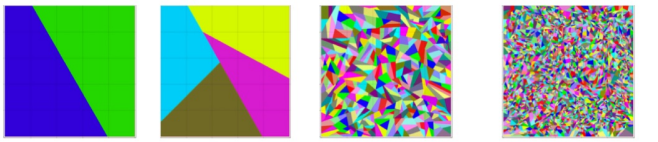
A. Overall Construction

Definition 4. *A BSP-tree in $\mathbb{R}^n$ is a binary tree data structure [15] $(V, \mathcal{C}, \pi, \sigma, \tau)$, where V is the set of nodes, $\mathcal{C}(v) = \{x \in \mathbb{R}^n | H_{\mathcal{C}(v)} x \leq h_{\mathcal{C}(v)}\}$ is the H-polytope of node $v \in V$, $\pi(v)$ is the parent of v, $\sigma(v)$ returns the set of polytopes associated with node v (defined later), and $\tau : V \to (s, \alpha) \cup \emptyset$ maps each node to its splitting hyperplane that is parameterized by normal vector $s \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$. A node without a splitting hyperplane is a leaf node. If v is not a leaf node, the H-polytopes of the left and right children, denoted by v_L and v_R, respectively, are recursively defined as:*

- $\mathcal{C}(v_L) = \{x \in \mathbb{R}^n | [H_{\mathcal{C}(v)}, s^T] x \leq [h_{\mathcal{C}(v)}, \alpha]\}$;
- $\mathcal{C}(v_R) = \{x \in \mathbb{R}^n | [H_{\mathcal{C}(v)}, -s^T] x \leq [h_{\mathcal{C}(v)}, -\alpha]\}$;

where $(s, \alpha) = \tau(v)$.

Assuming the admissible space is an H-polytope, a BSP-tree recursively partitions it into smaller H-polytopes. At every non-leaf node v, the splitting hyperplane cuts $\mathcal{C}(v)$ into two H-polytopes. The procedure for building a BSP-tree is outlined in Algorithm 5. We detail the subroutines `split` and `fill` in the subsequent subsections. An example of constructing a BSP-tree is illustrated in Fig. 2.



D=1, 2 Leaves D=2, 4 Leaves D=9, 499 Leaves D=15, 4138 Leaves

Fig. 2: BSP-tree polytopic partitions for a synthetic data of 500 zonotopes uniformly distributed in 2D box. The construction took 351 seconds. At each iteration, nodes with $|\sigma(v)| > 5$ were split. After reaching the depth of 15, 4138 leaves were obtained, where the mean and maximum of $|\sigma(v)|$ over the leaves was than 3.79 and 10, respectively.

Algorithm 5 BSP-Trees Construction

Require: $\mathbf{P}$, C_0, $N_{\max} > |\mathbf{P}|$, $D_{\max}$ ▷ List of polytopes, the admissible state polytope, the maximum number of polytopes in a leaf, and the maximum depth

$V = \{v_{\text{root}}\}, L = \{v_{\text{root}}\}$ ▷ Tree and list initialization
while $L \neq \emptyset$ and $D \leq D_{\max}$ **do**
  for $v \in L$ **do**
    $L \leftarrow L \setminus \{v\}$ ▷ Remove from the list
    `split`(v) to obtain v_L, v_R
    `fill` $\sigma(v_L)$ and $\sigma(v_R)$
    for $v' \in v_L, v_R$ **do**
      if $|\sigma(v')| > N_{\max}$ **then**
        $L \leftarrow L \cup \{v'\}$ ▷ Node to be split
  $D \leftarrow D + 1$ ▷ Add depth

Given a query point $q \in \mathbb{R}^n$, we traverse the tree in depth by evaluating $\epsilon = \alpha - s'q$ at each node. If $\epsilon > 0$, then we land at the left child. If $\epsilon \leq 0$, we land at the right hand. Note that the set of points resulting in $\epsilon = 0$ has zero measure. By evaluating a finite number of inequalities, we arrive at the leaf node that its H-polytope contains q. Ideally, BSP-trees end with all leaves containing only one polytope, making online queries very simple. However, such trees may be excessively large. We allow for truncated trees [26] where each leaf may contain up to user-defined $N_{\max}$ polytopes.

B. `split` node

One of the key components of all BSP-tree construction algorithms is finding the splitting hyperplane. For the point location problem, the authors in [27] used mixed-integer programming to find the optimal hyperplane that balances the tree in the sense that the number of polytopes associated for each child are equal, and ideally, half of those for the parent. However, solving combinatorial optimization problems for a large number of times is not feasible, especially in the shallower nodes of the tree that may contain a large number of polytopes. Moreover, it is not shown to produce signficant improvements over simpler methods [27].

Here we follow a simpler solution by picking random hyperplane directions. We sample $s \in \mathbb{R}^n$ from a normal distribution, but choose the right hand side α such that the splitting hyperplane roughly cuts $\mathcal{C}(v)$ into balanced halves (in the sense of volume instead). We solve the following linear programs:

$$\bar{\alpha} = \begin{array}{ll}\max. & s^T x, \\ \text{s.t.} & x \in \mathcal{C}(v),\end{array} \qquad \underline{\alpha} = \begin{array}{ll}\min. & s^T x, \\ \text{s.t.} & x \in \mathcal{C}(v).\end{array} \tag{4}$$

Then we choose $\alpha = \frac{1}{2}(\underline{\alpha} + \bar{\alpha})$. Thus, the hyperplane is guaranteed to cut through $\mathcal{C}(v)$.

C. `fill` Node

Given a node v, we need to find $\sigma(v)$. A filling procedure is sound if for all points in $\mathcal{C}(v)$, the nearest polytope is one of the elements in $\sigma(v)$. Formally, we have:

$$\sigma(v) \supseteq \{P \in \mathbf{P} | \exists x \in \mathcal{C}(v), P = \arg\min_{P' \in \mathbf{P}} d(P', x)\}. \tag{5}$$

By definition have $\sigma(v_{\text{root}}) = \mathbf{P}$. Also, it is straightforward to prove that $\sigma(v) \subseteq \sigma(\pi(v))$.

We define the following distances. First, we need a notion for the largest distance from a point in $C := \mathcal{C}(v)$ to an AH-polytope P:

$$D^{\overrightarrow{\max}}(P, C) := \max{}_{x \in C} d(P, x). \tag{6}$$

Note that, in general, $D^{\overrightarrow{\max}}(P, C) \neq D^{\overrightarrow{\max}}(C, P)$. Eq. (6) is also known as the directed Hausdorff distance [11]. We also define:

$$D^{\min}(P, C) := \min{}_{x \in C, y \in P} d(x, y). \tag{7}$$

We obtain the following result by construction.

Theorem 3 (Node filling)**.** *Given node $v \in V$, $\sigma(v)$ should contain all the elements in the following set:*

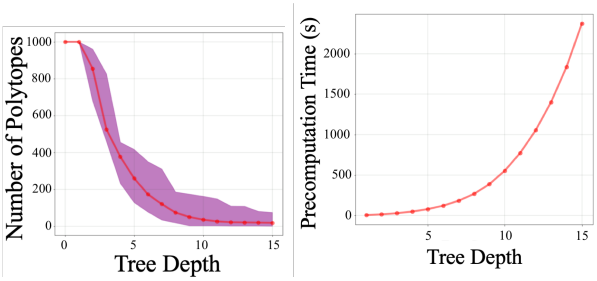
$$\{P \in \sigma(\pi(v)) | D^{\min}(P, C) \leq \min_{P' \in \mathcal{P}(\pi(v))} D^{\overrightarrow{\max}}(P', C)\}. \tag{8}$$

While (7) can be easily cast as a linear program, solving the max-min problem in (6) is more challenging. Using duality, we obtain a non-convex optimization problem for which a convex linear programming relaxation is obtained using polytope containment. The details are in [11]. The relaxation introduces some conservativeness as we over-approximate the set in (8), making the set $\sigma(v)$ larger. While this over-approximation does not break the soundness of the nearest polytope problem - the nearest polytope is still within $\sigma(v)$ - it can be inefficient.

D. Complexity

BSP-trees are computationally intensive to build offline, but provide very fast online queries as the tree is traversed in depth. The main bottleneck in the offline phase is the requirement to solve a large number of linear programs to fill the nodes. If the number of leaves has a polynomial relationship with the number of polytopes, then the query of BSP-trees, under mild assumptions, have average $\mathcal{O}(\log |\mathbf{P}|)$ time complexity [27]. However, obtaining theoretical bounds for the complexity of the number of leaves for generic polytopic data sets is not obvious.

One of the advantages of BSP-trees over the previous two algorithms is that it provides user-defined upper bounds for the number of polytopes that are required to be checked.



(a) Median number of polytopes evaluated in a single query versus tree depth.

(b) Precomputation time versus tree depth.

Fig. 3: Empirical complexity analysis of the BSP-tree on the first 1000 AH-polytopes of R3T implementation for the pendulum swing up. At $D = 15$, the number of leaves is 8725. We set $N^{\max} = 20$. 5629 of the leaves had less than or equal to 20 polytopes.

VII. Experimental Results

A. Overview

Two synthetic datasets of zonotopes distributed uniformly and along an axis ("linearly distributed"), and two real datasets of AH-polytopes from R3T [8] and MPC [6] were used to evaluate the algorithms. All code for the experiments can be found on GitHub[1]. All tests were performed on a personal computer with i7-7820HQ CPU.

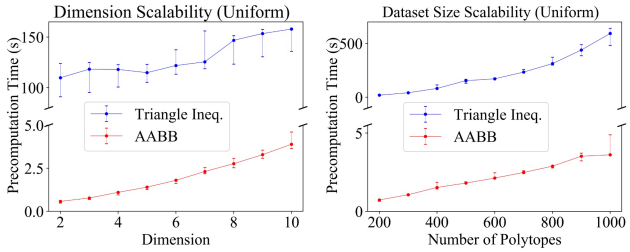
This section is organized as follows. First, we evaluate the scalability on dimensions and dataset size using synthetic datasets. Next, we test the real world performance of the algorithms using datasets generated by R3T and MPC. Unless otherwise noted, the medians of the results are plotted and the error bars denote maximum and minimum values. A total of 1000 queries were performed in each experiment.

Due to the limitations on BSP-tree's scalability to high dimensions, limited experiments were performed on on BSP-trees. Moreover, the online complexity of BSP-trees is set by the user. Therefore, we have plotted the complexity versus the depth of the tree (determined by the user) in Fig. 3. We leave further benchmarking of BSP-trees and their speedup optimization for the nearest polytope problem to future work.

B. Offline Scalability

The precomputation performances of the AABB algorithm and the triangle inequality algorithm were tested on synthetic datasets. Specifically, we test for the scalability against dimension and dataset size on randomly generated polytopes. Figure 4 shows the precomputation performance on uniformly distributed random polytopes, as motivated by sampling-based planning such as [8]. Figure 5 shows the precomputation performance on random polytopes distributed along an axis ("linearly distributed"), as motivated by trajectory stabilization techniques such as LQR trees [28]. All experiments were triplicated.

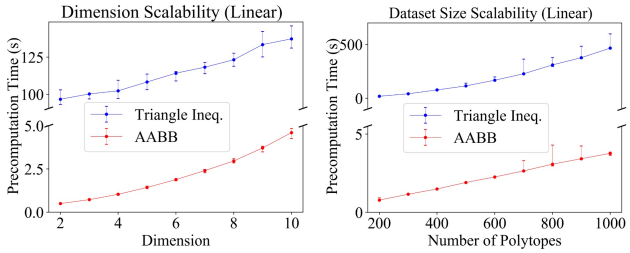
[1]https://github.com/wualbert/closest_polytope_algorithms.git



(a) Precomputation time dimension scalability with 500 polytopes.

(b) Precomputation time dataset size scalability with polytopes in 6D.

Fig. 4: Precomputation time with uniformly distributed random polytopes.



(a) Precomputation time dimension scalability with 500 random polytopes.

(b) Precomputation time dataset size scalability with random polytopes in 6D.

Fig. 5: Precomputation time with random polytopes distributed along an axis ("linearly distributed").

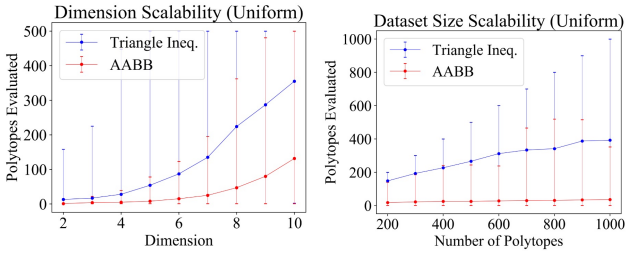
The AABB algorithm is much faster in precomputation. Both algorithms require longer precomputation in higher dimensions due to the increase in complexity of the optimization problems (see Sections III-A and IV-A). Note that the number of key points for triangle inequality was limited to $|K||\mathbf{P}| < 10^6$ in all experiments to maintain a reasonable precomputation time.

C. Online Scalability

The online performances of the AABB and triangle inequality algorithms were tested against dimension and dataset size. We measure performance with the number of polytope distances evaluated on each query. Figure 6 shows the query performance on uniformly distributed random polytopes. Figure 7 shows the query performance on random polytopes distributed along an axis ("linearly distributed").

The AABB algorithm outperformed the triangle inequality algorithm in both median and worst-case performances. The worst-case of both algorithms did not scale well with dimension, with both algorithms requiring checking over 40% of the polytopes starting in 7D. With the dataset size, scaling performance is much better in AABB. Section VII-F contains more discussions on AABB complexity.

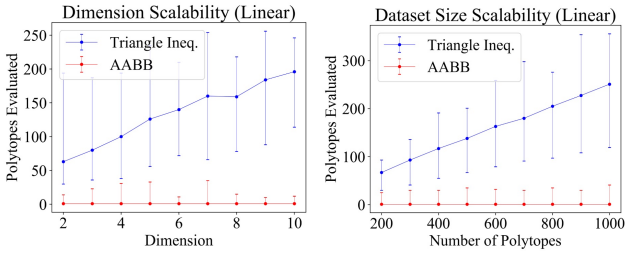
We note that these performances are highly dependent on the structure of the dataset and the query point choice. For instance, if a query point is very far from all of the polytopes, both algorithms will check nearly all of the polytopes. This



(a) Number of polytopes evaluated per query with 500-polytope synthetic datasets.

(b) Number of polytopes evaluated per query with synthetic datasets in 6D.

Fig. 6: Number of polytopes evaluated per query with uniformly distributed random polytopes.



(a) Number of polytopes evaluated per query with 500-polytope synthetic datasets.

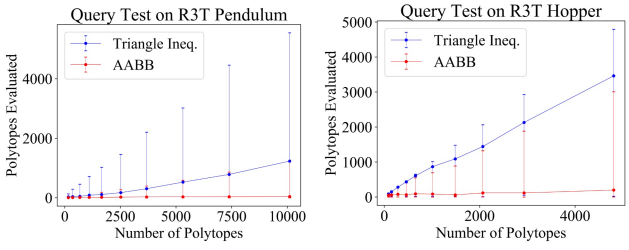
(b) Number of polytopes evaluated per query with synthetic datasets in 6D.

Fig. 7: Number of polytopes evaluated per query with random polytopes distributed along an axis ("linearly distributed").

dependency on the specific query point is supported by the much smaller median values.

D. Performance on R3T Datasets

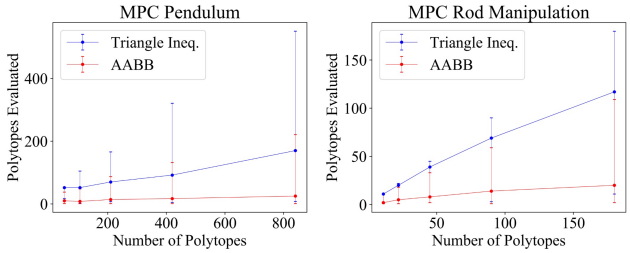
The AABB and triangle inequality algorithms were tested on real polytope data from R3T [8]. These polytopes are reachable sets generated by motion planning in pendulum swing-up (2D continuous system) and planner hopper (10D, 2 contact modes) problems [29], [30]. In this context, only the query performance is relevant since the polytopes are added sequentially during runtime as R3T explores the state space. Figure 8 summarizes the results.



(a) Number of polytopes evaluated per query on the R3T pendulum dataset.

(b) Number of polytopes evaluated per query on the R3T hopper dataset.

Fig. 8: Testing results of the algorithms on R3T datasets.



(a) Number of polytopes evaluated per query on the MPC pendulum dataset.

(b) Number of polytopes evaluated per query on the MPC rod manipulation dataset.

Fig. 9: Testing results of the algorithms on MPC datasets.

The AABB algorithm yielded significantly better performance on R3T datasets. We attribute this result to the process of generating these polytopes. In R3T, the polytopes are generated through simulating forward dynamics for a specific time horizon [8]. Therefore, the polytopes seldom possess elongated shapes that are unfavorable for AABB approximations. With the AABB algorithm, often less than 5% of all polytopes were checked online.

E. Performance on MPC Datasets

The AABB and triangle inequality algorithms were tested on two datasets generated by MPC [6]. The first dataset originates from stabilizing an inverted pendulum by bouncing against a vertical wall (2D, 2 contact modes) [31]. The second dataset represents manipulating a rod with 2 fingers[2] (10D, 16 contact modes). Similar to RRT, we considered different exploration stages of the MPC problem. The results are summarized in Figure 9.

With MPC, the AABB algorithm still outperformed the triangle inequality algorithm by a significant margin. Using the AABB algorithm, typically less than 20% of all polytopes were evaluated, which is a significant improvement from the naive algorithm.

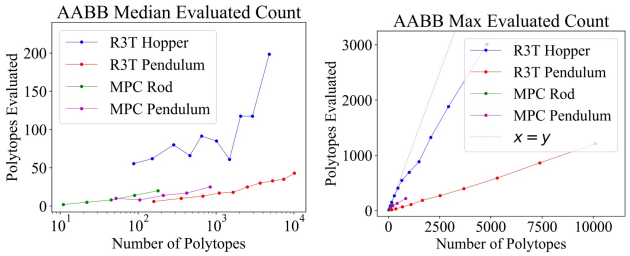
F. Empirical Online Complexity of the AABB Algorithm

Figure 10 shows the AABB algorithm's empirical online complexity in R3T and MPC datasets. The median computation time is roughly logarithmic to the number of polytopes, while the worst-case is linear. However, even in the worst case, only around 50% of all polytopes were evaluated.

VIII. Conclusion and Future Work

We developed 3 algorithms for efficiently solving the nearest polytope problem. Of the three algorithms, AABB and triangle inequality can be scaled up to high dimensions. Through testing on synthetic, R3T, and MPC datasets, we concluded that the AABB algorithm is the superior. All algorithms provide significant querying performance improvements over the naive exhaustive search, with AABB demonstrating roughly logarithmic median performance empirically. The algorithms developed in this paper can be

[2]The scripts and details of this example are available at https://github.com/sadraddini/PWA-Control.git



(a) Median number of polytopes evaluated per query. This is roughly logarithmic to the size of the dataset.

(b) Maximum number of polytopes evaluated per query. This is roughly linear to the dataset size.

Fig. 10: Empirical complexity analysis of the AABB algorithm.

applied to control applications that require solving the nearest polytope problem. Future work will focus on improving the heuristics, combining the algorithms, and further implementations. For instance, while the methods in this paper were purely geometrical, exploiting the underlying dynamics of the polytope data source (such as hybrid system dynamics) can potentially lead to performance improvements.

REFERENCES

[1] J. Snoeyink, "Handbook of discrete and computational geometry," J. E. Goodman and J. O'Rourke, Eds. Boca Raton, FL, USA: CRC Press, Inc., 1997, ch. Point Location, pp. 559–574. [Online]. Available: http://dl.acm.org/citation.cfm?id=285869.285901

[2] A. Bemporad, F. Borrelli, and M. Morari, "Piecewise linear optimal controllers for hybrid systems," in *American Control Conference, 2000. Proceedings of the 2000*, vol. 2. IEEE, 2000, pp. 1190–1194.

[3] P. Tondel, T. Johansen, and A. Bemporad, "Evaluation of piecewise affine control via binary search tree," *Automatica*, vol. 39, no. 5, pp. 945 – 950, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0005109802003084

[4] F. Bayat, T. A. Johansen, and A. A. Jalali, "Using hash tables to manage the time-storage complexity in a point location problem: Application to explicit model predictive control," *Automatica*, vol. 47, no. 3, pp. 571 – 577, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0005109811000240

[5] G. Ziegler, *Lectures on Polytopes*, ser. Graduate texts in mathematics. Springer-Verlag, 1995. [Online]. Available: https://books.google.com/books?id=WtJXAAAAYAAJ

[6] S. Sadraddini and R. Tedrake, "Sampling-based polytopic trees for approximate optimal control of piecewise affine systems," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 7690–7696.

[7] A. Shkolnik, M. Walter, and R. Tedrake, "Reachability-guided sampling for planning under differential constraints," in *2009 IEEE International Conference on Robotics and Automation*. IEEE, 2009, pp. 2859–2865.

[8] A. Wu and S. Sadraddini, "R3t: Rapidly-exploring random reachability set trees," https://github.com/wualbert/r3t, accessed: 2019-09-17.

[9] J. J. Kuffner Jr and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *ICRA*, vol. 2, 2000.

[10] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[11] S. Sadraddini and R. Tedrake, "Linear encodings for polytope containment problems," *arXiv preprint arXiv:1903.05214*, 2019.

[12] F. J. Christophersen, M. Kvasnica, C. N. Jones, and M. Morari, "Efficient evaluation of piecewise control laws defined over a large number of polyhedra," in *2007 European Control Conference (ECC)*, July 2007, pp. 2360–2367.

[13] W. C. Thibault and B. F. Naylor, "Set operations on polyhedra using binary space partitioning trees," in *ACM SIGGRAPH computer graphics*, vol. 21, no. 4. ACM, 1987, pp. 153–162.

[14] A. S. Winter *et al.*, "An investigation into real-time 3d polygon rendering using bsp trees," *Project Dissertation submitted to the University of Wales Swansea in Partial Fulfilment for the Degree of Bachelor of Science*, 1999.

[15] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.

[16] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[17] A. Guttman, *R-trees: A dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

[18] D.-T. Lee and C. Wong, "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees," *Acta Informatica*, vol. 9, no. 1, pp. 23–29, 1977.

[19] A. Alessio and A. Bemporad, *A Survey on Explicit Model Predictive Control*, 05 2009, vol. 384, pp. 345–369.

[20] J. Zhang and X. Xiu, "K-d tree based approach for point location problem in explicit model predictive control," *Journal of the Franklin Institute*, vol. 355, no. 13, pp. 5431 – 5451, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0016003218303685

[21] J. Spjotvold, S. V. Rakovic, P. Tondel, and T. A. Johansen, "Utilizing reachability analysis in point location problems," in *Proceedings of the 45th IEEE Conference on Decision and Control*, Dec 2006, pp. 4568–4569.

[22] M. Baotić, F. Borrelli, A. Bemporad, and M. Morari, "Efficient online computation of constrained optimal control," *SIAM Journal on Control and Optimization*, vol. 47, no. 5, pp. 2470–2489, 2008.

[23] F. Borrelli, *Constrained optimal control of linear and hybrid systems*. Springer, 2003, vol. 290.

[24] C. N. Jones, P. Grieder, and S. V. Raković, "A logarithmic-time solution to the point location problem for parametric linear programming," *Automatica*, vol. 42, no. 12, pp. 2215–2218, 2006.

[25] P. Houthuys, "Box sort, a multidimensional binary sorting method for rectangular boxes, used for quick range searching," *The Visual Computer*, vol. 3, no. 4, pp. 236–249, Dec 1987. [Online]. Available: https://doi.org/10.1007/BF01952830

[26] F. Bayat, T. A. Johansen, and A. A. Jalali, "Flexible piecewise function evaluation methods based on truncated binary search trees and lattice representation in explicit mpc," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 3, pp. 632–640, 2011.

[27] A. N. Fuchs, C. Jones, and M. Morari, "Optimized decision trees for point location in polytopic data sets-application to explicit mpc," in *Proceedings of the 2010 American Control Conference*. IEEE, 2010, pp. 5507–5512.

[28] R. Tedrake, I. R. Manchester, M. Tobenkin, and J. W. Roberts, "Lqr-trees: Feedback motion planning via sums-of-squares verification," *The International Journal of Robotics Research*, vol. 29, no. 8, pp. 1038–1052, 2010.

[29] M. H. Raibert, "Hopping in legged systems — modeling and simulation for the two-dimensional one-legged case," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-14, no. 3, pp. 451–463, May 1984.

[30] R. Tedrake, "Applied optimal control for dynamically stable legged locomotion," 01 2004.

[31] T. Marcucci, R. Deits, M. Gabiccini, A. Biechi, and R. Tedrake, "Approximate hybrid model predictive control for multi-contact push recovery in complex environments," in *Humanoid Robotics (Humanoids), 2017 IEEE-RAS 17th International Conference on*. IEEE, 2017, pp. 31–38.