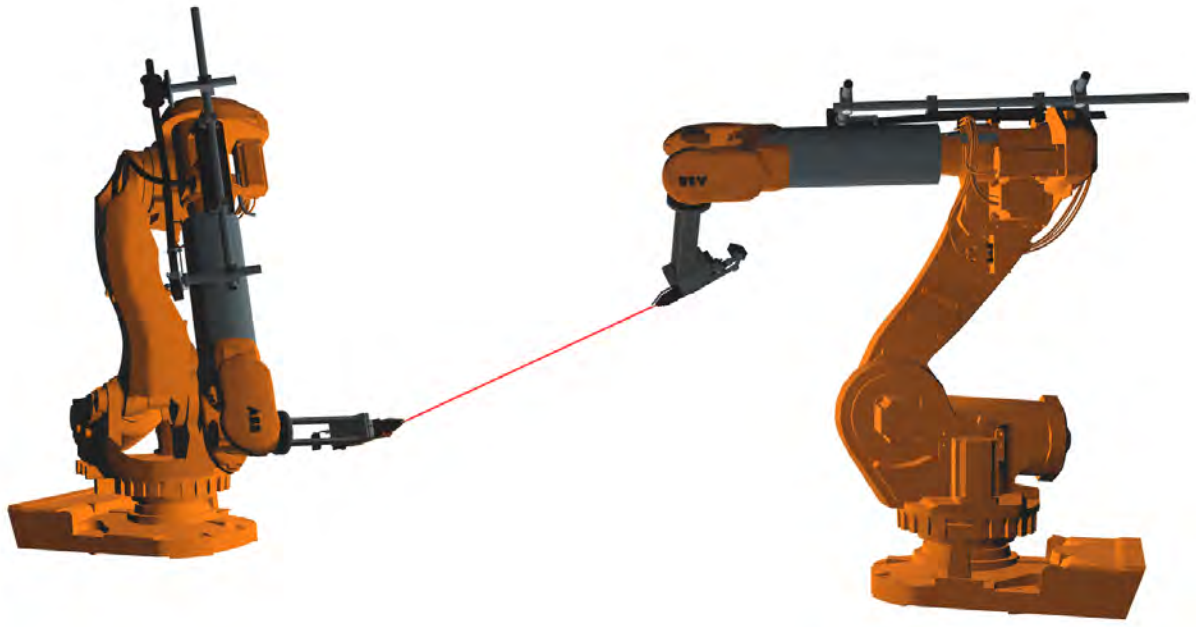
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



FRAUNHOFER CHALMERS
RESEARCH CENTRE FOR INDUSTRIAL MATHEMATICS

Accelerating Proximity Queries

Accelerating Proximity Queries for Non-convex Geometries in a Robot Cell Context

Master's thesis in Complex Adaptive Systems

JOAKIM THORÉN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Accelerating Proximity Queries

Accelerating Proximity Queries for Non-convex Geometries in a Robot Cell Context

JOAKIM THORÉN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Accelerating Proximity Queries
Accelerating Proximity Queries for Non-convex Geometries in a Robot Cell Context
JOAKIM THORÉN

Supervisor: Robert Bohlin, Fraunhofer-Chalmers Centre
Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A red line whose length is the separation distance between two robotic arms.

Typeset in LaTeX
Gothenburg, Sweden 2018

Accelerating Proximity Queries
Accelerating Proximity Queries for Non-convex Geometries in a Robot Cell Context
JOAKIM THORÉN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Sampling-based motion-planners, for example *rapidly exploring dense tree (RRT)* based planners, depend on fast proximity queries. Regrettably, bounding volume tests are significant bottlenecks of proximity queries. Sampling-based motion-planners are therefore accelerated by reducing the number of bounding volume tests. To this end, a novel algorithm called *Forest Proximity Query* (FPQ) is developed. Contrary to previous research, FPQ traverses several pairs of BVHs simultaneously, effectively exploiting an actuality that only a single minimal separation distance — out of several possible separation distances — is required during sampling-based motion-planning. An implementation of FPQ show that FPQ performs up to 67% fewer BV tests in comparison to the well-known Proximity Query Package, increasing proximity querying performance by up to 46%. In conclusion, FPQ is successful in its attempt at improving performance of sampling-based motion-planners.

Keywords: bounding volume traversal tree, priority directed search, front tracking, rectangle swept sphere, proximity query, distance computation, motion planning, non-convex, geometry, robotics.

Acknowledgements

I give my most sincere thanks to Robert Bohlin at Fraunhofer-Chalmers Centre for his kind assistance in the entirety of this thesis, his warm welcoming of me as a master thesis student into Fraunhofer-Chalmers Centre and for his patience with my never ending stream of, sometimes confused, questions. I would also like to thank Tomas Hermansson at Fraunhofer-Chalmers Centre for his valuable discussions regarding my thesis as well as his help with IPS software implementation details. It is only fair to thank all of Fraunhofer-Chalmers Centre, especially my fellow MSc thesis peers Hanna Berggren, Matteo Canavero, Christian Larsen and Fabian Melvås who kept me in compagnia eccellente during the past six months while we worked under the roof of the department of Geometry and Motion Planning, for providing such a friendly and comforting environment for me to carry out my master thesis.

I would also like to thank my supervisor at Chalmers, Erik Sintorn, for providing me valuable and well-needed feedback on my thesis. Ulf Assarsson, who besides silently accomplishing numerous administrative tasks due to his role as my examiner, introduced me to algorithms and data structures for collision-detection in his excellent course TDA362 Computer Graphics at Chalmers University of Technology. Without this clear and motivating introduction to concepts within Computer Graphics, it is not certain that I would have stumbled upon proximity querying.

I would like to thank my family: my mother Monica Thorén, my father Mikael Thorén and my little sister Mimmi Thorén for encouraging me to pursue an academic degree. They have always spurred my various interests without exception. I would be dishonest if I were not to thank my grandparents for firmly believing in me with my academic undertakings — thank you.

Finally and most importantly, I would like to thank my friend and partner Elvira Jonsson. Her reassuring support and sheer joy of life is contagious. She made the past four years a wonderful time.

Joakim Thorén, Gothenburg, May 2018

Contents

Glossary with acronyms

biclique also known as complete bipartite graph

bounding volume (BV) a bounding volume (BV) is a convex geometry enclosing some arbitrary geometry

bounding volume hierarchy (BVH) a bounding volume hierarchy (BVH) is a tree of bounding volumes enclosing some geometry

bounding volume traversal tree (BVTT) a bounding volume traversal tree (BVTT) is a tree, formed from two BVHs, whose nodes are pairs of bounding volumes that are traversed and possibly pruned

bounding volume traversal tree front (BVTT front) A set of roots of pruned sub-trees and a set of leaves of some BVTT, in which traversal terminated

configuration space ($\mathcal{C}$-space) a configuration space ($\mathcal{C}$-space) is a topological space in which a point represents the state of some robot or geometry

duplex priority queue (DPQ) A priority queue with $\mathcal{O}(1)$ insertions and deletions under certain access patterns

dynamic geometry geometry which undergoes rigid transformations

explicit BVTT A BVTT that is (partially) stored in memory

Forest Proximity Query (FPQ) A method of traversing a forest of BVTTs in order to efficiently compute a proximity query

Fraunhofer-Chalmers Centre (FCC) a research company situated in Johanneberg Science Park, Gothenburg

front tracking (FT) Caching collisions between root of BVH and root, internal or leaf nodes of other BVH

generalized front tracking (GFT) Caching of either root, internal or leaf nodes in in a BVTT

geometric object a set of geometric primitives

geometric primitive a set of discs, triangles, tori or points (or similar)

implicit BVTT BVTT which is generated on-the-fly

Industrial Path Solutions (IPS) a commercial motion-planning software developed by Fraunhofer-Chalmers Centre

optimal front Leaves of a minimal bounding volume traversal tree

priority directed search (PDS) Using a priority queue to schedule what nodes to traverse in BVTTs

Proximity Query Package (PQP) a C++ library for computing the separation distance between two geometric objects

rapidly exploring dense tree (RRT) A technique for sampling points in a configuration space by incrementally constructing a space-filling tree

rectangle swept sphere (RSS) A volume generated from dilating a sphere with some rectangle

separation distance minimum distance between two geometries

static geometry geometry which stays put

surface area heuristic (SAH) Splits bounding-volume nodes into children as a function of surface area

triangle soup a triangle soup refers to a set of triangles without any structure or nice properties

world configuration a point in the world configuration space

world configuration space cartesian product of all configuration spaces

List of Figures

List of Tables

1
Introduction

Modern factories rely on robotic arms for assembly, welding, and gluing during a manufacturing process. Efficient usage of robotic arms contributes directly to manufacturing costs and manufacturing run times. However, a problem which must be solved prior to utilizing robotic arms is that of motion-planning. Motion-planning is the process of running an algorithm, sometimes referred to as a motion-planner, which outputs a path for some robot arm. This path governs with next to exact precision the actions of a robot arm at any given time.

A motion-planner is often in need of finding a *separation distance* between two objects, for example robotic arms and a car bodies. A separation distance is the shortest distance between any two points on two separate objects. The problem of finding separation distances between two objects is also known as proximity querying, a well known problem within the field of computational geometry.

A fast motion-planner is preferable to a slow motion-planner for several reasons, some of which are:

- Real-time motion-planning
- Decreased down-time for humans waiting for a motion-planning process to terminate
- Scenes which were previously considered intractable may become tractable

Regrettably, proximity queries are oftentimes significant bottlenecks in motion-planning software. Therefore, there is a demand for efficient proximity querying methods. However, the set of available methods for proximity querying depends on geometry assumptions. Some common assumptions about geometries are: convexity, closedness or orientability. If these assumptions are valid for a problem at hand, then there are fast proximity query algorithms available. In practice, these geometrical properties are not always satisfied.

In a robot-cell context, proximity queries are computed on non-convex and non-closed polytopes, or more generally *triangle soups*. In particular, *Fraunhofer-Chalmers Centre* (FCC) develops a motion-planning software called *Industrial Path Solutions* (IPS) in which robotic arms and end-products are represented (in a virtual prototyping environment) as triangle soups. Further, the motion-planner shipped with IPS is a *sampling-based motion-planner*. Sampling-based motion-planners tries out several configurations of robotic arms during a motion-planning session. In terms of triangle soups, this means that the triangle soups which represent robotic arms or end-products undergo rigid motion. A proximity query is made for each tried out configuration. In summary, a robot-cell context is assumed to be a scene with rigid bodies undergoing rigid motion.

A smallest distance between any two triangle soups can be found by computing all pairwise distances. This method scales quadratically with the number of primitives. Instead, a triangle soup is enclosed by a tree of *bounding volumes* (BVs) commonly known as a *bounding volume hierarchy* (BVH). Bulks of triangle distance computations can be avoided by comparing two BVHs against each other, improving upon the quadratic complexity of testing each triangle against all other triangles. Finally, a *bounding volume traversal tree* (BVTT) defines how any two BVHs should be tested against each other. Hence, the problem of proximity querying is that of traversing a, preferably small, BVTT.

Contribution: I developed a novel method, called *Forest Proximity Query* (FPQ), which finds a separation distance in a forest of BVTTs where each underlying triangle soup undergoes rigid motion. FPQ utilizes two established methods: *Generalized front tracking* (GFT) and *Priority directed search* (PDS). Neither has previously been adapted to forests of BVTTs. Results show that FPQ reduces the number of BV-BV distance computations, in comparison to the state-of-the-art method *Proximity Query Package* (PQP). For a motion-planning scenario, the computation-time is reduced by approximately 40% due to a reduced number of BV tests.

In conclusion, FPQ is a promising method for acceleration of proximity queries in a robot cell context and especially so for motion-planning scenarios.

1.1 Outline

The thesis is divided into the following sections:

- A literature survey of relevant related methods. See Section 1.5 "Previous research".
- Preliminary theory required for understanding a formal problem definition and for understanding FPQ.
- Presentation of FPQ, a novel method for accelerating proximity queries. See Chapter 3 "Forest proximity querying".
- A set of benchmarks comparing various variants of FPQ against each other as well as comparing FPQ with IPS is presented in Chapter 5 "Results"
- A discussion FPQ is presented in Chapter 6 "Discussion".
- A conclusion on whether FPQ successfully accelerates proximity queries or not is presented in Chapter 7 "Conclusion".

In general, each chapter and section is dependent on the previous section or chapter. It is therefore recommended to read this thesis page by page. Readers who are familiar with the contents Section 2.1 "Preliminary Theory" may skip said section.

1.2 Background

Proximity querying can be used as a subroutine within motion-planning. This section explains in detail how proximity querying is used in motion-planning. An

understanding of how proximity querying is used in motion-planning should help the reader to understand why some assumptions are in place.

1.2.1 Proximity querying as subroutine of motion-planners

This section expands upon the dependency of proximity querying from a motion-planning point-of-view.

The problem of motion-planning is that of finding valid paths throughout a space known as *world configuration space* or simply *configuration space* ($\mathcal{C}$-space). A point in $\mathcal{C}$-space can be thought of as a vector of numbers that uniquely determine a state of some robotic arm. Therefore, a path throughout $\mathcal{C}$-space correspond to a motion of a robotic arm. For example, if a robotic arm is governed entirely by three independent real-valued numbers, then the state of the robotic arm is given by a point in $\mathbb{R}^3$ — the $\mathcal{C}$-space of the robotic arm is $\mathbb{R}^3$.

Globally optimal analytic paths from a starting-point to an end-point in a world configuration space, which may or may not satisfy some given set of constraints, are difficult to find. Hence, *sampling-based motion-planners* are introduced. A sampling-based motion-planner samples points on the world configuration space manifold. These points are turned into nodes in a graph. A path throughout the graph corresponds to a path throughout the world configuration space which in turn corresponds to a path for some robot arm. Edges in a path corresponds to a line segment throughout world configuration space.

It is crucial to ensure validity of every edge on any computed path, where validity means that each point on the line segment satisfies a set of given constraints. The set of given constraints is typically that no collisions between robot arms or end-products may occur or that some minimum safety distance between two objects must not be violated. Ensuring validity over a line segment is similar to that of *continuous collision detection*, with the exception that validity in this case means a collision-free *and* safe path with respect to minimum safety distances. Some motion-planners, including the motion-planner included in *Industrial Path Solutions* (IPS), checks validity of line segments one at a time by sampling points on said line. This means that most often, the distance between two points successive points q_i, q_{i-1} in *configuration space* ($\mathcal{C}$-space) is bounded from above: $\|q_i - q_{i-1}\| < d$. A (small) bounded change in configuration points is commonly referred to as *spatial coherence*.

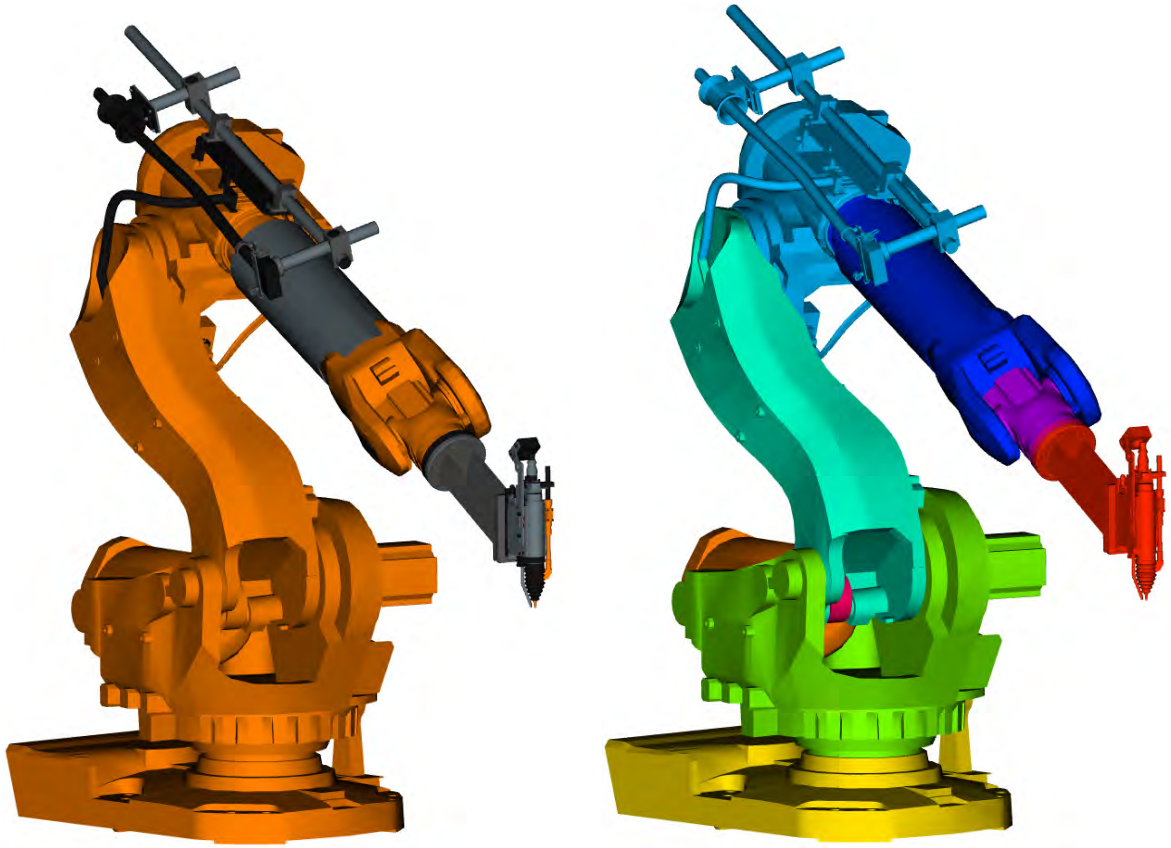
One way to ensure that a motion is collision free is to compute a separation distance between a robot arm and some object. If said separation distance is $r \in \mathbb{R}$, then it is known that a robot arm can move freely within a sphere of radius r.

It is understood that a sampling-based motion-planner does at least two things: Checks validity of sampled points on the world configuration space and checks validity of line segments in the world configuration space. In both cases, the problem of proximity querying arises — efficient proximity queries must be available for quickly determining if a point or line segment in world configuration space is valid or not, which is to say that the set of constraints is respected.

1.2.2 Robot arms as composite rigid bodies

In IPS, there are two sets of *geometric objects* A and B. A set of objects could be, for instance, all rigid parts of some robot arm or some machine element. A minimum distance between A and B corresponds to the smallest distance between some composite object, for example a robot arm, and some other composite object. This is precisely the distance which is required by motion-planners.

All independent rigid parts of an ABB robot arm are colored in Figure 1.1. The whole robot arm corresponds to some set of geometric objects A and its rigid (colored) parts correspond to elements of A.



(a) ABB robotic arm (b) Colorization of (a)

Figure 1.1: ABB robotic arm with colorized rigid components.

Sampling-based motion-planners tries out many different world configuration points. Hence, objects may jump around in space over several iterations. For each world configuration point, a minimum separation distance between two sets of geometric objects A and B must be found. It follows that computation of successive proximity queries is of great importance for motion-planners. Hence, I develop methods for successive computation of the smallest distance between two sets A and B of geometric objects.

1.2.3 Preliminaries

The reader of this thesis is expected to have knowledge acquired through any standard Computer Science programme. In particular, complexity theory, naïve set theory and some introductory course on data structures and algorithms. Some concepts typically found in a course on Computer Graphics, in particular *bounding volume* (BV) and *bounding volume hierarchy* (BVH), are also of high relevance. Other relevant concepts are presented in Chapter 2 "Theory".

1.2.4 FCC

FCC is a research centre located in Chalmers Science Park. Its main focus is mathematical problems in the industry. An expansive competence area at FCC is geometry and motion planning, where virtual assembly, inspection and robot programming is included. An important industrial application is automatic generation of optimized and collision free robot motions, where FCC performs a number of research and development projects in cooperation with customers.

The research partners of this master thesis are FCC and Chalmers.

1.3 Research question

I address the problem of accelerating proximity queries in a robot-cell context. If $\text{dist}(i; A, B)$ is a function which outputs a minimum distance between two sets A and B of geometric objects on some i:th iteration, then the research question can be phrased as follows: "What data structures or algorithms are suitable for acceleration of proximity queries in a robot-cell context? In particular, for successive computation of $\text{dist}(i; A, B)$ where A and B are triangle soups?". Note that this research question encompass development of a new data structure or algorithm if suitable.

It is premature to rigorously define $\text{dist}(i; A, B)$ just yet. A mathematically rigorous definition of $\text{dist}(i; A, B)$ is presented in Section 2.2 "Problem definition", after the necessary theory has been developed. For now, it is enough to think of $\text{dist}(i; A, B)$ as a function which outputs a minimum distance between two sets of geometric objects.

1.4 Limitations

This section presents and motivates various limitations.

Many customers of FCC use consumer-level computers. Therefore, FCC restricts usage of *general-purpose computing on graphics processing units* (GPGPU) algorithms or distributed computing. Threads should be avoided. This is also a constraint stemming from IPS which already utilizes most out of the threading capabilities on a consumer-level machine.

I do not investigate solutions to the problem of finding a separation distance between two BVs. Several papers have already studied said problem [1][2][3][4].

Further, FCC has already developed efficient methods for computing the separation distance between two BVs.

Note that finding the separation distance between two BVs is not the same problem as computing separation distance between two (or a set of) BVHs. The latter problem is presented in great detail in Chapter 3 "Forest proximity querying".

Efficient computation of a minimal separation distance between two geometric primitives, for example two triangles, is out of scope. This is because FCC has developed efficient methods for distance computations between primitives. Further, the topic of efficient primitive distance computations has been studied extensively. Instead, I focus on efficient computation of BVH-BVH tests.

In a virtual prototyping robot-cell context geometric objects may be arbitrary triangle soups. Hence no assumptions on convexity, closedness or orientabiltiy of geometry can be made.

1.5 Previous research

In this section, previous research on proximity queries is presented. Research on collision detection is presented too since the problem of collision detection shares the structure of proximity queries. Larsen at al. explains said correspondence in greater detail[1]. I intend to make it clear that there is a wide-range of possible methods for accelerating proximity queries.

A method by Lin and Canny [5] or another method by Gilbert, Johnson and Keerthi [6] assumes that all geometries in a scene are convex and exploits the convexity to speed up proximity queries. Ehmann and Lin present a method which decomposes non-convex geometries into convex geometries and then uses preexisting methods for convex geometries to solve the proximity query problem [3]. Jyh-Ming Lien and Nancy M. Amato presents a method for decomposing geometry into approximately convex components [7]. Convex decomposition of non-convex polygons has been used for computing Gromov-Hausdorff and Gromov-Fréchet metrics [8]. Convex decomposition is attractive since it enables usage of fast proximity querying algorithms on non-convex geometries. A standard proximity query algorithm for convex geometries is the GJK-algorithm[6]. However, convex decomposition is not universally applicable to all types of non-convex geometry. Hollow cylinders, for instance, are notoriously difficult to decompose into a set of convex geometries. Proximity queries in a robot-cell context must be applicable and efficient on arbitrary triangle soups.

A common approach to proximity querying is that of enclosing triangle soups within BVHs. Hierarchies of *rectangle swept spheres* (RSSs) can provide competitive performance in comparison to alternative methods [1][2][3][9]. The renowned PQP library[10] implements BVHs of RSSs. PQP does not rely on GPGPU techniques or other special hardware nor does it assume convexity of geometries. *Robotic Proximity Query package* (RPQ)[11] is an extension of PQP which utilizes kinematic information on robotic arms in order to quickly determine if there has been a collision or not.

Triangle caching[1] is suggested by Larsen et al.[1]. A pair of triangles that gave the separation distance in a former iteration is cached. This pair of triangles can be tested directly in a following iteration. If coherence is high, then it is likely that the obtained distance is also the shortest or at least close to the shortest distance.

Inner Sphere Trees have been suggested by Weller and Zachmann for performing approximate proximity queries[12]. An Inner Sphere Tree is a tree of spheres whose union approximates some geometry. This approach could be feasible for geometric objects with a low surface-area-to-volume ratio. Inner sphere trees reportedly perform 50x as fast as PQP with an average error of 0.15%. Inner Sphere Trees has been used by Kaluschke et al. for computing distances between robotic arms and point clouds [13]. Unfortunately, geometries in a robot-cell context may have a high surface-area-to-volume ratio. Further, triangle soups do not lend themselves well to sphere-packing since triangle soups are not necessarily closed.

Klosowski presents *front tracking* (FT) in his PhD dissertation. FT is a technique which may reduce the number of BV-tests by caching BV from a single BV for which traversal terminated, collectively denoted as a *front*. Klosowski measures an average 19% increase in performance for a scene containing a moving airplane that is tested against static geometry [14]. S. Ehmann and M. Lin generalizes FT by considering pairs of BVs, instead of just BVs, denoting the generalization as *generalized front tracking* (GFT).

GFT has been used extensively for collision detection. Otaduy and Lin utilizes fronts for multiresolution collision detection [15]. Tang et al. presents parallel GFT [16][17]. L. Yen-Tsai and C. Jin-Shin proposes resetting a front if it remains fixed over two consecutive frames [18]. O. Tropp et al. suggest to "prune" a front if motion is sufficiently large, where pruning decreases the size of a front [19]. A localized and streamed GFT is presented by M. Tang et al.[20], with "localized" meaning that only a (changed) subset of the GFT initiates traversal in a following iteration.

GFT has been used for distance computations in addition to collision detection. S. Ehmann and M. Lin studies both collision detection and distance computation by using fronts with operations "dropping" and "raising", together with a generalized type of front tracking denoted GFT in which a front is used to initiate traversal [3]. Said article concludes that GFT it beneficial in all of its tested benchmarks. Lauterbach et al. studies FT for distance computations, suggesting a GPU-based FT method and that two sibling nodes within a front can be replaced by their common parent [9]. To the best of my knowledge, only S. Ehmann and M. Lin with Lauterbach et al. investigates GFT for distance computations.

T. Akenine-Möller with T. Larsson reports that GFT may not pay off if flat BVHs are used [21]. However, no assumption on the height of a BVH is made in this thesis. Hence, GFT could still prove profitable. X. Zhang and Y. Kim highlights memory-issues which could occur with usage of a front [22]. Memory issues are investigated in Section 2.4.1 "Memory usage of front tracking".

Some authors and libraries use a broad-phase step for checking collisions. The *SWIFT*-library assumes convexity of geometries and applies a *Sweep and Prune* approach to cull away uninteresting pairs of geometries and then proceeds to perform distance-queries with a modified Lil-Canny algorithm [23][24][3]. SWIFT assumes

[1] A similar technique in a ray-tracing context is known as *shadow caching*.

that geometries are orientable 2-manifolds[3]. The *ncollide* library, developed mostly by S. Crozet, employs a broad-phase strategy to accelerate proximity queries and it relies on composition of convex geometries represented by *support mappings* for representation of non-convex geometry[25]. *I-COLLIDE*[26] and *Flexible Collision Library* (FCL)[27] uses Sweep and Prune to reduce exact collision tests. Broad-phase methods are typically employed in scenes with thousands of independent dynamic geometries. For few dynamic geometric objects it might not be worthwhile to employ a broad-phase proximity querying. In a robot-cell context, there are indeed many (millions) of triangles but not necessarily thousands of dynamic independent geometries. *The Computational Geometry Algorithms Library* (CGAL) project utilizes a kd-tree to accelerate proximity queries defined on axis-aligned bounding-boxes [28].

2
Theory

This chapter introduces theoretical concepts which are required for a comprehensive understanding of this thesis, leading up to a formal problem definition. After a presentation of the formal problem definition, a foundational concept known as *bounding volume traversal tree* (BVTT) is presented. Then, the concept of *front tracking* (FT) is presented in terms of BVTTs.

2.1 Preliminary Theory

The purpose of this section is threefold. Firstly, to introduce some simple constructions in graph theory which are used to define the problem in Section 2.2 "Problem definition". Secondly, I aim to introduce some concepts in computational geometry used throughout this thesis. In particular, concepts related to proximity querying. Finally, to introduce elementary concepts of topology which are required for introducing the concept of $\mathcal{C}$-spaces. $\mathcal{C}$-spaces are introduced in Section 2.2 "Problem definition". The following subsections can be skipped if the reader has prior knowledge of basic topology, basic graph theory and proximity querying.

2.1.1 Preliminary graph theory

While this thesis does not concern itself with graph theory to any greater extent, a useful concept called *bicliques* is used in order to define the problem of this thesis in Section 2.2 "Problem definition".

One may think of two disjoint sets of nodes with some edges in between said sets. Formally, a *bipartite* graph $G = (V, E)$ is an undirected graph where there exists two subsets of vertices $V_1, V_2 \subset V$ such that $V_1 \cap V_2 = \emptyset \wedge V_1 \cup V_2 = V$ with every edge $e \in E$ connecting one node from V_1 to another node from V_2.

A bipartite graph can be made into a *complete bipartite graph*, also known as a *biclique*, by connecting all nodes from one set to another set. Formally, a biclique is a bipartite graph $G = (V, E)$ with $E = \{e_{i,j} : \forall i \in V_1, \forall j \in V_2\}$. A biclique can be denoted $G = (V_1, V_2, E)$. The number of edges in a biclique $G = (A, B, E)$ is $|E| = |A| \cdot |B|$. In Section 2.2 "Problem definition" it will be shown that $|E|$ is the number of proximity queries which has to be computed. The number of nodes in a biclique is $n = |A| + |B|$. See Figure 2.1 for an illustration of a biclique.

See the book by Richard J. Trudeau "Introduction to Graph Theory"[29] for further introduction to graph theory.

A special case of a graph is a *tree*. I assume that a reader has previous experience with trees. In order to avoid any confusion, I present my terminology related to trees

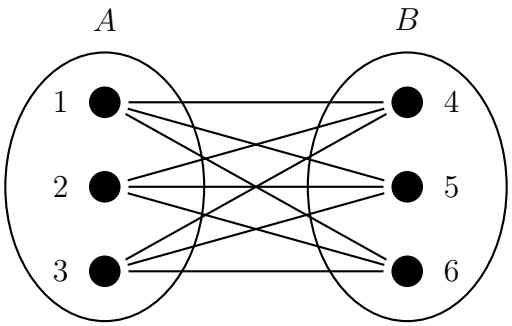


Figure 2.1: A biclique $G = (A, B, E)$ with $|A| = 3$, $|B| = 3$.

used throughout this thesis:

Tree: A *tree* $\mathcal{T} = (V, E)$ is a directed graph without any cycles, where all paths from one node to another are unique. I occasionally use "$v \in \mathcal{T}$" as a shorthand for "$v \in V$ with $\mathcal{T} = (V, E)$".

Root: A node $v \in \mathcal{T}$ is a root if it has no incoming edges.

Children and parents: A node $v_c \in \mathcal{T}$ is a *child* of $v_p \in \mathcal{T}$ if there is an edge going from v_p to v_c. In this case, I say that v_p is a parent of v_c.

Sibling: Two nodes $v_l, v_r \in V$ are siblings iff they have the same parent.

Parent set: A parent set $P(v) \subseteq \mathcal{T}$ is all nodes on the path from v to the root of $\mathcal{T}$ (not including v but including the root of $\mathcal{T}$. Intuitively, $P(v)$ simply holds the parent of v and the parent of the parent of v, and so forth, until the root which is also a member of the parent set.

2.1.2 Preliminary (computational) geometry

The definitions of *bounding volume* (BV) and *bounding volume hierarchy* (BVH) are not mathematically rigorous but instead aim to provide intuition.

Definition 1. ***Bounding volume:*** *A bounding volume (BV) $\mathcal{A}_{bounding}$ is some convex geometry, oftentimes an sphere, oriented block, or rectangle-swept-sphere, that bounds some other not necessarily convex geometry $\mathcal{A} \subseteq \mathcal{A}_{bounding}$, for example torii or teapots.*

A most common usage of BVs is to first check some distance or collision between BVs before computing exact and expensive tests between complicated geometries.

Definition 2. ***Bounding volume hierarchy:*** *A bounding volume hierarchy (BVH) is a tree of BVs, in which each parent (most commonly) bounds all of its children.*

In this thesis it is assumed that every leaf in a BVH holds exactly one triangle.

Further treatment of BVs and BVHs can be found in the book "Real-Time Rendering" by Tomas Akenine-Möller et al.[30, pp.647–650].

2.1.3 Preliminary topology

The bulk of this thesis does not depend on having solid understanding of the topological definitions below. Hence, the definitions below will not be explained thoroughly, although they are foundational with respect to the problem definition of this thesis. The reader should be able to understand this thesis by thinking of a $\mathcal{C}$-space as a set of "states" describing a robotic arm.

The topological definitions below originate from Steven LaValle's introduction to topology and $\mathcal{C}$-spaces in Chapter 4 "The Configuration Space" of the book "Planning Algorithms"[31, pp.127–155]. Mark de Berg et al. gives an introduction to $\mathcal{C}$-spaces for undergraduates in Chapter 13 of the book "Computational Geometry"[32, pp.283–303].

A set $\mathcal{X}$ is called a *topological space* if there is a collection of subsets of $\mathcal{X}$ called *open sets* for which the following axioms hold:

1. The union of any number of open sets is an open set.
2. The intersection of a finite number of open sets is an open set.
3. Both $\mathcal{X}$ and $\emptyset$ are open sets.

A *homeomorphism* is a continuous and bijective function $f : \mathcal{X} \to \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are two topological spaces. Two topological spaces are said to be *homeomorphic* if a homeomorphism exists in between them, $\mathcal{X} \cong \mathcal{Y}$. A homeomorphism induces an equivalence class. Hence, another way of saying that two topological spaces are homeomorphic is to say that they are *topologically equivalent*. A common saying is that coffee cups and doughnuts are equal (homeomorphic) since a coffee cup can be reshaped into a donut and vice versa.

A topological space $M \subseteq R^m$ is a *manifold* if for every $x \in M$, an open set $O \subset M$ exists such that:

1. $x \in O$
2. O is homeomorphic to R^n
3. n is fixed for all $x \in M$

An *n-dimensional real projective space* $\mathbb{RP}^n$ is the set of all lines $\mathbb{R}^{n+1}$ that pass through the origin. These spaces are used to represent *rigid transformations*. For this thesis, $n = 3$ unless otherwise stated.

2.2 Problem definition

A mathematical formulation of the problem which I address is developed in this section. The mathematical formulation of the problem depends on some crucial definitions. These definitions are presented prior to introducing the mathematical formulation of the problem. Note that these definitions are not necessarily standard.

A geometric object is a set $\mathcal{O} = \{o_1, \ldots, o_n\}$ of n *geometric primitives* $o_1, \ldots, o_n$. A geometric primitive is just some well-behaved set of points, for example a disc, triangle, torus or a single point.

The separation distance between two geometric primitives $o_{\mathcal{A}}$ and $o_{\mathcal{B}}$ is defined as:

$$\|o_{\mathcal{A}} - o_{\mathcal{B}}\| = \min_{a \in o_{\mathcal{A}}, b \in o_{\mathcal{B}}} |a - b| \tag{2.1}$$

A *separation distance* between two geometric objects $\mathcal{A}, \mathcal{B}$ is defined as:

$$\|\mathcal{A} - \mathcal{B}\| = \min_{o_{\mathcal{A}} \in \mathcal{A}, o_{\mathcal{B}} \in \mathcal{B}} \|o_{\mathcal{A}} - o_{\mathcal{B}}\| \tag{2.2}$$

Let A, B be two sets of geometric objects and let $G = (A, B, E)$ be the biclique induced by A and B with undirected edges E. Each undirected edge $e_{\mathcal{A},\mathcal{B}} \in E, \mathcal{A} \in A, \mathcal{B} \in B$ induces a *distance measure* $\mathcal{M}$ where $\mathcal{M}(e_{\mathcal{A},\mathcal{B}}) = \|\mathcal{A} - \mathcal{B}\|$ between the two geometric objects $\mathcal{A}$ and $\mathcal{B}$. The distance $\|\mathcal{A} - \mathcal{B}\|$ will be referred to as the separation distance between two geometric objects $\mathcal{A}$ and $\mathcal{B}$.

The distance between two sets of geometric objects A, B is defined as

$$\|A - B\| = \min \mathcal{M}(E) \tag{2.3}$$

A *configuration space* ($\mathcal{C}$-space) of a geometric object $\mathcal{A}$ is a topological space, usually a manifold, $\mathcal{C}_{\mathcal{A}}$. A *configuration* of some geometric object $\mathcal{A}$ is a point $q_{\mathcal{A}}(i) \in \mathcal{C}_{\mathcal{A}}$ that determines the state of a $\mathcal{A}$, for example, its position and orientation.

A *world configuration* $q(i) : \mathbb{N} \to \mathcal{C}$ outputs a point, in the topological space called *world configuration space* $\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2 \times \ldots \times \mathcal{C}_n$, which determines the position and orientation of all n geometric objects.[1] A world configuration $q(i)$ can be thought of as a complete description of some scene at time or iteration i.

A geometry $\mathcal{A}$ is *static* iff $|\mathcal{C}_{\mathcal{A}}| = 1$ ie its configuration $q_{\mathcal{A}}(i) = q_{\mathcal{A}}$ is a well-defined constant or *dynamic* iff $\mathcal{C}_{\mathcal{A}} \subseteq \mathbb{R}^3 \times \mathbb{RP}^3$, i.e. $\mathcal{A}$ is a rigid body. For convenience, an edge $e_{\mathcal{A},\mathcal{B}}$ is called static if both $\mathcal{A}, \mathcal{B}$ are static (and similarly for dynamic geometry). Consult Steven LaValle's book on Motion Planning named "Motion Planning"[31] for in-depth details of $\mathcal{C}$-spaces or the book "Principles of Robot Motion"[33] by Choset et al. for a less rigorous introduction to $\mathcal{C}$-spaces.

A technical detail should be illuminated: Now that $\mathcal{C}$-spaces have been introduced, $\mathcal{A}$ is no longer a geometric object but a function from a point in the $\mathcal{C}$-space $\mathcal{C}_{\mathcal{A}}$ to a geometric object. This should not cause any confusion. A geometric object *given some configuration* is simply denoted $\mathcal{A}(q_{\mathcal{A}})$. A geometric object *without a configuration* is referred to as $\mathcal{A}$.

It is time to define the problem that I address with this master thesis. Let $\mathcal{C}_A$ and $\mathcal{C}_B$ be two world configuration spaces, where geometric objects are either static or dynamic. The problem is to find an efficient way to successively compute:

$$\text{dist}(i; A, B) = \|A(q_A(i)) - B(q_B(i))\|, q_A(i) \in \mathcal{C}_A, q_B(i) \in \mathcal{C}_B, i = 1, 2, \ldots \tag{2.4}$$

The notion $A(q_A(i))$ simply means the i:th world configuration of some set of geometric objects A. Visually, Equation 2.4 represents the length of the red line in Figure 2.2 given $q_A(i)$ and $q_B(i)$. In plain English: the problem definition is to successively compute the minimum distance between two sets of dynamic geometric object, where the geometric objects jump around in some scene.

[1] The continuous function $q(t) : \mathbb{R} \to \mathcal{C}$ could be defined if the geometries were to move around continuously. In this thesis, geometries are not assumed to move around continuously in space.

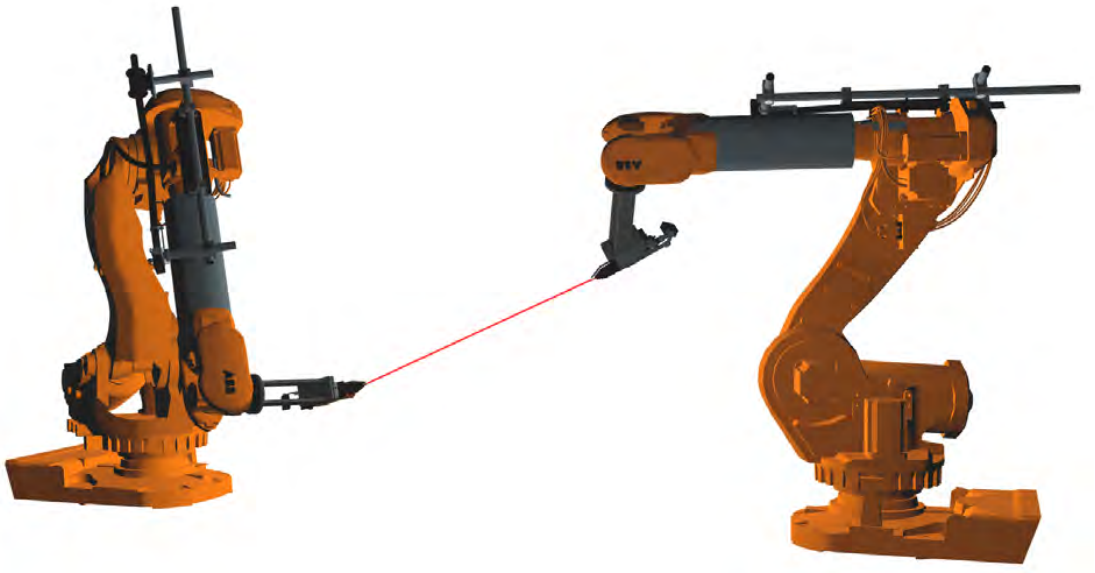


Figure 2.2: Two ABB robot arms with a red line whose end-points represents the two points which give the smallest separation distance between the two robot arms. Equation 2.4 represents the length of the red line for some iteration i.

2.3 Bounding volume traversal trees

This section introduces the concept of a BVTTs, which is crucial for understanding how proximity queries (successive computations of $\text{dist}(i; A, B)$) are computed. BVTTs are useful since they enable the language of trees (for example: children, parents, and siblings) to be used when reasoning about a BVH-BVH test. BVTTs could be considered moderately abstract objects. For this reason, a soft introduction to BVTTs is given. A formal description of BVTTs is given in Section 2.3.4 "BVTTs generation".

Before presenting BVTTs, some elementary topics in Computer Graphics are revisited.

2.3.1 Polygon mesh

Recall Equation 2.1. This equation states that the minimum distance between two geometric objects is the smallest separation distance over all its geometric primitives. Triangles are instances of a geometric primitives. A *polygon mesh* is typically a set of connected triangles. It is understood that Equation 2.1 in practice describes the separation distances between two triangles (or points, lines) of any two polygon meshes. In practice, geometric objects are most often polygon meshes.

2.3.2 Bounding volume hierarchies

Given a polygon mesh $\mathcal{A}$, it is common to compute a BVH which encloses $\mathcal{A}$. A *bounding volume hierarchy* (BVH) is a tree of BVs enclosing some geometric object. Common variants of BVHs include, but are not limited to, axis-aligned

bounding-boxes (AABBs), oriented bounding-boxes (OBBs), discrete oriented polytopes (DOPs) or RSSs. The PQP library uses *rectangle swept sphere* (RSS) as BVs to accelerate proximity queries[1]. I do not intend to provide a thorough survey of BVHs since methods developed throughout this thesis does not depend on a choice of BV-type.

2.3.3 BVTTs represent BV tests

BVHs are oftentimes concerned with the problem of separation distance as formulated in Equation 2.2, in particular by computing a BVH-BVH test. A BVH-BVH test can be represented by a *bounding volume traversal tree* (BVTT)[1][2][3][27][34].

A BVTT is formed from two different BVHs that encapsulates two different geometric objects. The BVTT determines what pairs of BVs must be tested against each other prior to testing two geometric primitives. BVTTs has been studied by various authors [1][2][3][17][27][20][34]. The concept of BVTTs is foundational to this thesis. See Figure 2.3 for an example of a BVTT formed from two BVHs.

Any BVH-pair gives rise to a BVTT and vice versa. Each node in a BVTT represents a pair of BVs. Hence, traversal of a BVTT is traversal of pairs of BVs. A BVTT for two BVHs encapsulating two geometric objects $\mathcal{A}, \mathcal{B}$ is used to compute Equation 2.1. When BVTT traversal terminates, a separation distance between the two underlying geometric objects has been found. More on generation and traversal of BVTTs in Section 2.3.4 "BVTTs generation" and Section 2.3.5 "BVTTs traversal", respectively.

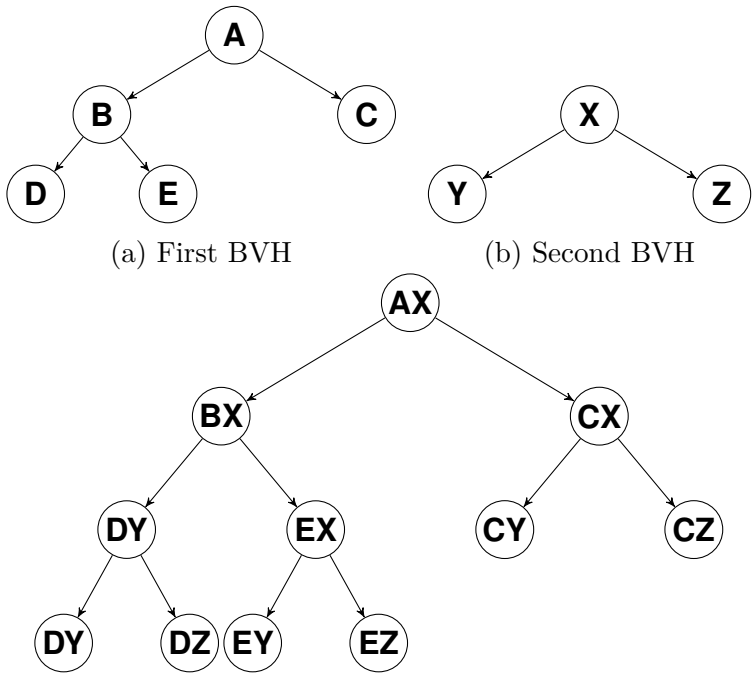
Note that a BVTT maps bijectively to an edge $e_{\mathcal{A},\mathcal{B}} \in E, G = (A, B, E)$ where A, B are two sets of geometric objects, since an edge $e \in E$ corresponds to two geometric objects which in turn correspond to two BVHs. This means that the number of BVTTs is $|A| \cdot |B|$.

2.3.4 BVTTs generation

A node $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ of some BVTT has a pair of BVs encapsulating two geometric objects $\mathcal{A}$ and $\mathcal{B}$, respectively. For convenience, the distance of a node $\|\mathcal{T}_{\mathcal{A},\mathcal{B}}\|$ in a BVTT is defined as the distance between its two underlying BVs, $\|\mathcal{A} - \mathcal{B}\|$. This means computing a BVTT node separation distance is the same as computing a separation distance between two BVs.

A common property of BVTTs is that if $\|\mathcal{T}_{\mathcal{A},\mathcal{B}}\| = d$ then $\|\mathcal{T}_c\| \geq d$, where $\mathcal{T}_c$ is a child BVTT node of $\mathcal{T}_{\mathcal{A},\mathcal{B}}$. Hence if some $\|\mathcal{T}_{\mathcal{A},\mathcal{B}}\| > \epsilon$ then every BV-pair of all children nodes are farther apart than ϵ. This property makes it possible to prune children who can not possibly give rise to a smallest separation distance and is by some authors referred to as the *bounding property*[12].

It is now explained how to construct a BVTT with the property mentioned above. A BVTT-node $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ is a pair of BVHs-nodes, $\mathcal{T}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{B}}$. A left child of $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ is $\mathcal{T}_{\mathcal{A}_{left},\mathcal{B}}$, where $\mathcal{T}_{\mathcal{A}_{left}}$ is the left child of BVH-node $\mathcal{T}_{\mathcal{A}}$. A right child of $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ is $\mathcal{T}_{\mathcal{A}_{right}}, \mathcal{T}_{\mathcal{B}})$. In order to ensure generation of all BVTT-nodes which represents leaf-leaf tests, a special rule must be enforced: A left child of $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ is given from $(\mathcal{T}_{\mathcal{A}}, \mathcal{T}_{\mathcal{B},left})$ if $\mathcal{T}_{\mathcal{A},left}$ does not exist and if $\mathcal{T}_{\mathcal{B},left}$ exists. If neither $\mathcal{T}_{\mathcal{A},left}$ nor $\mathcal{T}_{\mathcal{B},left}$ exists, then a left child of $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ does not exist. In summary:



(a) First BVH (b) Second BVH

(c) A BVTT formed from the first BVH in (a) and the second BVH in (b)

Figure 2.3: Illustration of how two BVHs combine into a single BVTT. All combinations of BVH leaf nodes D, E, C and Y, Z are leaf nodes in the BVTT. In practice, leaf nodes of BVHs are often triangles. BVTT-leaves are all triangle-triangle tests which has to be carried out in a BVH-BVH test.

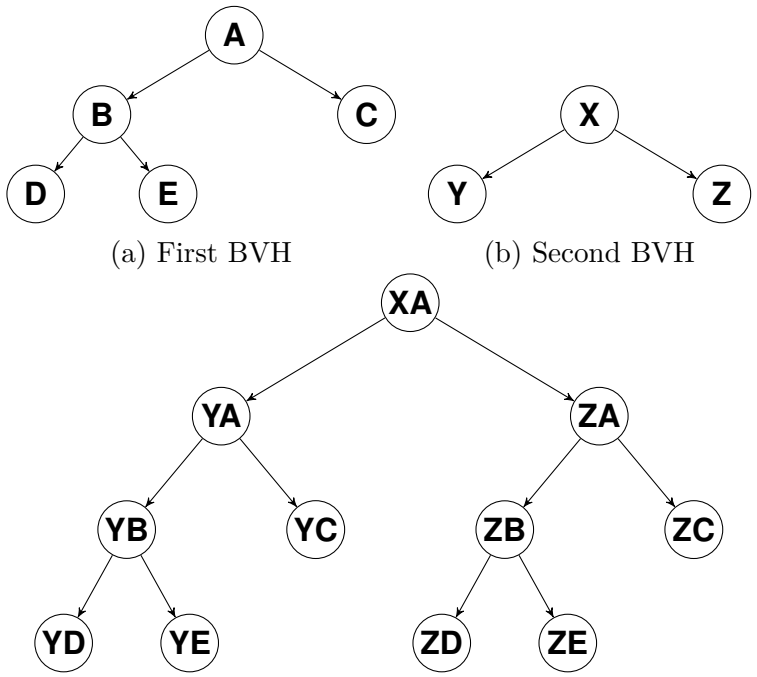
$$\text{left}(\mathcal{T}_{\mathcal{A},\mathcal{B}}) = \begin{cases} \mathcal{T}_{\mathcal{A}_{left},\mathcal{B}} & \text{if } \mathcal{A}_{left} \neq \emptyset \\ \mathcal{T}_{\mathcal{A},\mathcal{B}_{left}} & \text{if } \mathcal{A}_{left} = \emptyset \wedge \mathcal{B}_{left} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \tag{2.5}$$

A completely analogous rule is enforced for generation of right children within BVTTs.

Figure 2.3 should help illustrate how two BVHs can be combined into a BVTT.

Note that in general it is not certain that a child BV is enclosed by its parent BV. Phrased mathematically, $\exists \text{BV}_c : \text{BV}_c \not\subseteq \text{BV}_p$ where BV_p is a parent of BV_c. For example, BVHs built out of RSSs do not satisfy the bounding property since RSSs are inflated, during BVHs construction, without respect to parent BVs [1].

If children are not enclosed by their parents, then the bounding property may not hold. However, it is always true that a parent BV encloses geometric primitives enclosed by a child BV. Therefore, if some parent BVTT-node has a distance greater than some d, a child with distance $d' < d$ can be safely pruned if $d > \epsilon$ although $d' < \epsilon$ since geometric primitives within said children are at least d apart. A critical property has been demonstrated; if some BVTT-node has a distance $d > \epsilon$ then all children can be safely pruned since no geometric primitive-test can give rise to δ.



(a) First BVH (b) Second BVH

(c) A BVTT formed from the first BVH in (a) and the second BVH in (b)

Figure 2.4: Illustration of an alternative combination of two BVHs in (a) and (b) into a single BVTT in (c). All combinations of BVH leaf nodes D, E, C and Y, Z are leaf nodes in the BVTT.

Another (equivalent) way to think of this property is that any node can be safely pruned if it has a parent whose distance is greater than a smallest known distance.

Any BVTT node is a function of two known BVs — given $\mathcal{T}_{\mathcal{A},\mathcal{B}}$, its left child is generated as left($\mathcal{T}_{\mathcal{A},\mathcal{B}}$) and its right child is generated as right($\mathcal{T}_{\mathcal{A},\mathcal{B}}$). Information available on some parent is sufficient for generating its children.

A parent node of any BVTT-node can be computed in constant time, assuming that any BVH-node has constant time access to its parent:

$$\text{parent}(\mathcal{T}_{\mathcal{A},\mathcal{B}}) = \begin{cases} \mathcal{T}_{\mathcal{A},\mathcal{B}_{parent}} & \text{if } \mathcal{B}_{parent} \neq \emptyset \\ \mathcal{T}_{\mathcal{A}_{parent},\mathcal{B}} & \text{if } \mathcal{B}_{parent} = \emptyset \wedge \mathcal{A}_{parent} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \tag{2.6}$$

Any second BV-parent in a BVTT-node takes priority over any first BV-parent in said BVTT-node. In case no parent exists, then the BVTT-node is a root.

Another BVTT-generation scheme is illustrated in Figure 2.4. An important observation is that both BVTTs in Figure 2.4 and Figure 2.3 have identical leaves, but intermediate BVTT-nodes vary. A consequence is that two different generation-schemes for two BVHs may perform different BV-BV tests and as such have different pruning capabilities.

MacDonald and Booth has suggested a BVH space subdivision scheme based on BV surface area [35], commonly known as *surface area heuristic* (SAH) within

the ray-tracing community. An approach used by IPS, inspired by MacDonald and Booth, is the following: Choose AB and AC as children of AX (or equivalently XA) if and only if the surface area of A is greater than the surface area of X, otherwise choose AY and AZ as children:

$$\text{child}(\mathcal{T}_{\mathcal{A},\mathcal{X}}) = \begin{cases} (\mathcal{T}_{\mathcal{B},\mathcal{X}}, \mathcal{T}_{\mathcal{C},\mathcal{X}}) & \text{if } A(A) > A(X) \\ (\mathcal{T}_{\mathcal{A},\mathcal{Y}}, \mathcal{T}_{\mathcal{A},\mathcal{Z}}) & \text{otherwise} \end{cases} \tag{2.7}$$

This children generation scheme ensures that nodes with large surface areas are split into its two children as early as possible. This reduces BV-BV collisions and increases BV-BV distances, making more BVTT nodes prunable and therefore reducing the set of traversed nodes on any iteration. Note that parent($\mathcal{T}_{\mathcal{A},\mathcal{B}}$) has to be adjusted accordingly.

In conclusion: A BVTT is a tree whose nodes represent a BV-BV test. If $\|\mathcal{T}_{\mathcal{A},\mathcal{B}}\| > \epsilon$ then no children need further investigation and consequently all grand-children representing geometric primitive tests can be omitted.

2.3.5 BVTTs traversal

A depth-first BVTT-traversal scheme, which finds a smallest separation distance δ, is presented in this section. The depth-first traversal scheme functions as a demonstration of how traversal throughout a BVTT can be done. This demonstration also illustrates exactly how a separation distance can be found given a BVTT root, an upper bound ϵ on δ the separation distance δ and a configuration point q. *Priority directed search* (PDS) is presented in light of said demonstration.

A BVTT can be traversed by depth-first, breadth-first or by any other traversal scheme. If any two BVs of a node in a BVTT are farther away than ϵ, then the sub-tree with said node as root can be safely pruned, speeding up BVTT traversal. A distance between two BVs, given a world-configuration point q, on some BVTT node $\mathcal{T}$ is denoted $\|\mathcal{T}(q)\|$. A depth-first BVTT traversal scheme, which returns a smallest separation distance, is presented in Algorithm 1 "Depth-first search". Checking existence of left and right child is omitted for brevity. A distance between two geometric primitives on a BVTT leaf l is denoted as $\|l\|_{leaf}$.

Algorithm 1 Depth-first search

Require: $\mathcal{T}$ a node in a bounding volume traversal tree

```
1:  function TRAVERSE(T, ε, q)
2:      if ‖T(q))‖ ≤ ε then
3:          if T is leaf then
4:              ε ← min(ε, ‖T(q)‖)                    ▷ Update ε
5:          else
6:              ε_l ← TRAVERSE(left(T)), ε, q)
7:              ε ← min(ε, ε_l)
8:              ε_r ← TRAVERSE(right(T)), ε, q)
9:              ε ← min(ε, ε_r)
        return ε
10: end function
```

Note that in Algorithm 1 "Depth-first search", ϵ may be updated whenever a leaf has been hit. Also note that pruning of sub-trees is dependent on ϵ. It is clear that a small ϵ would cause many sub-trees to be pruned on line 2. However, it is not certain that greedily diving into the left-most leaf gives a small ϵ. It could be better to have a traversal scheme which intelligently dives towards a small ϵ, or similarly, to have a traversal scheme which solely traverses a *minimal BVTT*.

Definition 3. ***Minimal BVTT:*** *The set of BVTT nodes of a BVTT $\mathcal{T}$ that are traversed with $\epsilon = \|\mathcal{A} - \mathcal{B}\|$.*

A minimal subtree of a BVTT is shown in Figure 2.5.

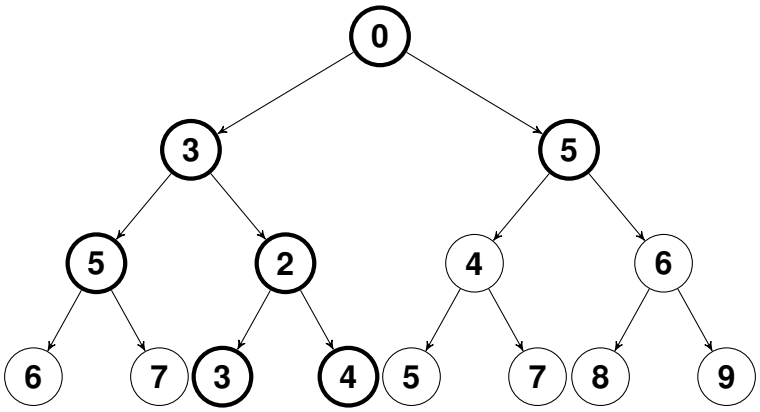


Figure 2.5: A full BVTT is shown. It has a minimal BVTT, indicated by thick nodes. Node distances are indicated within nodes. The smallest leaf has distance $\|\mathcal{A} - \mathcal{B}\| = 3$.

A smallest separation distance will henceforth be referred to as either the smallest separation distance or simply δ. The notion of a smallest separation distance δ can be found in two (closely related) papers by L. Eric et al. [1][2]. In other words, a minimal BVTT is traversed if Algorithm 1 "Depth-first search" is called with $\epsilon = \delta$.

In practice, it seems that traversing a minimal BVTT necessitates knowledge of δ a priori, the unknown number which is being computed by Algorithm 1 "Depth-first search". A traversal scheme called *priority directed search* (PDS) is presented in Section 3.3 "Priority directed search". PDS traverses a minimal BVTT without knowing δ a priori (see proof in Section 2.3.7 "Minimal BVTT forest"). PDS was first introduced in Larsen et al. [1] and further developed shortly thereafter [2][3].

At its very core, PDS schedules what nodes to visit according to their distances. Instead of recursively visiting left children followed by right children, PDS inserts a BVTT root $\mathcal{T}$ into a priority-queue sorted on separation distances in ascending order. The top-node of the priority-queue is popped and a child v_c is inserted into the priority-queue if and only if $\|v_c\| \leq \epsilon$. A lax description of PDS is that it is similar to Dijikstra's algorithm on trees with BVTT node distances as edge weights. A (naïve) pseudocode illustrating PDS is presented in Algorithm 2 "Priority directed search".

Algorithm 2 Priority directed search

Require: $\mathcal{T}$ a node in a bounding volume traversal tree
Require: PQ a priority queue
1: **function** TRAVERSE($\mathcal{T}, \epsilon, q$)
2: $PQ \leftarrow PQ \cup \{\mathcal{T}\}$
3: **while** $PQ \neq \emptyset$ **do**
4: $v \leftarrow \text{POP}(PQ)$
5: **if** $\|v(q)\| \leq \epsilon$ **then**
6: **if** v is leaf **then**
7: $\epsilon \leftarrow \min(\epsilon, \|v(q)\|_{leaf})$ ▷ Update ϵ
8: **else**
9: $PQ \leftarrow PQ \cup \{\text{left}(v), \text{right}(v)\}$
return ϵ
10: **end function**

A better PDS algorithm, with respect to the number of $\mathcal{O}(\log n)$ insertions and deletions, is presented in Section 3.3 "Priority directed search".

PDS is not for free. There is a trade-off: Traversal may greedily dive towards leaves that do not hold a small distance ϵ or traversal may, instead, slowly explore a BVTT using $\mathcal{O}(\log n)$ priority queue operations with the benefit approaching a leaf with a small distance ϵ.

2.3.6 Generating BVTTs on-the-fly

Any parent or child of some BVTT node $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ is well-defined given information available on $\mathcal{T}$, as shown in Section 2.3.4 "BVTTs generation". This means that if some traversal algorithm is currently standing on a node $\mathcal{T}_{\mathcal{A},\mathcal{B}}$, both children and its parent can be generated with left($\mathcal{T}_{\mathcal{A},\mathcal{B}}$), right($\mathcal{T}_{\mathcal{A},\mathcal{B}}$) and parent($\mathcal{T}_{\mathcal{A},\mathcal{B}}$), respectively.

Deallocation of right($\mathcal{T}_{\mathcal{A},\mathcal{B}}$) is possible after traversing to a child or parent. Consequently, a complete BVTT need not be stored in memory at all — only a current node need to be stored. I denote this as an *implicit BVTT*.

On the other end, *explicit BVTTs* are traditional trees in which every node has pointers to their children and optionally a parent. An explicit BVTT consumes twice as much memory as its implicit counterpart (since BVTTs are binary) and is capable of storing persistent information.

2.3.7 Minimal BVTT forest

Oftentimes there are several BVTTs, forming a *forest* $\mathcal{F}$ of BVTTs (analogous to the edge-set E with $G = (A, B, E)$), each of which may give rise to a smallest separation distance δ.

Recall from Section 2.2 "Problem definition" that $\|\mathcal{A}-\mathcal{B}\|$ is a minimal separation distance between *two geometric objects* and that $\|A - B\|$ is a minimal separation distance between *two sets of geometric objects*. With this in mind, consider a forest $\mathcal{F}$ of BVTTs. Each member of $\mathcal{F}$ has a smallest separation distance. For any BVTT $\mathcal{T}_{\mathcal{A},\mathcal{B}} \in \mathcal{F}$, a minimal BVTT is obtained by calling the traversal routine given

in Section 2.3.5 "BVTTs traversal" with $\epsilon = \|\mathcal{A} - \mathcal{B}\|$. However, a smaller BVTT is obtained if the traverse routine is called with $\epsilon = \|A - B\| \leq \|\mathcal{A} - \mathcal{B}\|$. Consider the forest of all such smaller BVTTs. I denote this forest as a *minimal BVTT forest*.

Definition 4. ***Minimal BVTT forest:*** *The set of BVTT nodes within a forest $\mathcal{F}$ of BVTTs which are traversed with $\epsilon = \|A - B\|$.*

A minimal forest of BVTTs is illustrated in Figure 2.6.

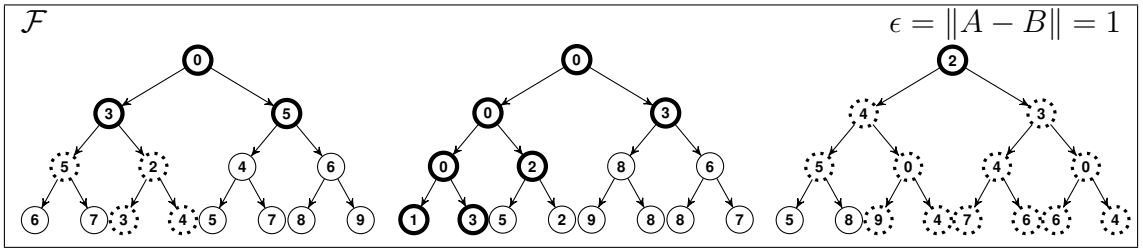


Figure 2.6: A forest $\mathcal{F}$ of three BVTTs is shown. Its minimal counterpart is indicated by thick nodes. Node distances are indicated within nodes. The smallest leaf within the forest has distance $\|A - B\| = 1$. Nodes which are members of minimal BVTTs but not members of the minimal forest of BVTTs are dotted.

Note that a minimal BVTT forest is equally small or smaller than the union of minimal BVTTs, since nodes of some BVTT $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ can be more aggressively pruned with $\epsilon = \|A - B\|$ as opposed to $\epsilon = \|\mathcal{A} - \mathcal{B}\|$ since $\epsilon = \|A - B\| \leq \|\mathcal{A} - \mathcal{B}\|$ by definition of $\|A - B\|$. These nodes are dotted within Figure 2.6. Hence, it is preferred to traverse minimal BVTT forests as opposed to unions of minimal BVTTs.

2.3.8 PDS traverses minimal BVTT forests

Many authors suggests that PDS traverses fewer BVTTs nodes than a depth-first search [1][2][3]. I make a stronger claim: PDS traverses a minimal BVTT forest. To this end, I develop a proof of Theorem 1 that support said claim.

Some notations are presented prior to proving lemmas leading up to Theorem 1:

- Let V be the set of visited nodes in a forest of BVTTs F using PDS with $\epsilon = \infty$
- Let V^* be the set of visited nodes of F using PDS with $\epsilon = \delta = \|A - B\|$ (so V^* is a minimal BVTT forest).
- Let $v^* \in V^*$ be a leaf node with $\|v^*\|_{leaf} = \delta = \|A - B\|$
- Denote $P(v)$ as the parent set of v (see Section 2.1 "Preliminary Theory").
- Let PQ be the priority queue used by PDS, initially populated by all BVTT roots of a forest $\mathcal{F}$.

Lemma 1. *If $P(v^*) \cap PQ \neq \emptyset$, then a node which is popped is smaller than or equal to δ*

Proof. Clearly, $|PQ \cap P(v)| \leq 1$ at all times — no two or more parents exists simultaneously in PQ. Therefore, $P(v^*) \cap PQ \neq \emptyset$ implies that $PQ \cap P(v^*) = \{v_p^*\}$. Note that $\|v_p^*\| \leq \delta$ since leaves are enclosed by parent bounding volumes.

Now, pop a node v from PQ, then $\|v\| \leq \min(PQ) \leq \min(PQ \cap P(v^*)) = \min\{v_p^*\} = \|v_p^*\| \leq \delta$. Hence $\|v\| \leq \delta$. □

Note that $P(v^*) \cap PQ \neq \emptyset$ implies that the best leaf v^* have not been visited and implies that $PQ \neq$ even if a node is popped (so $\min(PQ)$ is well-defined).

The proof that PDS traverses minimal BVTT forests leverages the fact that if a node v giving a smallest distance δ is yet to be encountered, then a node in its parent set must exist in PQ and this node must have a distance smaller than δ. So any node that is popped prior to visiting the node with distance δ must have a distance smaller than δ (since PQ is a priority queue). Without further ado, the proof that PDS traverses minimal BVTT forests is now presented:

Theorem 1. $V = V^*$

Proof. Let $\epsilon = \infty$. Assume $V \neq V^*$. Assume WLOG that $V = V^* \cup \{v\}$ with $\{v\} \cap V^* = \emptyset$, i.e. v is the sole node of in V which is not in V^*. I deal with the following two cases separately: $P(v^*) \cap PQ \neq \emptyset$ and $P(v^*) \cap PQ = \emptyset$

Case 1: $P(v^*) \cap PQ \neq \emptyset$: Consider the case where v is about to be popped from PQ. When any parent of v was popped, then $P(v^*) \cap PQ \neq \emptyset$ must have been true too. Hence, all parents of v are smaller than or equal to δ by lemma 1. In particular, the closest parent v_p of v is smaller than or equal to δ. But then traversal can not terminate at v_p even if $\epsilon = \delta$, hence $v \in V^*$ by definition of V^*.

Case 2: $P(v^*) \cap PQ = \emptyset$: If $P(v^*)$ has no nodes in common with PQ, it must be because (a) v^* has already been popped so $\epsilon = \delta$ or (b) v^* is the earliest leaf in PQ. Case (b) is easily transformed to case (a) by continuing the priority directed search until v^* is popped. Hence, a treatment of (a) is sufficient. Now, if $\epsilon = \delta$ prior to popping the closest parent v_p of v, then $\|v_p\| \leq \delta$ since $v \in PQ$. But if $\|v_p\| \leq \delta$ then $v \in V^*$ ($v_p \in V^*$ by WLOG-assumption). A contradiction. Hence, it must be the case that $\epsilon > \delta$ prior to popping the closest parent v_p of v. But if $\epsilon > \delta$ prior to popping the closest parent v_p of v, then v^* is yet to be popped. If v^* is yet to be popped, then we have either (c) or (d):

- **(c):** $P(v^*) \cap PQ \neq \emptyset \wedge v^* \notin PQ$ at the time of popping the parent v_p of v, thus $\|v_p\| \leq \delta$ by Lemma 1. Analogously with Case 1, if $\|v_p\| \leq \delta$ then $v \in V^*$.
- **(d):** $P(v^*) \cap PQ = \emptyset \wedge v^* \in PQ$ at the time of popping the parent v_p of v so $\|v_p\| \leq \|v^*\| = \delta$ and hence $v \in V^*$.

This concludes Case 2.

In conclusion, $v \in V^*$ for both Case 1 and Case 2, contradicting the assumption that $V \neq V^*$. □

Theorem 1 states that it does not matter if an ϵ is initialized to δ or ∞ — the two sets of visited BVTT nodes are equivalent, regardless. In particular, this theorem states that PDS traverses minimal BVTT forests. Nevertheless, initialization of ϵ

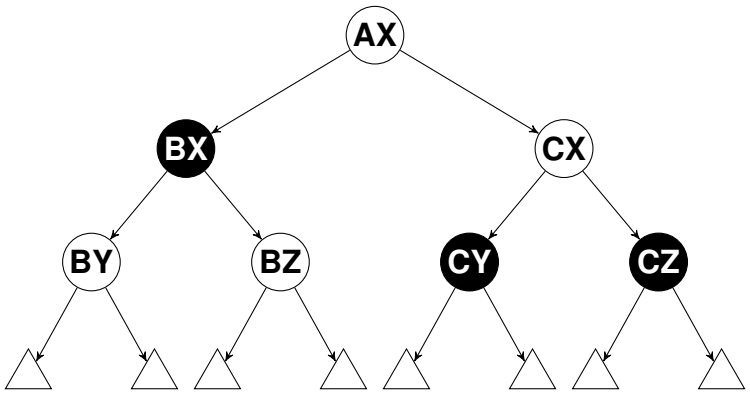


Figure 2.7: Filled nodes are members of a BVTT front. A triangle denotes some arbitrary BVTT-subtree.

is important for other reasons than that of visiting few BVTT nodes, which are presented throughout Section 3.3 "Priority directed search".

A simple corollary follows:

Corollary 1. *Let V be the set of visited nodes of $\mathcal{T}$ using PDS with $\epsilon = \infty$ and let V^* be the set of visited nodes of $\mathcal{T}$ PDS with $\epsilon = \|A - B\|$, then $V = V^*$*

Proof. Apply theorem 1 with $F = \{\mathcal{T}\}$. □

Corollary 1 is not surprising. If PDS traverses minimal BVTT forests, then it is not a long stretch to guess that PDS also traverses minimal BVTTs.

2.4 Front tracking

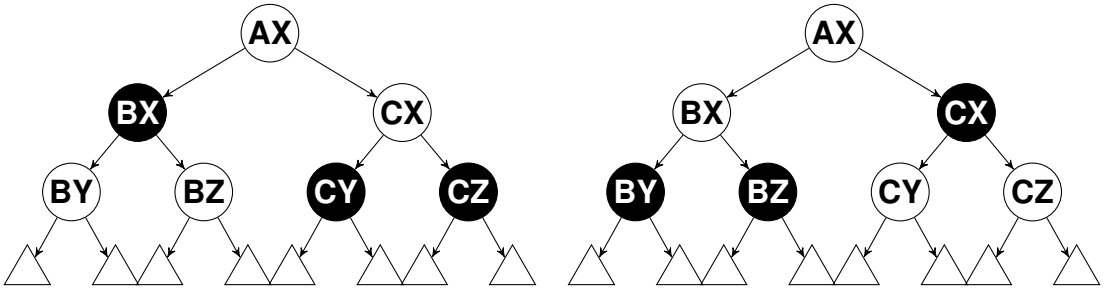
FT and variants thereof are techniques which reduce the number traversed BVs by remembering where queries terminated in previous iteration(s) [18][14][3][19][21][9][17]. The method of GFT saves a BVTT node v to a *bounding volume traversal tree front* (BVTT front), if traversal terminates on v due to $\|v\| \geq \epsilon$. In other words, a *bounding volume traversal tree front* (BVTT front) consists of cached BVTT nodes. In Figure 2.7, a BVTT front is shown as filled nodes. GFT initiates traversals from each node of a front instead of starting traversal from a BVTT root. For binary a BVTT, this could reduce the number of traversed nodes by at most half the front size and hence halves the number of BV-BV tests.

A BVTT front can be equipped with two operators:

Sprout operator: A sprout operator moves a front "downwards". Sprouting can be thought of as traversing a BVTT, saving nodes for which traversal terminated. This increases the size of a front.

Raise operator: A raise operator is the opposite of a sprout operator — it moves a front "upwards". A raise operator decreases the size of a front.

In coherent scenes where motions of geometric objects are small, it is expected that a BVTT front need only undergo minor changes — the set of pruned nodes



(a) Before updating the front (b) After updating the front

Figure 2.8: A BVTT front which undergo minor changes, (a) shows a front before updating it and (b) shows a front after updating it

is almost identical in a following iteration, as illustrated in Figure 2.8. Hence, it seems unnecessary to start traversal from a BVTT root when a BVTT front is a good guess of where traversal will terminate.

If traversal always starts at a BVTT front and if the BVTT front is always extended by nodes at which traversal terminates, then the BVTT front increases in size until it consists of all BVTT leaves. GFT has then degenerated into the very $\mathcal{O}(n^2)$ case which BVHs are designed to avoid. The problem is that the front no longer approximates the set of nodes at which traversal terminates. This explains why a front must be raised by some raise operator — otherwise the front no longer provides an accurate guess of where traversal will terminate.

A formalization of the notion of an accurate guess follows:

Definition 5. ***Optimal front (single BVTT):*** *The set of leaves in a minimal BVTT.*

This definition is inspired from O. Tropp et al. [19], who introduce *optimal front*s although their definition differ. Note that set of leaves in a minimal BVTT, i.e. an optimal front, is precisely the set of BVTT nodes at which traversal would terminate. Indeed, an optimal front is a perfect guess of where traversal will terminate (by definition).

An optimal front on a forest of BVTTs is defined analogously:

Definition 6. ***Optimal front (forest of BVTTs):*** *The set of leaves in a minimal forest of BVTTs.*

Figure 2.9 depicts an identical situation as in Figure 2.6 but the optimal front is indicated as filled nodes.

A front F is said to lie below an optimal front if nodes of the optimal front is a subset of the parents of F.

In conclusion, a good front tracker provides a good estimate of an optimal front, preferably with negligible overhead.

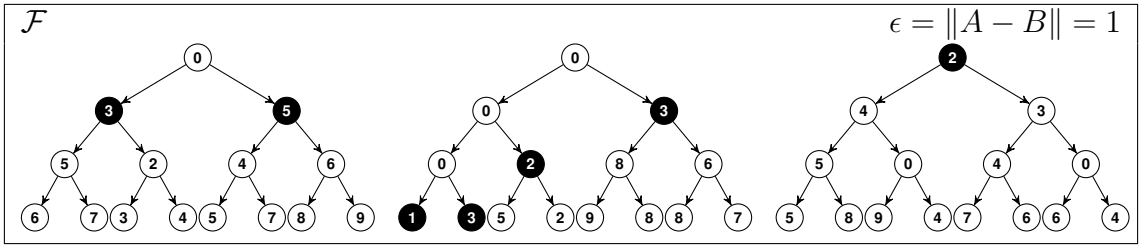


Figure 2.9: A forest $\mathcal{F}$ of three BVTTs is shown. An optimal front over $\mathcal{F}$ is indicated by filled nodes.

2.4.1 Memory usage of front tracking

Denote a *maximal front* of a BVTT as the set of all leaves in a BVTT. Consider two identical binary BVHs with n leaves respectively. There are n^2 pairings of leaves so there are n^2 leaves in the corresponding BVTT. Hence the memory usage of a (maximal) BVTT front is $\mathcal{O}(n^2)$. Assuming that the front stores BVTT nodes which hold 64bit pointers to two underlying bounding volumes or triangles, then a maximal front for two 10k meshes occupies $10000^2 \cdot 128\,\text{bit} \approx 1.6\,\text{GB}$.

In a robot-cell context there can be several BVHs with possibly (much) more triangles than 10k. If $|A| = |B| = 10$ then $|A| \cdot |B| = 100$ maximal fronts may be kept in memory in a worst-case scenario, occupying $100 \cdot 1.6\,\text{GB} = 160\,\text{GB}$ of memory. This may cause worry that FT is intractable. Experiments show that fronts are, in practice, consistently several orders of magnitudes smaller than the their maximal counterparts. See Section 5.8 "Front size and memory usage".

2.4.2 Invalid and redundant fronts

I introduce two front-properties: Invalid fronts and redundant fronts. These two definitions are convenient to be equipped with when discussing sprouting and raising operators.

Invalid front

A front F is *invalid* if there exists a leaf which is unreachable from a front. This could happen due to usage of incorrect sprouting or raising operators. A front should be neither invalid nor redundant if proximity queries should be correct and efficient. This bounds the design space of sprouting and raising operators.

Definition 7. ***Invalid front:*** *A front F over a forest $\mathcal{F}$ is invalid iff*

$$\exists l \in \mathcal{F} \wedge l \notin \text{subtree}(\mathcal{T}), \forall \mathcal{T} \in F$$

If traversal starts from an invalid front, then it is possible that a leaf node l_{unvis} is not visited after traversal ends. This is incorrect since if $\|l_{unvis}\| = \delta$, then an algorithm starting traversal from a front will not be able to return the correct separation distance. A invalid front is illustrated in Figure 2.10.

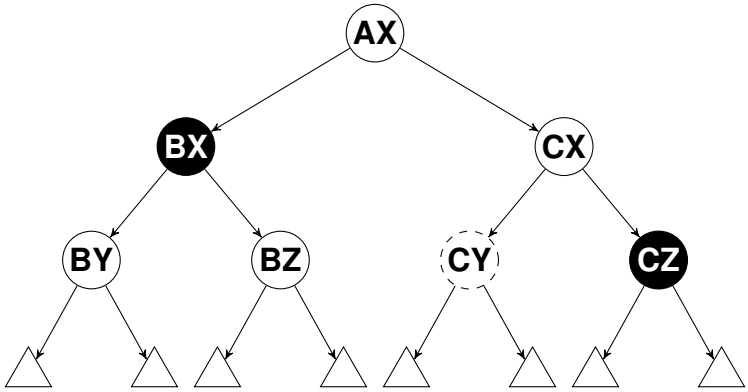


Figure 2.10: An invalid BVTT front. A dashed node denotes the missing node, i.e. the node which must be a member of a valid front.

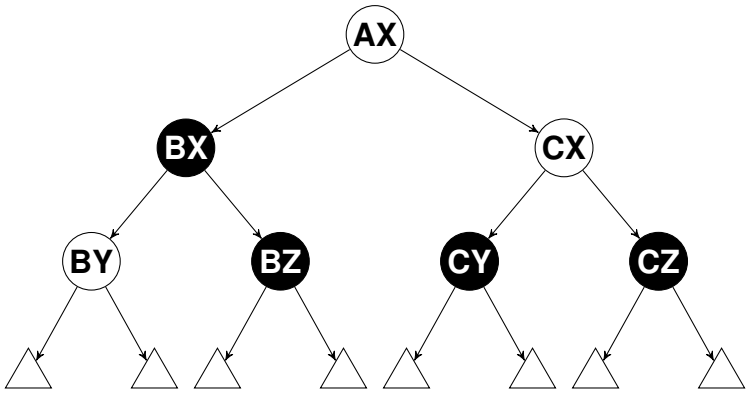


Figure 2.11: A redundant BVTT front. A child BZ and its parent BX are members of the illustrated front and hence the front is redundant.

Redundant front

A front F is *redundant* if there exists a node v in F and a node from its parent set $P(v)$ also exists in F.

Definition 8. ***Redundant front:*** *A front F over a forest $\mathcal{F}$ is redundant iff*

$$\exists v \in F : P(v) \cap F \neq \emptyset$$

A redundant front is unwanted because it makes it possible to initiate two different traversals which have a set of reachable leaves in common, introducing a risk of visiting some BVTT nodes v twice. A redundant front is shown in Figure 2.11, which can be made non-redundant by either removing BX and adding BY or removing BZ.

3

Forest proximity querying

I develop a novel method denoted *Forest Proximity Query* (FPQ) throughout sections under this chapter. FPQ aims to improve upon present state-of-the-art methods for distance computations by reducing the number of BV-tests, without resorting to approximate or special-purpose methods.

FPQ is a simple method that enables usage of established proximity querying methods on *forest* (collections of trees) of BVTTs. Conceptually, FPQ connects all BVTT roots in a forest of BVTTs to a super-root. Established BVTT techniques, for example *generalized front tracking* (GFT) or *priority directed search* (PDS), can then be applied to several BVTTs simultaneously by initiating traversal at the super-root.

I propose (and later on explain) two core benefits of FPQ: it explores a minimal forest of BVTTs when used in conjunction with PDS and it can reduce BVTT front size. These core benefits have potential to accelerate successive computations of $\text{dist}(i; A, B)$ (by reducing the number of distance computations between bounding volumes), and therefore, answers the research question of this thesis.

An implementation functions as a proof of concept (by demonstration) and is vital in comparing FPQ with PQP. Said implementation of FPQ benefits FCC, whom are eager to speed up their proximity queries — a major bottleneck in their motion-planner — within the *Industrial Path Solutions* (IPS) software. Indeed, proximity querying is typically a bottleneck in *rapidly exploring dense tree (RRT)* based motion-planners. For example, collision detection may amount up to 99% of total running time of RRT based motion-planners [36]. In summary, a reduced number of distance computations in between BVs targets a significant bottleneck of RRT based motion-samplers and is therefore believed to accelerate proximity queries in a robot cell context.

I answer the research question by developing and demonstrating FPQ, which is integrated within IPS. PQP is used by IPS, hence, a fair comparison between FPQ and PQP is possible. I will refer to my implementation as simply FPQ.

I choose to use distance routines within PQP for computing distances between BVs and between geometric primitives. A short presentation of how PQP is used within IPS is given in Chapter 4 "PQP". Using PQP within FPQ allows for a fair comparison with IPS. Otherwise, it would be difficult to tell if performance of FPQ is due to underlying BVHs and BVs or due to FPQ. Further, PQP uses RSSs which are regarded as a solid choice of BV-type for proximity querying. Some other possible choices of BVHs are hierarchies of AABBs, OOBs or k-DOPs, but I have not tried various types of BVHs since FPQ does not depend on some particular BVHs.

3.1 Ordering bounding volume traversal trees

Consider a forest of BVTTs (which have not been connected to a super-root), that each of which may give rise to a separation distance between two sets A, B of geometric objects. An ordering of BVTTs computations must be made. In other words, each edge $e \in E$ with $G = (A, B, E)$ must be assigned a rank determining the order of evaluation.

The ordering matters because BVTT traversal can be greatly accelerated given an upper-bound ϵ on the separation distance, due to Line 2 that enables pruning of BVTT subtrees if ϵ is sufficiently small. Naturally, if it is known that the separation distance must be smaller than some ϵ, then there is no need to examine BVTT nodes with distances larger than ϵ. Hence, the order of evaluation of BVTTs matters.

A most simple ranking is a fixed ranking where all BVTTs are of fixed order. This means that any pair of BVHs are always tested against each other in a fixed order. This ordering is not ideal since a BVTT that frequently gives rise to a minimum distance may be tested after testing other BVTTs.

IPS uses an ordering scheme which schedules a BVTT $\mathcal{T}$ to be examined first if $\mathcal{T}$ gave rise to the shortest distance in the previous iteration. Mathematically, $\mathcal{T}$ is examined first on iteration i if $\|\mathcal{T}(q_{i-1})\| = \delta_{i-1}$, where δ_{i-1} denotes the separation distance of the previous iteration. More on this in Chapter 4 "PQP".

A clever ordering scheme ensures that a BVTT $\mathcal{T}_{\mathcal{A},\mathcal{B}}$ with $\mathcal{T}_{\mathcal{A},\mathcal{B}} = \delta_i$ is tested first on iteration i. However, it is not clear-cut how to determine which BVTT in a forest of BVTTs gives a smallest separation distance without computing the separation distances of all BVTTs.

Since FPQ connects all BVTTs to a super-root, an ordering of BVTTs correspond to an ordering of edges between said super-root and BVTT roots. But it is the job of traversal algorithms to decide what edges to descend. Hence, FPQ implicitly uses a ordering scheme as a function of some traversal algorithm.

Depth-first search and breadth-first search are two well-known traversal algorithms. It is far from apparent that they visit BVTTs in a good order. Some authors suggests using PDS[2] instead of a depth-first search throughout a BVTT. PDS is particularly interesting when used with FPQ, since it traverses all BVTTs simultaneously. I proved that PDS traverses minimal forests of BVTTs if a priority queue is initialized with all BVTT roots. Hence, FPQ traverses minimal forests of BVTTs if all edge-weights in between a super-root and BVTT roots are set to zero. In practice, it is clearly sufficient to initialize a priority queue used by PDS with BVTT roots.

3.2 Depth-first search

Until this point, a lot of theory on BVTTs and FT has been presented. Prior to introducing FPQ with depth-first search, a brief summary of important key ideas that I have already presented follows:

1. Geometric objects consists of geometric primitives (see Section 2.2).
2. Each geometric object is associated with a BVH (see Section 2.3.2).

3. A BVTT is formed given two BVHs and each BVTT node represents a BV test or a geometric primitives test (see Section 2.3.3).
4. The problem of distance computation can be solved by computing BVTT node distances.
5. BVTT subtrees can be pruned if an small upper-bound ϵ on the shortest distance δ is known (see Line 2 of Section 1).
6. PDS with FPQ traverses minimal forests of BVTTs.
7. A robotic arm is a composition of rigid bodies, each of which is encapsulated by a BVH. Thus, the problem of computing separation distances between various robotic arms or static geometric objects is that of traversing a a forest of BVTT (see Section 1.2 and 2.3).
8. Separation distance computations constitute a bottleneck of RRT based motion-planners. Therefore, traversing small BVTTs may relieve said bottleneck and hence proximity queries in a robot cell context may be accelerated by traversing small BVTTs.
9. A point $\mathcal{C}$-space determines the state of a geometric object (see Section 2.2).
10. The problem I address is how to efficiently find a minimal separation distance between two sets of geometric objects for given a sequence of world configuration points $q_o, q_i, \ldots q_n$ (see Section 2.2).

The list above summarizes core ideas presented so far. With these ten key ideas in mind, I make the purpose of this section explicit: To weave previously presented ideas into a coherent whole, presented as an algorithm with well-defined inputs and outputs. In particular, a method for computing $\|A(q_i) - B(q_i)\|$, given a sequence of points in $\mathcal{C}$-space $q_0, q_1, \ldots, q_n$, using FPQ with depth-first traversal is now described. Finally, pseudocode for FPQ with depth-first traversal is presented.

FPQ aims to reduce the number of BV-BV tests, i.e. the number of times $\|v\|$ has to be computed where v is a BVTT node. In order to accomplish this, a front consisting of BVTT nodes within a forest of BVTTs is used. The front makes it possible to avoid traversing BVTT nodes which lie in between the front and the BVTT roots. The number of BV tests are halved if a front is optimal. However, FT relies on coherence in scenes. Fortunately, a distance between two successive $\mathcal{C}$-space points q_i and q_{i+1} is bounded from above $\|q_i - q_{i-1}\| < d$, ensuring at least partial coherence.

A list of symbols used throughout this section follows:

Symbols

A, B:	Sets of geometric objects
δ:	A separation distance between A and B, $\|A - B\|$
ϵ:	Upper-bound on δ (shortest distance known so far)
Front F:	A set of BVTT nodes approximating an optimal front
Leaf front L:	All leaf members of F belong to a leaf front $L \subseteq F$
Sproutees S:	Subset of F that is sprouted
Sprouted front SF:	Subset of F that was sprouted in the previous iteration
Rest front RF:	All members of F which are not member of either L or SF.

The algorithm is fairly large and therefore it is presented in independent fragments, in a chronological (with respect to execution) order. Finally, the fragments

are pieced together into a coherent whole.

3.2.1 Initialization

Initialization happens on two occasions:

A per BVTT forest initialization: An initialization is required every time a new set of geometric object A or B is given. These two sets remain fixed — geometric objects are by assumption rigid hence not deformable. Further, no geometric objects are added to said sets during motion-planning run-time (A and B can be regarded as constants). Therefore, this initialization-step occurs only once.

A per $\mathcal{C}$-space point initialization: This type of initialization occurs every time a $\mathcal{C}$-space point from the sequence of $\mathcal{C}$-space points $q_0, q_1, \ldots q_n$ is processed. This initialization occurs n times.

One-time initialization

Two sets of geometric objects A and B are given prior to computing $\text{dist}(i; A, B)$. All edges in $G = (A, B, E)$ correspond to the initial forest $\mathcal{F}$ of BVTTs. All roots within the initial forests are added to the front F.

Note that a super-root is a theoretical object which is not part of initialization (or other parts of FPQ). Instead of adding a super-root to a front, all BVTTs of $\mathcal{F}$ are added to the *same* front F. This is completely analogous to adding a hypothetical super-root to a front.

Per call initialization

PQP is used for computing BV distances and distances in between two geometric primitives. *World transformations* are computed in a $\mathcal{O}(|A| \cdot |B|)$ pass. PQP relies on world transformations when computing BV distances and distances between two geometric primitives.

An introduction to world transformations is given in Chapter 4 "PQP". For now, it is enough to know that a world transformation must be updated for $\|v(q)\|$ and $\|v(q)\|_{leaf}$ to be correctly computed.

3.2.2 Initializing an upper-bound on a separation distance via a leaf front

It is good to have a small upper-bound ϵ on a separation distance δ, sooner rather than later, since if $\|v\| > \epsilon$ for a BVTT node v, then all leaves below v have distances that are greater than ϵ. Therefore, traversal may terminate on v. For this reason, all BVTT leaves that are visited in one iteration are cached in a leaf front L, including the leaf which gave rise to the separation distance. This can be viewed as generalized triangle caching. Figure 3.1 illustrates a leaf front $L \subseteq F$, indicating members of L with a leaf-symbol.

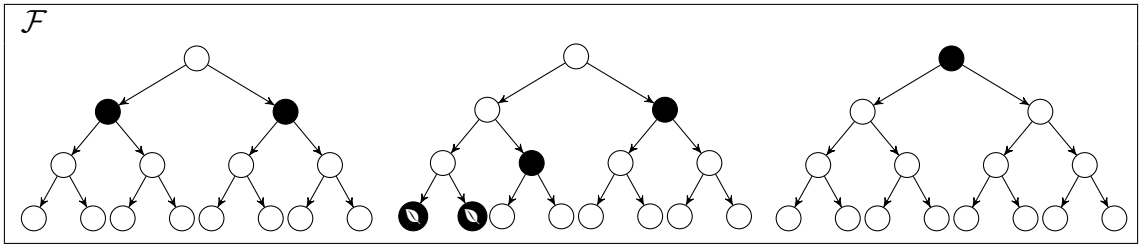


Figure 3.1: A forest $\mathcal{F}$ of three BVTTs is shown. A front is indicated by filled nodes. Members of the subset $L \subseteq F$ are indicated by a leaf-symbol.

A dynamic array is used for representing L because insertion at back is $\mathcal{O}(1)$ and there is no need for deletion.

At the start of the main routine, all leaf distances are computed in a $\mathcal{O}(|L|)$ pass, giving a smallest known upper-bound on the separation distance δ, denoted as ϵ. See Algorithm 3 "Initialization of ϵ".

Algorithm 3 Initialization of ϵ

```
1: function INIT_EPS(L, q)
2:     ε ← ∞
3:     for ∀v ∈ reverse(L) do
4:         if ‖v(q)‖ ≤ ε ∧ ‖v(q)‖_leaf ≤ ε then
5:             ε ← ‖v(q)‖_leaf
       return ε
6: end function
```

The leaves are traversed in a reversed order. This is because any leaf which is inserted into L is, at the time, smaller than ϵ. This means that small leaves are likely to be found in the right-most part (back) of L while larger leaves are likely to be found in the left-most part (front) of L.

The notion of $\|\mathcal{T}(q))\|_{leaf}$ is introduced to disambiguate between geometric primitive distance computations and BV distance computations. A geometric primitive distance computation is only carried out if the leaf node passes the (relatively cheaper) BV test.

It is not clear cut if it is beneficial to test a BV distance against ϵ, which may or may not allow us to discard a geometric primitive test, but benchmarks showed that it is slightly better to perform BV tests. A rationale for this is that geometric primitives tests are more expensive than BV tests, so if sufficiently many geometric primitive tests at leaf level are discarded due to BV tests yielding distances larger than ϵ, then a BV test at leaf level can prove beneficial (which it did). Said benchmark is not presented for it is a very minor result.

Finally, a cause for concern is initialization of ϵ via a leaf front is expensive — geometric primitives tests are expensive. This is not a problem in practice. Results indicate that initialization of ϵ correspond to roughly 1% of total running times (see Section 6.1 "Main results").

3.2.3 Leaf front, sprouted front, and rest of the front

A simple idea is to keep all of front nodes in one dynamic array. However, as already shown, a partitioning of the front can be beneficial — a leaf front is used to iterate over all leaves of the front instead checking for each front member if it is a leaf and if so, update ϵ.

Indeed, the front is partitioned further into a *sprouted front* SF, leaving a leftover part of the front denoted as rest front RF. The sprouted front is the set of leaves at which traversal terminated in the previous iteration. Therefore, SF can be regarded as the part of a front F which causes F to increase in size and, if coherency is high, the part of the front which is moving downwards. Consequently, RF can be regarded as the part of the front which is either fixed or moving upwards.

$$F = RF \cup SF \cup L \text{ with } RF, SF, L \text{ mutually exclusive} \tag{3.1}$$

where L is as before, with SF and RF:

$$SF = \text{Set of nodes for which traversal terminated} \tag{3.2}$$
$$RF = F \setminus (SF \cup L) \tag{3.3}$$

A front is partitioned into SF and consequently RF for this reason: If traversal terminates on node v at iteration $i-1$, i.e. v is a member of SF on iteration i, then v is likely to become further sprouted on iteration i, possibly reaching a leaf node l with $\|l\| < \epsilon$. Hence, by recalling what nodes were sprouted on iteration $i-1$, these nodes can be traversed prior to traversing other nodes. This motivates a partitioning of the front into SF.

It should be noted that this technique assumes that sprouted nodes have a higher probability of becoming sprouted in successive iterations. This is not a wild assumption in a robot-cell context. Consider a tool equipped at the tip of a robotic arm. All distances on BVTT nodes, representing a tool and a surface of some product, decreases as the tool approaches the product surface. These sprouted BVTT nodes may become further sprouted such that a leaf $l_{sprouted}$ with $\|l_{sprouted}\| < \min_{l \in L} \|l\|$ is visited, causing ϵ to be updated.

A stack S of sproutees, nodes for which a depth-first traversal must be initiated, is populated with BVTT nodes from RF and SF.

$$S = \{\|v\| < \epsilon : \forall v \in RF \cup SF\} \tag{3.4}$$

Nodes from RF are inserted into S if it is possible that their children give rise to a smaller ϵ, i.e. if $\|v\| < \epsilon, v \in RF$. Nodes from SF are inserted *after* nodes from RF has been inserted. The stack S will hence traverse nodes from SF first.

Pseudocode describing how a front is partitioned is found in Algorithm 4 "Front partitioning". Note that SF is cleared at the end. This is because SF is rebuilt during sprouting (sprouting is presented in Section 3.2.5 "Sprouting"). However, no node v from SF is lost since it keeps living in either SF or RF, depending on if it may have a child c with $\|c\| < \epsilon$ or not.

Algorithm 4 Front partitioning

```
1: function PARTITIONING(RF, SF, q, S)
2:     for ∀v ∈ RF do
3:         if ‖v(q))‖ ≤ ϵ then
4:             S ← S ∪ {v}
5:             RF ← RF \ {v}
6:     for ∀v ∈ SF do
7:         if ‖v(q))‖ ≤ ϵ then
8:             S ← S ∪ {v}
9:         else
10:            RF ← RF ∪ {v}
11:    SF ← ∅
12: end function
```

After Algorithm 4 "Front partitioning" have been executed, all front nodes with $\|v\| < \epsilon$ are members of S and it is precisely those nodes which must be sprouted in order to obtain a better approximation of the optimal front.

Note that nodes are removed from RF if they have a distance smaller than or equal to ϵ. Hence, after Algorithm 4 "Front partitioning" has been executed, RF consists of all nodes which are must be raised in order to better approximate the optimal front.

3.2.4 Raising

A simple raise operator is a sibling-raise operator. This type of raise operator is described by several authors [14][19][9]. A sibling-raise operator on F raises F by identifying sibling BVTT nodes of F and replace them by their common parent. Hence, raising siblings within a front is the problem of finding all sibling pairs: of nodes (v_1, v_2) with:

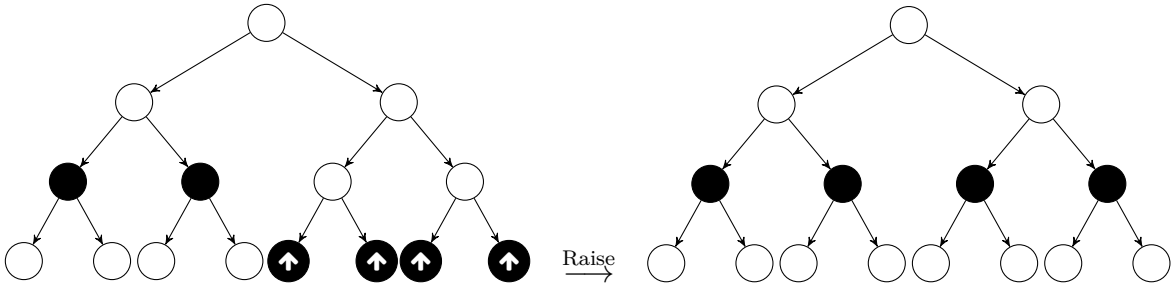
$$\{(v_1, v_2) \in RF \times RF : \text{parent}(v_1) = \text{parent}(v_2)\} \tag{3.5}$$

A front shown before raising and after raising, using a sibling-raise operator, is shown in Figure 3.2.

A sibling raise operator can never make a front invalid or redundant. Consider a front F, which is not invalid or redundant, and a raised front F_{i+1}. All reachable nodes of F_i are trivially reachable from F_{i+1}. Therefore F_{i+1} is valid. Further, both children of any (new) parent in F_{i+1} are removed from F_{i+1}, hence no child has a parent in F_{i+1}. Hence F_{i+1} is not redundant.

I apply a sibling-raise operator on RF post partitioning. All nodes of RF post partitioning constitutes the subset of F with distances greater than ϵ, i.e. those nodes which lie below an optimal front and thus those nodes which must be raised in order to obtain a better approximation of the optimal front.

I present an algorithm for raising a front F, with average time complexity $\mathcal{O}(n)$ and worst-case time complexity $\mathcal{O}(n^2)$, where $n = |RF|$, which utilizes a hash map H



(a) Before raisal (b) After raisal

Figure 3.2: A BVTT front before and after applying a sibling-raise operator. Nodes which are members of the BVTT front are filled with black. Nodes which are members of RF are marked with up-arrows.

to detect siblings in RF in Algorithm 5 "Raising". To the best of my knowledge, no authors have previously provided algorithmic descriptions of a sibling raise operator.

The algorithm traverses each element v_i of RF and inserts $(\text{parent}(v), i)$ into H. However, if $\text{parent}(v_j)$ of $v_j \in RF, i < j$ is already present in H then v_i is overwritten in-place by the common parent of v_i and v_j, $\text{parent}(v_i) = \text{parent}(v_j)$. The latter sibling v_j is finally removed from RF in $\mathcal{O}(1)$ time using the swap-and-pop idiom[1].

Algorithm 5 Raising

```
1: function RAISE(RF)
2:     i ← 0
3:     while i < |RF| do
4:         v_i ← RF(i)
5:         if IS_ROOT(v_i) then
6:             continue
7:         else if parent(v_i) ∉ H then
8:             H ← H ∪ {(parent(v_i), i)}
9:             i ← i + 1
10:        else
11:            R(H(parent(v_i))) ← parent(v_i)
12:            SWAP(v_i, BACK(RF))
13:            POP_BACK(RF)
14:    CLEAR(H)        ▷ Prevents false sibling-detections in a following iteration
15: end function
```

An illustration of Algorithm 5 "Raising" is shown in Figure 3.3.

Note that Algorithm 5 "Raising" is compatible with both implicit and explicit BVTTs. It does not use any persistent information of BVTT nodes since no persistent information on BVTT nodes is available for implicit BVTTs. A more efficient algorithm which is compatible with solely explicit BVTTs is devised in Section 3.4 "Efficient sibling-raise with explicit BVTTs".

[1]The swap-and-pop idiom: An element v of RF is swapped with the back element of RF. Popping the back element, which is now v, effectively erases it from the array without moving or copying ranges as per textbook deletion from dynamic arrays

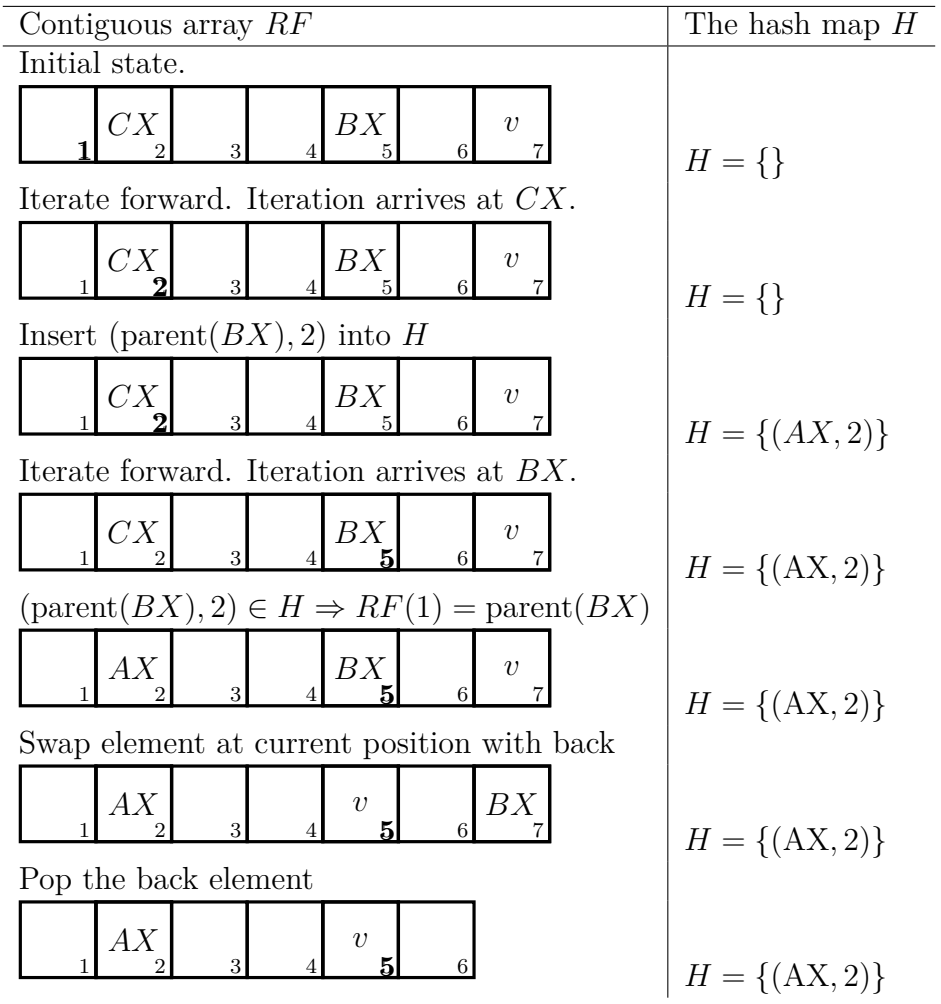


Figure 3.3: Illustration of how two siblings BX and CX are merged into their common parent AX within a contiguous array RF. The back node is some arbitrary BVTT node denoted v.

Two improvements

Let $d(v) \in \mathbb{N}$ denote the depth of $v \in \mathcal{T}$ where $\mathcal{T}$ is a BVTT. Then two simple invariants about sibling nodes hold:

- $w = \text{sibling}(v) \implies d(v) = d(w)$.
- $v \in \mathcal{T} \implies \text{sibling}(v) \in \mathcal{T}$

In other words, two nodes can only be siblings if they are members of the same BVTT $\mathcal{T}$ and have the same depth in $\mathcal{T}$. This enables using a hash map H per BVTT and per depth-level.

It is beneficial to have a hash map H per BVTT and per depth-level since insertions and lookups in hash maps are *on average* $\mathcal{O}(1)$ but worst-case $\mathcal{O}(n)$. By using several small hash maps instead of a single large hash map, the worst-case scenario is mitigated.

In terms of Algorithm 5 "Raising", the hash map H is retrieved from a two-dimensional dynamic array A after retrieving v_i from RF. Pseudocode of a sibling-

raise method utilizing several hash maps is presented in Algorithm 6 "Raising with several hash maps".

Algorithm 6 Raising with several hash maps

```
1: function RAISE(RF)
2:     i ← 0
3:     H ← A(d(v_i), GET_BVTT_ID(v_i))
4:     while i < |RF| do
5:         v_i ← RF(i)
6:         if IS_ROOT(v_i) then
7:             continue
8:         else if parent(v_i) ∉ H then
9:             H ← H ∪ {(parent(v_i), i)}
10:            i ← i + 1
11:        else
12:            R(H(parent(v_i))) ← parent(v_i)
13:            SWAP(v_i, BACK(RF))
14:            POP_BACK(RF)
15:    for H ∈ A do
16:        CLEAR(H)      ▷ Prevents false sibling-detections in a following iteration
17: end function
```

A subtle consequence of using several hash maps is that a BVTT node must allocate 8 bits for storing $d(v)$ and 16 bits for storing its BVTT ID, assuming that $d(v) < 2^8$ and that $|A| \cdot |B| < 2^{16}$. This increases memory consumption of FPQ.

I take all hash maps to be *flat hash maps*[2].

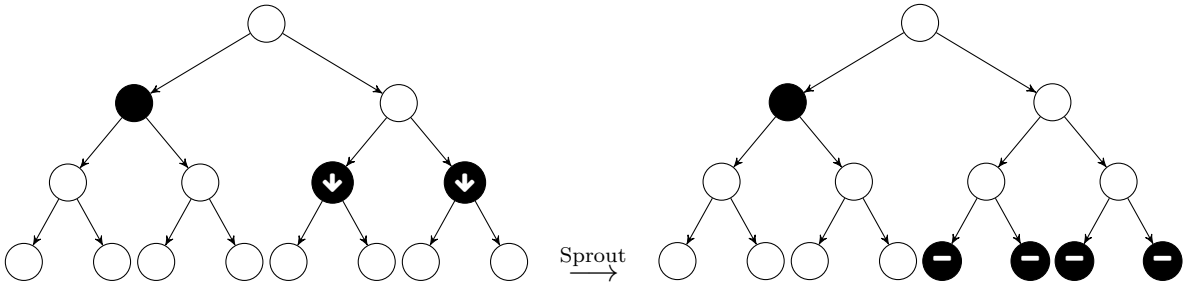
3.2.5 Sprouting

Conceptually, sprouting is the opposite of raising — sprouting moves a BVTT front downwards. Figure 3.4 illustrates how sprouting affects a BVTT front. In practice, sprouting is BVTT traversal. A presentation of depth-first sprouting follows.

A (naïve) depth-first traversal algorithm was previously presented in Algorithm 1 "Depth-first search". A depth-first sprouting operator behaves conceptually similar to said algorithm and works as follows: Take the top sproutee v of sproutees S, pop it from S, and feed it into a (modified) depth-first traversal algorithm. Whenever traversal terminates on a node, add it to SF. Repeat this process until S is empty.

The modified depth-first traversal algorithm works as follows: distances on two children v_l, v_r of a node $v \in S$ are computed. The smallest child $v_<$ is examined first. If $\|v_<\| < \epsilon$, then check if $v_<$ is a leaf and if so, proceed to compute its leaf distance $\|v_<\|_{leaf}$. Update ϵ if necessary. Proceed to examine $v_>$ in a similar fashion. If it turns out that both children are leaves with $v_>$ being the smaller leaf (although $v_>$ has a larger BV-BV distance), insert $v_<$ into L prior to inserting $v_>$ into L. Otherwise, insert $v_>$ into L prior to inserting $v_<$ into L.

[2] https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/



(a) Before sprouting (b) After sprouting

Figure 3.4: A BVTT front before and after sprouting. Nodes which are members of the BVTT front are filled with black. Nodes which are members of S are marked with up-arrows. Nodes which became members of SF after sprouting are marked with ⛔.

Indeed, there are many possible execution branches, each of which could lead to further sprouting of either v_l or v_r, updating ϵ, inserting either $v_>$ or $v_<$ into L, or rebuilding SF.

In fact, there are 19 possible interesting outcomes given in Table 3.1. Needless to say, most branches are omitted for brevity in Algorithm 7 "Sprouting". Regardless, this pseudocode should give the reader a thorough understanding of how the depth-first sprouting algorithm works.

Pseudocode is given in Algorithm 7 "Sprouting". It is possible to construct the complete pseudocode by enumerating all cases in Table 3.1 mechanically in addition to keeping the following guidelines in mind:

- Examine the smallest nodes first
- Never compute leaf distances which can not be smaller than ϵ
- Update ϵ whenever a leaf-test has been made
- If a smallest leaf-child is known, insert it into L last
- If a smallest intermediate node is known, insert it into S last
- Never compare any two distances more than once

Note that in practice, $\|v\|$ is only computed once per node and iteration — it is cached on v the first time $\|v\|$ is computed.

3.2.6 Complete FPQ with depth-first search

Initialization, partitioning, raising and sprouting are now woven together into a coherent algorithm represented as pseudocode in Algorithm 8 "Complete FPQ with DF search". This algorithm is a complete example of using FPQ for computing a separation distance between two sets of geometric objects, for successive points $q_1, q_2, \ldots, q_n$ in $\mathcal{C}$-space.

Algorithm 8 "Complete FPQ with DF search" initializes ϵ, partitions the front into nodes which must be either raised or sprouted in order to obtain a better approximation of the optimal front. Since leaves must be eligible for raisal too, they are temporarily copied to RF prior to raising RF. All leaves which did not get raised are copied back into L.

Table 3.1: Enumeration over possible sprouting cases for which comparisons are either **T**rue, **F**alse or **I**rrelevant. Irrelevant cases occur if (1) it is not relevant to perform a comparison due to termination of traversal at the node in question or (2) computing a leaf distance can not possibly yield a smaller ϵ. Columns with leaf distances are irrelevant if a corresponding node is not a leaf. On columns $\|v_>\| < \epsilon$ and $\|v_>\|_{leaf} < \epsilon$ it is implicit that a comparison againt ϵ is a comparison against $\|v_<\|_{leaf}$ if $\|v_<\|_{leaf} < \epsilon$ is true

Case	$\|v_<\| < \epsilon$	$v_<$ leaf	$\|v_<\|_{leaf} < \epsilon$	$\|v_>\| < \epsilon$	$v_>$ leaf	$\|v_>\|_{leaf} < \epsilon$
1	T	T	T	T	T	T
2	T	T	T	T	T	F
3	T	T	T	T	F	I
4	T	T	T	F	T	I
5	T	T	T	F	F	I
6	T	T	F	T	T	T
7	T	T	F	T	T	F
8	T	T	F	T	F	I
9	T	T	F	F	T	I
10	T	T	F	F	F	I
11	T	F	I	T	T	T
12	T	F	I	T	T	F
13	T	F	I	T	F	I
14	T	F	I	F	T	I
15	T	F	I	F	F	I
16	F	T	I	F	T	I
17	F	T	I	F	F	I
18	F	F	I	F	T	I
19	F	F	I	F	F	I

An improvement which eradicates copying on line 9 and line 11 in Algorithm 8 "Complete FPQ with DF search" is suggested in Section 6.7 "Avoiding copy-overhead during raisal".

3.3 Priority directed search

In this section, I present FPQ with PDS in light of FPQ with depth-first search. The two methods are, as we shall see, similar.

As suggested by [1], PDS guides BVTT traversal towards a small ϵ, promoting exploration of a BVTT closer to the minimal BVTT. GFT and PDS have been combined and found efficient when applied to a single BVTT[3].

FPQ with PDS is an adaption of GFT with PDS to a forest $\mathcal{F}$ of BVTTs, exploiting the fact that only a *single* smallest distance, rather than *all* smallest distances, is requested. I showed, in Section 2.3.8 "PDS traverses minimal BVTT forests", that PDS explores a minimal forest of BVTTs. Hence, PDS is a motivated approach for reducing BV-BV tests in any forest of BVTTs.

However, a single yet significant negative consequence is introduced with PDS: Insertion and deletion from S becomes $\mathcal{O}(\log n)$. Yet, there are several benefits of using PDS since a front may never sprout below that of the optimal front:

- Fewer BV-BV tests are performed
- Fewer geometric primitives tests are performed
- Front memory consumption decreases (ameliorating possibly intractable memory consumption, as presented in Section 2.4.1 "Memory usage of front tracking")
- Raise-operator does not have to work as hard
- Theorem 1 states that PDS traverses minimal forests of BVTTs *although* ϵ is initialized to ∞. Hence, PDS is expected to perform well in situations even when $\epsilon >> \delta$, i.e. in low-coherence scenarios.

FPQ with PDS complement each other since FPQ is expected to work well with high-coherence scenarios and PDS is expected to ameliorate low-coherence scenarios. Indeed, I propose that the relationship is symbiotic — PDS helps a front F to remain small, increasing likelihood of F being close to an optimal front, while a front offloads several $\mathcal{O}(\log n)$ operations from PDS by skipping top-level BVTT nodes.

Nonetheless, it is possible that a front lie below that of an optimal front due to conservative raising. Therefore, using a front with PDS destroys any guarantees that minimal forests of BVTTs are visited. I make no attempts at estimating if the positive consequences outweigh the negative consequence. Instead, both versions (depth-first and PDS) are implemented.

3.3.1 Complete FPQ with PDS in terms of FPQ with DFS

All hard work has already been dealt with in either Chapter 2 "Theory" or Section 3.2 "Depth-first search" — PDS traverses minimal forests of BVTTs and the core FPQ algorithm has been presented. Using FPQ with PDS requires one trivial change in Algorithm 7 "Sprouting": Let S be a priority queue, sorted on distances in increasing order, instead of a stack.

3.3.2 Partitioning is particularly important with PDS

Insertions and removals are slow with PDS. This further motivates a partitioning of the front. A priority queue PQ used by PDS is kept small by ensuring that no leaves nor node v with $\|v\| > \epsilon$ are inserted into PQ.

One approach[3] is to avoid partitioning. Simply insert all BVTTs nodes, including raised nodes post raisal, into a priority queue and sprout. Encountered leaves are also inserted into PQ. However, this approach can be inadequate beyond usability since $|PQ|$ becomes large.

3.3.3 PDS allows for early traversal termination

PDS guarantees that a node v^* with $\|v^*\| < \epsilon$ can never be popped from a priority queue PQ used for PDS if a node v with $\|v\| > \epsilon$ has been popped. This follows from Lemma 1.

But if v^* can not be popped after popping $\|v\| > \epsilon$, then it is not useful to continue sprouting the front since it will not approach an optimal front. Hence, traversal may terminate when $\|v\| > \epsilon$ is encountered although S is not empty, saving $\mathcal{O}(n \log n)$ pop-operations where n denotes $|S|$ at the time v is encountered.

All remaining elements of S must be copied into SF in order to ensure that the front $F = RF \cup SF \cup L$ remains valid. This must be done in a pass linear in the number of remaining elements of S. Regardless, an $\mathcal{O}(|S|)$-pass without computing any BV-BV tests is more efficient than continued computation of BV-BV tests during $\mathcal{O}(|S| \log |S|)$-traversal.

A single simple modification to Algorithm 7 "Sprouting" realizes this optimization — after line 30, copy remaining elements in S into SF and break out of the while-loop.

3.3.4 Analysis of FPQ vs GFT

A single large priority queue can be used to explore a minimal forest of BVTTs or several small priority queues can be used to explore several minimal BVTTs. I proved, in Section 2.3.7 "Minimal BVTT forest", that a minimal forest $\mathcal{F}$ of BVTTs is smaller than the union of minimal BVTTs. But it is not clear if a single large priority queue is beneficial over several smaller priority queues.

A (rough) analysis of when it is expected to benefit from FPQ in comparison to GFT when using PDS follows. Let n denote the number of BVTTs and let k denote the average number of sproutees per BVTT fronts, using GFT. Let k' denote the average number of sproutees per BVTT in a front using FPQ. GFT uses n priority queues to maintain k elements each while FPQ uses one priority queue to maintain nk' elements, hence GFT with PDS is $\mathcal{O}(nk \log k)$ and FPQ with PDS is $\mathcal{O}(nk' \log nk')$. It follows that if $nk' \log nk' < nk \log k$, then FPQ is expected to be beneficial.

Let $c = \frac{k}{k'} \geq 1$. A quick calculation which hints of a necessary condition follow:

[3] the approach which was initially implemented in early stages of this master thesis

$$nk \log k > nk' \log nk' \Leftrightarrow c \log k > \log nk' \Leftrightarrow k^c > nk' \Leftrightarrow ck^{c-1} > n$$

From this calculation it is understood that the ratio c must be sufficiently large, which is to say, the set of sproutees with GFT which are not sproutees with FPQ must be sufficiently large.

For example, if FPQ manages to move the BVTT front up one level in comparison to the union of smaller GFT fronts, then $c = 2$ (since BVTTs are binary trees and the number of sproutees is assumed to be proportional to the front size). It follows that a sufficient condition is $k > \frac{n}{2}$ when $c = 2$. However, if $c < 2$, then the condition may or may not be fulfilled. For example, if $k = 27$ and $n = 5$ then $\frac{3}{2}(27)^{\frac{3}{2}-1} = \frac{9}{2} \not> 5$.

A conclusion which can be made from this brief calculation is that FPQ can be more costly than GFT since FPQ must maintain a single large priority queue instead of n smaller priority queues — but FPQ is expected to pay off if the set of sproutees with FPQ is sufficiently smaller than the union of all sproutees obtained with GFT.

3.4 Efficient sibling-raise with explicit BVTTs

It has been previously stated, in Section 2.3.6 "Generating BVTTs on-the-fly", each child is completely determined from its parent and each parent is completely determined from its children. This makes it, at a first glance, attractive to use a implicit BVTT as opposed to a explicit BVTT since there does not seem to be any need to store nodes that lie in between roots and front nodes.

A sibling-raise routine, which is applicable to both implicit or explicit BVTTs, was presented in Algorithm 5 "Raising". It detects siblings by generating parents and storing them in a hash map H. However, this is a source of overhead.

If we limit ourselves to solely explicit BVTTs, then it is possible to get rid of the hash map H for detecting siblings: siblings of visited nodes can be marked with some stamp $k \in \mathbb{N}$, and check for each visited node if its sibling is marked. If so, the two siblings can be merged into their common parent at the position of the first encountered sibling, effectively raising the front in a linear pass. The second sibling is removed with the swap-and-pop idiom. Complexity-wise, the sibling-raise algorithm has been transformed from an *average* $\mathcal{O}(n)$-algorithm to an *always* $\mathcal{O}(n)$-algorithm.

A more efficient sibling-raise strategy, which relies on usage of explicit BVTTs, has been described and is presented as pseudocode in Algorithm 9 "Raising (explicit BVTTs)".

Compare Algorithm 9 "Raising (explicit BVTTs)" with Algorithm 5 "Raising". All traces of the hash map H is gone, thanks to the possibility of persistently marking nodes with $mark$ and storing sibling-indices $sibidx$.

An integer stamp k is necessary, as opposed to just marking nodes as visited with some boolean, since the raise-routine is called in all iterations (i.e. for each given $\mathcal{C}$-space point q_i). A mark must not be kept over iterations, since a sibling which was encountered on a previous iterations could have been sprouted on the current iteration — invalidating the mark. However, it is possible to reset marks in

a linear pass, but this introduces unnecessary overhead. Hence, an integer stamp k is used to mark siblings in the front.

In conclusion, usage of explicit BVTTs enables removal of H which in turn increases performance at the cost of increased memory consumption. This result is presented in Section 5.10 "Raising and sprouting on implicit and explicit BVTTs".

3.5 Recursively raising siblings

A recursive raisal scheme is presented in this section. It is a generalization of the previously introduced raise-sibling strategy, in which raised parents are further raised if their sibling exist in the front.

Raising siblings a single level is viable whenever the decrease in size, over iterations, of an optimal front is sufficiently bounded. If nodes need only be raised a single level in order to obtain an optimal front, then a single level raisal routine is sufficient. However, if an optimal front does decrease in size more rapidly, then a single level raisal routine functions as a friction for a front that is trying to keep up with the optimal front. For this reason, a recursive sibling-raise routine is developed.

3.5.1 A $\mathcal{O}(n^2)$ recursive sibling-raise routine

A simple approach is to apply either Algorithm 5 "Raising" or Algorithm 9 "Raising (explicit BVTTs)" to RF until RF does not change in size any longer. Both algorithms are, at least on average, linear-time algorithms.

A worst-case scenario is that only a single pair of siblings is raised on each recursion, effectively decreasing $|RF|$ by one per recursion. Mathematically, the number of visited nodes during a naïve recursive raisal algorithm is bounded from above by:

$$n + (n-1) + \ldots + 1 = n^2 \tag{3.6}$$

where n denotes the size of RF at each recursion. It follows that this naïve approach to recursively raising siblings has quadratic time-complexity $\mathcal{O}(|RF|^2)$.

3.5.2 A $\mathcal{O}(n)$ recursive sibling-raise routine

I developed an algorithm which performs a recursive raise in $\mathcal{O}(n)$, where $n = |RF|$. Due to time constraints, only a recursive raise routine for explicit BVTTs was implemented.

The linear-time recursive raise algorithm works as follows: Traverse all elements of RF as usual and mark their siblings. If some node v is marked, replace its sibling by the common parent, erase v from RF and call a recursive subroutine on $v_p = \text{parent}(v)$. The recursive subroutine checks if v_p is marked and if so, merge v_p and its sibling into their common parent and apply the recursive subroutine on the former back element and then on $\text{parent}(v_p)$, otherwise mark the sibling of v_p and terminate the recursion. See Algorithm 10 "Raising (explicit BVTTs)" for a pseudocode description.

There are two recursive calls of the recursive raise subroutine. The latter correspond to the straight-forward recursive call where a raised parent is further raised. The former call, however, is made to ensure index validity.

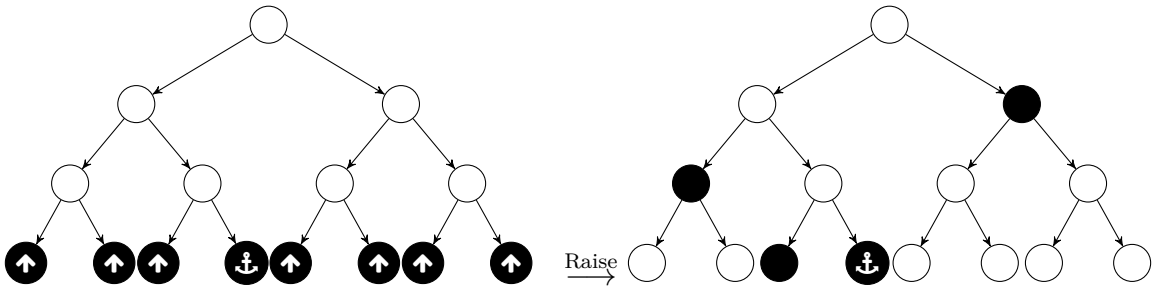
It is important that the sibling index *sibidx* on any node remains valid, even after removing elements from RF. Otherwise, it is possible that an arbitrary node is overwritten by a parent of two siblings. Only back nodes are deleted from RF. Hence, it is a left-most sibling node $v_s = \text{sibling}(\text{BACK}(RF))$ which may get an invalidated sibling index. However, by visiting the (former) back node at $RF(r)$ first, there is no risk that the invalid index on v_s is used, since v_s is overwritten directly by $\text{parent}(RF(r))$. This is the reason why the recursion recurses into former back nodes prior to recursing into parent nodes.

Each node of RF is visited at most once. In addition to nodes of RF, all nodes of the parent set image $P(RF)$ are visited at most once, i.e. all nodes "above" RF are visited at most once. Note that $|P(RF)| = |RF| - 1$ since BVTTs are binary. It follows that at most $2|RF| - 1$ nodes are visited. Hence Algorithm 10 "Raising (explicit BVTTs)" is $\mathcal{O}(n)$ where $n = |RF|$.

3.5.3 Anchor nodes

It might appear that a recursive sibling-raise always raises a front into BVTT roots. BVTT roots does, usually, not constitute a good approximation of some optimal front. However, thanks to *anchor nodes*, a recursive sibling-raise does not raise a front into the set of roots in a BVTT forest.

An anchor node v_a is any node which is not a member of RF during raisal. Hence, any sibling of v_a can not become raised, effectively "anchoring" the front at v_a. See Figure 3.5 for an illustration of anchoring nodes which prevents a front from becoming raised to forest roots.



(a) Before recursive raisal (b) After recursive raisal

Figure 3.5: A BVTT before raising and after raising. Nodes which are about to become raised are marked with up-arrows. An anchor node is marked with an anchor.

3.6 Resetting a front

An $\mathcal{O}(n)$ recursive raise strategy, where two sibling nodes are raised into a common parent, has been presented in Algorithm 10 "Raising (explicit BVTTs)". However, a recursive raise may "over-raise", that is, raise a front F far above that of an optimal

front F^*. In that case, it could have been more efficient to raise the node into its root node and sprout it instead.

Some authors suggests resetting a front, according to some heuristic, to either get rid of a raise-operator all together or to complement a sibling-raise strategy [16][17][9]. In the context of FPQ, resetting a front replaces all front nodes by all BVTT root nodes, after which an approximation of an optimal front is rebuilt.

Ideally, a front F is reset whenever it is computationally cheaper to approximate an optimal fronts by sprouting from BVTT roots than it is to approximate an optimal front by raising F. A heuristic must decide when a front should be reset, however, it is far from obvious what heuristic should be used.

I have experimented with several simple heuristics, all of which decide if a front should be reset or not. A simple heuristic that performs decently is:

$$\text{do\_reset} := \begin{cases} \text{true}, & \text{if the number of raised nodes exceeds 9\% of the total front size} \\ \text{false}, & \text{otherwise} \end{cases} \tag{3.7}$$

The rationale for this heuristic is that if an optimal front F^* decreases in size, a raise routine will cause an approximation F to decrease in size, however, the rate of decrease may not be as rapid as that of F^*. Hence, if a measured decrease in F is sufficiently large, then $|F^*| \ll |F|$ such that a reset is efficient. The magic threshold number 9% was found, through a binary search, to minimize the total number of BV-BV calculations for a particular scene.

There are possibly better heuristics for resetting a front [17]. However, advanced reset methods have not been prioritized and hence not implemented.

A heuristic determining if a front should be reset has been presented. However, it is not clear when Equation 3.7 should be evaluated nor when the front should be reset in terms of Algorithm 8 "Complete FPQ with DF search". A front could be reset either immediately or post ϵ initialization, post partitioning, post raising or post sprouting.

3.6.1 Evaluating heuristic during raisal

It is unnecessary to raise nodes if a front-reset is inbound. A parent need only be raised if Equation 3.7 is false. This motivates marking a front for reset during raisal, since only then are unnecessary raisals avoidable.

A simple counter is maintained in all raise routines, which keeps tracks of the number of raised nodes. Prior to merging two siblings into their common parent, Equation 3.7 is evaluated with said counter. If true, no nodes are further raised and the front is marked for reset. Otherwise, the algorithm proceeds to raise nodes as usual.

3.6.2 Reset after initializing ϵ

It is important for partitioning and sprouting to initialize an upper-bound ϵ on the separation distance $\delta \leq \epsilon \neq \infty$ since more BVTT nodes can be pruned. This is

intuitively true for depth-first traversal, but also for a PDS traversal despite that PDS traverses minimal forests of BVTTs.

With PDS, the number of popped nodes (deletions) from a priority queue PQ does not depend on an initial value of ϵ. However, the number of insertions decreases if $\epsilon \neq \infty$ since insertions into PQ can be avoided for several execution branches, given in Table 3.1. Hence it is important to initialize $\epsilon \neq \infty$, even though minimal forests of BVTTs are visited when using PDS.

In conclusion, a front is reset after initializing ϵ with INIT_EPS but before front partitioning, if and only if Equation 3.7 evaluated to true during the previous iteration.

3.7 Duplex priority queue

Depth-first search traverses more nodes than PDS in a forest of BVTTs or as many nodes (by Theorem 1). However, the number of visited BVTT nodes is for some test-scenarios only marginally larger with depth-first in comparison to the number of visited nodes when using PDS). This is a result, that is presented in Section 5.5 "Bounding volume and triangle distance computations", that motivates development of *duplex priority queue (DPQ)*.

A hypothesis which provides an answer as to why this is the case follows: A depth-first search, in which the child with the smallest distance is traversed first, generates a path throughout the forest of BVTTs which is *roughly* the same as a path generated by PDS.

If the hypothesis holds true, then PDS oftentimes traverses down a child v of the current node as opposed to jumping to a different place in the BVTT forest. In particular, $\|v\| < \text{top}(PQ)$. Inserting v into PQ only to remove it is of no use. Hence, it could be beneficial to avoid a costly $\mathcal{O}(\log n)$ insert and simply insert it into a stack instead.

A priority queue, which I call DPQ[4], with $\mathcal{O}(1)$ insertion iff $\|v\| < \min\{PQ\}$ and $\mathcal{O}(1)$ deletion if the stack is not empty is presented in Algorithm 11 "DPQ". Three standard operations are presented: insert, top and pop. The underlying stack of DPQ is denoted DPQ_S and the underlying priority queue of DPQ is denoted DPQ_{PQ}.

Note that DPQ is a priority queue since a smallest BVTT node is read or deleted at all times.

Under the following access pattern, DPQ performs insertion and deletion in $\mathcal{O}(1)$ time:

1. Insert a BVTT node v with $\|v\| \leq \min\{DPQ\}$ into a DPQ
2. Pop a node from said DPQ

In essence, DPQ attains the efficiency of a fast depth-first search whenever PDS traversal coincide with depth-first traversal. This comes at the cost of if-check overhead during insertion and deletion.

[4]To the best of my (lacking) knowledge, this type of priority queue is novel. However, I strongly believe this has been done before and I do not claim myself as the inventor

It is, perhaps contrary to intuition, crucial to perform the distance comparison in POP and TOP since DPQ_PQ may hold smaller nodes than DPQ_S even though the smallest nodes are always inserted into DPQ_S. This occurs in the following scenario:

1. Insert a BVTT node v_0, with $\|v_1\| = 3.0$ (inserts into DPQ_S)
2. Insert a BVTT node v_1, with $\|v_2\| = 1.0$ (inserts into DPQ_S)
3. Insert a BVTT node v_2, with $\|v_3\| = 2.0$ (inserts into DPQ_{PQ})
4. Pop a node (pops v_2 from DPQ_S).

At this state, $DPQ_S = \{v_1\}$ and $DPQ_{PQ} = \{v_3\}$ so $\min\{DPQ_{PQ}\} < \min\{DPQ_S\}$. Hence it is possible that $\min\{DPQ_{PQ}\} < \min\{DPQ_S\}$ which warrants a distance comparison in POP and TOP.

Remark: Two children v_l and v_r of a BVTT node v may both be of smaller distance than ϵ. If the larger child is inserted into first then both children can be inserted into the underlying stack in $\mathcal{O}(1)$-time — otherwise, the smallest node is inserted first, after which the larger node is inserted into the underlying priority queue in $\mathcal{O}(\log n)$-time. Hence, the order of insertion into S during Algorithm 7 "Sprouting" matters if S is a DPQ. In terms of pseudocode, changes are superfluous since the ordering warranted by DPQ coincide with the previous order of insertion, as presented in Section 3.2.5 "Sprouting".

3.8 Sampling points in $\mathcal{C}_{obs}$

The sampling-based motion-planner used in IPS is based on RRTs. IPS may sample points in the *obstacle-region* $\mathcal{C}_{obs} \subset \mathcal{C}$, which is a subset of $\mathcal{C}$-space where geometric objects collide [31]. By definition of $\mathcal{C}_{obs}$:

$$\exists v : \|v(q)\|_{leaf} = 0, \forall q \in \mathcal{C}_{obs} \tag{3.8}$$

This is trivially true, since if there does not exist a leaf with distance zero, then the configuration point can not possibly lie in $\mathcal{C}_{obs}$.

Since no separation distances are negative, any BVTT node v with $\|v(q)\|_{leaf} = 0$ gives a smallest separation distance δ. Searching for a smaller upper-bound $\epsilon > \delta$ or a smaller δ is futile. Therefore, FPQ may terminate early whenever a BVTT node v with $\|v(q)\|_{leaf} = 0$ is encountered.

I suggest two places in which early termination, due to collision, is possible:

- Return $\delta = 0$ immediately if a leaf with distance zero is encountered in Algorithm 3 "Initialization of ϵ"
- Check if $\epsilon = 0$ before line 3 in Algorithm 7 "Sprouting". If true, break out of the while-statement and copy remaining elements of S into SF in order to preserve front validity. Otherwise, continue traversal as usual.

These optimizations are only fruitful if FPQ is used for points in $\mathcal{C}$-space which are also members of $\mathcal{C}_{obs}$.

3.9 A safe ϵ

In this subsection, a method of computing a safe initial ϵ_1 is presented. Here, safe means that it is guaranteed that no minimum distance between two sets of geometric objects can be greater than ϵ_1.

Two static geometric objects, $\mathcal{A}$ and $\mathcal{B}$, have a fixed minimum distance for all world configurations:

$$\|\mathcal{A}(q_{\mathcal{A}}(i)) - \mathcal{B}(q_{\mathcal{B}}(i))\| = \|\mathcal{A}(q_{\mathcal{A}}) - \mathcal{B}(q_{\mathcal{B}})\| = c \tag{3.9}$$

This gives a simple method for finding a safe ϵ. If the following holds:

$$\exists \mathcal{A} \in A : |q_{\mathcal{A}}| = 1 \wedge \exists \mathcal{B} \in B : |q_{\mathcal{B}}| = 1 \tag{3.10}$$

then simply compute the minimum distance for every static edge. Take the smallest minimum distance and denote it ϵ_{safe}.

It is geometrically obvious why $\epsilon_i \leq \epsilon_{safe}$: Two objects which stays put forever has some fixed minimum distance ϵ_{safe} in between them — a minimum distance ϵ_i can not exceed ϵ_{safe} for then it would not be the minimum distance. Finding ϵ_{safe} can be done naïvly in $O(n \cdot d)$ where $n = |\{e_{\mathcal{A},\mathcal{B}} \in E : \mathcal{A}, \mathcal{B} \text{ static}\}|$ and d is the time complexity for computing the separation distance. A safe ϵ_{safe} can in an initialization phase and proceeds to set $\epsilon_0 := \epsilon_{safe}$. Further, it is safe to always set $\epsilon_i := \min(\epsilon_i, \epsilon_{safe})$. If Equation 3.10 is false, set $\epsilon_{safe} := \infty$.

Another way to compute a possibly smaller ϵ_{safe} is compute a smallest maximal distance between two geometric objects $\mathcal{A} \in A$ and $\mathcal{B} \in B$ over all world configurations:

$$\epsilon_{safe} := \min_{\mathcal{A}\in A, \mathcal{B}\in B} \max_{q_{\mathcal{A}}\in\mathcal{C}_{\mathcal{A}}, q_{\mathcal{B}}\in\mathcal{C}_{\mathcal{B}}} \|\mathcal{A}(q_{\mathcal{A}}) - \mathcal{B}(q_{\mathcal{B}})\| \tag{3.11}$$

Unfortunately, it is not straight-forward how to compute Equation 3.11. Naïvly, every point $q \in \mathcal{C}_A \times \mathcal{C}_B$ must be examined.

Further, I believe that ϵ_{safe} is never smaller than ϵ, post ϵ-initialization as presented in Algorithm 3 "Initialization of ϵ", if a minimum over static edges is used to initialize ϵ prior to calling Algorithm 3 "Initialization of ϵ". However, it is difficult to estimate how often a ϵ_{safe}, computed by Equation 3.11, is smaller than ϵ post ϵ-initialization. Regrettably, due to the sheer complexity of Equation 3.11, computation of Equation 3.11 it not considered.

3.10 Correctness of FPQ

This section briefly presents a list of tests that are used to verify correctness of FPQ and its subcomponents.

Correct distance: Check if the distance output by FPQ coincides with the distance output of IPS for all configuration points. This test is clearly sufficient for determining if FPQ returns a correct distance on some scene, given a sequence of configuration points.

Necessary-test for forest equivalence: Exhaustively traverse all BVH-pairs with IPS and count all visited BVH-pairs and check if the count match the number of counted BVTT nodes visited during exhaustive search with FPQ. This test is a necessary-test which does not necessarily guarantee that the traversed BVTT correspond to the traversed set of BVH-pairs. However, if this test fails, it means that the forest of BVTTs which FPQ traverses is not the same as the forest of BVTTs which IPS traverses. This test is only enabled for small scenes, for exhaustive enumeration of large BVTTs is practically intractable as hinted by Section 2.4.1 "Memory usage of front tracking".

Front redundancy: Recursively enumerate all parents and denote the set of visited parents as P. Then $P \cap F = \emptyset$ must hold. By definition 8, this test is sufficient for determining if a front is redundant or not.

Front correctness #1: Check that there are no duplicate nodes in a front F. This test is a necessary-test.

Front correctness #2: Recursively enumerate all parents of a front F, denote the set of visited parents as P. Then $|P| - |F| = |E|$ must hold, where $G = (A, B, E)$ as usual. If $|E| = 1$, then this check ensures that the number of visited parents of a sole BVTT is precisely the size of the front minus unity, which must be the case since BVTTs are binary. In general $|E| \neq 1$, which explains why unity is exchanged for $|E|$. This test is a necessary-test.

DPQ correctness: Generate a set S of 10k random floating point values in range $[0, 1]$. Insert each element of S into a known working implementation of a priority queue (`std::priority_queue`) and into an instance of DPQ. Check if the two sequences of popped floating-point numbers are equivalent.

I conjecture that front correctness #1 and #2 combined constitute a sufficient-test for front correctness.

All of the listed tests are implemented in C++ in order to ensure that FPQ behaves as expected.

Algorithm 7 Sprouting

```
1: function SPROUT(S, SF, L, q, ε)
2:     while S ≠ ∅ do
3:         v ← TOP(S)
4:         POP(S)
5:         if ‖v(q)‖ < ε then
6:             v_l ← LEFT(v), v_r ← RIGHT(v)
7:             v_< ← SMALLEST(v_l, v_r, q), v_> ← LARGEST(v_l, v_r, q)
8:             if ‖v_<(q)‖ < ε then
9:                 if LEAF(v_<) then
10:                    if ‖v_<(q)‖_leaf < ε then
11:                        ε ← ‖v_<‖_leaf
12:                        if ‖v_>(q)‖ < ε then
13:                            if LEAF(v_>) then
14:                                if ‖v_>(q)‖_leaf < ε then        ▷ Case 1 in Table 3.1
15:                                    ε ← ‖v_>(q)‖_leaf
16:                                    L ← L ∪ {v_<}, L ← L ∪ {v_>}
17:                                else                             ▷ Case 2 in Table 3.1
18:                                    L ← L ∪ {v_>}, L ← L ∪ {v_<}
19:                            else                                 ▷ Case 3 in Table 3.1
20:                                L ← L ∪ {v_<}, S ← S ∪ {v_>}
21:                        else
22:                            if LEAF(v_>) then                    ▷ Case 4 in Table 3.1
23:                                L ← L ∪ {v_>}, L ← L ∪ {v_<}
24:                            else                                 ▷ Case 5 in Table 3.1
25:                                L ← L ∪ {v_<}, SF ← SF ∪ {v_>}
26:                    else...
27:                else...
28:            else...
29:        else
30:            SF ← SF ∪ {v}
31:    return ε
32: end function
```

Algorithm 8 Complete FPQ with DF search

function ONE_TIME_INIT(A, B, SF)
  for $\forall \mathcal{A} \in A$ **do**
    for $\forall \mathcal{B} \in B$ **do**
      $SF \leftarrow SF \cup \{\text{CREATE\_BVTT\_ROOT}(\mathcal{A}, \mathcal{B})\}$
end function
function DIST(L, SF, RF, q_i)
  $\epsilon \leftarrow \text{INIT\_EPS}(L, q_i)$
  $\text{PARTITIONING}(RF, SF, q, S, \epsilon)$
  $RF \leftarrow RF \cup L$
  $\text{RAISE}(RF)$
  $L \leftarrow RF \cap L$
  $\epsilon \leftarrow \text{SPROUT}(S, SF, L, q, \epsilon)$
  $\delta \leftarrow \epsilon$
  return δ
end function

Algorithm 9 Raising (explicit BVTTs)

function RAISE(RF, k)
  $i \leftarrow 0$
  while $i < |RF|$ **do**
    $v_i \leftarrow RF(i)$
    if IS_ROOT(v_i) **then**
      $i \leftarrow i + 1$
    else if $\text{SIBLING}(v_i)_{mark} \neq k$ **then** ▷ True if v_i has not been visited by raise-routine on this iteration
      $\text{SIBLING}(v_i)_{mark} \leftarrow k$
      $\text{SIBLING}(v_i)_{sibidx} \leftarrow i$
      $i \leftarrow i + 1$
    else ▷ Sibling of v_i has been visited so both exist in front
      $R(RF(i)_{sibidx}) \leftarrow \text{parent}(v_i)$
      $\text{SWAP}(RF(i), \text{BACK}(RF))$
      $\text{POP\_BACK}(RF)$
end function

Algorithm 10 Raising (explicit BVTTs)

```
1: function BASE_RAISE(RF, k)
2:     while i < |RF| do
3:         v_i ← RF(i)
4:         if IS_ROOT(v_i) then
5:             i ← i + 1
6:         else if SIBLING(v_i)_mark ≠ k then
7:             SIBLING(v_i)_mark ← k
8:             SIBLING(v_i)_sibidx ← i
9:             i ← i + 1
10:        else                ▷ Sibling of v_i has been visited so both exist in front
11:            j ← RF(i)_sibidx
12:            R(j) ← parent(v_i)
13:            SWAP(RF(i), BACK(RF))
14:            POP_BACK(RF)
15:            RECURSIVE_RAISE(RF, k, j)
16: end function
17: function RECURSIVE_RAISE(RF, k, j)
18:     if j ≥ |RF| ∨ IS_ROOT(RF(j)) then
19:         return
20:     else if SIBLING(RF(j))_mark ≠ k then
21:         SIBLING(RF(j))_mark ← k
22:         SIBLING(RF(j))_sibidx ← j
23:     else
24:         l ← RF(j)_sibidx
25:         RF(j) ← PARENT(RF(j))
26:         SWAP(RF(l), BACK(RF))
27:         POP_BACK(RF)
28:         RECURSIVE_RAISE(RF, k, l)      ▷ Recurse into former back node first
29:         RECURSIVE_RAISE(RF, k, j)                    ▷ Recurse into parent last
30: end function
```

Algorithm 11 DPQ

```
1: function INSERT(DPQ, v)
2:     if PQ(DPQ) ≠ ∅ then
3:         if ‖v‖ ≤ ‖TOP(DPQ_S)‖ then
4:             DPQ_S ← DPQ_S ∪ {v}                          ▷ O(1)
5:         else
6:             DPQ_PQ ← DPQ_PQ ∪ {v}                        ▷ O(log n)
7:     else if DPQ_PQ = ∅ ∨ ‖v‖ < ‖TOP(DPQ_PQ)‖ then
8:         DPQ_S ← DPQ_S ∪ {v}                              ▷ O(1)
9:     else
10:        DPQ_PQ ← DPQ_PQ ∪ {v}                            ▷ O(log n)
11: end function
12: function TOP(DPQ)
13:     if DPQ_PQ = ∅ ∨ (DPQ_S ≠ ∅ ∧ ‖TOP(DPQ_S)‖ ≤ ‖TOP(DPQ_PQ)‖ then
14:         return TOP(DPQ_S)
15:     else
16:         return TOP(DPQ_PQ)
17: end function
18: function POP(DPQ)
19:     if DPQ_PQ = ∅ ∨ (DPQ_S ≠ ∅ ∧ ‖TOP(DPQ_S)‖ ≤ ‖TOP(DPQ_PQ)‖ then
20:         POP(DPQ_S)
21:     else
22:         POP(DPQ_PQ)
23: end function
```

4
PQP

The *Proximity Query Package* (PQP) is used by internally IPS for computing separation distances between two BVs and for computing them between two geometric primitives. In particular, for computing separation distances between two *rectangle swept spheres* (RSSs) and for computing them between two triangles. I also use PQP for the aforementioned computations.

PQP is capable of computing $\|\mathcal{A} - \mathcal{B}\|$ where $\mathcal{A}$ and $\mathcal{B}$ are two geometric objects. In other words, PQP can compute a separation distance between two BVHs. However, PQP does not provide any means to compute a distance $\|A - B\|$ between two sets of geometric objects A and B. Analogously, PQP does not provide any means for computing a separation distance between sets of rigid bodies (for example robotic arms). FCC has developed an algorithm which enables PQP to compute $\|A - B\|$. I present that algorithm in Section 4.2 "IPS traversal algorithm over forests of BVTTs" and I will refer to this extended version of PQP as simply IPS. IPS can be regarded as a superset of PQP that is capable of traversing forests of BVTTs.

In summary, PQP has three relevant use cases: (1) to compute a separation distance between two BVs (2) to compute a separation distance between two geometric primitives and (3) for comparing FPQ with IPS. This chapter settles how a separation distance between two BVs $\|v(q)\|$ is computed and how a separation distance between two triangles $\|v(q)\|_{leaf}$ is computed with PQP. I also present how IPS uses PQP for computing $\|A - B\|$.

4.1 Computing separation distances

This section describes how separation distances between either two BVs or two geometric primitives are computed in terms of algorithms of Chapter 3 "Forest proximity querying" by using PQP. Some brief presentations of two PQP technicalities follows:

- PQP computes distances under the assumption that one RSS volume out of two is axis-aligned [2].
- A BV is expressed relative its model space via a local transformation L.

A transformation $M_{1\to 2}$ ensures that the first bullet-point is satisfied:

$$M_{1\to 2} = L_2^{-1} \times M_{world} \times L_1 \text{ with } M_{world} = M_{world,2}^{-1} \times M_{world,1} \tag{4.1}$$

Equation 4.1 takes the first BV from its local space to its BVH model space via L_1, then takes it to model space of second BVH via a *world transformation* M_{world}

and finally to the local space of the second BV, in which the second BV is axis aligned.

An algorithm for computing a real number representing a separation distance between two BVs on a BVTT node v is given in Algorithm 12 "Computing RSS distances".

Algorithm 12 Computing RSS distances

```
1: function DIST(v, q)
2:     M_{1→2} ← L_2(v)^{-1} × M_world(q) × L_1(v)
3:     return PQP_RSS_DIST(BV1(v), BV2(v), M_{1→2})
4: end function
```

Separation distances between geometric primitives are computed analogously by replacing instances of BV-calls by geometric primitive-calls and finally replacing the RSS distance call with a geometric primitive distance call.

Note that M_{world} depends on a $\mathcal{C}$-space point q but L_1 and L_2 does not depend on q. This is because geometric objects are by assumption rigid and therefore BV nodes within BVHs remain fixed in relation to BVH roots. In practice, a representation of a $\mathcal{C}$-space point is not present in any implementation presented in this thesis[1]. Instead, each BVTT nodes owns pointer to its corresponding M_{world}, all of which are updated in a $\mathcal{O}(|A| \cdot |B|)$ pass.

In conclusion, any BVTT node v must store:

- A pointer to a transformation M_{world}
- Two pointers to underlying BVHs, granting access to underlying BVs with local transformations

Pseudocode for computing $\|v(q)\| = \text{dist}(v, q)$ has been presented and a method for computing $\|v(q)\|_{leaf}$ in terms of Algorithm 12 "Computing RSS distances" has been presented, effectively covering two out of three use cases of PQP.

4.1.1 Transformations are relative to parents in IPS

All calculations of $M_{1\leftarrow 2}$ within FPQ are computed according to Equation 4.1, for a total of two transformation multiplications per BVTT node. IPS does a single transformation multiplication per BVTT node. This is achieved by letting BV nodes within BVHs own a transformation which takes any child to its parent space, instead of model space. This allows for computation of $M_{1\leftarrow 2}$ by using the following equation, which depends on if a child of BVH_1 or BVH_2 is descended:

$$M_{1\to 2,i} = \begin{cases} L_1^{-1} \cdot M_{1\to 2,i-1} & \text{if descending the first BVH} \\ M_{1\to 2,i} = M_{1\to 2,i-1} \cdot L_2 & \text{if descending the second BVH} \\ M_{1\to 2,0} = M_{world} & \text{if i=0} \end{cases} \tag{4.2}$$

[1]The notion $\|v(q_i)\|$ and $\|v(q_i)\|_{leaf}$ is used to emphasize that that distances depend on a current iteration.

L_1 denotes the transformation of the current BV of BVH_1 and similarly for L_2.

Exactly one of the equations are computed on visiting a BVTT node with IPS, effectively updating $M_{1\to 2}$ instead of recomputing it. However, this trick is not used by FPQ since a combination of sprouting and raising implies several transformation multiplications back and forth, introducing a risk of numerical instability. A numerical instability could cause two severe issues:

- BVTT distances become either too large or too small, causing both false positives and false negatives with respect to BVTT-subtree pruning.
- BVTT leaf distances become inaccurate. This is critical since exact distances (with only a very small error) must be returned in all situations.

All FPQ algorithms under Chapter 3 "Forest proximity querying" use Equation 4.1 instead of Equation 4.2.

4.2 IPS traversal algorithm over forests of BVTTs

This section describes how IPS computes separation distances over forests of BVTTs by utilizing PQP[1][2]. An understanding of how IPS performs proximity queries is necessary since IPS is compared against FPQ in Chapter 5 "Results".

IPS uses the following techniques for computing proximity queries:

Common: Tests between geometric primitives.
C: Test between BVs. In particular, tests between RSSs.
C: BVTTs are generated with the SAH-inspired technique, described by Equation 2.7.
C: A smallest sibling is traversed first.
Different: Depth-first traversal (PQP) over a forest of BVTTs, where a BVTT giving rise to a smallest separation distance is ordered first in the following iteration by swapping it with the (former) first BVTT.
D: Triangle caching per BVTT.

A bold **C** indicates that FPQ uses the listed method too, whereas a bold **D** indicates that FPQ does not use the listed method.

All listed methods, but the swapping scheme, originate from PQP[1][2][3]. A swapping scheme, in which the previously best BVTT is tested first, is used to guess what BVTT will give a smallest separation distance in a following iteration. This is plausible since $\|q_{i-1} - q_i\| < d$: a BVTT which gave the smallest separation distance in the previous iteration is likely to give a smallest separation distance in the current iteration.

A comparison between IPS and FPQ is fair since they traverse identical BVTTs forests, using the same PQP-routines for computing distances in between BVs and geometric primitives. The two differences is that of traversal, where PQP creates an ordering of BVTTs in contrast to FPQ which creates an ordering of BVTT-subtrees, and that of triangle caching as opposed to using a leaf front.

IPS computes $\|A(q_A(i)) - B(q_B(i))\|$ according to the pseudocode in Algorithm 13 "IPS distance routine", by calling PQP_DIST on each BVH-pair of a ordered set P.

PQP_DIST checks if $\mathcal{A}$ and $\mathcal{B}$ are leaves and if so, computes a test between the two corresponding geometric primitives. Otherwise, two pairs of BVHs are generated according to Equation 2.7 and descend down the pair with the smallest distance. Finally, check if further traversal is necessary and if so, recurse and update ϵ.

Algorithm 13 "IPS distance routine" is simplified: In reality, computed distances are cached in temporaries in order avoid duplicate distance computations.

This concludes the presentation of how IPS performs proximity queries with PQP.

Algorithm 13 IPS distance routine

```
1: function INIT(A, B)
2:     P ← ∅                                  ▷ Populated with pairs of BVHs (BVTTs)
3:     T ← ∅                                  ▷ Will hold cached triangle pairs per BVH-pair
4:     for ∀A ∈ A do
5:         for ∀B ∈ B do
6:             P ← P ∪ (A, B)
       return (P, T)
7: end function
8: function IPS_DIST(P, T, q_i)
9:     ε* ← ∞
10:    i_best ← 0
11:    for i = 0, ..., |P| do
12:        (A, B) ← P(i)
13:        ε ← PQP_DIST(A, B, q_i, ε*, T, i)
14:        if ε < ε* then
15:            ε* ← ε
16:            i_best ← i
17:        i ← i + 1
18:    SWAP(FIRST(P), P(i_best))
19:    SWAP(FIRST(T), T(i_best))
20:    δ ← ε* return δ
21: end function
22: function PQP_DIST(A, B, q_i, ε, T, i)
23:    if A and B are leaves then
24:        d ← ‖A(q_i) − B(q_i)‖_leaf
25:        if d < ‖T(i)(q_i)‖_leaf then
26:            T(i) ← (A, B)      ▷ Cache triangles between the two BVHs A and B
           return d
27:    (c_left, c_right) ← CHILD((A, B))                    ▷ See Equation 2.7
28:    if ‖c_left(q_i)‖ < ‖c_right(q_i)‖ then
29:        if ‖c_left(q_i)‖ < ε* then
30:            ε ← min(ε, PQP_DIST(left(c_left), right(c_left), q_i, ε, T, i))
31:        if ‖c_right(q_i)‖ < ε* then
32:            ε ← min(ε, PQP_DIST(left(c_right), right(c_right), q_i, ε, T, i))
33:    else
34:        if ‖c_right(q_i)‖ < ε* then
35:            ε ← min(ε, PQP_DIST(left(c_right), right(c_right), q_i, ε, T, i))
36:        if ‖c_left(q_i)‖ < ε* then
37:            ε ← min(ε, PQP_DIST(left(c_left), right(c_left), q_i, ε, T, i))
           return ε
38: end function
```

5

Results

I have motivated development of *Forest Proximity Query* (FPQ) which on some occasions performs better than state-of-the-art methods of proximity querying, in particular, better than *Industrial Path Solutions* (IPS) with respect to the number of *rectangle swept sphere* (RSS) tests and time. This is done by reducing the number of RSS calculations. FPQ variations are compared with each other using IPS as a reference point where applicable. Several measurements which helps understanding characteristics of FPQ and its variants are presented in this section. I explicitly point out for each figure what is particularly interesting.

Chapter 3 "Forest proximity querying" described and motivated FPQ, whose key selling-point is that it traverses fewer *bounding volume traversal tree* (BVTT) nodes than IPS, which in turn uses the well-known PQP library. I have implemented FPQ, using C++, and integrated it with the IPS software. The FPQ algorithm outputs a separation distance on any IPS-compatible scene with dynamic rigid bodies. An implementation within IPS enables a range of complex test-scenarios as well as a fair comparison of FPQ with PQP.

All results presented in this chapter are referenced in Chapter 6 "Discussion", which summarizes results as necessary.

5.1 Variations

I have developed variations of FPQ. The variations are:

- **I**mplicit versus **E**xplicit
- **S**ingle sibling raise versus **R**ecursive sibling raise
- **D**epth first search versus **P**riority directed search (with DPQ)
- (Optionally) **N**o reset — front-resetting is used unless explicitly stated otherwise.[1]

For example, **I+S+P** indicates that measurements are generated using an implicit BVTT where a front is (1) raised using the single level raise strategy and (2) sprouted using PDS.

Further, I present measurements for which front tracking has been disabled. I then write either **D** or **P** as in depth-first or priority directed search, respectively.

[1] A subtle detail is made explicit: If reseting is disabled, then raise routines still terminate early if the number of raised nodes exceeds 9% as described in Section 3.6.1 "Evaluating heuristic during raisal". This effectively limits the number of raised nodes per iteration.

FPQ with disabled front tracking is Algorithm 7 "Sprouting", with the minor modifications of (1) initializing S with all BVTT roots of a forest of BVTTs and (2) omitting insertions into L and SF.

There are four variations, three scenarios and eleven measures — in total $2^4 \cdot 3 \cdot 11 = 528$ combinations — thus an exhaustive examination is impractical. Instead, combinations that emphasize how different methods affect end-results are presented.

Finally, note that **I+S+P** often overlap exactly with **E+S+P**. This should not be surprising since they visit identical BVTTs. These two variants do not overlap for all timing results.

5.2 Benchmarking environment

Several benchmarking libraries are available for C++, such as Nonius, google/benchmark, Hayai and Celero. However, for the purposes of this thesis, no micro-benchmark library has been used since benchmark most often takes more time than 100ms. Instead, a simple timer based on `std::chrono::high_resolution_clock::now()` is used for all reported timings.

All measurements are averaged over 20 independent runs, unless explicitly stated otherwise. All measurements are conducted on a Windows 10 64bit computer with the Intel Core i7-7700k 4.20GHz processor and with 32GB of RAM installed.

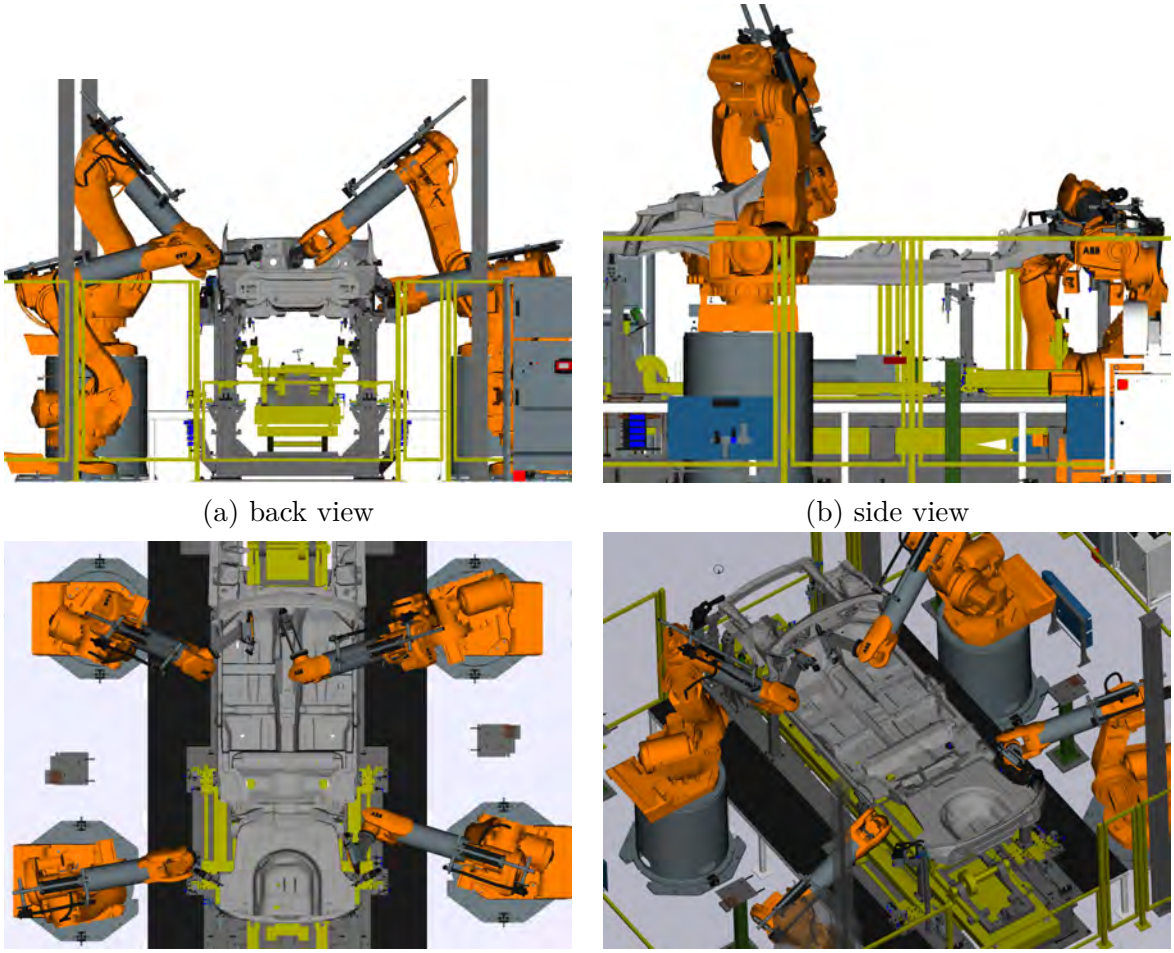
A test scene from Volvo Cars has been granted to FCC for demonstration of various projects and is shown in Figure 5.1. This test-scene models a real-world robot-cell in which four different ABB robot arms, composed of several rigid bodies, perform stud-welding on a car body. The Volvo test-scene induces various scenarios by letting the two sets of geometric objects A and B contain different combinations of robot-arms (dynamic) and car bodies (static) with static geometry, respectively.

There are four robotic arms in said test-scene, labeled IRB71, IRB72, IRB73 and IRB74. For all intents and purposes, they are to be regarded as identical. For the sake of simplicity, I write $A = \{\text{IRB71}\}$ when A holds all geometric objects that compose IRB71.

There are in total 2.5 million triangles, out of which 0.3 million triangles constitute robotic arms, 0.9 million are used by the car body and 1.3 million triangles comprise the static environment (some pillars, the assembly line and miscellaneous geometry). In Figure 5.2, a robotic arm is close to a car body. A close-up of said situation is shown together with an image showing a wireframe of the robotic arm and an imposed wireframe of the car body. The wireframe image demonstrates a high triangle resolution of the geometric objects.

The car body and the static environment are merged into a single large geometric object for which a single large BVH of RSSs is built. All four robotic arms are composed of rigid parts (see Section 1.2.2 "Robot arms as composite rigid bodies"), where each rigid part is represented as a BVH of RSSs.

Computing a minimum separation distance between a robotic arm and the car with static geometry corresponds to computing $\text{dist}(i; A, B)$ with A being the set of rigid parts and B being the car geometry with static geometry.

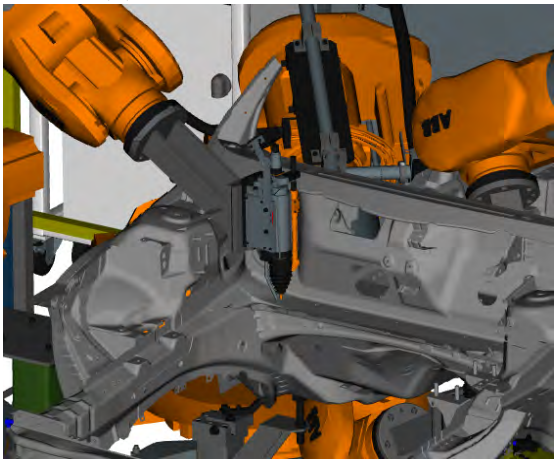


(a) back view (b) side view

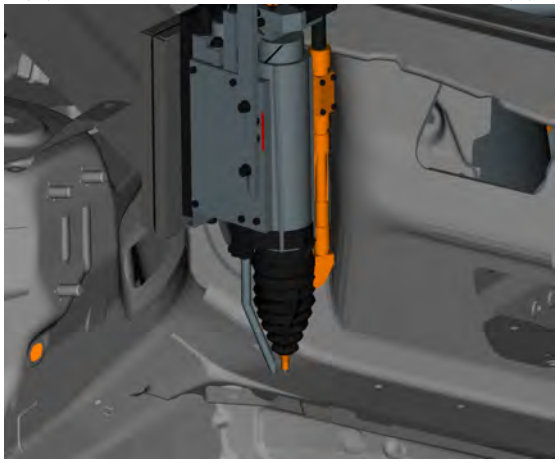
(c) top-down view (d) isometric view

Figure 5.1: Scene consisting of four ABB robotic arms. Each arm is stud-welding on a car body. All measurements and benchmarks are performed on this scene. Geometry by courtesy of Volvo Cars.

(a) Overview of robotic arm



(b) Close-up of robotic arm shown in (a)



(c) Wireframe of robotic arm with imposed wireframe of the car body

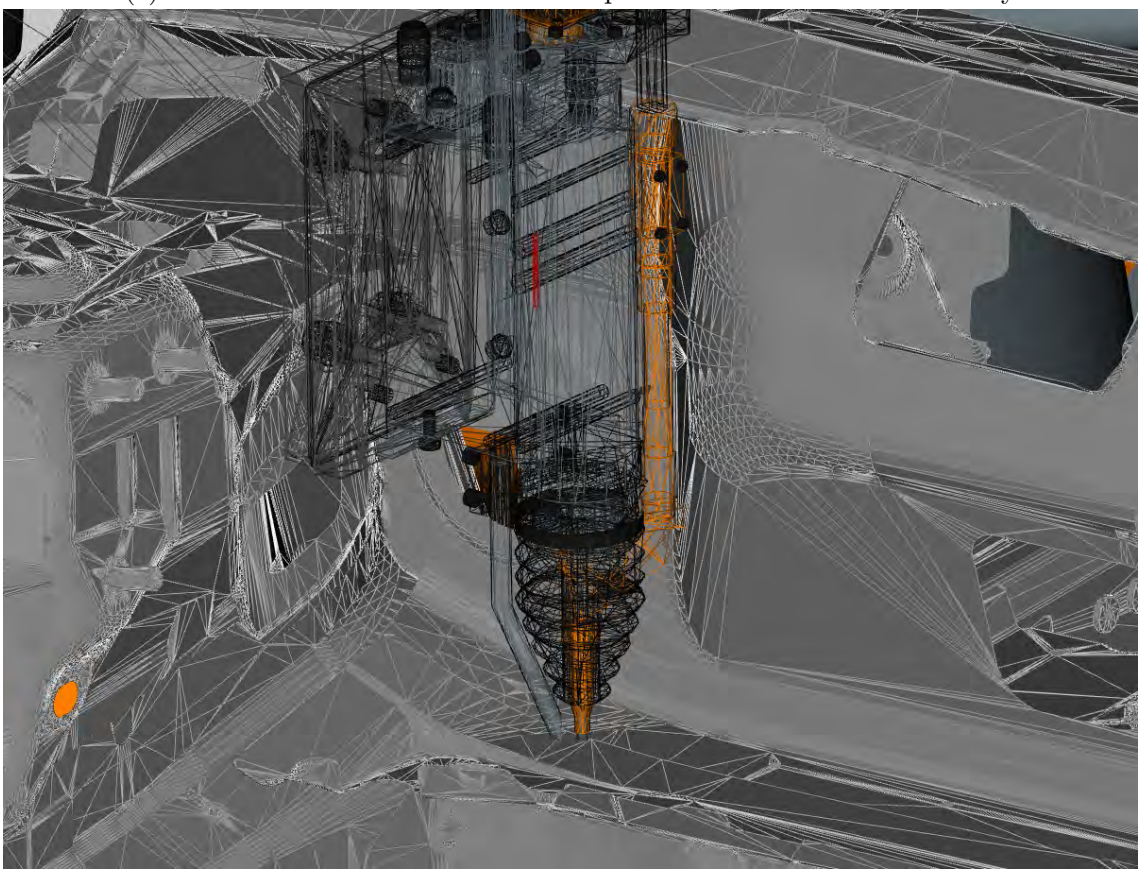


Figure 5.2: Demonstration of triangle resolution of geometries in the test-scene. Geometry by courtesy of Volvo Cars.

5.2.1 Scenarios in a Volvo Cars test scene

Within the Volvo Cars test-scene, several scenarios are tested by varying A, B and the sequence of $\mathcal{C}$-space points $q_1, q_2, \ldots, q_n$. The set of scenarios is:

Play-forward: A play-forward scenario where rigid bodies undergo continuous motion. In particular, robot arms of the scene are animated to perform stud-welding on a car body. A sequence of $\mathcal{C}$-space points $q_1, q_2, \ldots, q_n$, that give a highly coherent scenario, govern robot arm configurations. The set of geometric objects A and B is taken to be the set of all rigid parts of each robot arm and the car body with miscellaneous geometry, respectively. That is to say, $A = \{\text{IRB71}, \text{IRB72}, \text{IRB73}, \text{IRB74}\}$ and $B = \{\text{car}, \text{miscellaneous geometry}\}$.

Sampling-based motion-planning: A medium-coherence scenario where rigid bodies undergo non-continuous motion. A sequence of $\mathcal{C}$-space points $q_1, q_2, \ldots, q_n$, satisfying $\|q_{i-1} - q_i\| < d$ for almost all i where d is some small constant, govern robot arm configurations. This sequence is the sequence of $\mathcal{C}$-space points which the IPS motion-planner generate during motion-planning. The two sets of geometric objects A and B are taken to be a single the set of rigid parts belonging to a single robot arm and the car body with miscellaneous geometry, respectively. In particular, $A = \{IRB72\}$ and $B = \{\text{car}, \text{miscellaneous geometry}\}$.

No-coherence: A non-coherence scenario is taken to be the play-forward scenario but the sequence $q_1, q_2, \ldots, q_n$ is randomly permuted.

Note that detail within time series for the motion-planning scenario and the non-coherent scenario are of little interest. Rather, it is the general behaviour of time series that is interesting. Yet, I provide detailed time series as is. This gives the reader an opportunity to verify that methods behave as expected.

5.3 Minimal separation distances

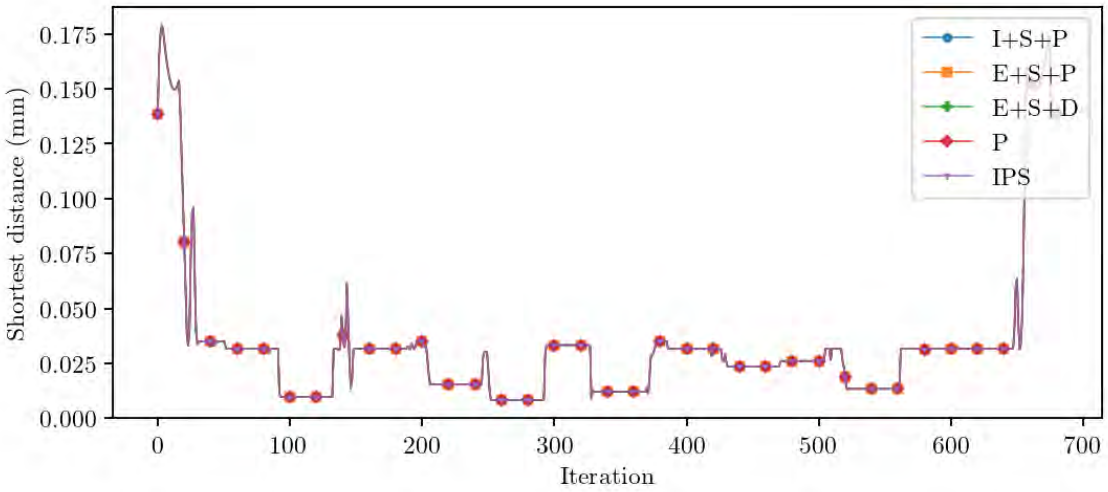
Minimal separation distances, per scenario, are not interesting results in the sense that they do not help towards a comparison of FPQ and IPS. However, minimal separation distances do help towards gaining an intuition of the level of coherency in a scenario. For this reason, minimal separation distances per scenario are shown in Figure 5.3.

In subplot (a) of Figure 5.3 indicate that coherence is indeed high for the play-forward scenario.

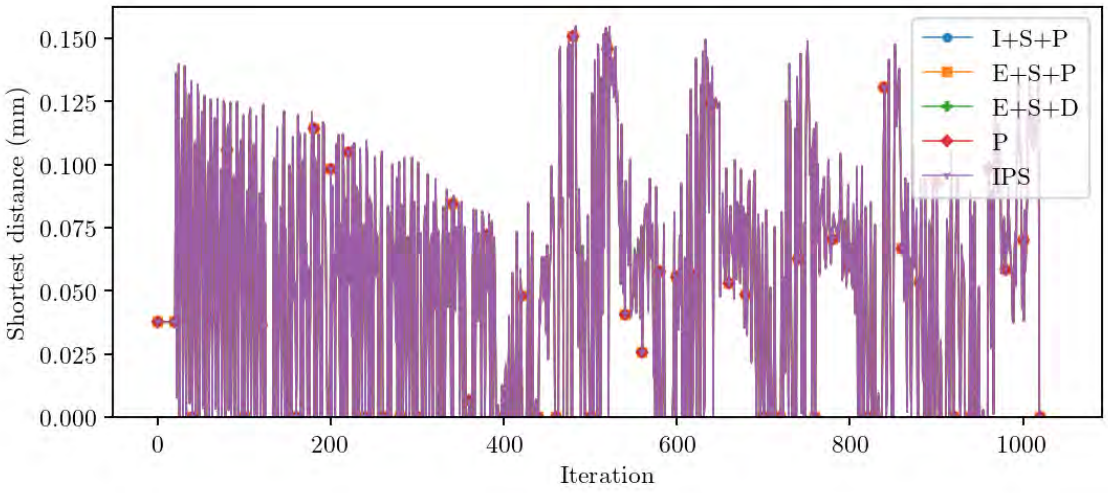
The motion-planning scenario in subplot (b) of Figure 5.3 indicate that a sampling-based motion-planner generate configuration points that causes separation distances to vary wildly, but there are some iteration-intervals for which there is some coherence. Unfortunately, a distance series does not capture a complete behaviour of geometry during a motion-planning scenario. I assure the reader that some level of coherence is observable when visually examining the motion-planning scenario within IPS.

Finally, separation distances of subplot (c) in Figure 5.3 do indicate that coherence is low for the non-coherent scenario.

(a) Minimal separation distances, play-forward scenario



(b) Minimal separation distances, motion-planning scenario



(c) Minimal separation distances, non-coherent scenario

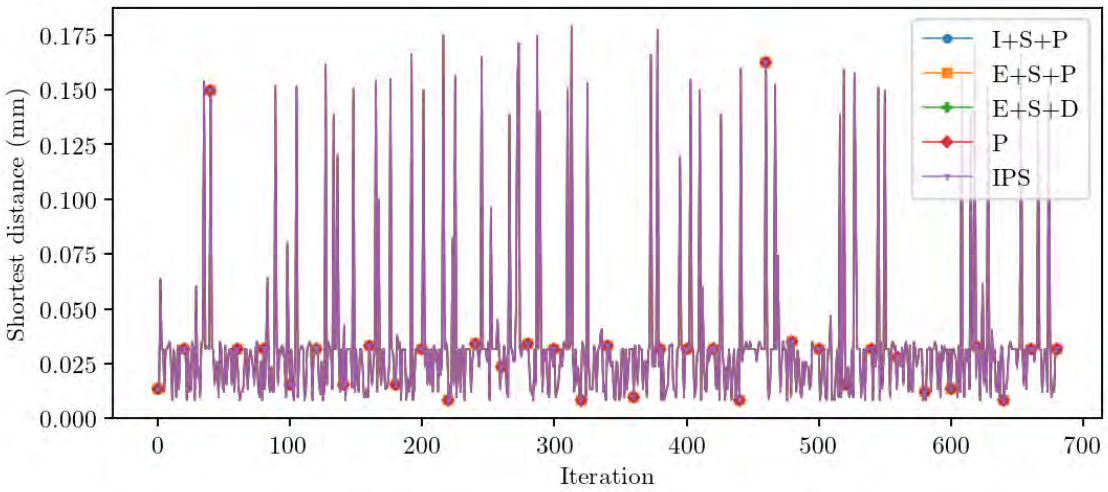


Figure 5.3: Minimal separation distances for the (a) play-forward scenario, (b) motion-planning scenario and the (c) non-coherent scenario. Minimal separation distances are naturally equal for all methods. Note that separation distances jumps more with lower coherency levels.

5.4 Timings

Timings of FPQ variants and IPS are measured on several scenarios. Timing results for each scenario and a subset of variants is shown in Figure 5.4. Timing results for recursive raise strategies is presented later. A clipped version of timing results for the play-forward scenario is shown in Figure 5.5.

The fastest reported method for the play-forward scenario is **E+S+D**. This indicates that overhead associated with a front is smaller than the gains of using a front, if a scenario is coherent.

Observe that **P** is the fastest reported method for the motion-planning scenario and the non-coherent scenario, but it is the slowest reported method for the play-forward scenario. This indicates that PDS is improves upon depth-first search (which is used by IPS) when coherency is low, but that it does not improve upon a depth-first search when coherency is high.

Compare **I+S+P** with **E+S+P** and conclude that explicit BVTTs are capable of reducing run-times, as shown by subfigures (a) and (b) of Figure 5.4. This is expected — Section 3.4 "Efficient sibling-raise with explicit BVTTs" presents a raise routine Algorithm 9 "Raising (explicit BVTTs)" that is only compatible with explicit BVTTs. Said raise routine is capable of raising any two siblings without using a, possibly inefficient, hash map H. Recall that H is used with implicit BVTTs in Algorithm 5 "Raising". However, **E+S+P** is slower than **I+S+P** for the non-coherent scenario. This is further discussed in Section 6.4 "Raise times and memory usage with implicit and explicit BVTTs".

Finally, there are observable spikes in timing results (and in other figures as well). This is due to front resetting. Later, in Section 5.11 "Disabling front resetting", we shall see that there are no spikes if front resetting is disabled.

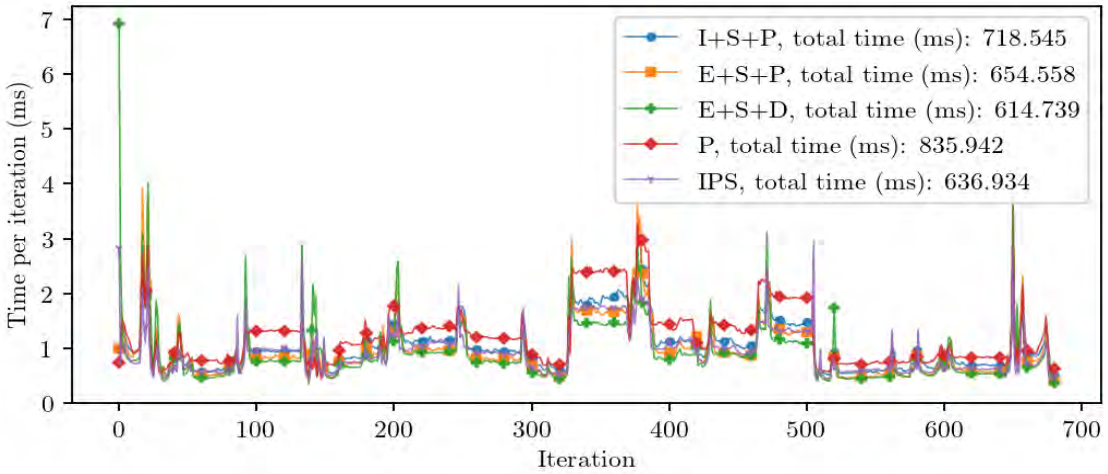
5.5 Bounding volume and triangle distance computations

A proposed benefit of FPQ is that it may perform fewer BVTT node distance computations, which is a major bottleneck of traditional proximity querying algorithms and in turn a major bottleneck of sampling-based motion-planners. A key measurement is the total number of BVTT node distance computations, i.e. RSS tests or triangle tests.

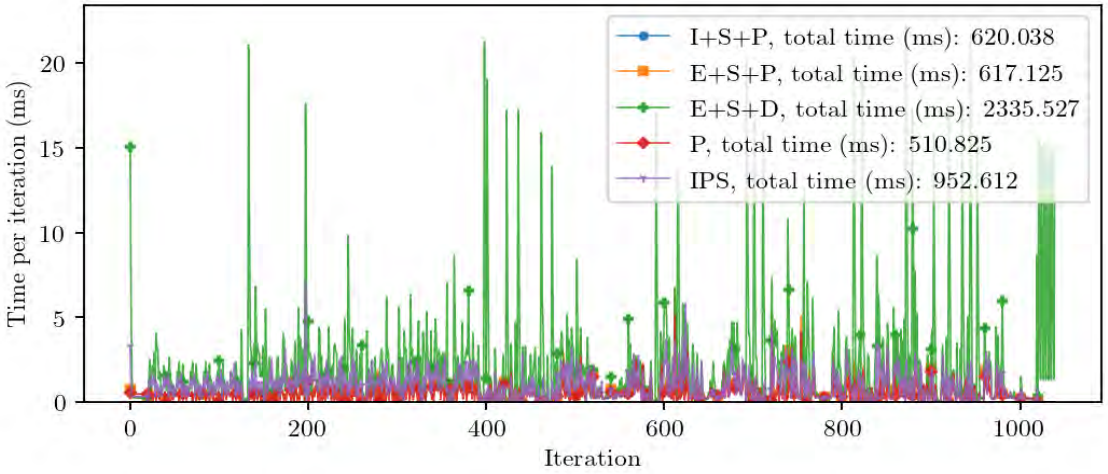
RSS test results are presented in Figure 5.6. The time series show the number of RSS tests per iteration. The total number of RSS tests is shown in the upper right corner for each scenario. Triangle test results are presented in Figure 5.7.

Compare the general shape of all RSS series in Figure 5.6 with corresponding time series previously presented in Figure 5.4. The RSS series and the time series

(a) Timings, play-forward scenario



(b) Timings, motion-planning scenario



(c) Timings, non-coherent scenario

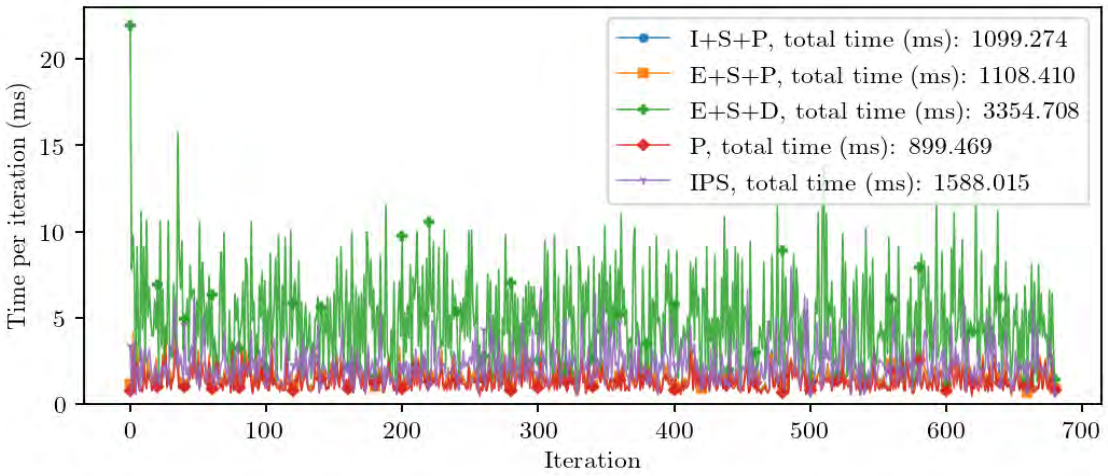


Figure 5.4: Timing results for the (a) play-forward scenario, (b) motion-planning scenario and the (c) non-coherent scenario. Timing results per iteration is shown in the time-series. Total times are shown in the upper-right corners. Note that **E+S+D** is slightly faster than IPS in the play-forward scenario but that **P** is significantly faster than IPS for the other two scenarios.

(a) Timings, play-forward scenario (clipped)

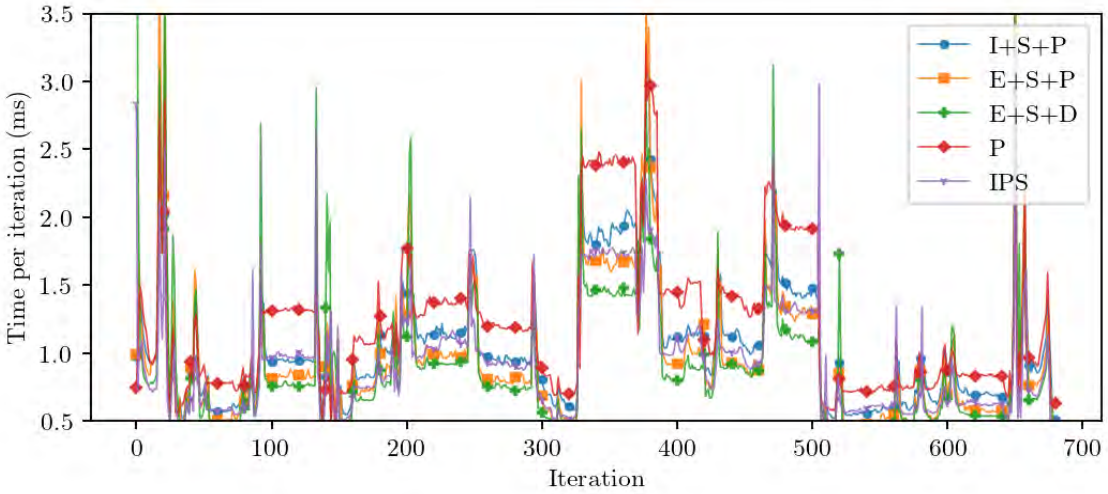


Figure 5.5: A clipped version of subplot (a) of Figure 5.4

resemble each other. While this does not conclude that run-times are dominated by RSS tests, it does indeed indicate that run-times are dominated by RSS tests.

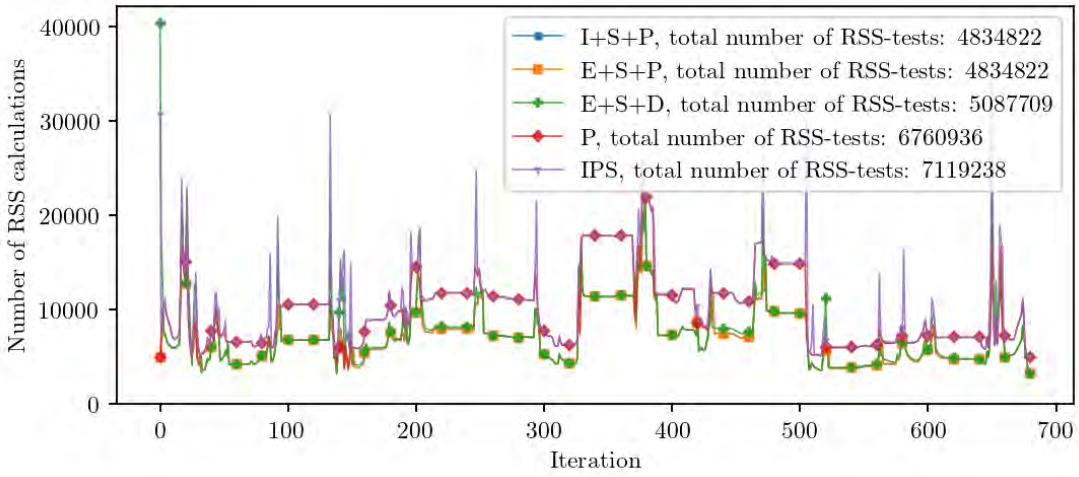
5.6 Leaf front measurements

Section 3.2.2 "Initializing an upper-bound on a separation distance via a leaf front" presents a method for initializing ϵ. This method may incur additional overhead due to a increase in the number of RSS tests or triangle tests. Figure 5.8 show how much time is spent traversing the leaf front during initialization of ϵ, for each iteration. Timings include the time spent computing triangle tests during traversal. Figure 5.9 show how the size of the leaf per iteration.

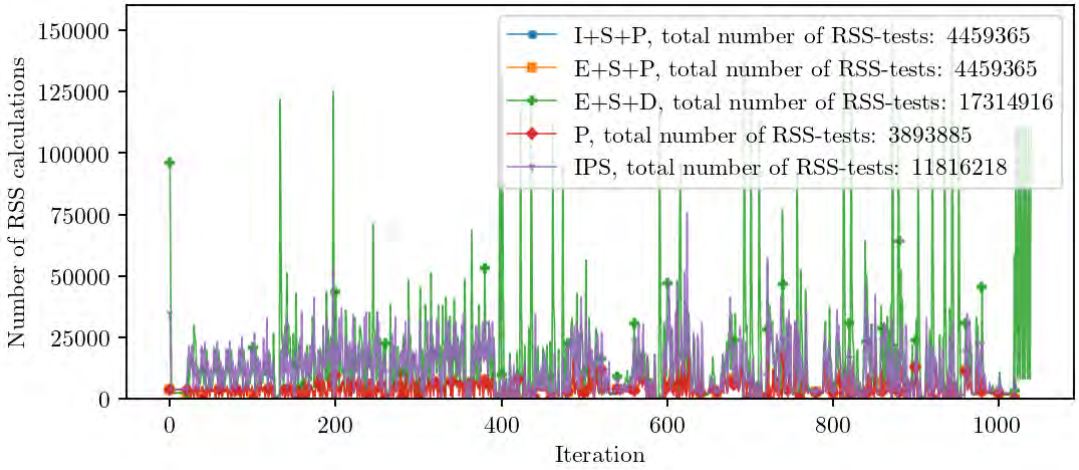
Leaf front timings shown in Figure 5.8 demonstrate that leaf front traversal, for all methods in all scenarios, are fast in comparison to corresponding total run-times, shown in Figure 5.4. This is not surprising since the number of triangle tests, shown in Figure 5.7, are few in comparison to the number of RSS tests shown in Figure 5.6.

Further, the number of triangle tests shown in Figure 5.7 demonstrate that FPQ does *not* trade RSS tests for more triangle tests since the number of triangle tests is lower for all FPQ methods, in comparison to IPS, with the exception of **E+S+D** under low-coherency scenarios. The **E+S+D** exception is further discussed in Section 6.1 "Main results".

(a) RSS tests, play-forward scenario



(b) RSS tests, motion-planning scenario



(c) RSS tests, non-coherent scenario

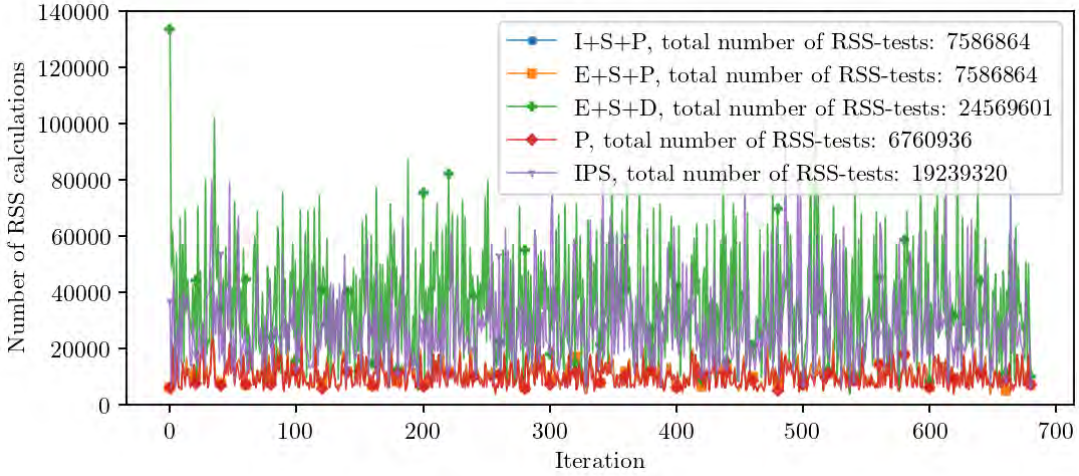
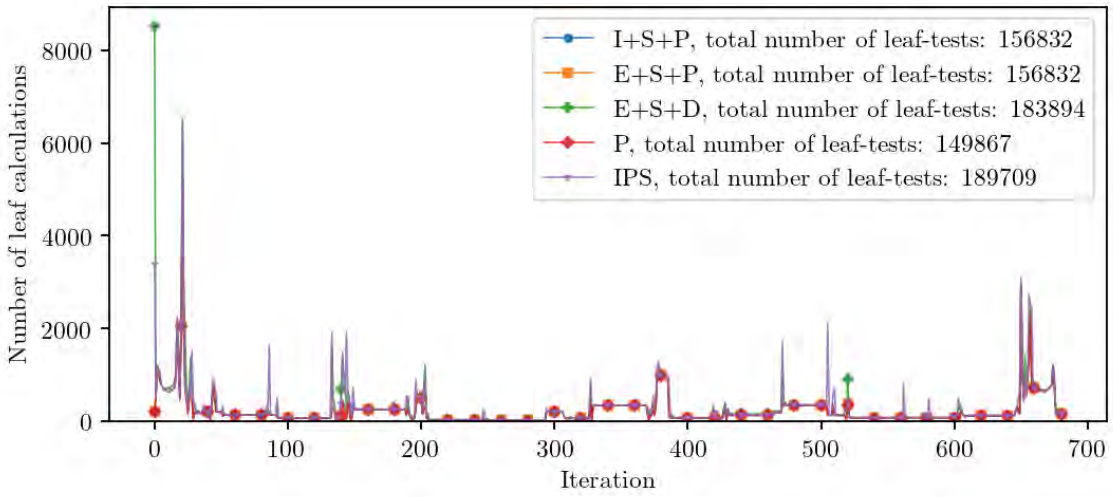
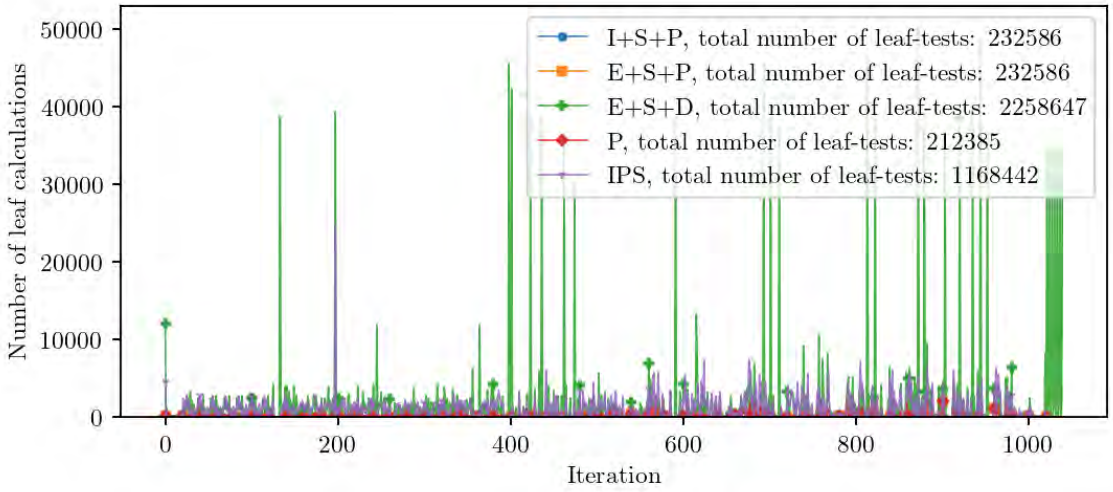


Figure 5.6: Number of RSS tests for the (a) play-forward scenario, (b) motion-planning scenario and the (c) non-coherent scenario. The number of RSS tests per iteration is shown in the time-series. Total number of RSS tests are shown in the upper-right corners. Note that these curves resembles timing results in Figure 5.4, which could mean that RSS tests are indeed bottlenecks of separation distance computation.

(a) Triangle tests, play-forward scenario



(b) Triangle tests, motion-planning scenario



(c) Triangle tests, non-coherent scenario

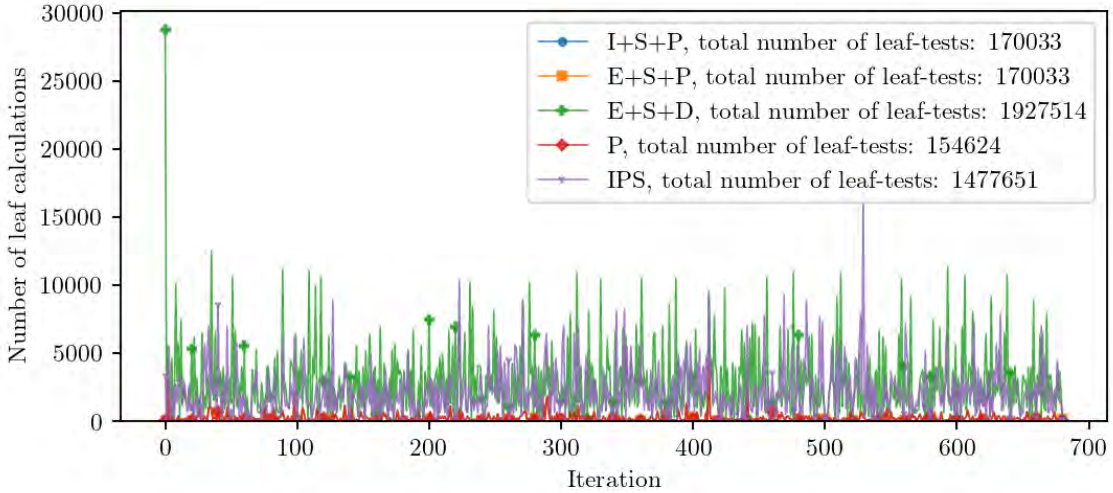
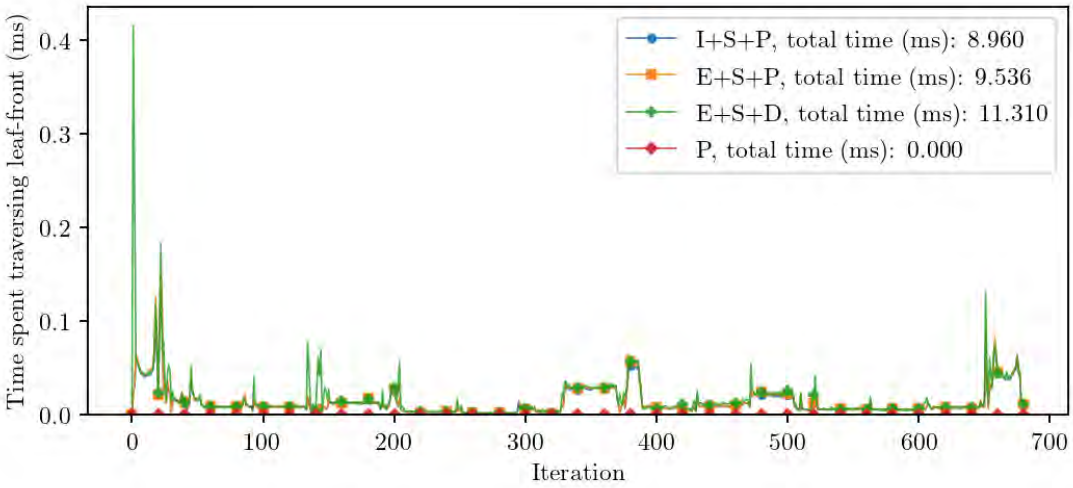
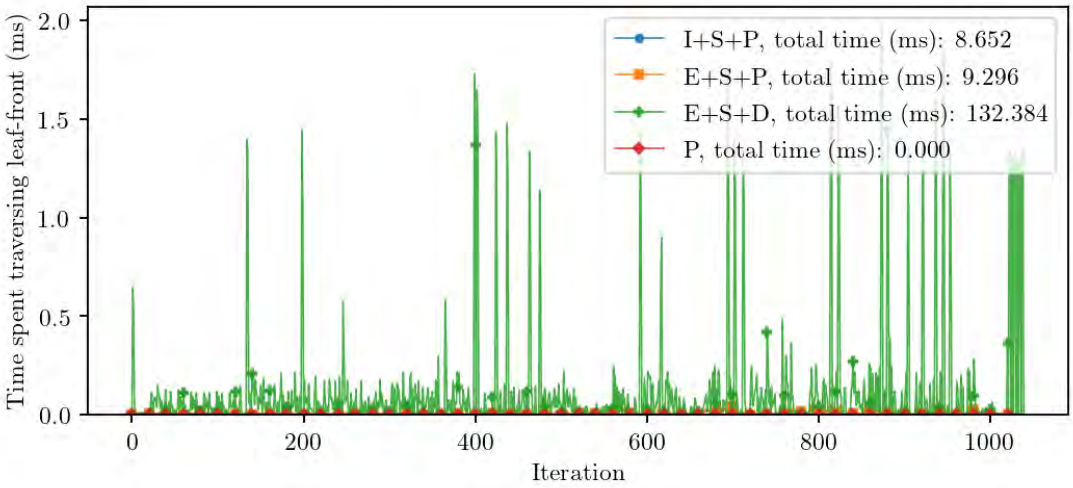


Figure 5.7: Number of triangle tests for the play-forward scenario, motion-planning scenario and the non-coherent scenario. The number of triangle tests per iteration is shown in the time-series. Total times are shown in the upper-right corners. Note that the number of triangle tests per variant is always small in comparison to the number of RSS tests shown in Figure 5.6

(a) Leaf front timings, play-forward scenario



(b) Leaf front timings, motion-planning scenario



(c) Leaf front timings, non-coherent scenario

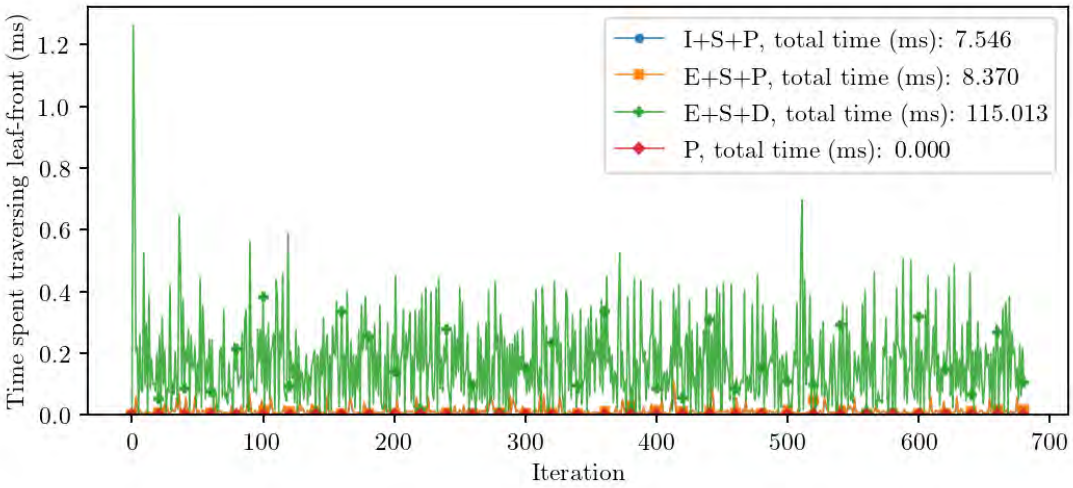
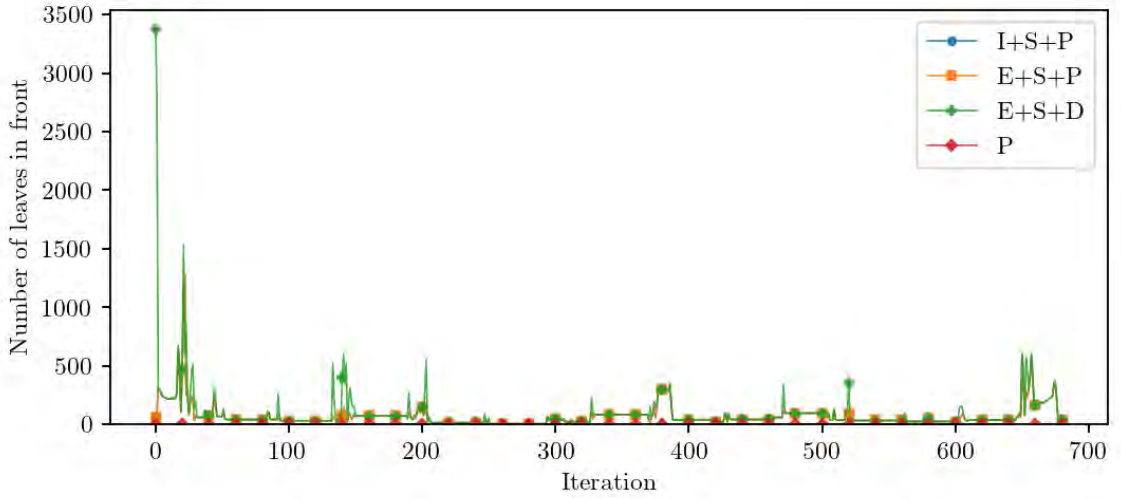
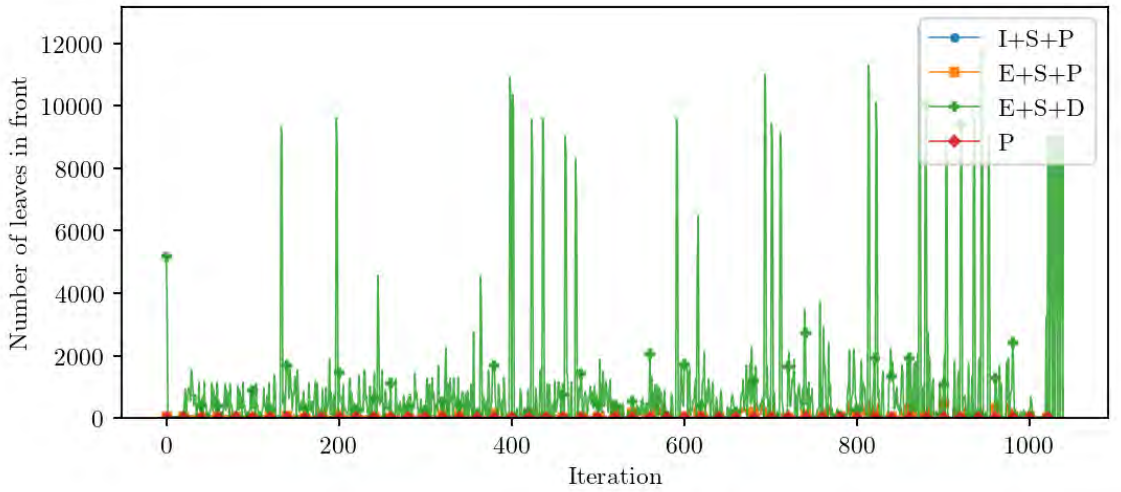


Figure 5.8: Time spent traversing leaf fronts, computing triangle tests, when initializing ϵ for the (a) play-forward scenario, (b) motion-planning scenario and the (c) non-coherent scenario. Time spent per iteration is shown in the time-series. Total times are shown in the upper-right corners. This means that time spent initializing ϵ small in comparison to total timings shown in Figure 5.4

(a) Size of leaf front, play-forward scenario



(b) Size of leaf front, motion-planning scenario



(c) Size of leaf front, non-coherent scenario

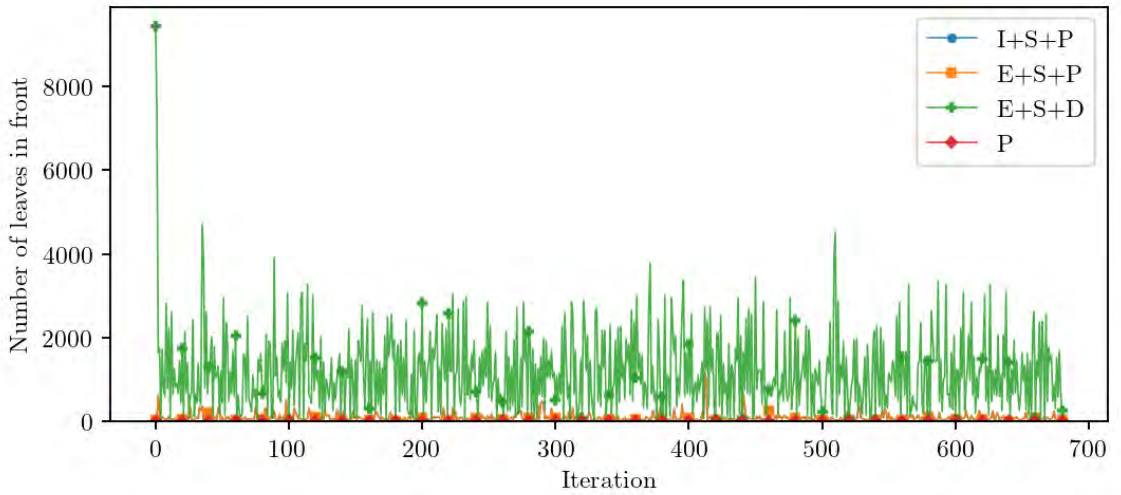


Figure 5.9: Number of elements within leaf fronts, per iteration, for the (a) play-forward scenario, (b) motion-planning scenario and the (c) non-coherent scenario. Note that depth-first causes leaf fronts to become large in comparison to priority directed search.

Finally, leaf front sizes shown in Figure 5.9 indicate that a depth first search is more likely to traverse BVTT leaves than a priority directed search. This is expected since a PDS traverse minimal forests of BVTTs by Theorem 1, which is not certain for depth first search — PDS traverses few leaves regardless of coherency while depth first search degenerates with decreased coherency levels.

5.7 Minimal forests has fewer nodes than sets of minimal BVTTs

A relevant comparison with respect to to Section 3.3.4 "Analysis of FPQ vs GFT" is that of traversing a minimal forest of BVTTs in contrast to traversing a forest of minimal BVTTs. Table 5.1 is generated by traversing a forest BVTTs using solely Algorithm 7 "Sprouting".

Table 5.1: A comparison of traversing several minimal BVTTs against a minimal forest of BVTTs. Statics are used to denote the car body and the static geometry. The measurement is carried out on the play-forward scenario, using the **P** variant of FPQ (only PDS without FT) on a forest of BVTTs.

Robot arm identifier	Number of RSS tests
IRB71 against statics	4304365
IRB72 against statics	3461201
IRB73 against statics	3519009
IRB74 against statics	3298305
Total amount of RSS tests	14582880
All IRBs against statics	6760936

As is seen from Table 5.1, the number of RSSs tests when traversing a minimal forest of BVTTs is roughly half of that of traversing four minimal BVTTs for the play-forward scenario. Results in Table 5.1 motivates FPQ — performing a proximity on a whole forest of BVTTs in concert, in comparison to treating BVTTs independently, is clearly beneficial with respect to the number of RSSs tests (since an upper-bound ϵ on a separation distance δ is "shared" within the forest).

Finally, there is no reason to suspect that this ratio is a function of scene-coherency, due to Theorem 1, which is why no measurements on the motion-planning scenario nor the non-coherent scenario is presented.

5.8 Front size and memory usage

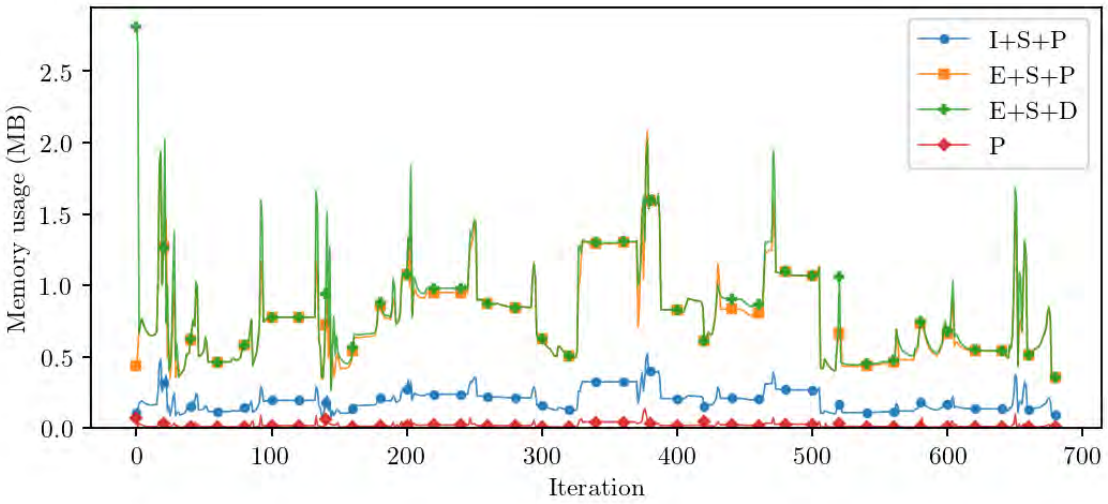
A large front causes high memory usage. Memory usage is a concern with FT, GFT and especially FPQ. Two techniques ameliorates memory usage of FPQ:

- Implicit BVTTs
- PDS

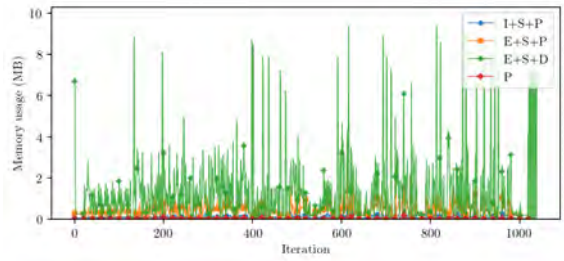
Figure 5.10 shows memory usage per iteration for **I+S+P**, **E+S+P** and **E+S+D**. Memory usage is approximated. For implicit BVTTs, memory is approximated by multiplying the bytesize of an implicit BVTT node with the front size. For explicit BVTTs, memory is approximated by multiplying the bytesize of an explicit BVTT node with twice the front size.[2]. Front sizes for each method and for each scene is shown in Figure 5.11.

Note that memory usage of **P** is not defined in terms of a front size, since **P** does not utilize a front at all. Instead, memory usage of **P** is approximated by multiplying the bytesize of an implicit BVTT node with the largest size of the priority attained during an iteration.

(a) Memory usage per iteration, play-forward scenario



(b) Memory usage per iteration, motion-planning scenario



(c) Memory usage per iteration, non-coherent scenario

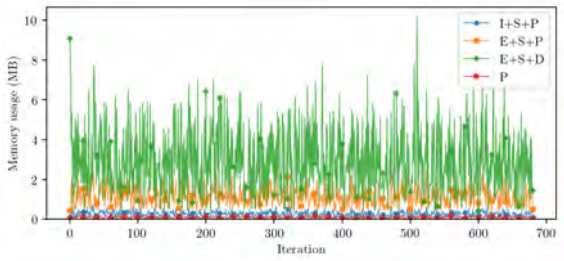
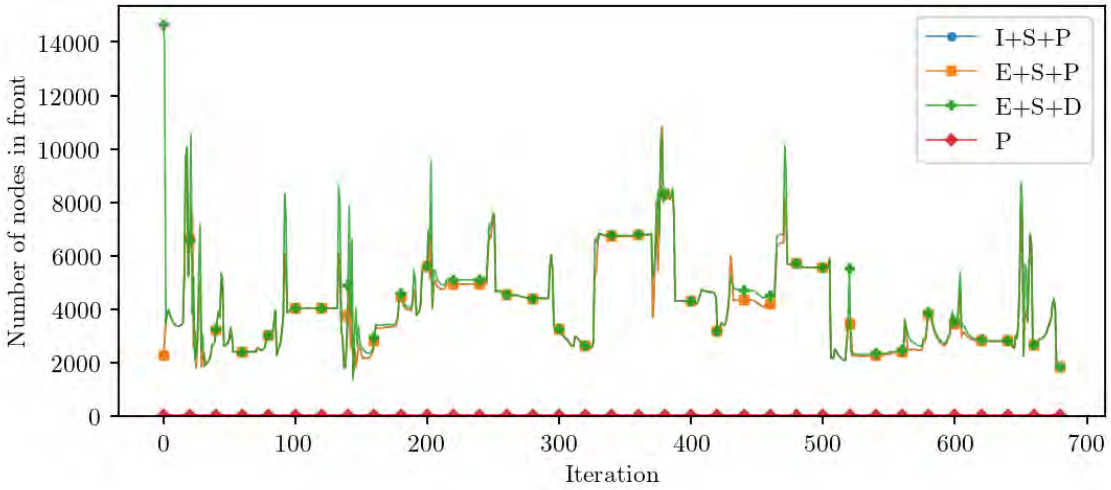


Figure 5.10: Approximated memory usage per iteration for the play-forward scenario, motion-planning scenario and the non-coherent scenario. Note that **P** has the smallest memory usage. Note that out of the variants that use fronts, **I+P+S** has the smallest memory usage.

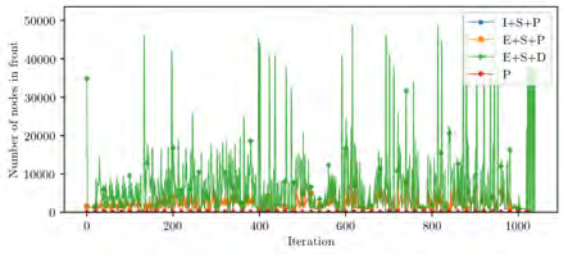
Figure 5.10 shows that memory usage does not exceed 10 MB for any method in any scenario. Any concerns that FT is intractable due to memory issues can be discarded (unless triangle counts become sufficiently large).

[2]Recall that BVTTs are binary, so the total number of allocated BVTTs nodes is exactly $2 \cdot |F| - 1$

(a) Front size per iteration, play-forward scenario



(b) Front size per iteration, motion-planning scenario



(c) Front size per iteration, non-coherent scenario

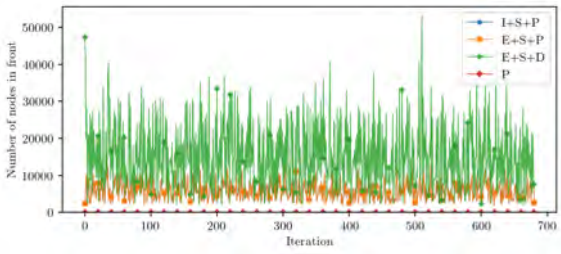


Figure 5.11: Front sizes per iteration for the play-forward scenario, motion-planning scenario and the non-coherent scenario. Note that these curves are proportional to corresponding memory curves in Figure 5.10. Note that priority directed search variants have smaller front sizes.

I+S+P front size series are not visible in Figure 5.11 since they overlap exactly with **E+S+P** front size series. This is expected since the two methods are equivalent with respect to members of a front and traversal. However, they differ with respect to memory usage. **I+S+P** need not store a BVTT in memory and hence it has lower memory usage than **E+S+D**, although it is true that their front sizes are equivalent.

Lastly, fronts become large with depth-first search when coherence is low, as indicated by **E+S+D** curves of subplots (b) and (c) in Figure 5.11. This motivates that if a front is used in a low coherency scenario, then PDS can reduce the size of the front.

5.9 Single raise and recursive raise

A sibling raise-operator can raise nodes one level or several levels, as described in Section 3.5 "Recursively raising siblings". I present timing results and RSS-test results for **E+S+P**, **E+S+D** with **E+R+P** and **E+S+D**. Timing results and the number of RSS tests can be seen in Figure 5.12 and Figure 5.13, respectively.

Timings results in subplot (a) of Figure 5.12 show that a recursive raise algorithm can be beneficial in comparison to a single level raise algorithm. However, this claim does not carry over to depth first variants for the motion-planning scenario and the non-coherent scenario, for which a recursive raise algorithm degrades performance with respect to time. An optimal front varies rapidly for scenarios with low coherency, so it becomes increasingly difficult to approximate an optimal front. For example, if a front is recursively raised on iteration i such that it approaches the optimal front on iteration i, then the lack of coherence makes the front less useful on iteration $i+1$ — the additional overhead caused by a recursive raise does not pay off as much for incoherent scenarios. Finally, this is not observed when using PDS since there is no great need for raisal to begin with since PDS traverses minimal forests of BVTTs (PDS does not "over-sprout" as much as a depth-first search).

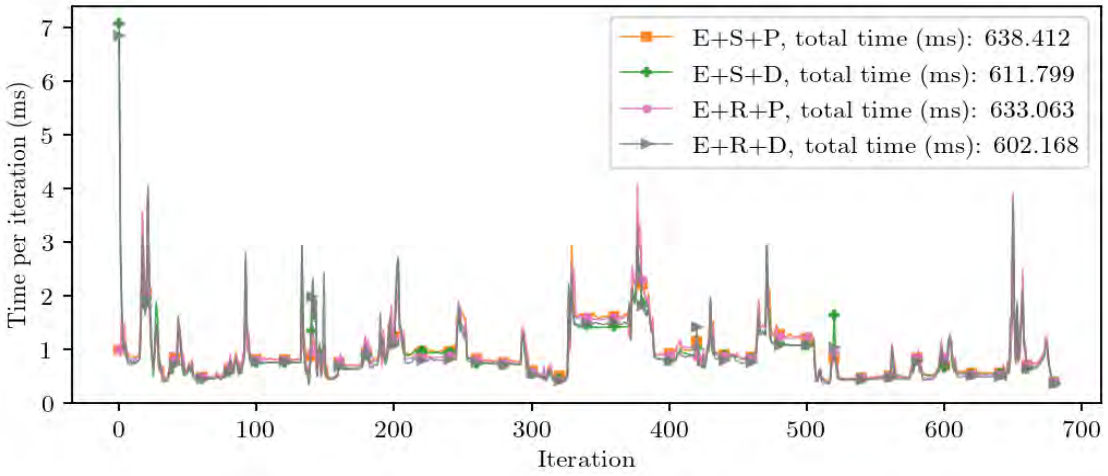
It is of interest to compare differences between PDS and depth-first traversal, since depth-first traversal may sprout below that of an optimal front (over-sprout), causing more raise-operations. Raise times for **E+S+P**, **E+S+D**, **E+R+P** and **E+S+D** are presented in Figure 5.14.

Note that raise times are lower for all recursive variants **R** in comparison to their single level raise variants **S**. I discuss and explain in Section 6.5 "Recursive raisal decrease raise times" why total raise times are lowered with recursive raisal, in comparsion to single level raisal.

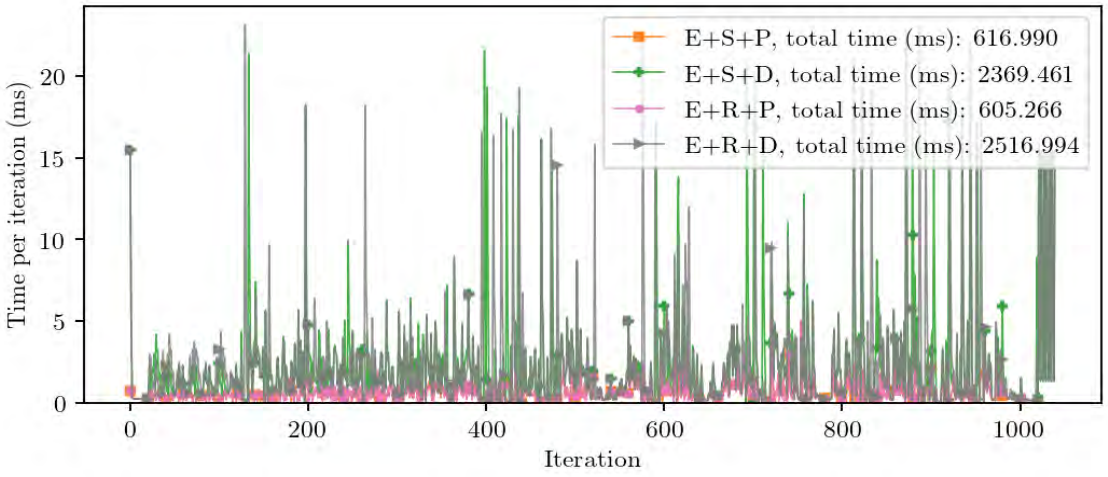
Raise-operators may raise an approximate front F above an optimal front F^*, causing more sprout-operations by F. Sprout times are presented in Figure 5.15.

A recursive raise algorithm can "over-raise", i.e. raise nodes above an optimal front (more so than a single level raise algorithm can!). This causes sprouting algorithms to work more. This may explain why sprout times, shown in Figure 5.15, are consistently higher for FPQ variants using a recursive raisal algorithm than FPQ variants using a single level raisal algorithm.

(a) Time spent per iteration, play-forward scenario



(b) Time spent per iteration, motion-planning scenario



(c) Time spent per iteration, non-coherent scenario

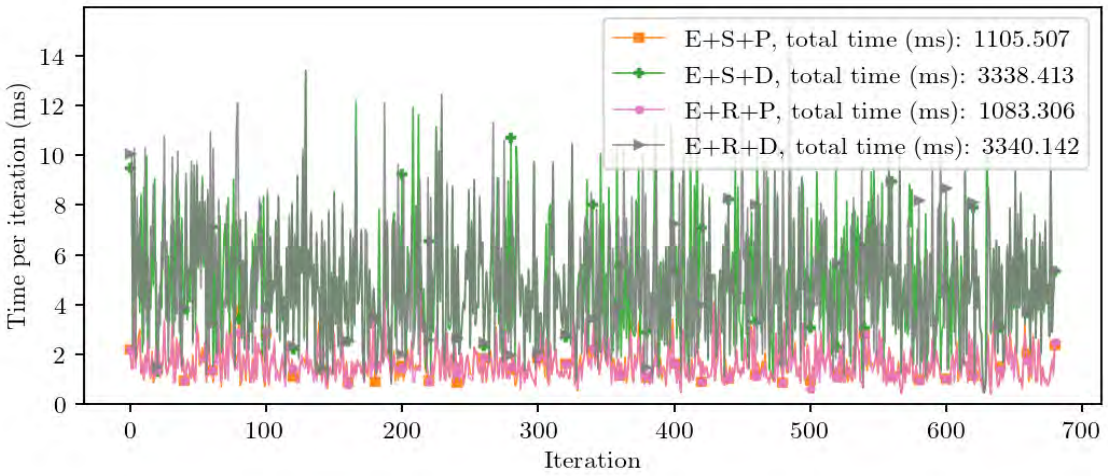
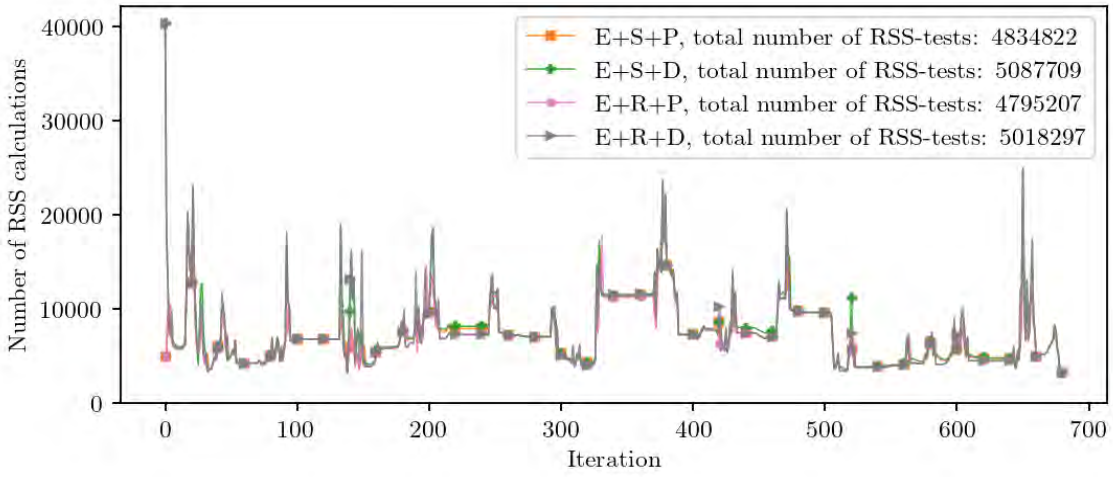
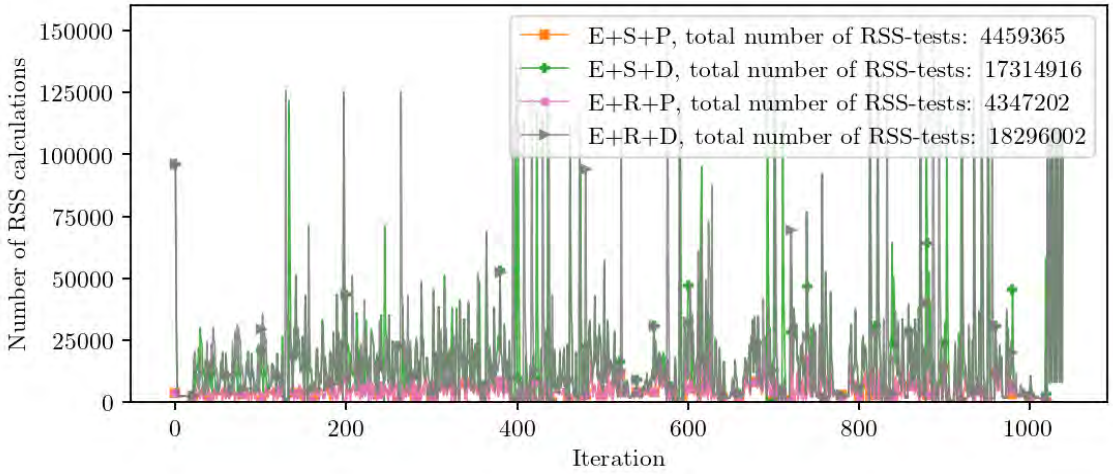


Figure 5.12: Timing results comparing a single level raise operator with a recursive raise operator, using either PDS or depth-first search, over all three scenarios. Note that **E+R+D** attains the smallest running time out of the other recursive variants.

(a) Single raise vs recursive raise RSS tests, play-forward scenario



(b) Single raise vs recursive raise RSS tests, motion-planning scenario



(c) Single raise vs recursive raise RSS tests, non-coherent scenario

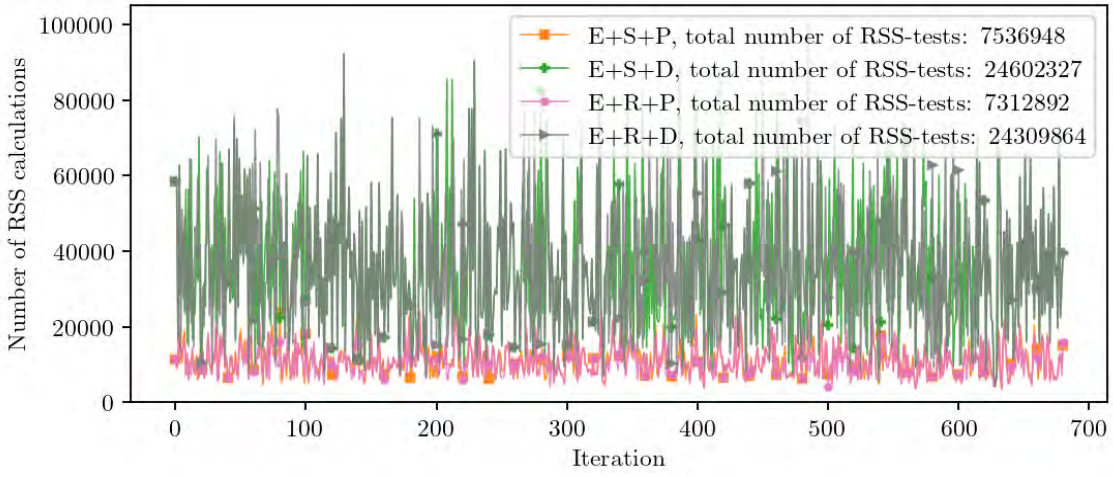
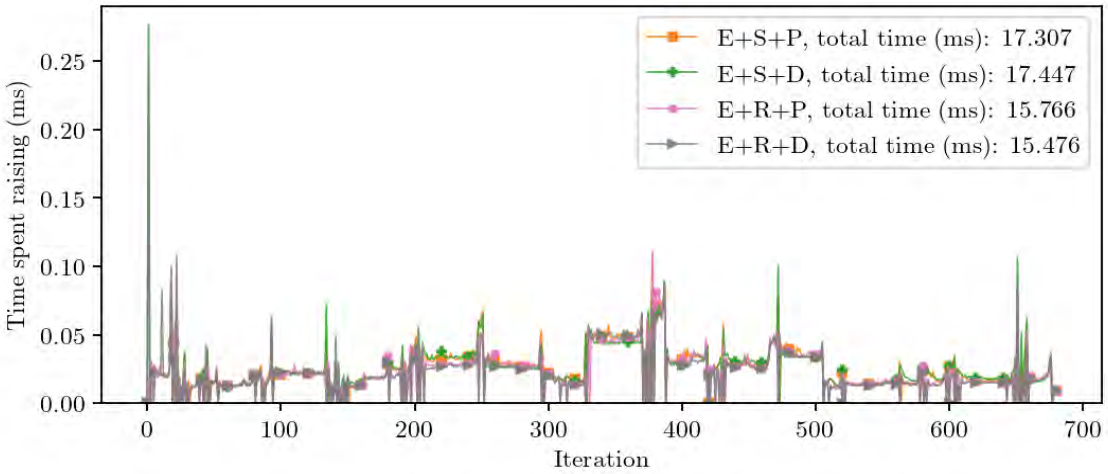
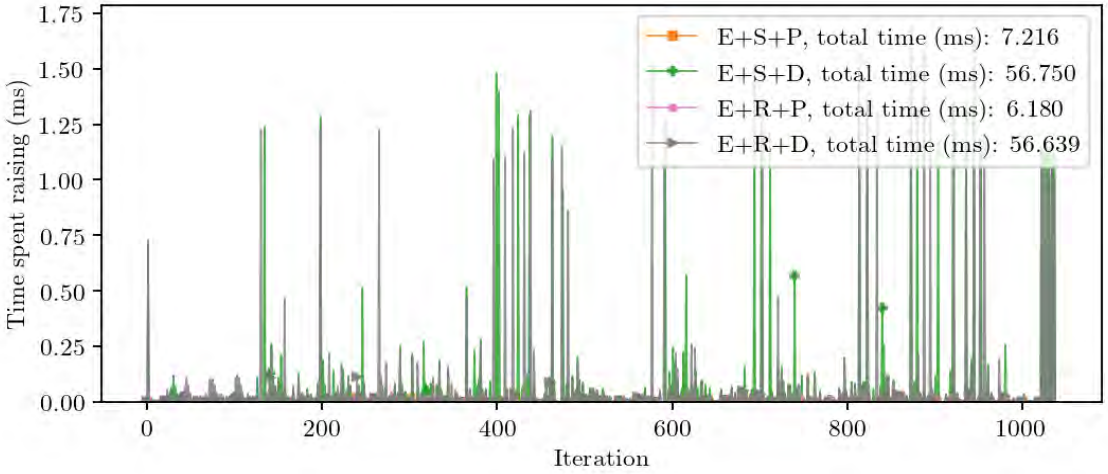


Figure 5.13: Number of RSS tests. These results compare a single level raise operator with a recursive raise operator, using either PDS or depth-first search, over all three scenarios. Note that no variant that uses front tracking performs fewer RSS tests than **E+R+D** in any scenario.

(a) Single raise vs recursive raise raise-times, play-forward scenario



(b) Single raise vs recursive raise raise-times, motion-planning scenario



(c) Single raise vs recursive raise raise-times, non-coherent scenario

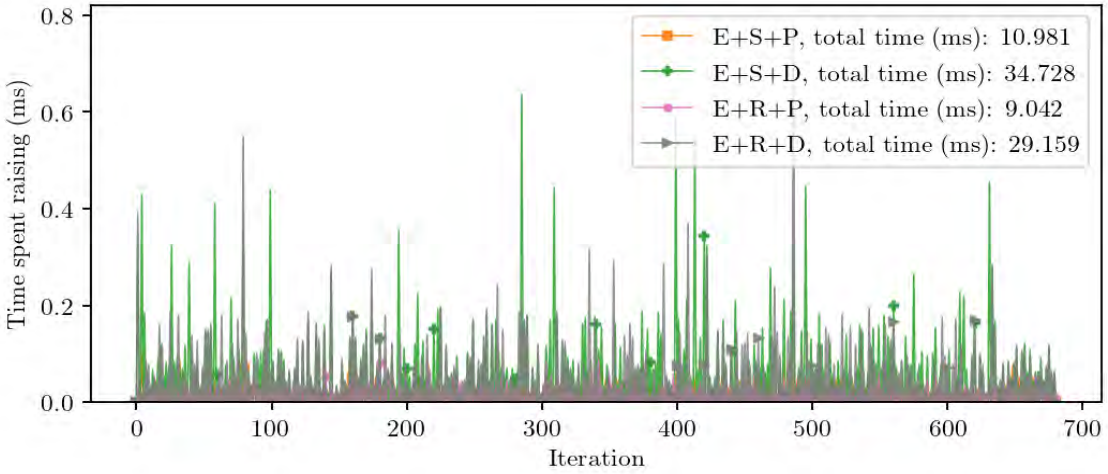
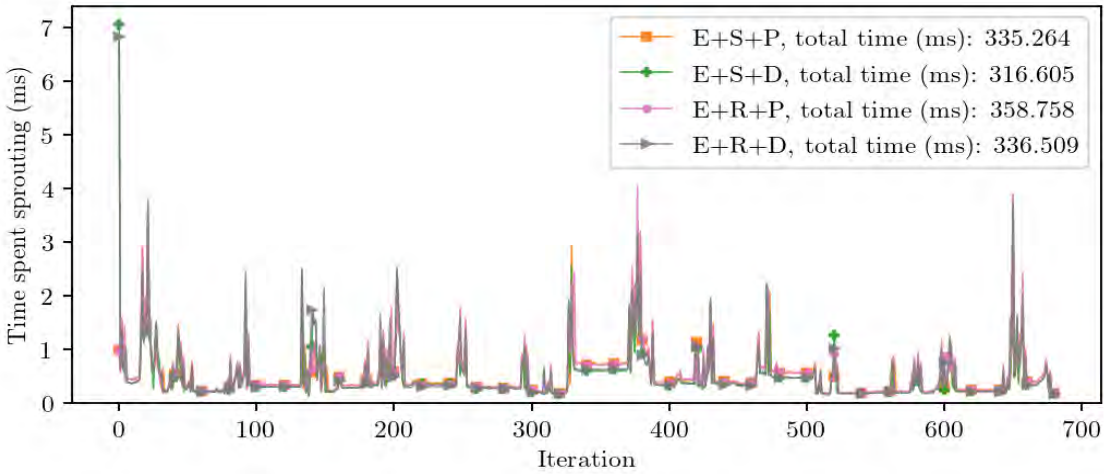
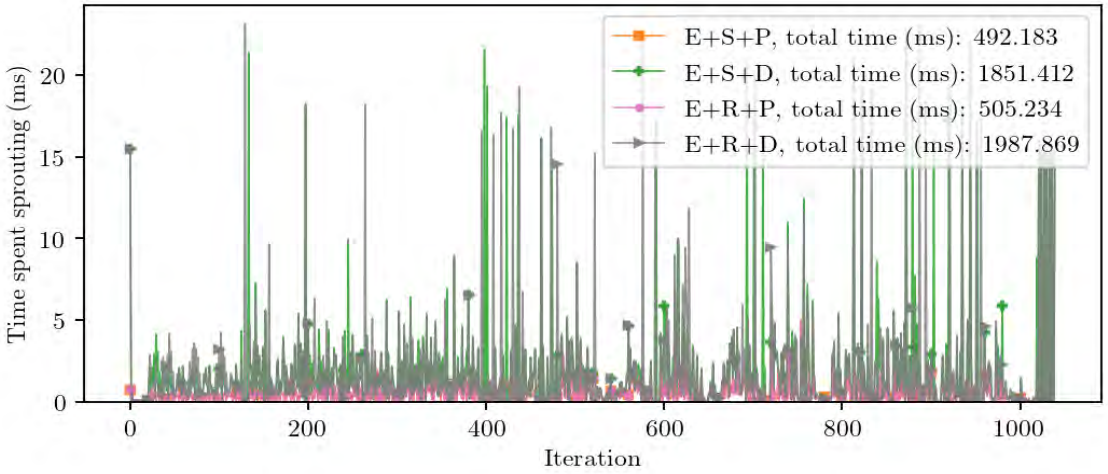


Figure 5.14: Raise-times per iteration, for each method, are shown in the series. Total raise-time per method is shown in upper-right corners. These results compare a single level raise operator with a recursive raise operator, using either PDS or depth-first search, over all three scenarios. Note that recursive raisal is consistently faster than single level raisal with respect to raise times.

(a) Single raise vs recursive raise sprout-times, play-forward scenario



(b) Single raise vs recursive raise sprout-times, motion-planning scenario



(c) Single raise vs recursive raise sprout-times, non-coherent scenario

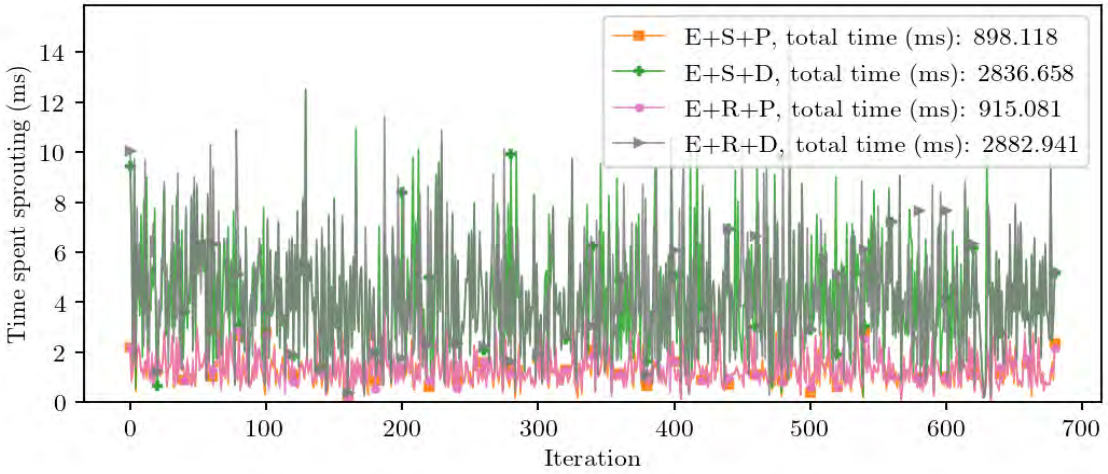


Figure 5.15: Sprout-times per iteration, for each method, are shown in the series. Total sprout-time per method is shown in upper-right corners. These results compare a single level raise operator with a recursive raise operator, using either PDS or depth-first search, over all three scenarios. Note that recursive raisal is consistently slower than single level raisal with respect to sprout times.

5.10 Raising and sprouting on implicit and explicit BVTTs

A key motivator for explicit BVTT is that they enable storage of persistent data which in turn allows for $\mathcal{O}(n)$ raisal as opposed to average $\mathcal{O}(n)$ raisal, as stated in Section 3.4 "Efficient sibling-raise with explicit BVTTs".

Time spent raising using **I+S+P**, **E+S+P** and **E+S+D**, for all levels of coherency, is shown in Figure 5.16. Time spent raising is captured by measuring time spent within raise routines. Time spent sprouting using **I+S+P**, **E+S+P** and **E+S+D**, for all levels of coherency, is shown in Figure 5.17. Time spent sprouting is captures by measuring the time spent in Algorithm 7 "Sprouting" (including RSS tests).

Raise times shown in Figure 5.16 show that explicit BVTTs allows for lower raise times, as anticipated due to the efficient raise algorithm for explicit BVTTs presented in Section 3.4 "Efficient sibling-raise with explicit BVTTs". However, sprout times are increased with explicit BVTTs, as shown by Figure 5.17. This is discussed in Section 6.4 "Raise times and memory usage with implicit and explicit BVTTs".

5.11 Disabling front resetting

The impact of front resetting can be discussed if there are results for which front resetting is activated and if there are results for which front resetting is inactivated. Until now, all results of Chapter 5 "Results" are generated with front resetting enabled. This section presents results for which front resetting has been disabled.

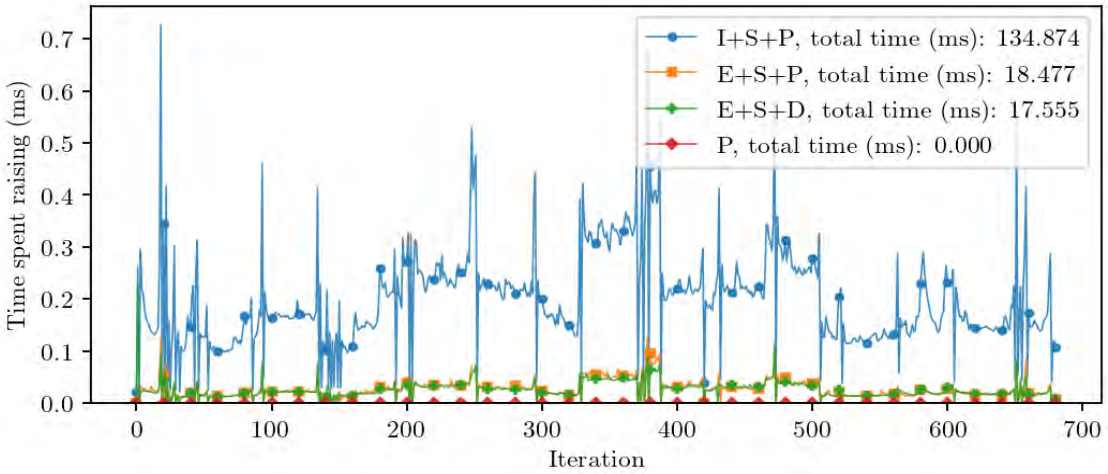
Figure 5.18 shows timing results, Figure 5.19 shows the number of RSS tests, Figure 5.20 shows the number of triangle tests and Figure 5.21 shows front sizes.

Compare timing results in subplot (a) of Figure 5.18 have a half-life characteristic in comparison to the main timing results obtained with reset enabled in Figure 5.4. This indicates that a front eliminates iteration intervals where half-life characteristics emerge, as intended.

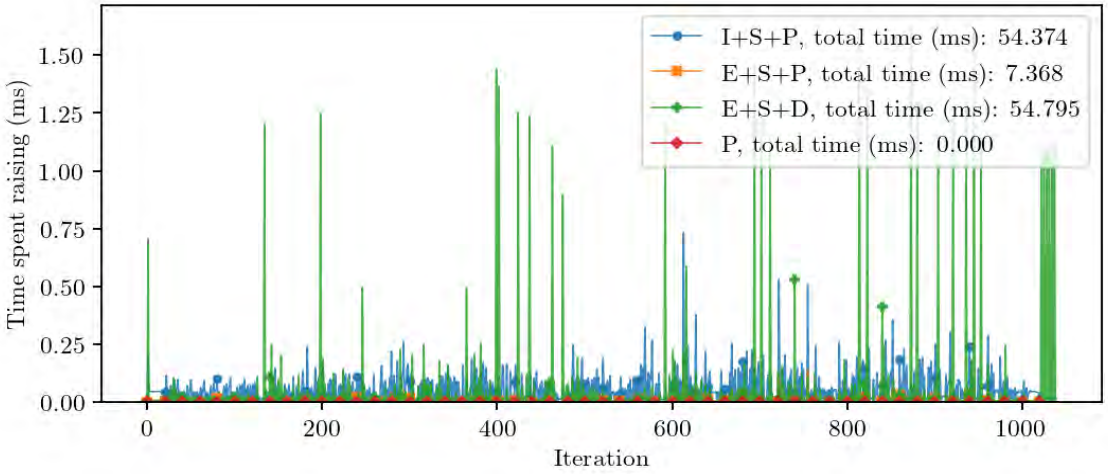
By comparing the number of RSS *without* front resetting, shown in Figure 5.19, with the number of RSS tests *with* front resetting, shown in Figure 5.6, we see that all methods for all scenarios perform better (with respect to the number of RSS tests) when front tracking is enabled — except for depth-first variants. I will not provide any in-depth explanation to this observation. A brief explanation is that front resetting triggers too often for depth-first variants if coherency is low, causing traversal to visit large BVTTs.

This brief explanation is supported by comparing the number of triangle tests without front resetting, shown in Figure 5.20, with the number of triangle tests with front resetting, shown in Figure 5.7. The number of triangle tests in the motion-planning scenario and the non-coherent scenario is higher, for **E+S+D**, when front resetting is enabled. A front reset replaces a front F with the initial set of BVTT roots. There is no guarantee that an ordering of BVTTs is beneficial after a reset and there is no guarantee that a leaf-front is relevant for low coherency scenarios. Hence, it is possible that a large portion of several BVTTs are visited before a small

(a) Time spent raising, play-forward scenario



(b) Time spent raising, motion-planning scenario



(c) Time spent raising, non-coherent scenario

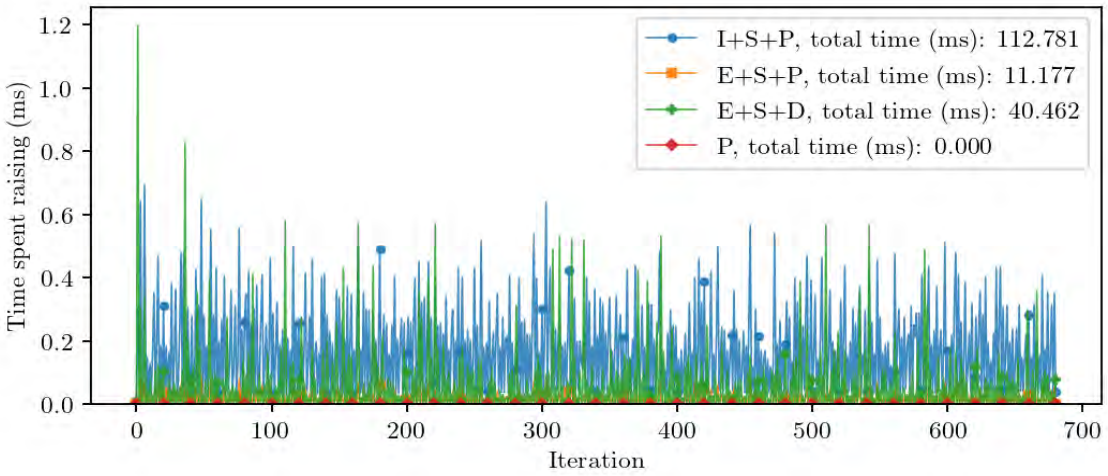
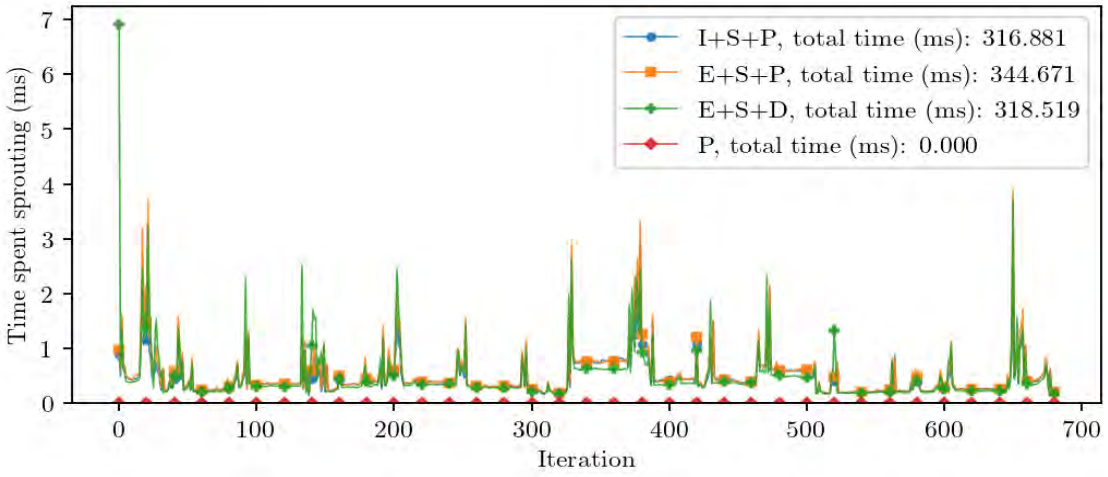
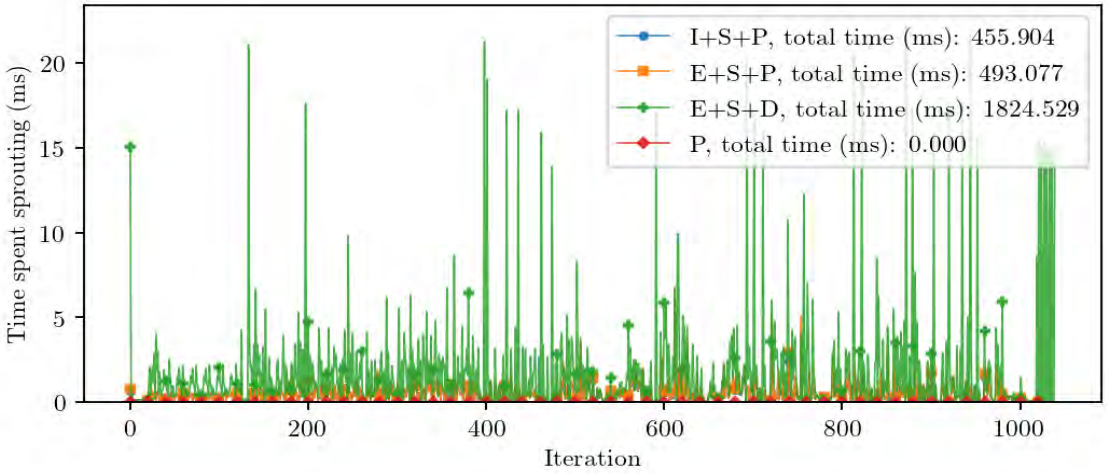


Figure 5.16: Time spent raising with play-forward scenario, motion-planning scenario and the non-coherent scenario. Note that explicit BVTT total timing results are lower than implicit BVTT timing results. Effects of resetting is also apparent — the sudden drops in raise-times could be due to front resets, which causes no node to be raised in the following iteration.

(a) Time spent sprouting, play-forward scenario



(b) Time spent sprouting, motion-planning scenario



(c) Time spent sprouting, non-coherent scenario

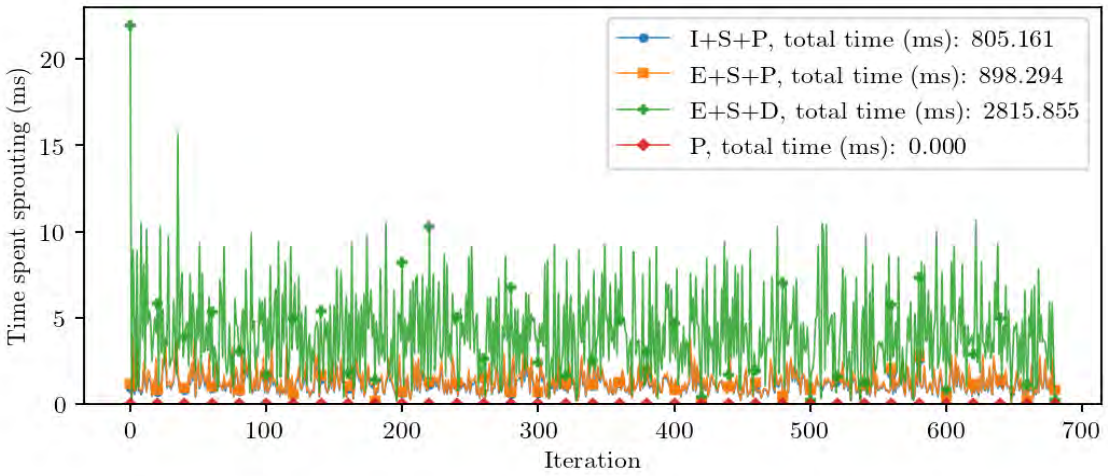
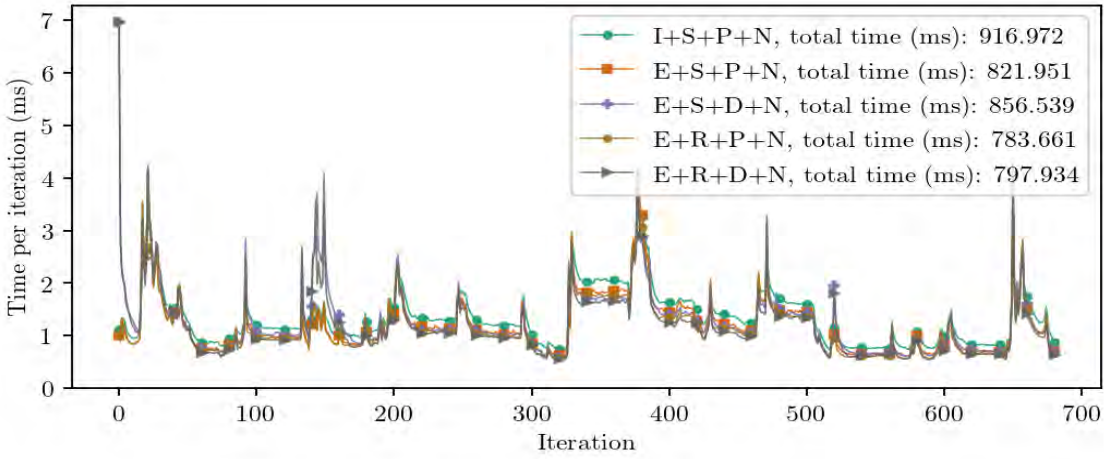
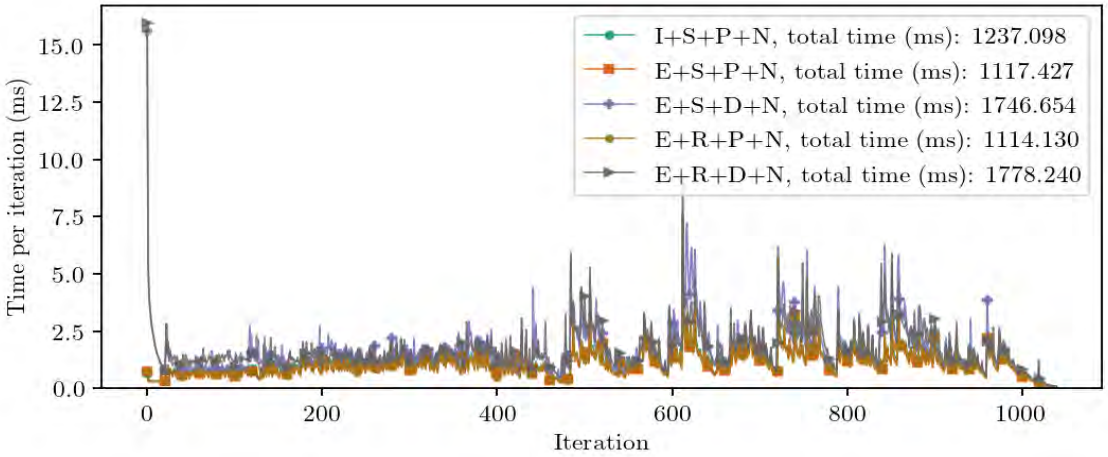


Figure 5.17: Time spent sprouting with (a) play-forward scenario, (b) motion-planning scenario and (c) the non-coherent scenario. Note that **I+S+P** is always faster than **E+S+P** and that **E+S+D** is fast for the play-forward scenario but degrades in the motion-planning scenario and the non-coherent scenario.

(a) Time spent per iteration, play-forward scenario



(b) Time spent per iteration, motion-planning scenario



(c) Time spent per iteration, non-coherent scenario

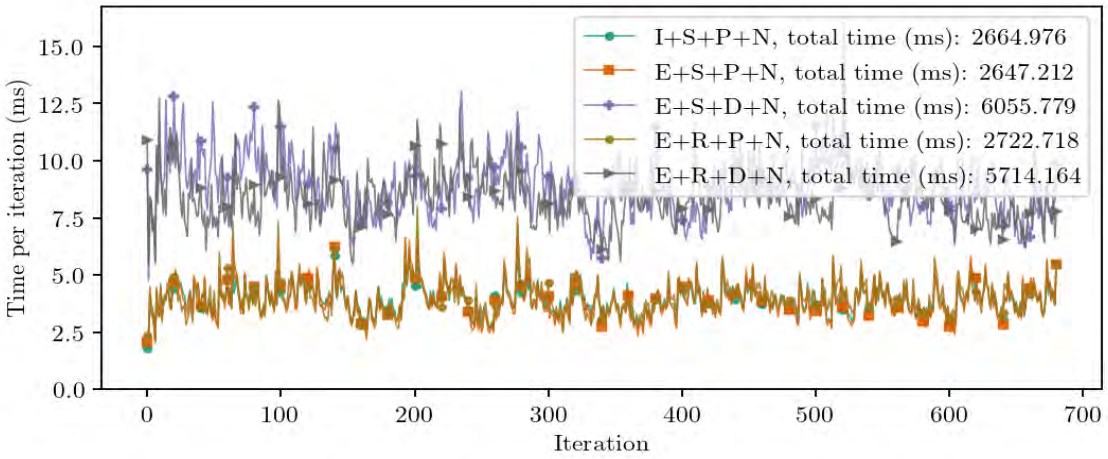
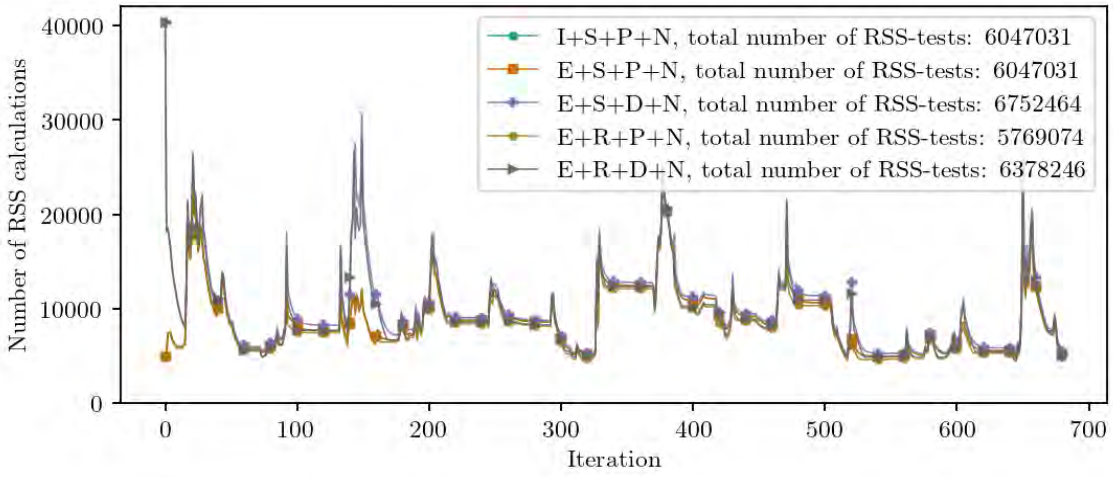
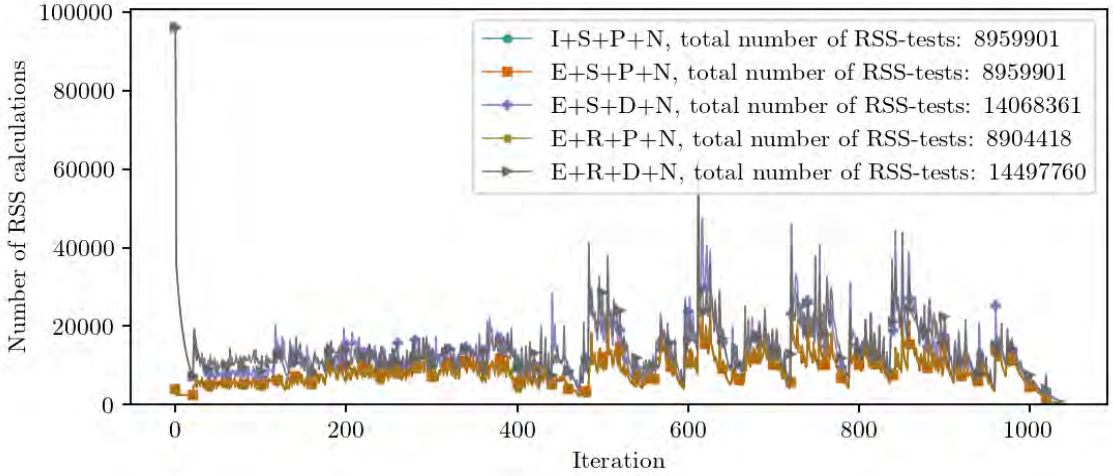


Figure 5.18: Timing results with front resetting disabled. Timing results per iteration is shown in the time-series. Total times are shown in the upper-right corners. Note that the shown time series has a half-life characteristic. Compare this figure with Figure 5.4 and Figure 5.12. Note that the total timings for all variants in this figure are greater than their corresponding series (with front resetting enabled) in Figure 5.12, except depth-first variants for the motion-planning scenario. This is discussed in Section 6.6.2 "Enabled front resetting".

(a) Number of RSS tests per iteration, play-forward scenario



(b) Number of RSS tests per iteration, motion-planning scenario



(c) Number of RSS tests per iteration, non-coherent scenario

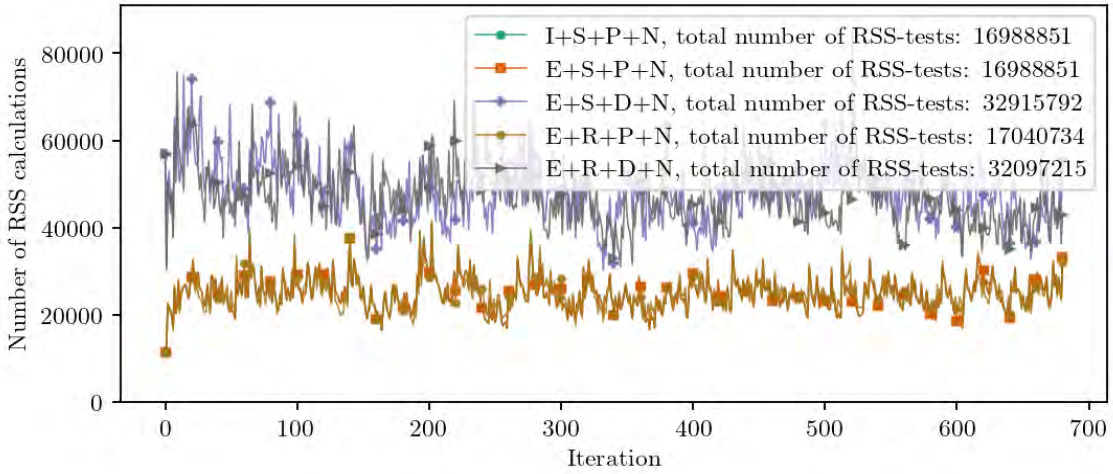


Figure 5.19: Number of RSS tests with front resetting disabled. The number of RSS tests per iteration is shown in the time-series. Total number of RSS tests are shown in the upper-right corners. Again, note that depth-first variants in the motion-planning scenario are outliers — disabled front reseting reduces the number of RSS tests in comparison (compare with Figure 5.13).

enough ϵ, that enables any significant pruning, is found. This manifests itself as an increase in the number of triangle tests.

Finally, compare Figure 5.21, that show front sizes obtained when front resetting is disabled, with Figure 5.11, that show front sizes obtained when front resetting is enabled. Note that front resetting causes front sizes to fluctuate for the motion-planning scenario and the non-coherent scenario, but that front sizes overall are higher without front resetting.

5.12 A bad and coherent scenario for E+R+D

E+R+D performs on par with IPS for the play-forward scenario, with respect to timings. I present a new scenario, denoted IRB72+IRB73, for which **E+R+D** performs worse in comparison to IPS with respect to time.

The scenario is the play-forward scenario but with $A = \{\text{IRB72}\}$ and $B = \{\text{IRB73}\}$. Coherency can be considered lower than the play-forward scenario since two moving robotic arms are tested against each other. Yet, coherence is still higher than the motion-planning scenario and the non-coherent scenario.

A sense of coherency for the IRB72+IRB73 scenario can be provided by the minimal separation distance measures, given in Figure 5.22. Timing result are presented in Figure 5.23 and the number of RSS tests computed is presented in Figure 5.24. Finally, the number of triangle tests computed is presented in Figure 5.25.

Recall that timing results shown in Figure 5.4 and Figure 5.12 indicate that **E+R+D** is faster than all FPQ variants as well as faster than IPS. Figure 5.23 show that IPS performs better than **E+R+D** for the IRB72+IRB73 scenario. Therefore, **E+R+D** is not always faster. This is discussed in Section 6.1.1 "A brief discussion on the IRB72+IRB73 scenario".

5.13 Two new motion-planning scenarios

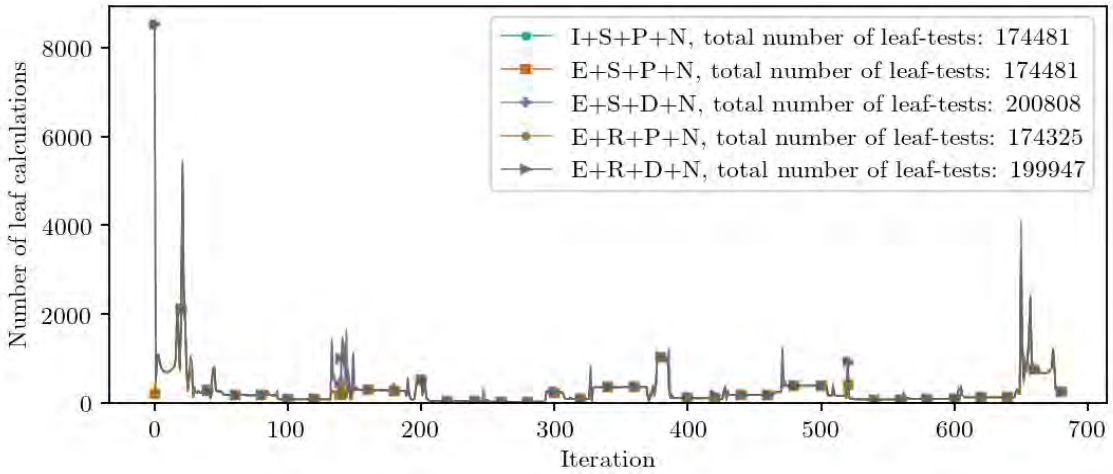
Two new motion-planning scenarios are devised in an attempt to understand how **P** scales in comparison to IPS. They are similar to the previously presented motion-planning scenario but static geometry is made to consist of either more or fewer triangles. The two motion-planning scenarios are:

40m MP: A motion-planning scenario where static geometry consists of 40 million triangles.

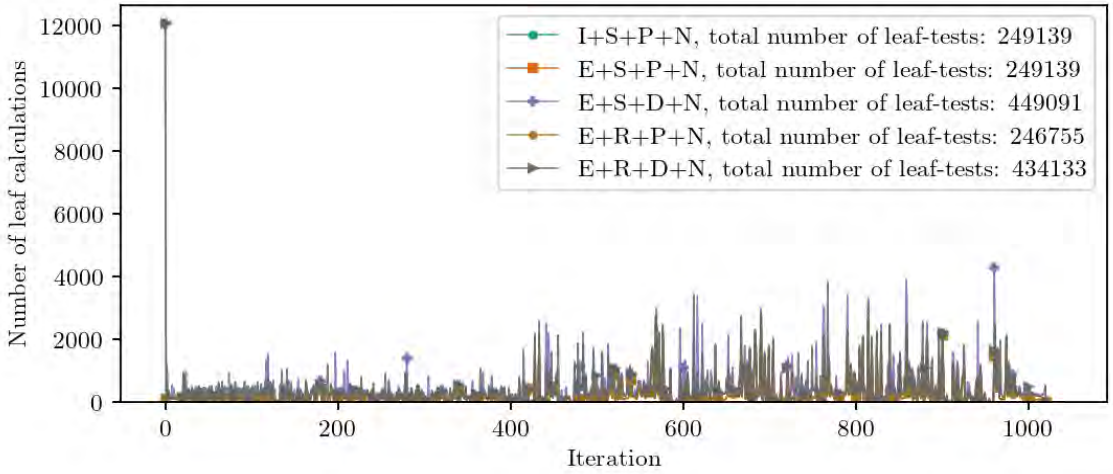
44k MP: A motion-planning scenario where static geometry consists of 44k triangles.

The 40m MP scenario is shown in Figure 5.26 and the 44k MP scenario is shown in Figure 5.27. Timings, number of RSS tests and number of triangle tests are shown in Figure 5.28, 5.29 and 5.30, respectively.

(a) Number of triangle tests per iteration, play-forward scenario



(b) Number of triangle tests per iteration, motion-planning scenario



(c) Number of triangle tests per iteration, non-coherent scenario

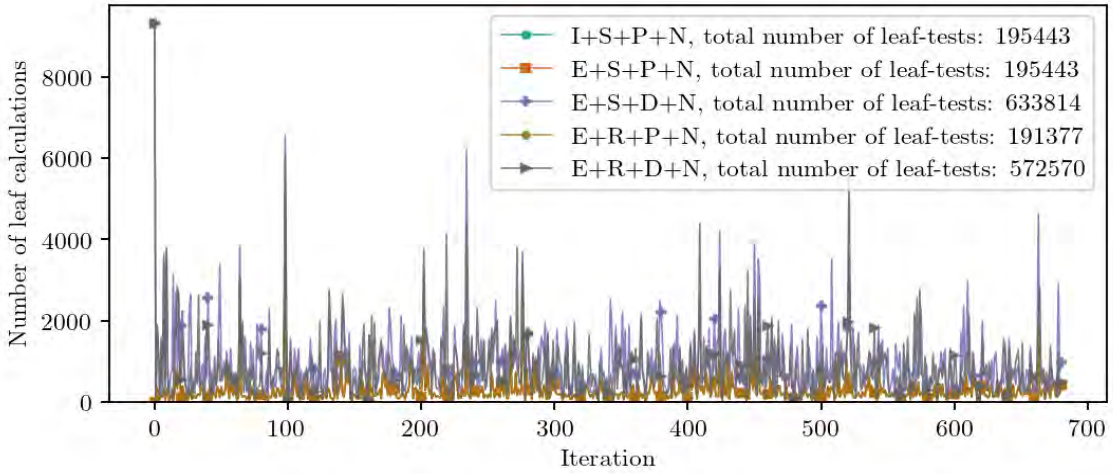
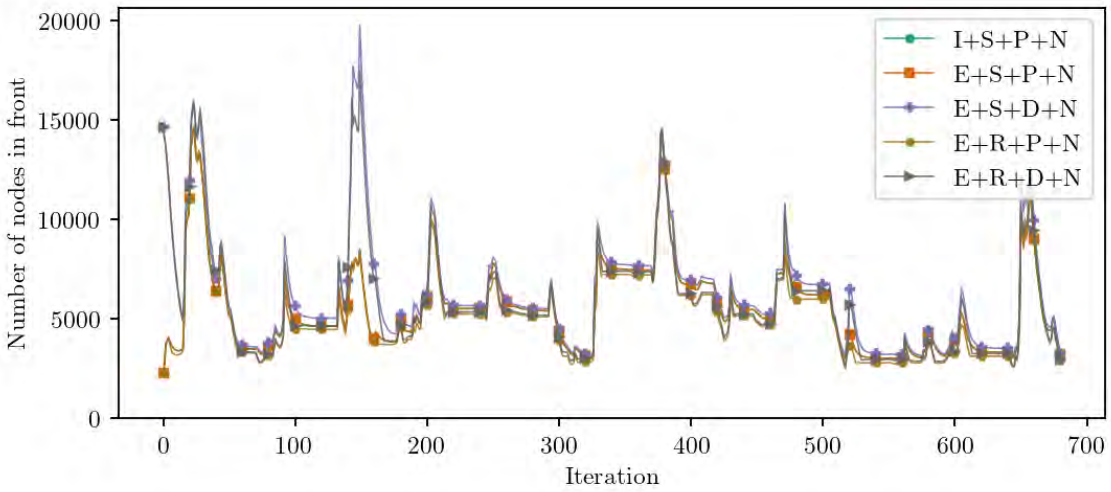
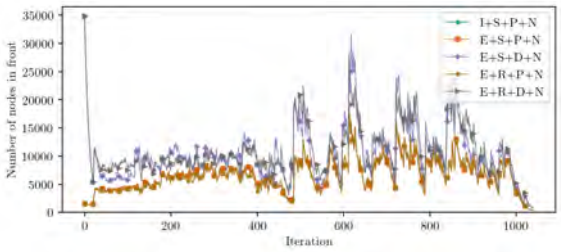


Figure 5.20: Number of triangle tests with front resetting disabled. The number of triangle tests per iteration is shown in the time-series. Total number of triangle tests are shown in the upper-right corners. Compare this figure with Figure 5.7, which shows leaf tests with enabled front resetting, and note that **E+S+D** performs more triangle tests with front resetting enabled for the motion-planning scenario and the non-coherent scenario.

(a) Front size per iteration, play-forward scenario



(b) Front size per iteration, motion-planning scenario



(c) Front size per iteration, non-coherent scenario

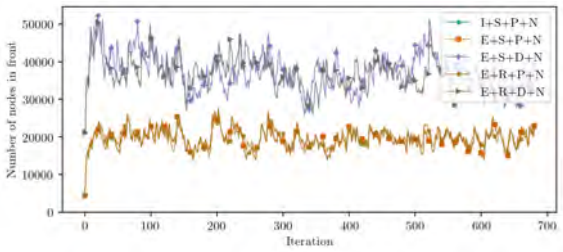


Figure 5.21: Front size per iteration, for each variant, is shown in the time series. Front resetting is disabled. Compare this figure with Figure 5.11 and note that front sizes become larger without front resetting

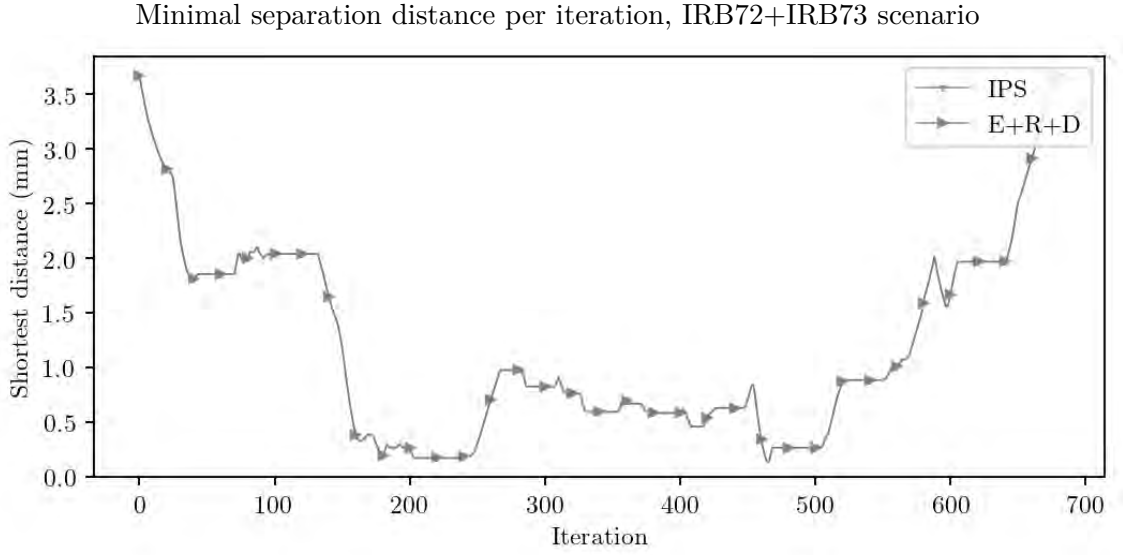


Figure 5.22: Minimal separation distance measurements for the IRB72+IRB73 scenario. Note in particular that the minimal separation distance increase and decrease rapidly during early and late iterations, respectively.

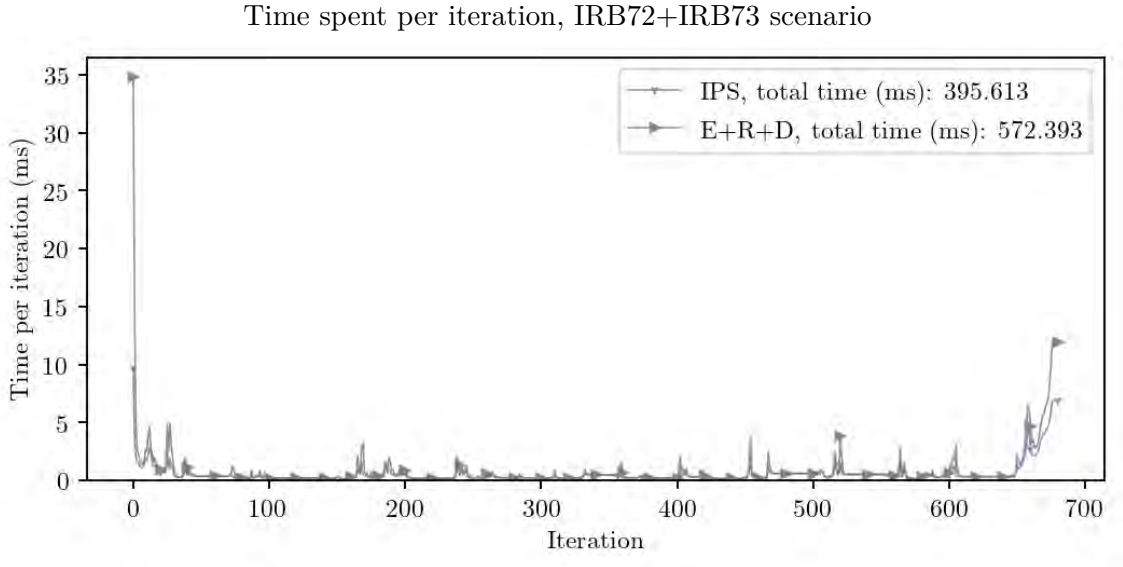


Figure 5.23: Timing results for the IRB72+IRB73 scenario. Note that **E+R+D** was for the play-forward scenario faster than IPS, but this is not the case for the IRB72+IRB73 scenario.

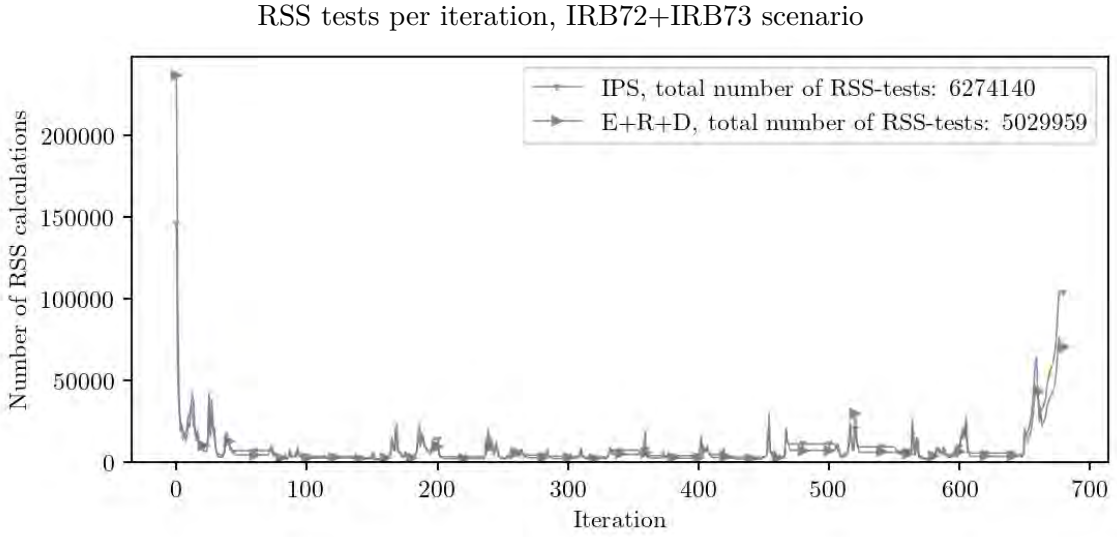


Figure 5.24: Number of RSS tests for the IRB72+IRB73 scenario.

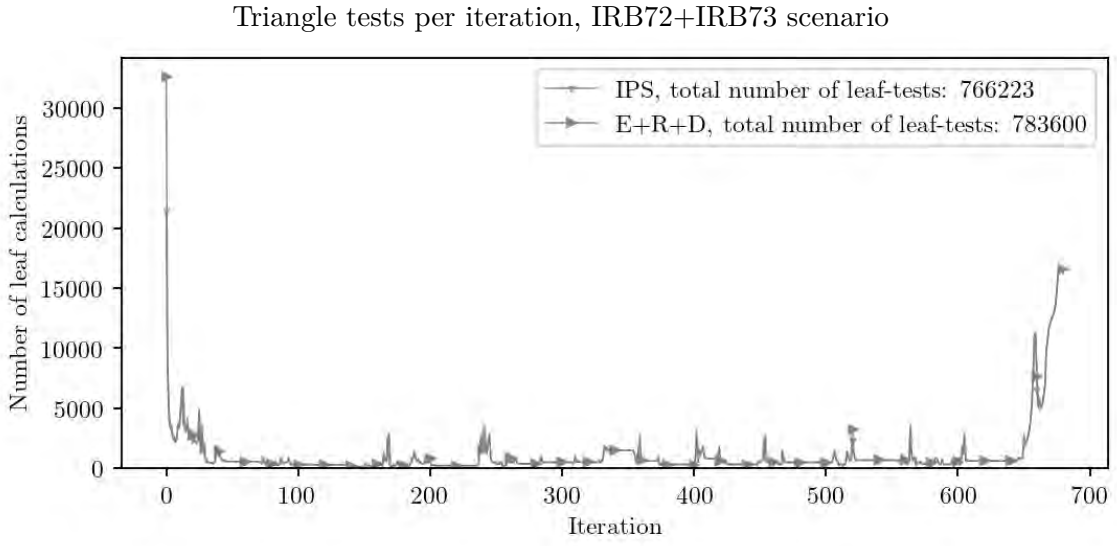


Figure 5.25: Number of RSS tests for the IRB72+IRB73 scenario. Note that **E+R+D** calculated fewer RSS tests for the play-forward scenario faster compared with IPS, but this is not the case for the IRB72+IRB73 scenario.

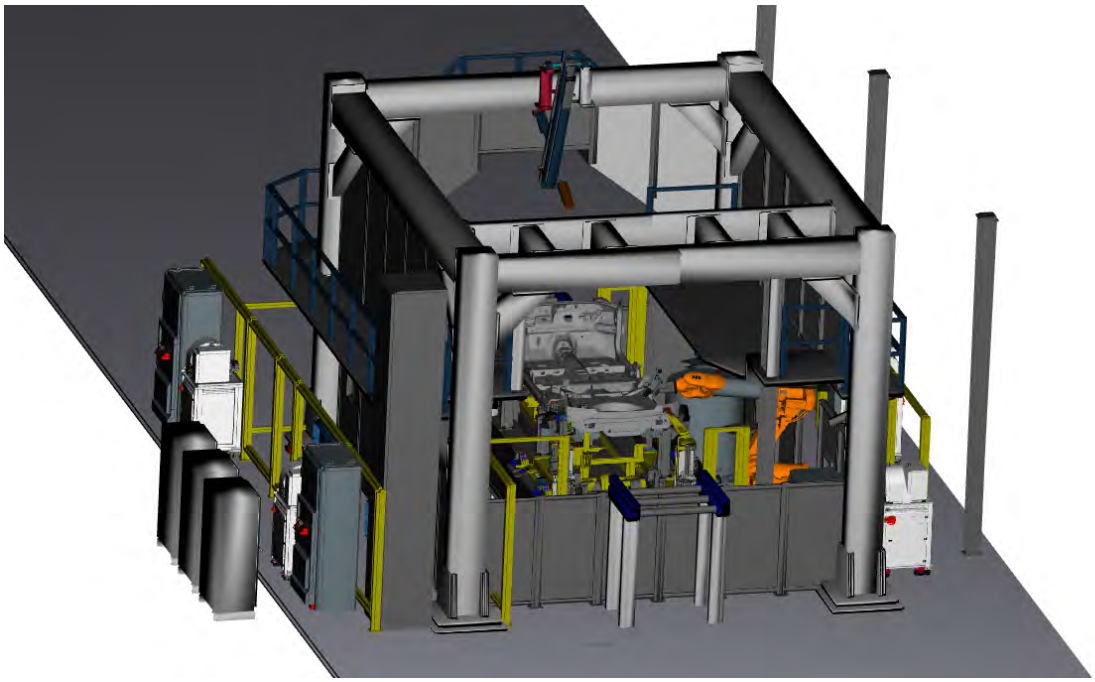
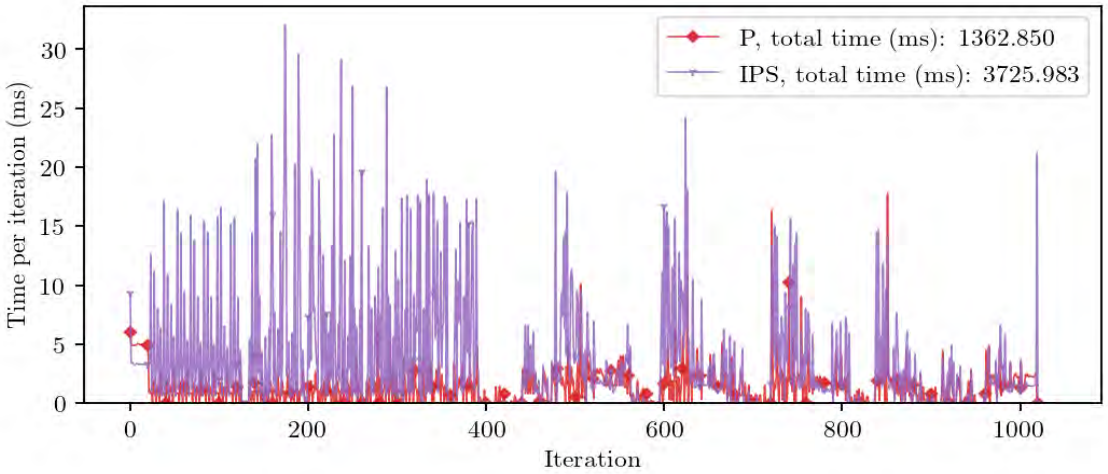


Figure 5.26: Overview of the 40m MP scenario. Geometry by courtesy of Volvo Cars.

Figure 5.27: Overview of the 44k MP scenario. Geometry by courtesy of Volvo Cars.

(a) Timings, 40m MP scenario



(b) Timings, 44k MP scenario

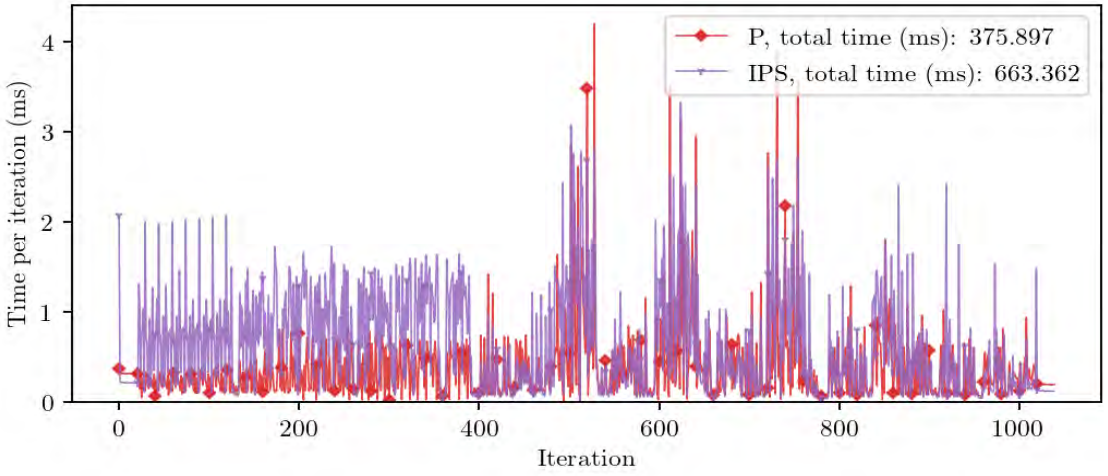
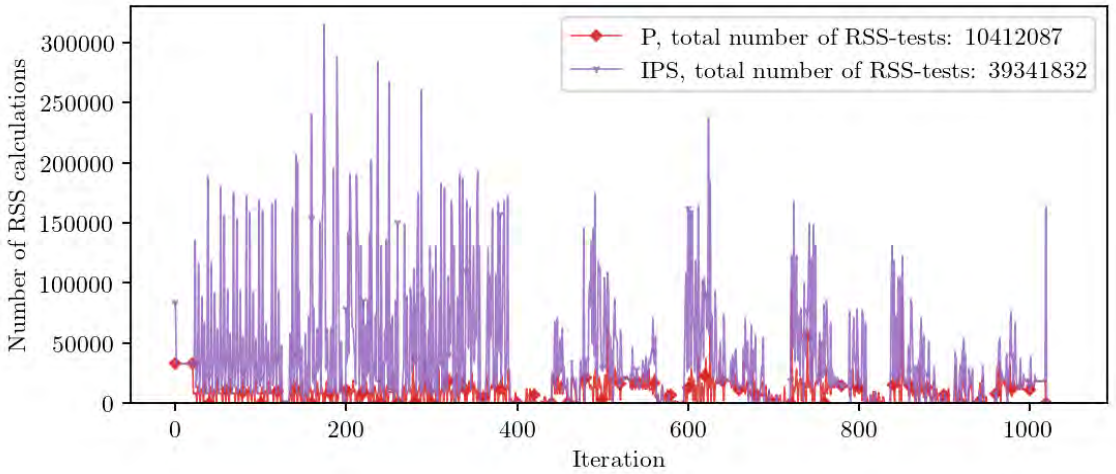


Figure 5.28: Timing results for (a) the 40m MP scenario and (b) the 44k MP scenario. The series show timing results per iteration. Time spent in total is shown in upper right corner. Note that **P** is faster than IPS in both scenarios.

(a) Number of RSS tests, 40m MP scenario



(b) Number of RSS tests, 44k MP scenario

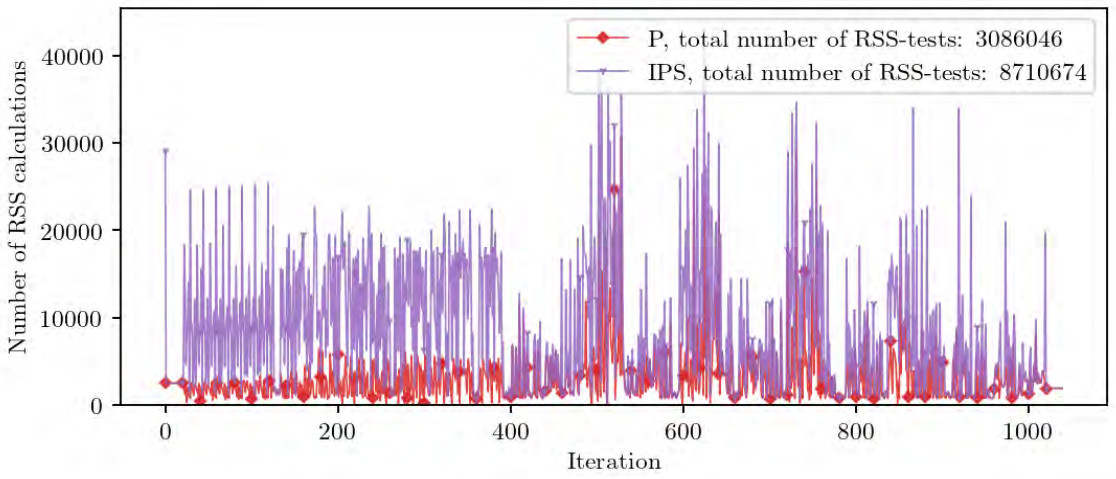
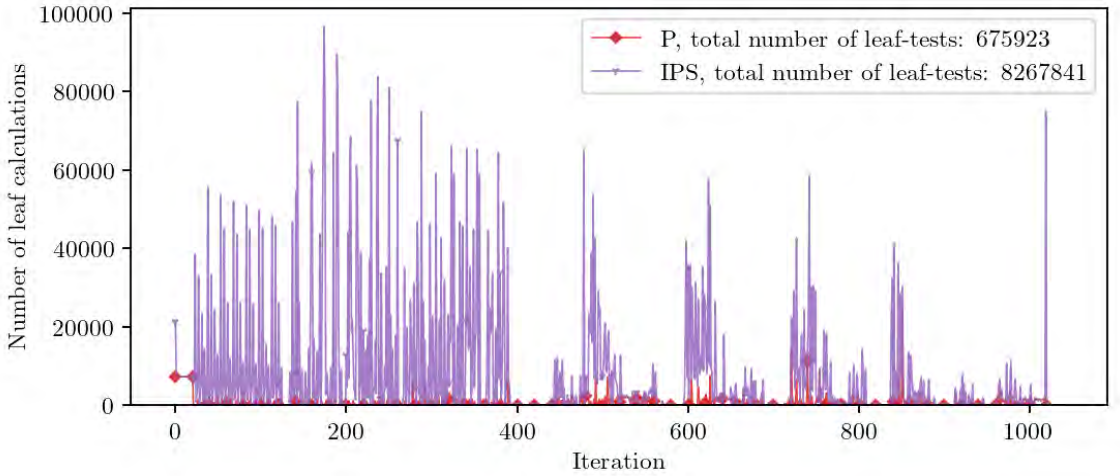


Figure 5.29: Number of RSS tests for (a) the 40m MP scenario and (b) the 44k MP scenario. The series show the number of RSS tests per iteration. The total number of RSS tests is shown in upper right corner. Note that **P** performs fewer RSS tests than IPS in both scenarios.

(a) Number of triangle tests, 40m MP scenario



(b) Number of triangle tests, 44k MP scenario

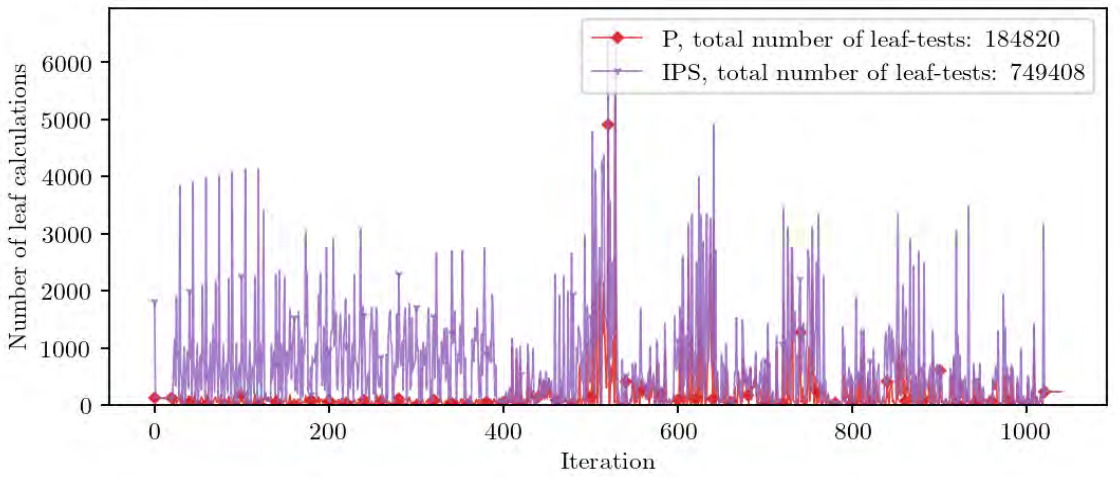


Figure 5.30: Number of triangle tests for (a) the 40m MP scenario and (b) the 44k MP scenario. The series show the number of triangle tests per iteration. The total number of triangle tests is shown in upper right corner. Note that **P** performs fewer triangle tests than IPS in both scenarios.

For now, it is enough to gather that measurements of timings, RSS tests, and triangle tests are smaller for **P** in comparison to **IPS** without exception for the 40m MP and the 44k MP scenarios. A discussion on how **P** scales in comparison to IPS, with respect to said scenarios, is presented in Section 6.2.2 "Scaling of PDS with respect to triangle counts".

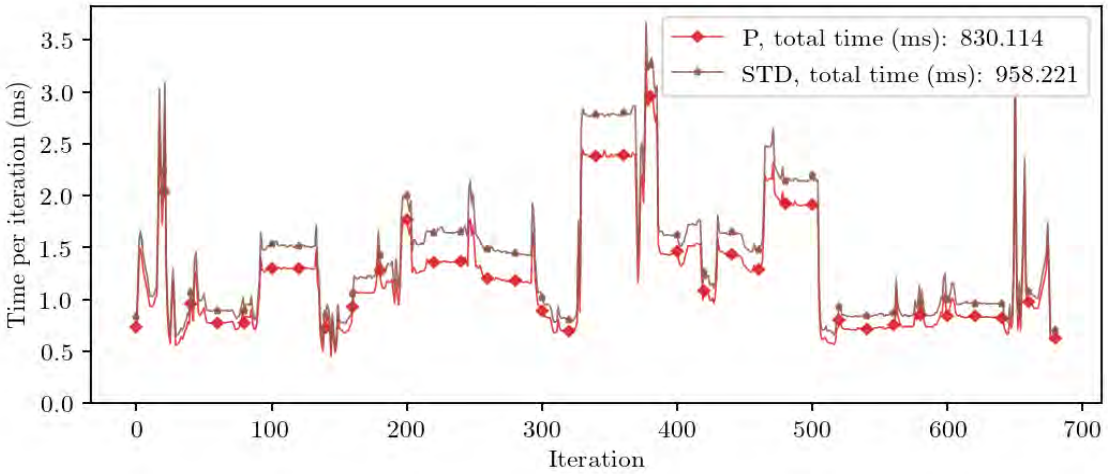
5.14 DPQ versus `std::priority_queue`

A DPQ is used by FPQ in contrast to a C++ standard priority queue. A comparison, with respect to timing results and number of RSS tests, between DPQ (denoted as **P**) and the C++ standard priority queue `std::priority_queue` (denoted as **STD**) is presented in figures within this section.

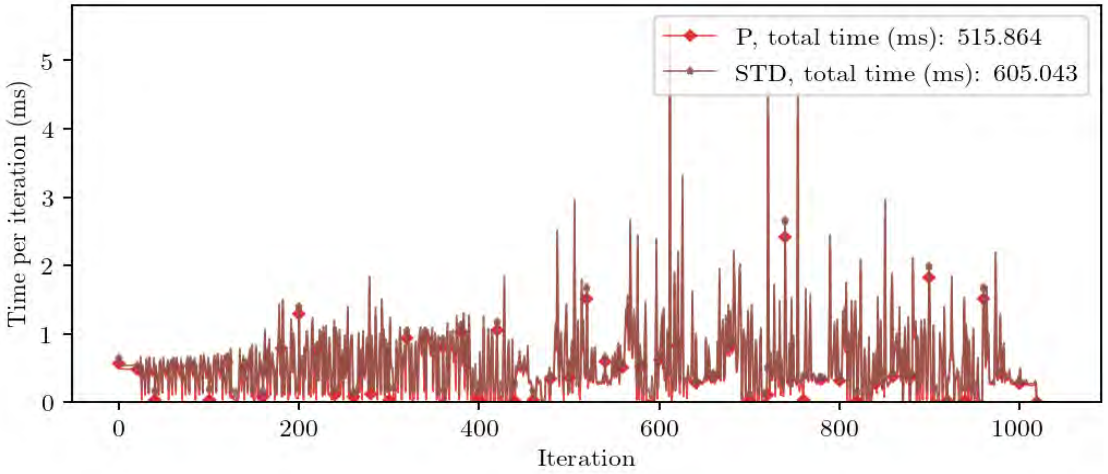
Timing results, per scenario, of **STD** and **P**, are shown in Figure 5.31. These timing results show that DPQ perform better than **STD** without exception, for the scenarios at hand.

The number of RSS tests, per scenario, for **STD** and **P** is shown in Figure 5.32. Note that the number of RSS tests is equivalent for the play-forward scenario and the non-coherent scenario, regardless of using DPQ or `std::priority_queue` . This is expected due to Theorem 1. However, for the motion-planning scenario, DPQ and `std::priority_queue` deviate with respect to the number of RSS tests. This is discussed in Section 6.8.2 "DPQ traverses few nodes if $q \in \mathcal{C}_{obs}$".

(a) DPQ vs STD timings, play-forward scenario



(b) DPQ vs STD timings, motion-planning scenario



(c) DPQ vs STD timings, non-coherent scenario

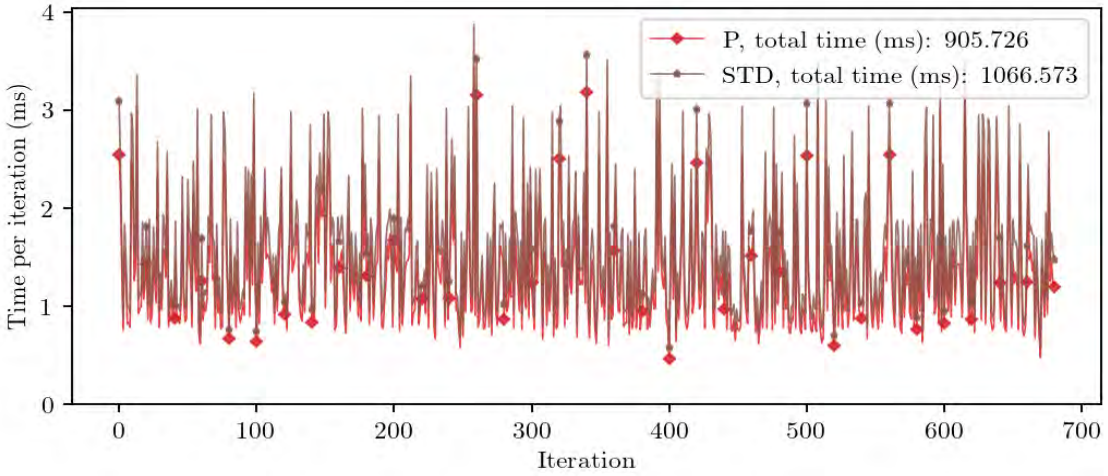
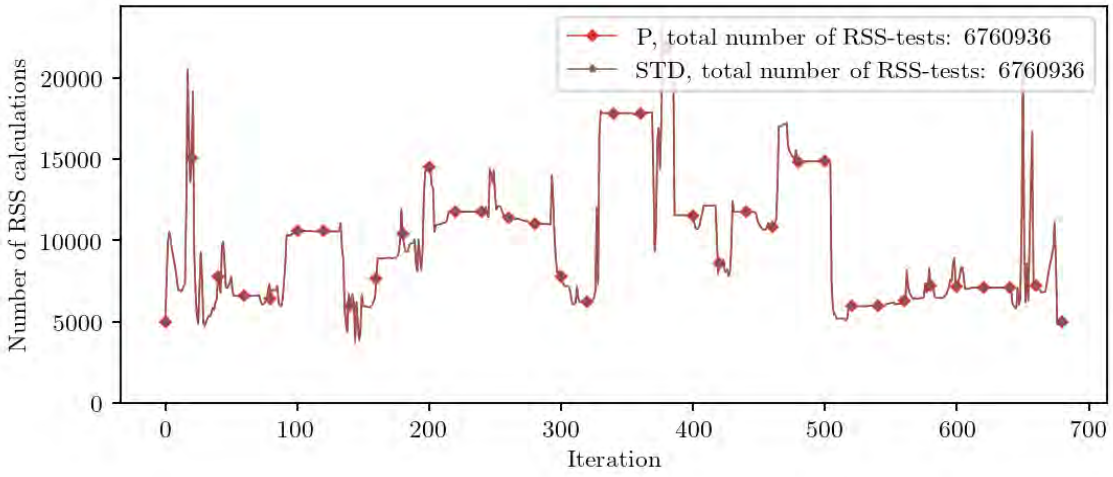
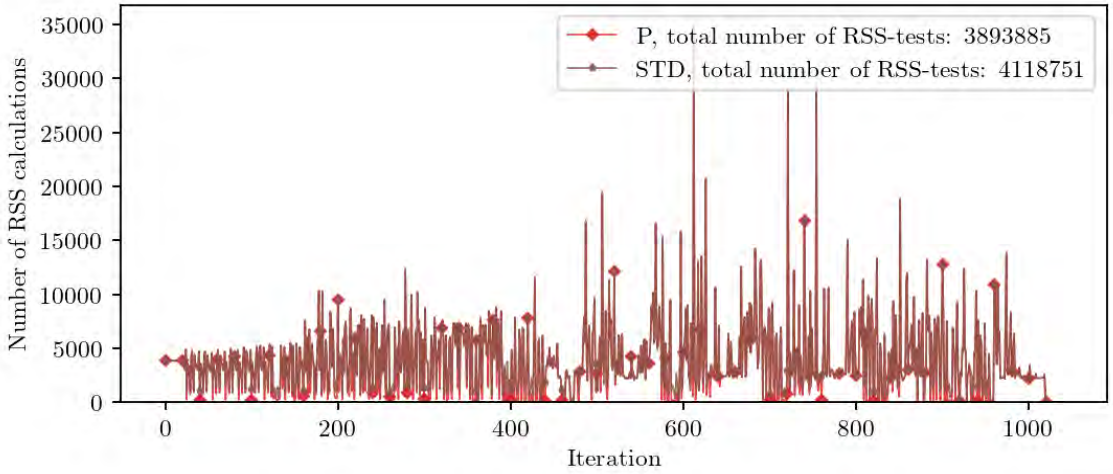


Figure 5.31: Timing results for the play-forward scenario, motion-planning scenario and the non-coherent scenario. The series show timing results per iteration. Time spent in total is shown in upper right corner. DPQ is denoted as **P** and the standard priority queue is denoted as **STD**. Note that DPQ is faster than **STD** for all scenarios.

(a) DPQ vs STD RSS tests, play-forward scenario



(b) DPQ vs STD RSS tests, motion-planning scenario



(c) DPQ vs STD RSS tests, non-coherent scenario

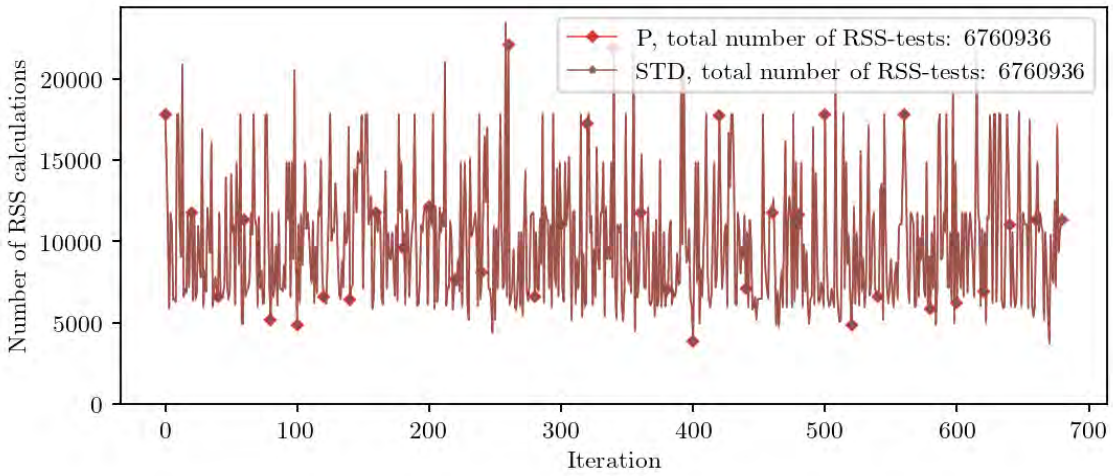


Figure 5.32: Number of RSS tests for the play-forward scenario, motion-planning scenario and the non-coherent scenario. The series show the number of RSS tests per iteration. The total number of RSS is shown in the upper right corner. DPQ is denoted as **p** and the standard priority queue is denoted as **STD**. Note that the number of RSS tests vary only for the motion-planning scenario.

6
Discussion

In this chapter, I discuss the results of this thesis and how they answer the research question as posed in Section 1.3 "Research question".

Sampling-based motion-planners are used to find paths throughout $\mathcal{C}$-spaces that govern motions of robotic arms. Sampling-based motion-planners depends on proximity queries, which constitute a major bottleneck of the IPS motion planner. For this reason, acceleration of proximity queries translates to acceleration of the IPS motion-planner.

A major bottleneck of proximity querying is that of BV-BV separation distance computations. IPS uses PQP for proximity queries and hence BVs are taken to be RSSs. Therefore, a reduction in the number of RSSs tests may reduce the computational time of motion-planning within IPS.

To this end, I developed a method called *Forest Proximity Query* (FPQ) together with a depth-first and a *priority directed search* (PDS) algorithm for BVTT traversal. A depth-first algorithm may be faster than PDS due to $\mathcal{O}(\log n)$ overhead associated with priority queues but I suggested a symbiotic relationship between FPQ and PDS in Section 3.3 "Priority directed search" which may cause FPQ with PDS to perform better than FPQ with depth-first search.

Timing results are presented in Figure 5.4. These timing results indicate that using a front together with depth-first search is viable in comparison to IPS. However, FPQ with PDS reduces the number of RSS tests, for coherent scenarios, more than any other FPQ-variant or IPS. Yet, a front with depth-first search perform better with respect to time, in comparison to using a front with PDS. In conclusion, overhead associated with $\mathcal{O}(\log n)$ insertion and deletions cancels any performance gains obtained from a reduced number of RSS tests.

In the motion-planning and the non-coherent scenario, PDS without FT (**P**) compares favorably against all other presented methods with respect to time and the number of RSS tests. This discussion is continued in Section 6.1 "Main results".

There are several minor techniques whose results, in isolation, are minor. However, when combined, these results form an algorithm which performs on par with IPS for the play-forward scenario and performs better than IPS for motion-planning and non-coherent scenarios with respect to time. Yet, the following techniques are discussed in isolation such that it is possible to understand how much they contribute to the final results:

- Depth-first versus PDS. In short, depth-first performs better than PDS for coherent scenes but worse on non-coherent scenes.
- Explicit BVTTs are intended to reduce raise-times. Results indicate that explicit BVTTs are successful in doing so, however, memory usage increases

by up to x25 in comparison to implicit BVTTs.
- A recursive raise strategy and front resetting is supposed to help an approximate front approach a rapidly changing optimal front. The interplay between raise strategies and front resetting is discussed in Section 6.6 "Single raise, recursive raise and reset".
- DPQ was introduced to alleviate overhead of PDS in coherent scenarios. DPQ performs better than a C++ standard priority queue implementation in all scenarios. However, performance gains are relatively lower with DPQ in the non-coherent scenario in comparison to the play-forward scenario and the motion-planning scenario. This is not because DPQ is particularly bad when coherence decreases below some threshold, its because DPQ is better for coherent scenarios by design and better for motion-planning scenarios by a fortunate coincidence. A discussion on DPQ is presented in Section 6.8 "DPQ"

Without further ado, main results are discussed.

6.1 Main results

The FPQ-variant notation from Chapter 5 "Results" is reused in this section. An acceleration of proximity queries is naturally measured by timing a proximity before and after some change of method. The change of method in question is that of BVTT traversal by using FPQ for traversal, instead of the IPS traversal algorithm, effectively reducing the number of RSS tests.

Fewer RSS tests may entail increased performance. However, decreasing RSS tests with FPQ implies front-overhead or priority queue overhead. Timing results indicate if benefits of FPQ outweigh drawbacks of FPQ, or vice versa. Two core results are therefore (1) timings and (2) number of RSS tests.

RSS tests form an upper-bound on the potential performance gain. For example, if FPQ computes half the number of RSS tests that IPS computes, than FPQ can potentially be, at most, twice as fast as IPS. Results shows conclusively that the number of RSS tests is reduced significantly, by up to 60%, with FPQ.

The research question is "What data structures or algorithms are suitable for acceleration of proximity queries in a robot-cell context? [...]" (see Section 1.3 "Research question"). Table 6.1 summarizes, for a given scenario, what method is suitable with respect to either time or the number of RSS tests. Table 6.1 motivates the following answer to the research question: It depends on the scenario at hand. I briefly motivate the entries in Table 6.1:

High-coherency scenarios: IPS is a solid choice for high-coherency situations where triangle soups undergo continuous rigid motion, as displayed by Figure 5.4. Nonetheless, **E+R+D** can possibly become more robust with further development. Indeed, results in Figure 5.12 show that **E+R+D** takes approximately 602 ms and results in Figure 5.4 show that IPS takes approximately 637 ms. Hence, **E+R+D** takes 35 less milliseconds than IPS.

On the other hand, Figure 5.23 show that **E+R+D** (572 ms) can be significantly worse than IPS (396 ms) with respect to time. In summary: **E+R+D**

Table 6.1: A summarizing table that recommends which method to use with respect either time or the number RSS tests (column). Note that suggested method vary depending on scenario (rows). All entry-methods are minimal with respect to the corresponding measure for the corresponding scene, except the Play-forward/Time entry for which I explicitly recommend IPS instead of **E+R+D** due to robustness.

Scenario	Time	RSS tests
Play-forward	IPS	**E+R+D**
Motion-planning	**P**	**P**
Non-coherent	**P**	**P**

can be 35 ms faster than IPS on one scenario but IPS can be 396 ms faster than **E+R+D**. In this regard, IPS is more robust than **E+R+D**.

Low-coherency scenarios: PDS, without FT, on forests of BVTTs is a solid choice for scenarios where coherency is low. One such scenario is motion-planning, for which PDS finishes at 511 ms while IPS finishes at 953 ms. This shows that PDS on a forest of BVTTs can accelerate proximity queries by up to 46%. This gain in performance is most likely due to the decreased number of RSS tests carried out when using PDS in comparison to IPS (from 11816218 tests with IPS to 3893885 tests with RSS, i.e. a 67% decrease in the number of RSS tests). Two synthetic test scenes with lower and higher triangle counts indicate that PDS scales favourably against IPS. No motion-planning scenario where IPS is advantageous in comparison to PDS on forests of BVTTs has been found.

A minor remark on the main results is that FPQ does not merely reduce the number of BVTT tests by increasing geometric primitive tests. On the contrary, all FPQ variants computes fewer geometric primitive tests in comparison to IPS, as shown by Figure 5.7. Two exceptions are **E+S+D** and **E+R+D** for the motion-planning scenario and the non-coherent scenario. More geometric primitive tests are computed with either **E+S+D** or **E+R+D** than IPS. Needless to say, neither **E+S+D** nor **E+R+D** trade fewer RSS tests for more geometric primitive tests — they're not suitable for scenarios without high-coherency.

Compare Figure 5.9 with Figure 5.11 and note leaf front sizes are arguably smaller than total front sizes. It should follow that time spent initializing ϵ is small in comparison to total timings. This is supported by a comparison of Figure 5.4 with Figure 5.8, which show that **E+S+D** is the variant with the largest fraction of time spent initializing ϵ: $11.310\,\text{ms}/614.739\,\text{ms} \approx 0.018$. This shows initialization of ϵ constitute a small fraction of total running times.

6.1.1 A brief discussion on the IRB72+IRB73 scenario

The IRB72+IRB73 scenario consists of two robotic arms performing stud-welding on a Volvo car body. **E+R+D** performs badly with respect to time for this scenario, as shown by Figure 5.23. On the other hand, Figure 5.24 shows that **E+R+D**

still performs approximately 20% fewer RSS tests in comparison to IPS. Further, **E+R+D** computes approximately 2% more geometric primitive distances than IPS, as shown by Figure 5.25.

It is no easy task to conclusively state why **E+R+D** is better (with respect to time) for the play-forward scenario but worse for the IRB72+IRB73 scenario (again, with respect to time). A possible explanation is that coherency is low during early iterations and late iterations since both robot arms move towards their first and last stud-welding jobs, respectively.

Finally, the IRB72+IRB73 indicate that **E+R+D** is potentially a bad fit for general-purpose proximity querying since possible losses with respect to time is large in comparison to possible gains with respect to time. In particular, Figure 5.23 show that possible losses, using **E+R+D** in comparison to IPS, amount to 572.393 ms − 395.613 ms = 176.78 ms longer computation times (45% increase in computation times), while Figure 5.4 with Figure 5.12 show that a possible gain, using **E+R+D** in comparison to IPS, amounts to 636.934 ms − 602.168 ms = 34.766 ms shorter computation times (5% decrease in computation times). Hence, possible gains are significantly smaller than possible losses and therefore I consider IPS more robust.

6.2 Depth-first and PDS

Timing results in Figure 5.4 indicate that depth-first search is generally better on the play-forward scenario. This is not surprising — a depth-first search traverses a close to minimal forest of BVTTs if coherency is high. High coherency implies $\epsilon \approx \delta$ due to either triangle caching (as used by IPS) or leaf fronts (as used by FPQ).

In Figure 5.6, it is clear that **E+S+P** visits fewer nodes than **E+S+D** at all times. This is not surprising due to Theorem 1. Yet, in Figure 5.4 it is shown that **E+S+D** performs slightly better than **E+S+P**, with respect to time, for the play-forward scenario. The time difference is due to sprout-times, shown in Figure 5.17, which are higher for PDS due to $\mathcal{O}(\log n)$ priority queue operations. Note that raise-times are higher for **E+S+D** than for **E+S+P**. Raise-times are, evidently, small in comparison to sprout-times. Hence **E+S+D** performs better than **E+S+P** for a coherent scenario. Raise-times are shown in Figure 5.16.

Results obtained for the motion-planning and the non-coherent scenario share many characteristics. Both scenarios have low levels of coherency but the motion-planning scenario generates points in obstacle-region $\mathcal{C}_{obs} \subseteq \mathcal{C}_{space}$ (recall, from Section 3.8 "Sampling points in $\mathcal{C}_{obs}$", that $\mathcal{C}_{obs}$ is the subset of a $\mathcal{C}$-space that contains all $\mathcal{C}$-space points with collisions, i.e. the region of a $\mathcal{C}$-space where a robot arm path is "blocked"). While **E+S+D** is seemingly competitive for coherent scenarios, **E+S+D** is the worst method with respect to all presented measures when coherence is low. A single exception is sprout-times, for which **I+S+P** is worse.

6.2.1 Depth-first and front tracking combines poorly for for low-coherency scenarios

E+S+D performs poorly for low-coherency scenarios. This is expected due to the following two reasons:

- A low-coherency scenario may render a leaf front close to useless since ϵ is not necessarily a small upper-bound on δ. In turn, this could cause a depth-first search to traverse far below an optimal front, provoking further costly RSS tests.
- A single level raise does not help towards moving upwards an optimal front in case of over-sprouting.

In short, **E+S+D** has a hard time approximating an optimal front that undergo rapid changes.

A consequence is that the front size is increased, see Figure 5.11, which in turn creates an abundance of BVTT nodes which must be raised in a following iteration. This increases raise-times. The front becomes a worse approximation of an optimal front while causing even more overhead. Therefore, it is plausible to state that no front is better than a bad front. I firmly conclude that front tracking with depth-first search should not be used for low-coherence scenarios.

6.2.2 Scaling of PDS with respect to triangle counts

P performs well in comparsion to IPS, with respect to the number of RSS tests, for motion-planning scenarios. Two new motion-planning scenarios, a 40 million triangle motion-planning scenario and a 44.000 triangle motion-planning, are introduced in Section 5.13 "Two new motion-planning scenarios". Hence, I have presented three motion-planning (MP) scenarios: A 44k MP scenario, a 2.2m MP scenario and a 40m MP scenario. These three scenarios can provide a sense of how PDS scale, in comparsion to IPS, with respect to triangle counts.

Table 6.2 compares timings results for the 44k MP scenario, the 2.2.m MP scenario, and the 40m MP scenario, for both PQP and **P**. Results of Table 6.2 are found in Figure 5.4 and Figure 5.28.

Table 6.2: Timings results for **P** and IPS in the motion-planning scenario as well as the 40m MP scenario.

Scenario	44k MP	Motion-planning (2.2m)	40m MP
P	376 ms	511 ms	1363 ms
IPS	663 ms	953 ms	3726 ms
Speedup	43%	46%	63%

The summary provided in Table 6.2 indicates that the performance gap between **P** and IPS increases with the number of triangles. This can be explained due to Theorem 1 — **P** will visit a minimal forest of BVTTs regardless of coherence, while IPS may explore much larger forests. This claim is supported by Figure 5.29 and Figure 5.30, both of which show that **P** visits fewer BVTT nodes in total than IPS. In a sense, it is not **P** that becomes arbitrarily fast with increasing triangle counts. Rather, it is IPS that is having increasing difficulties navigating a (large) forest of BVTTs when coherence is low.

6.3 FPQ and GFT

Recall the analysis of FPQ versus GFT (Section 3.3.4 "Analysis of FPQ vs GFT"). It was stated that the ratio of pruned BVTT nodes had to be sufficiently large in order for FPQ to be better than GFT with respect the number of to priority queue operations. This ratio has not been measured nor are timing measurements are available on GFT versus FPQ.

However, the number of RSS tests for both methods are roughly halved with FPQ in comparison to GFT, as shown by Table 5.1 ($1458288/676093 \approx 2.16$). Further, the number of insertions i on a priority queue used by PDS is equal to the number of RSS tests $t = i$. The number of deletions r is trivially bounded from above by the number of insertions i. Hence the number of operations is at most $2i = 2t \geq i + r$, i.e. the number of operations on the priority queue is at most twice that of the number of RSS tests. But since the ratio of RSS tests is approximately $2.16 > 2$, it follows that FPQ does fewer priority queue operations GFT since GFT does at least twice the amount of RSS tests i.e. at least twice as many priority queue operations as FPQ.

This discussion showed, with help of experimental results, that the number of priority queue operations must be smaller with FPQ than with GFT for coherent scenarios. There is no reason to believe that other levels of coherency destroy this result, due to Theorem 1. It follows that FPQ reduces overhead associated with PDS in comparison with GFT.

6.4 Raise times and memory usage with implicit and explicit BVTTs

Explicit BVTTs were introduced in order to improve raise times at the cost of increased memory usage. From Figure 5.10 we gather that memory usage is increased in all scenarios when using explicit BVTTs. At best, explicit BVTTs quadruples memory usage in comparison to implicit BVTTs. At worst, memory usage increases by roughly x25 in comparison to implicit BVTTs.

For coherent scenes, memory usage of FPQ remains relatively small but as coherence decreases, memory usage increases. This is not a surprising result since fronts become large in scenarios with low coherence, as shown by Figure 5.11. Yet, memory usage never exceeds 10 MB in any scenario, which is tractable on any consumer level computer.

Fortunately, the increased memory usage improves upon raise-times as anticipated. In Figure 5.16, we observe that raise times are significantly reduced with **E+S+P** and **E+S+D** in comparison to **I+S+P** for all scenarios. In the end, time spent in the play-forward scene is reduced from 0.703594 to 0.647196 (see Figure 5.4), improving total performance by roughly 13%.

As for the motion-planning scenario and the non-coherent scenario, no significant net-gain in performance is observed in Figure 5.4, despite the fact that raise-times are reduced, as shown in Figure 5.16. This could be because BVTT bytesize increases with explicit BVTTs, causing copy operations and move operations, commonly used

when sprouting, to become slightly more expensive. Sprout-times are presented in Figure 5.17. Observe that sprout-times for the motion-planning and the non-coherent scenario increases about as much as the decrease in raise-times, effectively cancelling gains obtained by smaller raise times.[1]

In conclusion, explicit BVTTs increases total performance for coherent scenarios but no significant gain is observed for the motion-planning scenario nor the non-coherent scenario. Nevertheless, the relative increase of memory usage is high but negligible in comparison to modern day capacities of consumer level laptops. Concerns raised in Section 2.4.1 "Memory usage of front tracking" are relieved.

6.5 Recursive raisal decrease raise times

Interestingly, Figure 5.14 show that recursive raisal *reduce* total raise times for all scenarios, no matter the traversal algorithm, contrary to intuition. It seems valid to argue that a recursive raisal algorithm should increase raise times since it may traverse more nodes than a single level raisal.

This is incorrect reasoning — while it is true that a recursive raisal may traverse more nodes in a single iteration, it may raise only a select few nodes in a following iteration. Single level raise, on the other hand, may traverse n nodes and raise one pair of siblings. On the next iteration, it traverses $n - 1$ nodes and raise on pair of siblings. This argument is highly similar to that of the näive recursive raisal algorithm presented in Section 3.5.1 "A $\mathcal{O}(n^2)$ recursive sibling-raise routine". Hence, a single level raise may behave like a naïve recursive raise spread across several iterations.

This explains why Figure 5.14 show that recursive raisal is consistently beneficial over single level raisal with respect to total raise times.

6.6 Single raise, recursive raise and reset

A problem with front tracking is that a front may linger below an optimal front unless an aggressive raise strategy or a reset strategy is adapted. This is measurable by examining the size of a front — a large front which slowly decreases over iterations is likely a front that lie below an optimal front.

I compare the single level raise strategy with the recursive raise strategy in a reset-free context. Thereafter, I compare the two strategies with enabled front resetting. There are two motivations for comparing raise strategies with and without front resetting:

1. Front resetting may obfuscate a comparison between a single level raise strategy and a recursive raise strategy
2. The impact of front resetting emerges when comparing raise strategies with and without front resetting.

[1]The arithmetical reader notices that summing raise-times, sprout-times and leaf-times do not sum up to the total measured time. This is because time measurements do not cover all lines within the C++ implementation. For example, front partitioning is not timed.

As we shall see, front resetting significantly improves upon performance.

6.6.1 Disabled front resetting

Raising one level at a time with a sibling raise strategy causes front sizes, shown in Figure 5.21, to have half-life characteristics — front sizes decrease rapidly at first, after which decrease rates diminishes. This phenomena is especially visible in subplot (a) of Figure 5.21, by examining the **E+S+D+N** curve.

The corresponding variation using a recursive raise strategy, **E+R+D+N**, seems to have a half-life characteristic that is less pronounced. Instead, **E+R+D+N** seems to approach the optimal front more rapidly, as intended. This indicates that a recursive raise strategy does indeed help a front approach an optimal front more swiftly. It remains to discuss if a recursive raise strategy pays off in terms of time.

Table 6.3: A table summarizing differences between the single raise strategy and the recursive raise strategy, for each scenario and for both traversal methods, when front resetting is disabled.

	Play-forward		Motion-planning		Non-coherent	
	PDS	DF	PDS	DF	PDS	DF
S	822 ms	857 ms	1117 ms	1746 ms	2647 ms	6056 ms
R	784 ms	798 ms	1114 ms	1778 ms	2723 ms	5714 ms
Diff.	38 ms	59 ms	3 ms	32 ms	76 ms	342 ms

Timing results obtained with disabled front resetting, shown in Figure 5.18, are summarized in Table 6.3. The **I+S+P+N** variant is not represented in said table, since it behaves similarly to **E+S+P+N**, with exceptions already discussed in Section 6.4 "Raise times and memory usage with implicit and explicit BVTTs".

Table 6.3 indicates that a recursive raise is more beneficial when using depth-first search in comparison to when using PDS (for all scenarios). This is not surprising. A depth-first search may sprout below minimal forests of BVTTs, warranting an aggressive raise strategy in order to compensate for over-sprouting.

Differences between the single level raise strategy and the recursive raise strategy may be considered small. I speculate why the difference is small: The lack of difference is perhaps due to early termination of raise routines, as described by Section 3.6.1 "Evaluating heuristic during raisal". If there is little next to no coherency in the motion-planning scenario or the non-coherent scenario, it could be that 9% of front nodes are raised regardless of using a single level raise or recursive raise. This may explain why the differences are relatively small.

6.6.2 Enabled front resetting

I have compared the single raise strategy with the recursive raise strategy when front resetting is disabled. It is of high interest to see what happens when front resetting is enabled. Table 6.4 summarizes timing results from Figure 5.12.

Note that all entries of Table 6.4, disregard Diff. entries, are smaller than their corresponding entries of Table 6.3 *except* depth-first search for the motion-planning

Table 6.4: A table summarizing differences between the single raise strategy and the recursive raise strategy, for each scenario and for both traversal methods, when front resetting is enabled.

	Play-forward		Motion-planning		Non-coherent	
	PDS	DF	PDS	DF	PDS	DF
S	638 ms	612 ms	617 ms	2369 ms	1106 ms	3338 ms
R	633 ms	602 ms	605 ms	2517 ms	1083 ms	3340 ms
Diff.	5 ms	8 ms	12 ms	148 ms	23 ms	2 ms

scenario. This means that resetting is beneficial except for the motion-planning scenario with depth-first search.

At a first glance, this is surprising. However, there is an intuitive explanation: Early termination is always possible if there exists a collision, i.e. that $q \in \mathcal{C}_{obs}$. This was shown in Section 3.8 "Sampling points in $\mathcal{C}_{obs}$". However, a reset annihilates a leaf front that could otherwise hold a BVTT node v with $\|v\| = 0$, which is to say, a reset could forcefully remove an opportunity of early termination.

Note that the argument above is applicable to the PDS variant too. However, PDS does indeed benefit from front resetting in the motion-planning scenario. This oddity is due to DPQ and is thoroughly explained in Section 6.8.2 "DPQ traverses few nodes if $q \in \mathcal{C}_{obs}$".

Regardless, Table 6.4 and Table 6.3 in conjunction show that front resetting is, in general, beneficial as a complement to either a single level raise strategy or a recursive raise strategy.

Finally, observe that the depth-first entry on the play-forward scenario, using recursive raisal, finishes in 602 ms while IPS finishes in 637 ms (see Figure 5.4). I have demonstrated that **E+R+D** can, on some occasions, be 5.5% faster than IPS.

6.6.3 Reset reduces BV tests for BVHs lacking the bounding property

Front resetting was introduced to assist raise-operators. A reset ought to be beneficial if it is cheaper to sprout from BVTT roots (performing RSS calculations during sprouting) than it is to raise (without performing any RSS calculations). Intuitively, it seems that in most cases it should be cheaper to raise rather than reset. Yet, resetting arguably increases performance, perhaps more than expected.

The bounding property does not hold for RSS hierarchies. This means parents may have larger distances than their children. Hence, it is possible that traversal may terminate at a parent parent(v) with $\|\text{parent}(v)\| > \epsilon$ but termination is not possible at v, since $\|v\| \leq \epsilon$. Hence, if a front is raised to v but no further due to anchor nodes, pruning is not possible. But if the front would approach parent(v_c) from above, then v and its subtree could be pruned. A reset forces a front to approach nodes from above.

This argument explains why a reset is particularly important for BVHs for which the bounding property does not hold. In particular, this argument motivates why resetting is as beneficial as indicated by Table 6.4 and Table 6.3.

6.7 Avoiding copy-overhead during raisal

Copying on line 9 and line 11 in Algorithm 8 "Complete FPQ with DF search" is unnecessary. An improved raise algorithm could pop and swap from two different containers (RF and L) without resorting to copying L to RF. I briefly outline the improved raise algorithm now:

First, raise RF with Algorithm 5 "Raising" as usual. Then raise leaves as usual with the following modification: If the first encountered sibling lie in RF and second encountered sibling lie in L, pop and swap second sibling and overwrite first sibling in RF by their common parent.

The second encountered sibling can never lie in RF when traversing over L since RF is traversed prior to traversing L.

The complete (improved) FPQ with depth-first search algorithm is seen in Algorithm 14 "Complete FPQ with DF search (improved)". Compare Algorithm 8 "Complete FPQ with DF search" with Algorithm 14 "Complete FPQ with DF search (improved)" — line 9 and line 11 in Algorithm 8 "Complete FPQ with DF search" are no longer present in Algorithm 14 "Complete FPQ with DF search (improved)".

Algorithm 14 Complete FPQ with DF search (improved)

```
1: function ONE_TIME_INIT(A, B, SF)
2:     for ∀A ∈ A do
3:         for ∀B ∈ B do
4:             SF ← SF ∪ {CREATE_BVTT_ROOT(A, B)}
5: end function
6: function DIST(L, SF, RF, q_i)
7:     ϵ ← INIT_EPS(L, q_i)
8:     PARTITIONING(RF, SF, q, S, ϵ)
9:     RAISE(RF)
10:    RAISE_LEAVES(RF, L)
11:    ϵ ← SPROUT(S, SF, L, q, ϵ)
12:    δ ← ϵ
13:    return δ
14: end function
```

This concludes the suggested improvement of the raisal operator.

The improved raise algorithm was not implemented due to other more pressing issues taking priority over implementation of this algorithm. Results presented in Chapter 5 "Results" does not use Algorithm 14 "Complete FPQ with DF search (improved)".

6.8 DPQ

A DPQ is used to reduce the number of $\mathcal{O}(\log n)$ insertions and deletions when using PDS. Timing results shown in subplots (a-f) of Figure 5.31 show that DPQ

is faster than `std::priority_queue` for all scenarios. Table 6.5 summarizes time differences between DPQ and `std::priority_queue` .

Table 6.5: A table summarizing differences between DPQ and `std::priority_queue` .

	Play-forward	Motion-planning	Non-coherent
`std::priority_queue`	958 ms	605 ms	1067 ms
DPQ	830 ms	516 ms	906 ms
Relative gain with DPQ	13.3%	14.7%	15.1%

Table 6.5 seem to be some (weak) correlation between timings results and coherency, however, these results might just as well be due to measurement inaccuracies.

6.8.1 DPQ hypothesis

I considered DPQ due to a hypothesis stating that a path traversed by PDS is close to a path traversed by depth-first search (see Section 3.7 "Duplex priority queue"). This hypothesis stemmed from Figure 5.6, in which it is apparent from subplot (a) that PDS visits somewhat fewer nodes and from (b) it is apparent PDS and IPS computes close to equally many RSS tests per iteration, even when the number of RSS tests is decreasing or increasing. Note that subplot (a) and subplot (b) correspond to the play-forward scenario, which has high coherence for successive $\mathcal{C}$-space points q_i, q_{i+1}.

Since PDS traverses a minimal forest of BVTTs but IPS traverses almost as few nodes as PDS, then IPS must traverse a close-to minimal forest of BVTTs or conversely, PDS must traverse a close-to depth-first path. This makes PDS compare unfavorably for high-coherence scenes, not because PDS fails to traverse minimal forests of BVTTs, but since depth-first searches traverses close to minimal forests of BVTTs.

Comparative results on DPQ and `std::priority_queue` , presented in Figure 5.31, is strengthening the aforementioned hypothesis since an increase in performance is obtained with DPQ over `std::priority_queue` . Further, DPQ internally depends on `std::priority_queue` and `std::stack` , so the improved results can not be due to a better priority queue implementation within DPQ. Hence, the results presented in Figure 5.31 are fair and hence strengthens the hypothesis.

6.8.2 DPQ traverses few nodes if $q \in \mathcal{C}_{obs}$

DPQ and `std::priority_queue` visit equally many RSS for the play-forward scenario and the non-coherent scenario but DPQ traverses fewer nodes for the motion-plannings scenario (see Figure 5.32) — the motion-planning scenario is an exception. This was, at the time of result-generation, an unexpected result. It seems incorrect that either DPQ or `std::priority_queue` does not traverse a minimal forest of BVTTs. However, there is an explanation:

DPQ performs a depth-first search if BVTT nodes have equal distances, whereas `std::priority_queue` may traverse nodes with equal distances in an undefined order. Therefore, DPQ hits leaves quickly while `std::priority_queue` may visit several intermediate nodes prior to hitting a leaf. In addition, hitting a leaf l with $\|l\|_{leaf} = 0$ allows for early termination, as described by Section 3.8 "Sampling points in $\mathcal{C}_{obs}$", causing DPQ to terminate earlier than `std::priority_queue` .

Now, any colliding BVTT node is a member of of a minimal BVTT, by Definition 4, because a node is traversed if it has a distance which is less than *or equal* to $\|A - B\|$. If l is a colliding BVTT leaf, then $\|A - B\| = 0$ and all parents of l are colliding too. Hence l is always traversed and hence always a member of a minimal forest of BVTTs. However, early termination is possible if $\epsilon = 0$, as described in Section 3.8 "Sampling points in $\mathcal{C}_{obs}$". For this reason, a forest of BVTTs smaller than that of a minimal forest of BVTTs is (oftentimes) traversed by DPQ and (seldomly) traversed by `std::priority_queue` if $q \in \mathcal{C}_{obs}$. Hence, DPQs are, unexpectedly, particularly good for motion-planning scenarios if techniques presented in Section 3.8 "Sampling points in $\mathcal{C}_{obs}$" are used.

I have discussed why both DPQ and `std::priority_queue` can traverse forests of BVTTs that are smaller than minimal forests of BVTTs and I have discussed why DPQ does so more frequently than `std::priority_queue` .

On a side note, one may think of a *strictly* minimal forest of BVTTs, generated by nodes which are traversed if $\|v\| < \epsilon$ as opposed to $\|v\| \leq \epsilon$ (see Algorithm 1 "Depth-first search"). Both DPQ and `std::priority_queue` , with techniques presented in Section 3.8 "Sampling points in $\mathcal{C}_{obs}$", may never traverse a forest of BVTTs which is strictly smaller than a strictly minimal forest of BVTTs.

6.9 FPQ with other BVs than RSSs

FPQ does not exploit many properties of RSSs. There are two properties which FPQ rests upon:

- Surface areas of RSSs are well-defined and computable, enabling the SAH.
- BVs must enclose their leaves.[2]

Therefore, any two BVHs that consists of BVs with (1) well-defined surface areas and of (2) enclosing BVs, can be tested against each other with FPQ. Needless to say, BVs usually enclose geometric primitives and have well-defined surface areas. Hence, FPQ is usually applicable to any type of BVHs.

Since FPQ reduces BV tests at the price of overhead, it would be interesting to check if another type of BVH could prove more efficient for motion-planners. A traditional equation, introduced by S. Gottschalk, M. Lin and D. Manocha[37], models cost of interference detection (in this case, proximity querying) as follows:

$$T = N_v \times C_v + N_p \times C_p \tag{6.1}$$

[2]This ought to be a universal property of *bounding* volumes

where

- T: cost function for proximity querying
- N_v: number of RSS tests, i.e. number of visited intermediate BVTT nodes
- C_v: cost of a RSS test, i.e. $\|v\|$
- N_p: number of geometric primitives pairs, i.e. number of visited BVTT leaves
- C_p: cost of testing a pair of geometric primitives, i.e. $\|v\|_{leaf}$

A slightly modified version of this cost function when using FPQ is:

$$T = \frac{N_v \times C_v}{r_v} + \frac{N_p \times C_p}{r_p} + C_o, \text{ with } r_v, r_p \geq 1 \tag{6.2}$$

where $r_v \geq 1, r_p \geq 1$ are ratios of pruned RSS tests and triangle tests, respectively, with C_o denoting overhead cost induced by FPQ. For example, if FPQ computes half as many RSS tests and half as many triangles tests, then $r_v = r_p = 2$.

It seems plausible that a tighter BV decreases the size of some front F, in turn decreasing the C_o-term. It is difficult to argue that r_v and r_p should increase or decrease significantly with a changed type of BV. I conjecture that they remain largely the same regardless of BV type.

Now, an argument against tight and expensive BVs is that C_V becomes to large, causing $N_v \times C_v$ to become large. However, FPQ reduces the rate at which $N_v \times C_v$ grows at the cost of some overhead term that is believed to become smaller if a tighter BV type is used. Therefore, the total cost T may become smaller with BVs that are tighter than RSSs.

6.10 Parent-relative transformations

In Section 4.1.1 "Transformations are relative to parents in IPS", an optimization used in IPS was presented. The optimization reduces the number of transformation multiplications from two to one per BVTT. However, this optimization is not used in any variant of FPQ due to a possibility of numerical issues that may arise with sprouting and raising back and forth.

It is possible to use a parent-relative transformation solely when sprouting, possibly speeding up RSS tests: use Equation 4.2 per default but whenever a node is raised, revert back to using Equation 4.1, avoiding numerical issues. This could possibly improve on sprout-timings, shown in Figure 5.17, since fewer transformation multiplications are carried out during sprouting.

However, an incremental transformation $M_{1\to 2,i}$ has to be stored on BVTT nodes, increasing memory usage. All in all, it is unclear if this optimization would be beneficial when fronts are used since it may improve sprout-timings due to fewer multiplications but may decrease total timings due to increased BVTT node size.

6.11 Ethical considerations

Acceleration of proximity queries have no obvious direct ethical impact, with respect to either societal or gender considerations. However, there is a clear ecological benefit of acceleration of proximity queries — acceleration of proximity queries can reduce energy spent per proximity query, in turn reducing energy consumption of algorithms that depend on computation of separation distances. This is arguably a positive consequence.

On the other hand, efficient separation distance computations may be an enabler of real-time motion-planning for mobile robots. Therefore, acceleration of proximity queries may indirectly help advancement in the field of autonomous robotics. This field in itself is problematic due to several reasons, some of which are: (1) possibility of machines replacing humans in various industries, causing either temporary or lasting unemployment, (2) autonomous warfare robotics that in worst-case go rogue or (3) increasing difficulties in determining if it the manufacturer, the owner or the user of an autonomous robot that should be held accountable whenever something bad happens. Still, it is difficult to estimate to what extent acceleration of proximity queries contributes to these possible end.

The set of jobs that could be destroyed by autonomous robotics may, in general, be dominated by either males or females. If so, then there are ethical issues with respect to gender. I shall make no attempt at estimating if it is either males or females that to any greater extent become unemployed with advancements of autonomous robotics.

In conclusion, I believe that acceleration of proximity queries may in turn accelerate development of autonomous robotics. This latter development may in turn cause various ethical problems, some of which I have highlighted within this section.

7
Conclusion

I have investigated methods of accelerating proximity queries in a robot cell context. In particular, *separation distance* computations in between two sets of rigid bodies. Acceleration of proximity queries enables motion-planners to compute paths more rapidly. Possible benefits of sufficiently efficient motion-planners include:

- Enabling of real-time motion-planning
- Decreased down-time for humans waiting for a motion-planning process to terminate
- Scenes which were previously considered intractable may become tractable

I found that performance of proximity querying as a subroutine of sampling-based motion-planners can be increased by up to 46% in a real-world robot-cell, in comparison to the well-regarded *Proximity Query Package* (PQP), by using *priority directed search* (PDS) on several BVTTs. The performance gain stems from reducing the number of *bounding volume* (BV) tests by approximately 67%. In addition, PDS scales better with respect to increasing triangle counts than PQP when coherence is low.

Front tracking (FT) reduces BV tests on highly coherent scenarios and in turn reducing computation time by up to 5.5% in comparison to IPS. FT introduces overhead due to raising and sprouting operators. This overhead is significant on low-coherence scenarios. Further, overhead associated with FT may become significant even for highly-coherent scenarios. I show that FT may decrease performance by up to 45% for another coherent scenario. In conclusion, FT is capable of increasing performance marginally but it may hurt performance severely.

A specialized priority-queue called *duplex priority queue (DPQ)* was developed. It improves performance by approximately 14% for a motion-planning scenario in comparison to using a standard priority queue implementation available in the C++ standard library.

In conclusion, using a DPQ on a forest of BVTTs is a competitive approach to proximity querying as a subroutine of sampling-based motion-planners.

7.1 Future research

This section briefly presents some deprioritized ideas which could either improve or generalize FPQ. Some ideas for further measurements are presented too.

7.1.1 Front memoization

A, to the best of my knowledge, novel idea which could ameliorate situations where $\|q_i - q_{i-1}\| < d$ is close to d, i.e. where temporal coherency is relatively low, is to associate every i:th front with q_i. A previous BVTT front could be re-used if q_i is sufficiently close to some previous configuration point, otherwise a new front is built. This would require a nearest neighbour search in $\mathcal{C}$-space and memory for storing all previous BVTT fronts. However, $\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2 \times \ldots \times \mathcal{C}_n$ can be of high dimension. It has been showed a sequential scan is often faster than space-partitioning methods for $d > 10$[38].

A simple way to reduce the memory consumption of memoizing BVTT fronts is to limit the number of saved BVTT fronts. This could be done by deleting a front in a FIFO-manner or deleting a front with lowest access-frequency, or some merge thereof.

Hence, a simple spatial memoization of fronts could be made by storing k fronts in a FIFO-manner and finding the closest k:th previous q if $|q_i - q_{i-1}| > d$ for some fixed d. It is unclear if the cost associated with this technique is smaller than the gain from having a better BVTT front in low-coherent scenarios.

A possible extension of this technique is to avoid a point-location search throughout $\mathcal{C}$-space by letting a motion-planner actively select what front to use. If a motion-planner knows that points are generated from some j:th neighbourhood of $\mathcal{C}$-space, then the motion-planner could select the j:th front, effectively emulating coherence from a FPQ point-of-view.

7.1.2 FPQ for other problems within computational geometry

A brief discussion on applicability of FPQ to other problems within computational geometry follows. It would indeed be interesting to investigate how FPQ compares with existing methods in other domains of computational geometry.

Collision detection

FPQ could be transformed into a method for collision detection, by replacing instances of $\|v(q_i)\|_{leaf}$ by a boolean function $\|v(q_i)\|_{col}$ which returns true if the underlying geometric primitives of the BVTT node v collides, or false if the underlying geometric primitives does not collide. FPQ would answer the decision problem which yields true if there exists a collision in a forest of BVTTs or false otherwise.

Instead of sorting a priority queue on distances, the priority queue could be sorted on BV penetration distances. However, it is unclear if this method is feasible since penetration distances can take $\mathcal{O}(n^2)$ time even for convex geometries[39]. For guiding a search throughout a forest of BVTTs, it is not a strict requirement to have exact penetration distances. Therefore, approximate methods for penetration distance computation could be used instead.

FPQ would guide a search in a forest of BVTTs in a completely analogous fashion as with FPQ for distance computations.

Tolerance verification

Tolerance verification is a decision problem which yields true if some given geometries are at least δ apart [3][11]. Minor changes can be introduced to FPQ, using PDS, for it to handle tolerance verification:

- Let $\|v(q_i)\|_{tol} = \|v(q_i)\| - \delta$ and replace all instances of $\|v(q_i)\|$ by $\|v(q_i)\|_{tol}$.
- Instead of checking if some distance is smaller than ϵ, check if $\|v(q_i)\|_{tol}$ is positive (farther than δ apart) or negative (closer than δ).
- If some leaf $l \in L$ satisfy $\|l\|_{tol} < 0$, return false.
- If some leaf l has negative distance during sprouting, move all elements within S to SF and return false.
- If a BVTT v with $\|v\|_{tol} > 0$ is popped during sprouting, then no node may have a negative tolerance and then early termination, akin to what is described in Section 3.3.3 "PDS allows for early traversal termination", is possible.

These minor changes (along with possibly other minor changes) makes FPQ solve the tolerance verification decision problem.

7.1.3 A safe ϵ revisited

It would be interesting to further investigate, possibly existing, methods for computing Equation 3.11. If Equation 3.11 could be used to find a safe ϵ which is occasionally smaller than ϵ post ϵ-initialization, then both FPQ and IPS could perform fewer BV-BV computations.

7.1.4 More measurements

There are many possible measures which may provide further insights into FPQ. A brief list of measures which would be interesting to look at is presented below:

Size of front partitions per iteration: Observing how the sizes of SF, RF and L evolve over time could provide new insights into their interplay.

Time consumption by sprouting, raising, RSS tests and others: A deeper investigation on time consumption of front-operators and other parts of FPQ would be interesting, since it could direct further developments of FPQ to be precise and well-motivated.

Distance between a front F to an optimal front F^* over time: A distance between two fronts could be defined as $\|F - F^*\| = |F| - |F^*|$. The problem of front tracking is then the problem of minimizing $\|F - F^*\|$. By inspecting this metric, various raising techniques or reset strategies could be more accurately evaluated.

Index

References

[1] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast Proximity Queries with Swept Sphere Volumes", *technical report of Department of Computer Science, UNC Chapel Hill*, pp. 1–32, 1999, ISSN: 1050-4729. DOI: 10.1109/ROBOT.2000.845311.

[2] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha, "Fast distance queries with rectangular swept sphere volumes", *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 4, pp. 3719–3726, 2000, ISSN: 1050-4729. DOI: 10.1109/ROBOT.2000.845311. [Online]. Available: http://ieeexplore.ieee.org/document/845311/.

[3] S. A. Ehmann and M. C. Lin, "Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition", *Computer Graphics Forum*, vol. 20, no. 3, pp. 500–511, Sep. 2001, ISSN: 0167-7055. DOI: 10.1111/1467-8659.00543. [Online]. Available: http://doi.wiley.com/10.1111/1467-8659.00543.

[4] M. Lin and J. Canny, "A fast algorithm for incremental distance calculation", *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, vol. 2, no. April, pp. 1008–1014, 1991, ISSN: 10504729. DOI: 10.1109/ROBOT.1991.131723. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=131723.

[5] M. C. Lin, "Efficient Collision Detection for Animation and Robotics", PhD thesis, University of California, Berkeley, 1993. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.1507&rep=rep1&type=pdf.

[6] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space", *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 193–203, Apr. 1988, ISSN: 0882-4967. DOI: 10.1109/56.2083.

[7] J. M. Lien and N. M. Amato, "Approximate convex decomposition of polyhedra and its applications", *Computer Aided Geometric Design*, vol. 25, no. 7, pp. 503–522, 2008, ISSN: 01678396. DOI: 10.1016/j.cagd.2008.05.003.

[8] O. M. Berezsky and O. Y. Pitsun, "Computation of the minimum distance between non-convex polygons for segmentation quality evaluation", in *Proceedings of the 12th International Scientific and Technical Conference on Computer Sciences and Information Technologies, CSIT 2017*, vol. 1, 2017, pp. 183–186, ISBN: 9781538616383. DOI: 10.1109/STC-CSIT.2017.8098764.

[9] C. Lauterbach, Q. Mo, and D. Manocha, "gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries", *Computer Graphics Forum*, vol. 29, no. 2, pp. 419–428, May 2010, ISSN: 01677055. DOI: 10.1111/j.1467-8659.2009.01611.x. [Online]. Available: http://doi.wiley.com/10.1111/j.1467-8659.2009.01611.x.

[10] UNC Gamma research group. (1999). Pqp, [Online]. Available: http://gamma.cs.unc.edu/SSV/ (visited on 10/31/2017).

[11] A. Hernansanz and X. Giralt and A. Rodriguez and J. Amat, "RPQ: ROBOTIC PROXIMITY QUERIES - Development and Applications", in *Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics*, SciTePress - Science, 2007, pp. 59–66, ISBN: 978-972-8865-82-5. DOI: 10.5220/0001629100590066. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0001629100590066.

[12] R. Weller and G. Zachmann, "Inner Sphere Trees for Proximity and Penetration Queries", *2009 Robotics: Science and Systems Conference (RSS)*, vol. c, pp. 151–159, 2009, ISSN: 2330765X. DOI: 10.1145/1581073.1581097. [Online]. Available: http://cg.in.tu-clausthal.de/research/ist.

[13] M. Kaluschke, U. Zimmermann, M. Danzer, G. Zachmann, and R. Weller, "Massively-Parallel Proximity Queries for Point Clouds", in *Virtual Reality Interactions and Physical Simulations (VRIPhys)*, J. Bender, C. Duriez, F. Jaillet, and G. Zachmann, Eds., The Eurographics Association, 2014, pp. 19–28. DOI: 10.2312/vriphys.20141220.

[14] J. T. Klosowski, "Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments", PhD thesis, State University of New York at Stony Brook, 1998.

[15] M. a. Otaduy and M. C. Lin, "CLODs: Dual hierarchies for multiresolution collision detection", *SGP '03 Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pp. 94–102, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=882382.

[16] M. Tang, D. Manocha, and R. Tong, "Multi-core collision detection between deformable models", in *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling on - SPM '09*, New York, New York, USA: ACM Press, 2009, p. 355, ISBN: 9781605587110. DOI: 10.1145/1629255.1629303. [Online]. Available: http://doi.acm.org/10.1145/1629255.1629303%20http://portal.acm.org/citation.cfm?doid=1629255.1629303.

[17] ——, "MCCD: Multi-core collision detection between deformable models using front-based decomposition", *Graphical Models*, vol. 72, no. 2, pp. 7–23, Mar. 2010, ISSN: 15240703. DOI: 10.1016/j.gmod.2010.01.001. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1524070310000020.

[18] T.-Y. Li and J.-S. Chen, "Incremental 3D collision detection with hierarchical data structures", *Proceedings of the ACM symposium on Virtual reality software and technology 1998 - VRST '98*, pp. 139–144, 1998. DOI: 10.1145/293701.293719. [Online]. Available: http://portal.acm.org/citation.cfm?doid=293701.293719.

[19] O. Tropp, "Temporal coherence in collision detection for bounding volume hierarchies", *International Journal on Shape Modeling*, vol. 12, no. 2, pp. 159–178, 2006. DOI: 10.1142/S0218654306000883.

[20] M. Tang, H. Wang, L. Tang, R. Tong, and D. Manocha, "CAMA: Contact-aware matrix assembly with unified collision handling for GPU-based cloth simulation", *Computer Graphics Forum*, vol. 35, no. 2, pp. 511–521, 2016, ISSN: 14678659. DOI: 10.1111/cgf.12851.

[21] T. Larsson and T. Akenine-Möller, "A dynamic bounding volume hierarchy for generalized collision detection", *Computers & Graphics*, vol. 30, no. 3, pp. 450–459, Jun. 2006, ISSN: 00978493. DOI: 10.1016/j.cag.2006.02.011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0097849306000689.

[22] Xinyu Zhang and Y. J. Kim, "Scalable Collision Detection Using p-Partition Fronts on Many-Core Processors", *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, pp. 447–456, Mar. 2014, ISSN: 1077-2626. DOI: 10.1109/TVCG.2013.239. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/24434225%20http://ieeexplore.ieee.org/document/6620867/.

[23] S. Ehmann and M. Lin, "Accelerated proximity queries between convex polyhedra by multi-level Voronoi marching", in *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*, vol. 3, 2000, pp. 2101–2106, ISBN: 0-7803-6348-5. DOI: 10.1109/IROS.2000.895281. [Online]. Available: http://ieeexplore.ieee.org/document/895281/.

[24] UNC Gamma research group. (2001). Swift, [Online]. Available: http://gamma.cs.unc.edu/SWIFT/ (visited on 10/31/2017).

[25] S. Crozet. (2017). N-collide, [Online]. Available: http://ncollide.org/ (visited on 10/31/2017).

[26] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi, "I-COLLIDE: An interactive and exact collision detection system for large-scale environments", *I3D '95 Proceedings of the 1995 symposium on Interactive 3D graphics*, pp. 189–197, 1995. DOI: 10.1145/199404.199437. [Online]. Available: http://dl.acm.org/citation.cfm?id=199437.

[27] J. Pan, S. Chitta, and D. Manocha, "FCL: A general purpose library for collision and proximity queries", *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3859–3866, 2012, ISSN: 10504729. DOI: 10.1109/ICRA.2012.6225337.

[28] P. Alliez, T. Stéphane, C. Wormser, and L. Rineau. (2017). CGAL 4.11 - 3D Fast Intersection and Distance Computation (AABB Tree), [Online]. Available: https://doc.cgal.org/latest/AABB_tree/index.html#Chapter_3D_Fast_Intersection_and_Distance_Computation (visited on 01/24/2018).

[29] R. J. Trudeau, *Introduction to Graph Theory*. Mineola, New York: Dover Publications, 1994, ISBN: 0486678709.

[30] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 3rd. A K Peters/CRC Press, Jul. 2008, ISBN: 978-1568814247. DOI: 10.1201/b10644-23. [Online]. Available: http://www.crcnetbase.com/doi/10.1201/b10644-23.

[31] S. M. Lavalle, *Planning Algorithms*. Urbana, Illinois, U.S.A: Cambridge University Press, 2006, p. 842, ISBN: 9780511546877. DOI: 10.1017/CBO9780511546877. [Online]. Available: http://planning.cs.uiuc.edu/.

[32] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry*, 3rd ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ISBN: 978-3-540-77973-5. DOI: 10.1007/978-3-540-77974-2. [Online]. Available: http://link.springer.com/10.1007/978-3-540-77974-2%5Cnhttp://medcontent.metapress.com/index/A65RM03P4874243N.pdf%20http://link.springer.com/10.1007/978-3-540-77974-2.

[33] H. M. Choset, *Principles of Robot Motion: Theory, Algorithms, and Implementation*. 2005, pp. 1–161, ISBN: 9780262033275. DOI: 10.1017/S0263574706212803. [Online]. Available: http://books.google.com/books?hl=en&lr=&id=S3biKR21i-QC&oi=fnd&pg=PR15&dq=Principles+of+Robot+Motion&ots=bjuSP9WESQ&sig=nEyml1cFTZ8zC8iYezZXtQFrB44.

[34] S.-E. Yoon, B. Salomon, M. Lin, and D. Manocha, "Fast collision detection between massive models using dynamic simplification", in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing - SGP '04*, New York, New York, USA: ACM Press, 2004, pp. 136–146, ISBN: 3905673134. DOI: 10.1145/1057432.1057450. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1057432.1057450.

[35] D. MacDonald and K. Booth, "Heuristics for ray tracing using space subdivision", *The Visual Computer*, vol. 6, no. 3, pp. 153–166, 1990, ISSN: 1432-2315. [Online]. Available: http://link.springer.com/10.1007/978-3-319-01020-5.

[36] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT", in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, CA, USA: IEEE, Sep. 2011, pp. 3513–3518, ISBN: 978-1-61284-456-5. DOI: 10.1109/IROS.2011.6048813. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6048813.

[37] S. Gottschalk, M. C. Lin, D. Manocha, and C. Hill, "OBB Tree: A Hierarchical Structure for Rapid Interference Detection", *Proceedings of SIGGRAPH 96*, no. 8920219, pp. 171–180, 1996, ISSN: 00978930. DOI: http://doi.acm.org/10.1145/237170.237244.

[38] R. Weber, S. Blott, M. Ave, and M. Hill, "Similarity-Search Analysis Methods and Performance Study for in High-Dimensional Spaces", in *Proceedings of the 24th VLDB Conference*, New York, NY, USA: Morgan Kaufmann Publichers Inc., 1998, pp. 194–205.

[39] D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri, "Computing the Intersection-Depth of Polyhedra", *Algorithmica*, vol. 9, no. 6, pp. 518–533, 1993.