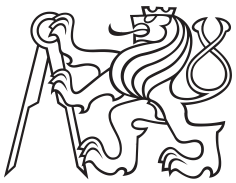
Bachelor Project



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Motion Planning for Disentanglement of Puzzles

Vojtěch Volprecht

Supervisor: Ing. Vojtěch Vonásek, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Volprecht Vojtěch** Personal ID number: **491976**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Motion Planning for Disentanglement of Puzzles

Bachelor’s thesis title in Czech:

Metody plánování pohybu pro rozkládání hlavolamů

Guidelines:

1. Get familiar with robotic motion planning, particularly with sampling-based methods, e.g. RRT and PRM [1]. Get familiar with the narrow passage problem and basic techniques to handle it [2].
2. Implement algorithm for puzzle disassembly inspired by [3].
3. Design an alternative way to detect ‘notches’ (definition in [3]) in a 3D triangle mesh.
4. Design a modification of the blooming process of [3] to enable multiple connections between the individual trees. Further, extend the blooming process using different type of random trees, e.g. [5].
5. Experimentally verify all implemented methods on the dataset of 3D objects (will be provided by the advisor), compare with suitable methods from the OMPL benchmark [4].

Bibliography / sources:

[1] LaValle, Steven M. Planning algorithms. Cambridge university press, 2006.
[2] J. Denny, R. Sandström, A. Bregger, and N. M. Amato. Dynamic region-biased rapidly-exploring random trees. In Twelfth International Workshop on the Algorithmic Founda-tions of Robotics (WAFR), 2016.
[3] Xinya Zhang, Robert Belfer, Paul G. Kry, and Etienne Vouga. 2020. C-Space tunnel discovery for puzzle path planning. ACM Trans. Graph. 39, 4, Article 104 (July 2020), 14 p. https://doi.org/10.1145/3386569.3392468
[4] Mark Moll, Ioan A. Şucan, Lydia E. Kavraki, Benchmarking Motion Planning Algorithms: An Extensible Infrastructure for Analysis and Visualization, IEEE Robotics & Automation Magazine, 22(3):96–102, September 2015. doi: 10.1109/MRA.2015.2448276.
[5] V. Vonásek and R. Pěnička, "Space-filling forest for multi-goal path planning," 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2019, pp. 1587-1590, doi: 10.1109/ETFA.2019.8869521.

Name and workplace of bachelor’s thesis supervisor:

Ing. Vojtěch Vonásek, Ph.D. Multi-robot Systems FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **24.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Vojtěch Vonásek, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Acknowledgements

Most of all I would like to thank my supervisor Ing. Vojtěch Vonásek, Ph.D. for his generous help and assistance. And last but not least, I would also like to thank my family who supported me during my studies and brought a critical insight into my thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 20 May 2022

Vojtěch Volprecht

Abstract

Path planning, also known as motion planning, is one of the most fundamental problems in robotics. This work combines both the algorithmic part of planning and the use of geometric properties of the environment. Geometric properties include those that help us solve the puzzle faster. We used heuristics to do that. After all, we configured our Rapidly-exploring random dense tree to find the path. The results show that the algorithm that uses even the slightest knowledge about the environment is better and faster than the algorithm itself without any external information.

Keywords: Euclidean-Geodesic ratio, Medial Axis, Local planners, Path planning, Planning algorithms, Probabilistic roadmaps, Puzzle path planning, Puzzle solving, Rapidly-exploring dense tree, Rapidly-exploring random tree, Skeletonization, 3D geometry

Supervisor: Ing. Vojtěch Vonásek, Ph.D.

Abstrakt

Plánování cesty, známé také jako plánování pohybu, je jedním z nejzákladnějších problémů robotiky. Tato práce kombinuje jak algoritmickou část plánování, tak využití geometrických vlastností prostředí. Mezi geometrické vlastnosti patří ty, které nám pomáhají řešit hlavolam rychleji. K tomu jsme použili heuristiku. Nakonec jsme nakonfigurovali RDT algoritmus k nalezení cesty. Výsledky ukazují, že algoritmus, který využívá i ty nejmenší znalosti o prostředí, je lepší a rychlejší než samotný algoritmus bez jakýchkoli vnějších informací.

Klíčová slova: Euclidovsko-Geodesické měřítko, Hledání cesty hlavolamu, Lokální plánování, Mediální osy, Plánovací algoritmy, Plánování cesty, PRM, RDT, RRT, Řešení hlavolamů, Skeletonizace, 3D geometrie

Překlad názvu: Metody plánování pohybu pro rozkládání hlavolamů

Contents

Figures

Tables

Chapter 1

Introduction

Motion planning is a challenge in robotics during past decades. Its fundamental problem is to find a sequence of valid configurations that moves the object from one position to the other. Another way of looking at it is the well-known *Piano Mover's Problem*, which deals with moving a piano between two rooms without crashing into a wall or other obstacles. In our case, it means to disentangle 3D rigid bodies of puzzles without colliding. These puzzles (Figures 1.1, 1.2) we solve are always two bodies, one of which is moving and the other is static.[1] Disentangle them involves sliding through narrow tunnels in the puzzles configuration space, which we will call C-space. We aim to speed up planning on these puzzles using their geometric properties to obtain the right configurations based on the location of the important narrow tunnels (Figures 1.1, 1.2).

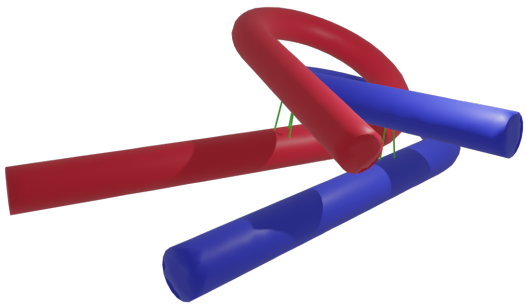


Figure 1.1: Example of the Alpha puzzle. Green lines mark the positions of our important narrow tunnels.

[1] Here and throughout the thesis, *Robot* is always considered as a moving part and *Env* as a static part.

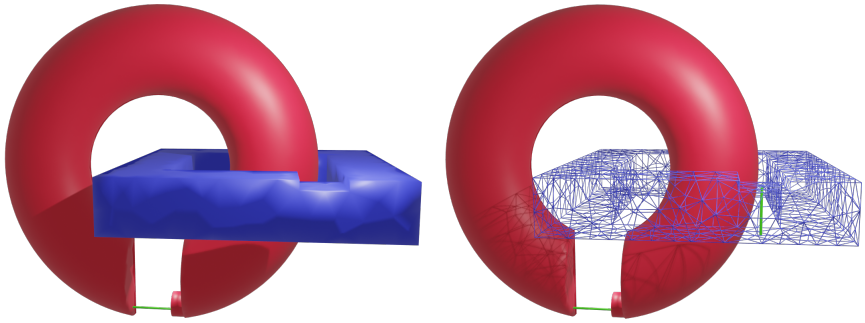


Figure 1.2: Example of the Duet puzzle. Green lines mark the positions of our important narrow tunnels. The green line in the grid of the Duet puzzle is inside the puzzle, focused in the picture on the right.

We make use of the theoretical approach from the article [23] from which we dealt with geometric heuristics such as Euclidean-Geodesic ratio (EGR) and Notches detection, later in Chapter 3.

We also propose a variant of motion planner based on Rapidly-exploring random tree (RRT) that builds roadmaps from each configuration and connects each closed pair together, described in Section 2.1.

Together with geometric procedures, we let our algorithm to solve a group of puzzles. For the overall analysis we also test our planner with the OMPL library [14], results in Chapter 5.

1.1 Insight Into The Issue

Motion planning or path planning nowadays reaches all areas where any mechanical movement needs to be planned. Whether it is the movement of a robotic arm consisting of several joints, drone moving in a building or solving a puzzle, the goal is still the same.

Before we move on, it is important to explain what C-space means and what such a configuration contains.

Similar to description from the article [23], let's imagine a simple 2D puzzle from Figure 1.3. We see a blue moving object which we call *Robot* and a red spot which will be our goal state. We consider the C-space as a unification

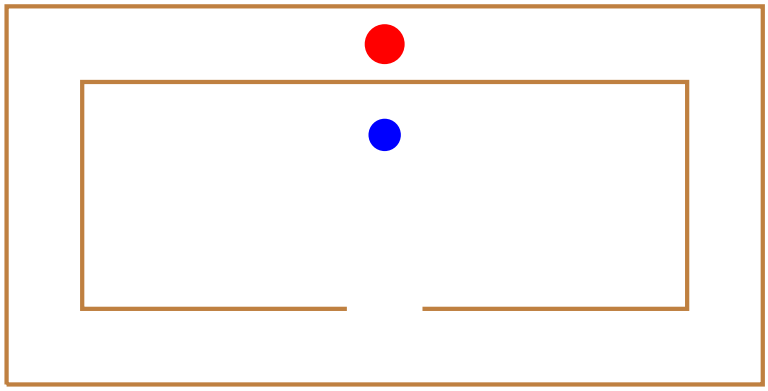


Figure 1.3: 2D puzzle with a blue *Robot* that needs to be moved forward to the red spot. C_{free} represents all white free space and C_{obst} represents every static brown wall.

of C_{free} and C_{obst} spaces. C_{free} represents all free space, including the big region where the *Robot* is initially spotted, connected via narrow passage to two chambers. That passage we call *Narrow Tunnel*. All this also includes C_{obst} as obstacles with static brown walls.

The configuration is the setting of the position (including rotation) of the *Robot* with respect to the surrounding world, i.e. C_{free} and C_{obst}. That simply means that we are looking for a places inside a C-space that are somehow important for solving our puzzle. In our case, these will be the mentioned *Narrow Tunnels*. Once we get them, we use them as members of our planning algorithm to create a whole tree that includes our collision-free path.

The main aim, as recalled above, is to get the *Robot* on the red spot. From our human view, it can be seen that the *Robot* has only one option and it is to get through the *Narrow Tunnel*. Any planning algorithm has to deal with that knowledge.

The cornerstone for path planning is the sampling of the given C-space. For such purposes there are many variants of sampling based algorithms. Some of the basic algorithms are Probabilistic roadmaps (PRM) and Rapidly-exploring random tree (RRT). More about them in the next chapter.

The main problem with motion planning is dealing with the space-dimension. If we imagine sampling 2D space, we got two axes: x and y. In addition, we have one rotation degree. If we had a problem like in Figure 1.3 with the *Robot* and the strategy like "up, down, up", it would be completely different from "down, up, up". After few steps, the number of possibilities

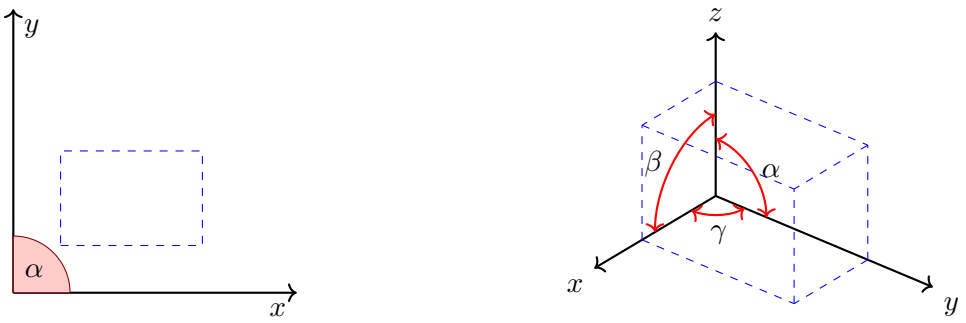


Figure 1.4: The blue object representing ***Robot*** with translation over $\boldsymbol{x}$ or $\boldsymbol{y}$ axis and with opportunity to rotate over $\boldsymbol{\alpha}$ angle (*left*). The same representing but in 3D with the addition of one axis and two angles (*right*).

grows exponentially. Unlike to human perspective, it is unclear for planner which samples are better than the others, so the planner has to evaluate them all. Moreover in 3D space, we have now three axes: x, y, z and beside that, 3 angles to care about. That is now 6-dimensional C-space problem (Figure 1.4). Further explanation can be taken from the article [16], where the complexity of motion planning is described as PSPACE-hard.

Even if we sample the configurations right into the tunnel(s) with probability p and the number of tunnels which the path goes through is N, then the probability of finding a path that leads through the tunnels we found is p^N. Simply said, sampling C-space without taking the C_{free} structure into consideration will not be enough to find the proper path.

1.2 The Challenge

The goal of our bachelor thesis, as mentioned in the introduction, is to implement theoretical approach of the article [23], focused on improvement of the planning algorithm with heuristics using geometric properties of our puzzles.

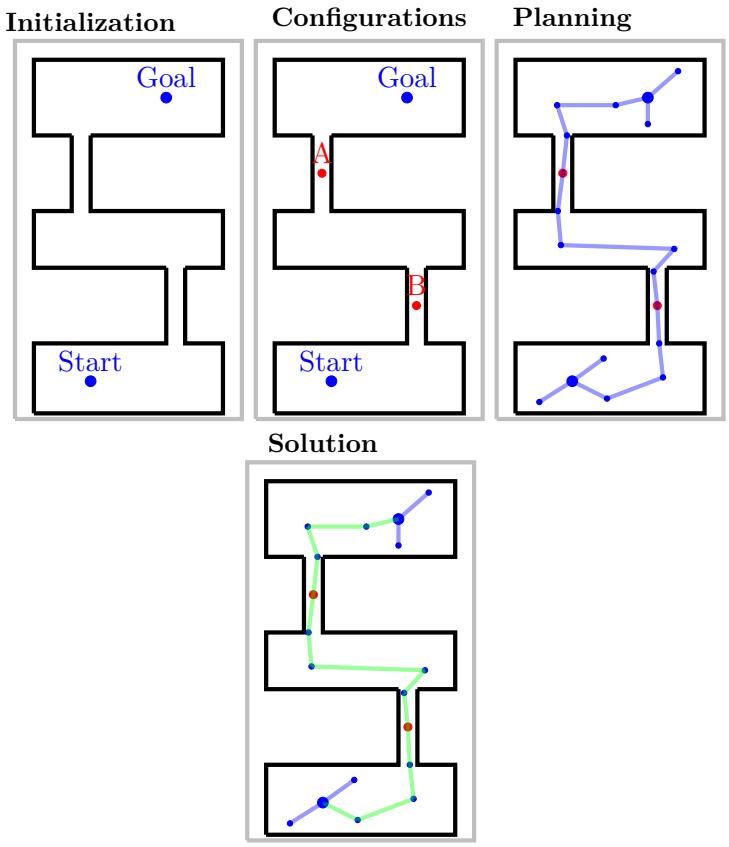


Figure 1.5: Pipeline of our implemented procedure simplified to 2D space. Initialize Start and Goal, find all configurations (A,B), run a planner, find path. A and B represent the important narrow tunnels we ought to find.

In our work we present a procedure, illustrated in Figure 1.5, how to find such important configurations through two geometric heuristics, which are EGR and Notches detection. For visualization, such configuration points in puzzles will be e.g. the center points in place of the green pipes from Figures 1.1 and 1.2. We further process such important configurations as roots of expanding trees, which sample our C-space. We provide the advantages and disadvantages of using several simple planning algorithms along with the reason why they cannot solve even the most basic puzzle. For this reason we will present a functional modification of a simple expanding tree.

Chapter 2

Related Work

By reviewing the latest works, sampling valid configurations in important positions of C-space, narrow tunnels, is still remaining a big challenge. No wonder that the state-of-the-art path planners depend on the quality of sampling. In recent years, sampling based algorithms have shown great advantages in terms of their flexibility in higher dimensions and at the same time their computational efficiency.

Since the geometrical structure of the environment can be very complex, its analysis may not always be directly given. Beside robotics the problem of narrow passages can be also dealt in molecular protein dynamics in biology [17] as illustrated in Figure 2.1. Furthermore this problem is extended by the movement in the time frames. That means the tunnels need to be detected in the right order in right time. To deal with this problem, they proposed a modification of Rapidly-exploring random tree (RRT) together with three new features. It needs to handle false detection of sampling outside of proteins. On the other hand it expands more new nodes in each iteration to boost expansion of right tunnels and at last it uses modification from Voronoi-Diagram [13] based sampling of the configuration space.

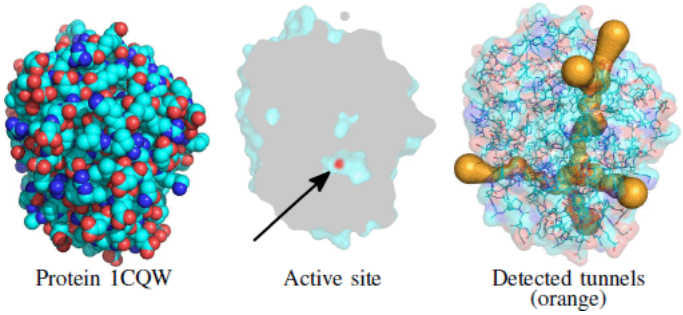


Figure 2.1: Example of the tunnel detection in haloalkane dehalogenase. Image taken from the article [17].

But apart from planning in a complex environment, we can look for a path with constraints in a robot. A typical example in robotics is the path planning under kinematic constraints [5]. The robots constraints include in particular the limitation of its movement. Among the popular robots are certainly humanoid robots [2] (Figure 2.2) whose main limitation are the joints in arms, which are supposed to symbolize the most natural human movement.

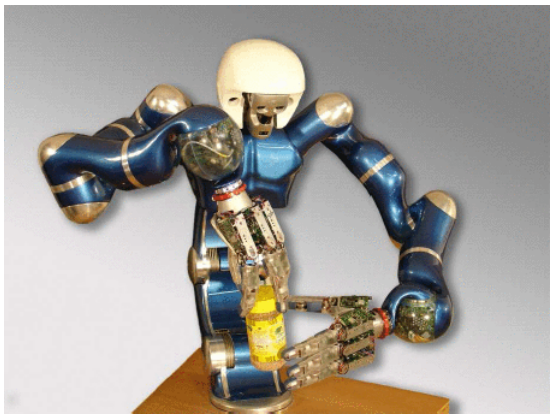


Figure 2.2: Humanoids upper body system: Justin. Image taken from the article [2].

Thanks to the huge interest of motion planning in robotics, our planning itself was inspired by several sources, but mostly from Steven M. Lavalle [12]. We took the inspiration especially from Chapters: *Geometric Representations and Transformations*, *Sampling-Based Motion Planning* and *The Configuration Space*.

2.1 Planning algorithms

Because we are trying to find our way in motion planning without colliding into an obstacle, we have to define some sampling based planners that efficiently explore C_{free}. Finding path can be solved with any sampling based algorithm e.g. Probabilistic roadmaps (PRM) [10] [12] or Rapidly-exploring random tree (RRT) [9] [12]. All sampling-based algorithms are based on constructing a graph containing collision-free paths, but many of them differ in the way they construct the graph.

This section will introduce a few of the sampling based algorithms, their pros and cons in relation to our problem, but what is the most important, we will also present derived planner from RRT called Rapidly-exploring dense tree (RDT) [12].

The most basic planner PRM [10] is a procedure divided into two phases (see Figure 2.3). Firstly it samples whole C_{free} with points trying to connect them to each other in a predetermined distance. That builds a roadmap constructed as the graph. In the second phase, the start and goal configurations are connected to the roadmap using their nearest neighbors. Then, any graph-search method (DFS [21], BFS [20]) tries to find a path in the roadmap. We immediately realize that the problem with this approach is that the probability of a sampled point in a narrow passage is very low. On the other hand, PRM performs quite well in high-dimensional spaces [10].

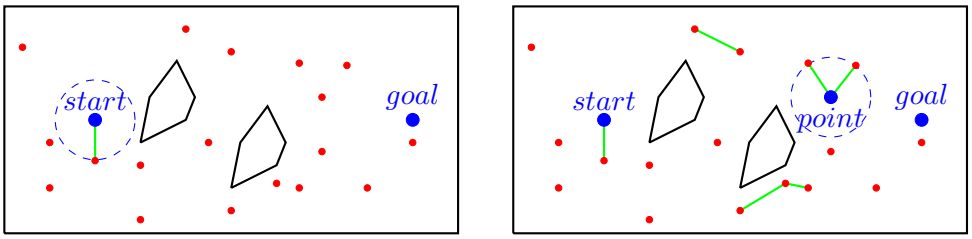


Figure 2.3: PRM planner: After sampled the whole C-space (*left*), we try to connect every single point to each other within a particular distance (*right*).

The most developed and researched planner is RRT [9]. Together with its various modifications it has found a great application not only in robotics [17], [3], [2]. This algorithm, illustrated in Figure 2.4, is based on expanding tree as much as possible. That means we root a tree in a start position and begin sampling the C-space. Every collision-free sample is attempted to connect to the tree. If there exists any collision-free path between the sample and its nearest neighbor in the tree, sample is joined to the tree. Eventually, when sampled point is near the goal, the procedure ends. Many other similar algorithms are based on this principle. For example, RRT-connect [11], which

roots two trees at the beginning, one in start and the other in goal and grow them with greedy heuristics towards each other. Or advance smoothed RRT techniques for kinematic car-like robot trajectory planning [7].

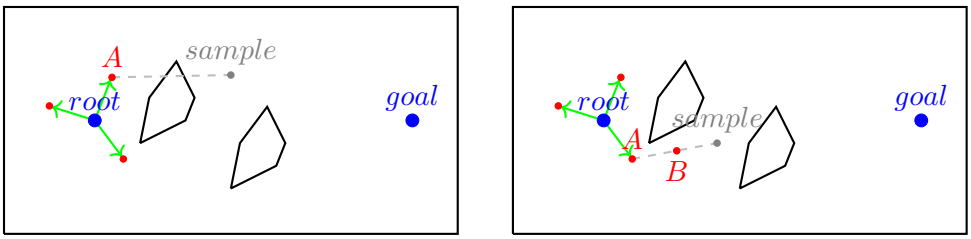


Figure 2.4: RRT planner: wrong sample because of collision on the trajectory (*left*), right expansion to the direction of a sample within the distance of a single unit (*right*).

Like every other planner, even RRT and PRM have their own improved variants. It is proven that both of them are not asymptotically optimal and therefore, in order to reduce the limitation, new algorithms were proposed [8], i.e. PRM* and RRT*. Proven to be probabilistically complete and asymptotically optimal. For a planner to be asymptotically optimal, it means that the cost of its found path converges almost to the optimum.

The RRT modification we used and was crucial for solving our puzzles is called RDT [12], which is listed in Algorithm 1. This modification brings one major change and that are the limitations for expanding its tree. In particular, there are two constraints. Instead of sampling $q_n \in C_{space}$ from q_0 till reaching q_{goal}, the RDT repeats sampling until it reaches $\mathcal{G}.dense$ of user defined size of the tree $\mathcal{G}$ or exceeds the predefined expansion *time*. This guarantees that the algorithm will end at some point.

Algorithm 1 Rapidly-Exploring Dense Tree

```
1:  function EXPANSION_RDT(q_0)
2:      G.init(q_0);
3:      while !G.dense OR !time do
4:          q_new ∈ C_space
5:          q_n ← NEAREST(G,q_new)
6:          G.add_vertex(q_new);
7:          G.add_edge(q_n, q_new);
8:          if G.close(q_n, q_goal) then break;
9:          end if
10:     end while
11: end function
```

Chapter 3

Geometric features

As said in the introduction, the goal is to locate narrow tunnel(s), because we assume they are necessary for solving such a puzzle. It is not a surprise that searching for the configurations has something to do with locating narrow tunnels. In our thesis we focus on creating the best possible configurations right inside the narrow tunnels. The important fact to care about is also that narrow tunnels correspond to the alignment of geometric features. Thanks to this we can look for connections between the geometry and the narrow tunnels.

As said above, the source of the geometric ideas are from the article [23] from which we present two geometric heuristics for detection of important configurations, narrow tunnels. Let's imagine our puzzles from Figures 1.1 and 1.2, we presume that narrow tunnels are either two gaps facing each other (that is the case of Alpha puzzle) or a gap aligned with a notch (that is the case of Duet puzzle). We will be dealing especially with these two kinds of features called gaps and notches.

3.1 Gaps Detection

We imagine that the gap on the puzzle is a narrow space, which is closed by the opposite parts of the given puzzle. Illustration of the gap on the Alpha puzzle is in Figure 3.1. We present EGR as a good candidate to detect the gaps.

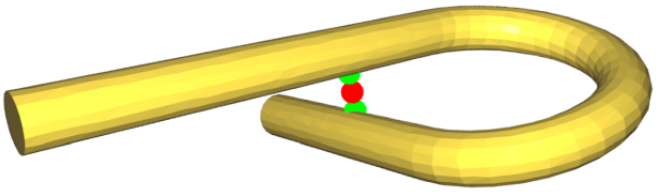


Figure 3.1: The red dot marks the midpoint of the found gap. It will also be paired against the other piece of alpha puzzle to construct a key configuration. Green pairs are minimized from EGR equation to construct particular red dot. Image taken from the article [23].

Let's have the surface of the puzzle formed as a triangulated mesh $\mathcal{M}$ of 3D points. The idea of how to find gaps is that we find pairs of points on the surface of a puzzle that form a local minimum of the EGR equation. It combines the Euclid distance to hold the pair close in the Euclidean space, while Geodesic ratio will also respect the shortest distance on the surface of the puzzle, illustrated in Figure 3.2.

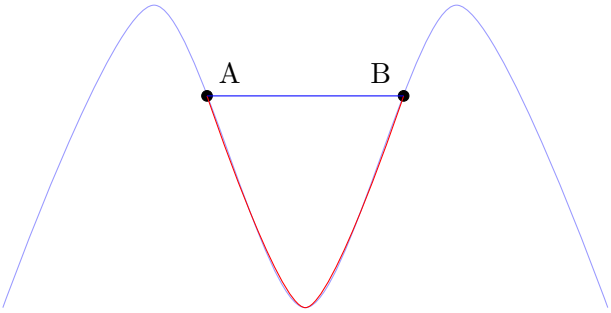


Figure 3.2: Red line represents Geodesic distance from **A** to **B** while blue line represents Euclidean distance.

More specifically, we will search for those pairs of point $\mathbf{u}$,$\mathbf{v} \in \mathcal{M}$, which meet the equation

$$r(\mathbf{u}, \mathbf{v}) = \frac{e(\mathbf{u}, \mathbf{v})}{g(\mathbf{u}, \mathbf{v})} + \alpha \cdot e(\mathbf{u}, \mathbf{v}), \tag{3.1}$$

where the function r represents our distance ratio we want to minimize. The function e is the Euclidean distance and g is the Geodesic distance (Figure 3.2). All those functions are computed above points $\mathbf{u}$ and $\mathbf{v}$ from the puzzle's surface mesh $\mathcal{M}$. In the equation is also α bias, that overestimates the Euclidean distance. That helps to eliminate the pairs, which are close, but do not form useful features. It is e.g. pair, each point on the other side of the edge or corner.

Implementation. We followed the procedure from the article [23], see Algorithms 2 and 3. We initially select a random vertex $\mathbf{u}$ and a random

vertex **v** from $\mathcal{M}$. They should not be neighbors, because rest of the procedure would not make sense. We need to minimize the function r so we iteratively improve ($\mathcal{M}.improve()$) our selection by moving each either **u** or **v** to adjacent vertex until it lowers its value. We do this procedure for user-defined number N of pairs. We then reject ($FILTER()$) duplicate pairs and the pairs that include neighbouring vertices. What can also occur is that the pair consists of vertices whose line intersects with the puzzle. Those we need to get rid of as well.

We found out that it helps if we run the algorithm described above three times, to get three independent groups and find the intersection points among them ($INTERSECT()$). We say two pairs are the same if the position difference between them is lower than h, where h is a user-defined constant (we used 1 as a normalized unit of the surrounding world). The intersected pairs are used as a source for the configuration points, which are constructed as midpoints of each pair as illustrated in Figure 3.1.

Algorithm 2 EGR

```
1: function GET_GAPS(M)        ▷ M represents puzzle's mesh of points
2:     G = ∅;
3:     for k ← 1 to 3 do
4:         G.insert(GET_PAIRS(M));
5:     end for
6:     G ← INTERSECT(G))
7:     return G.midpoints();      ▷ midpoints represent the configurations
8: end function
```

Algorithm 3 EGR

```
1: function GET_PAIRS(M)
2:     P = ∅;
3:     for k ← 1 to N do
4:         u, v ← M.random()
5:         u, v ← M.improve(u, r)          ▷ r represents our Equation 3.1
6:         P.insert(u, v);
7:     end for
8:     FILTER(M, P))                        ▷ rejection of unused pairs
9:     return P;
10: end function
```

What we also tried, and it is also suggested in the article [23], is to select the second point at the beginning locally not globally. This means that we choose the first point for the pair randomly, but the second one within a predefined Euclidean distance. This will minimize the function r just in the selected area. This, together with α bias, creates a great opportunity for experimentation. From our experience we mainly chose 0.5 for α bias, when we did global search and clear zero for local search. Both tries created

moreover the same configuration points. We came to the conclusion that it eventually depends on the quality of the random selection.

3.2 Notches Detection

The second feature we dealt with were the mentioned notches. However, EGR does not find notches so reliably so we cannot completely rely on it. The article [23] mentions a better idea how to detect these notches through medial skeleton [19]. The procedure is as follows. We find a medial skeleton of the puzzle through computing its medial axis [18]. We look for the points on the skeleton, which have the smallest radius to their neighbors on the surface. We hope that when we find such a point, it will indicate a notch in that area.

We also tried this approach. To find the skeleton and the medial axis we used the CGAL library [15]. Unfortunately from our own testing, the skeleton was just an approximation and the results were worse than we had expected and could work with. We believe that the skeleton approximation was not good enough because of the irregular distribution of the vertices of the mesh of the given puzzle.

Figure 3.3: Duet puzzle (*left*), focus on the notch and knot (*right*).

We had to come up with our own solution, Algorithm 4. We mainly dealt with finding notches on the grid of the Duet puzzle (Figure 3.3), where we took advantage of the properties that this puzzle is mostly regular.[1] Mainly because the notches are planar and therefore have a certain surface area. As it was said before, a notch is aligned with a gap. So if we take advantage from the previous knowledge about gaps, it is clear that we are going to look for the gaps on the knot and the notches on the grid. Our assumption from the geometry idea is that such a notch is located between two parallel planes (Figure 3.4). The configuration points will be then in the middle of these planes (Figure 3.9).

[1] To be clear from illustrated Figure 3.3, knot represents the part that looks like banana and grid the one with notch.

Implementation. Similarly to the gaps detection, let's have the surface of the puzzle formed as a triangulated mesh of points. To construct the planes we need to select the points from the mesh, which are not on the edges as illustrated in Figure 3.5. Because we know that points lying on the edge always belong to two or more planes. We will not continue to work with these points. Without these points we can easily create our desired planes. Specifically we create a plane from three adjacent vertices and then we mark every other adjacent vertex as lying on our plane if its distance to the plane did not exceed a certain threshold. Simply, a vertex whose neighbors do not lie on the same plane can be marked as one lying on the edge. It is important to note that this can only be done if the whole mesh is structured as a graph. Means, that every vertex is connected to its neighbors.

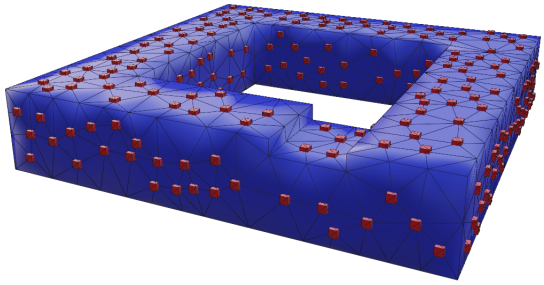


Figure 3.5: All plane points without those on the edges. The small red cubes represent the vertices on the triangulated mesh.

Algorithm 4 Notches detection

1: **function** GET_NOTCHES($\mathcal{M}$) ▷ $\mathcal{M}$ represent the puzzle's triangulated mesh
2: $\mathcal{M}, \mathcal{P}, \mathcal{C}, Pairs = \emptyset$;
3: $\mathcal{M}.filter\_edges()$ ▷ we reject vertices lying on the edges
4: $\mathcal{P} \leftarrow$ GET_PLANES($\mathcal{M}$)
5: $\mathcal{C} = \mathcal{P}.centers()$; ▷ centers of each plane
6: $Pairs \leftarrow$ PROJECT($\mathcal{C}$, $\mathcal{P}$) ▷ each plane projects it's center onto each parallel plane
7: **return** $Pairs.midpoints()$; ▷ midpoints represent configurations
8: **end function**

Once we have such planes, we are able to determine their mutual dependence. Because we can simply compute normal vectors from the vertices of our planes, we can say that two planes are equal if one normal vector is a multiple of the other. Specifically, we have two normal vectors **a** and **b** of two planes. We say that they are equal if the ratio of these normal vectors along the coordinates is equal as well, following the Equation 3.2.

$$\frac{a_1}{b_1} = \frac{a_2}{b_2} = \frac{a_3}{b_3}. \tag{3.2}$$

We cannot be completely precise with our equation, because there can be situations such as one vector being (0.001, 1, -0.001) and the other (0, 1, 0) although we want them to be exact. We did this calculation by rounding **r** the normal vectors to two decimal places and at the same time we weighted the division by the **threshold**, following the Equation 3.3.

$$\begin{aligned} \left|\frac{r(a_1)}{r(b_1)} - \frac{r(a_2)}{r(b_2)}\right| &\leq threshold \\ \left|\frac{r(a_1)}{r(b_1)} - \frac{r(a_3)}{r(b_3)}\right| &\leq threshold. \end{aligned} \tag{3.3}$$

Thanks to the fact that notch is a kind of a cutout in the body of the puzzle, it has its own plane. Lets have an arbitrary plane parallel to that notch plane. When constructing such a plane we have all the points lying on this plane at our disposal. Let us construct a bounding box in the shape of a rectangle, which will minimize the space containing all these points. The center of that bounding box should be approximately in the same place as the center of its plane (Figures 3.6 and 3.7). These centers are very important for us, because they form one part of pairs, that will be necessary for creation of our configurations.

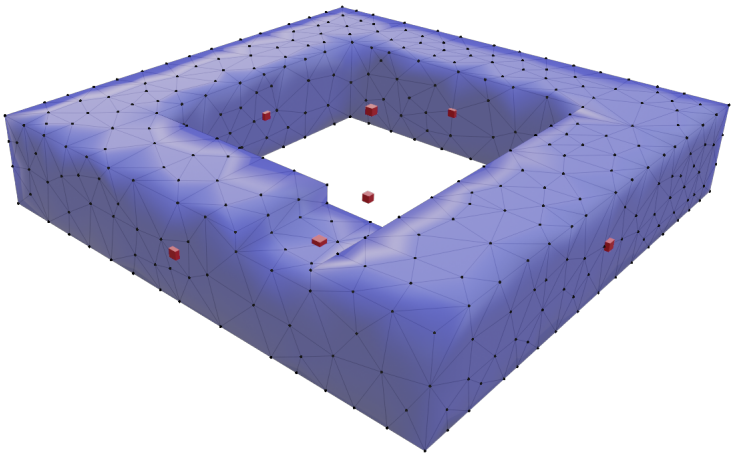


Figure 3.6: Centers of the planes.

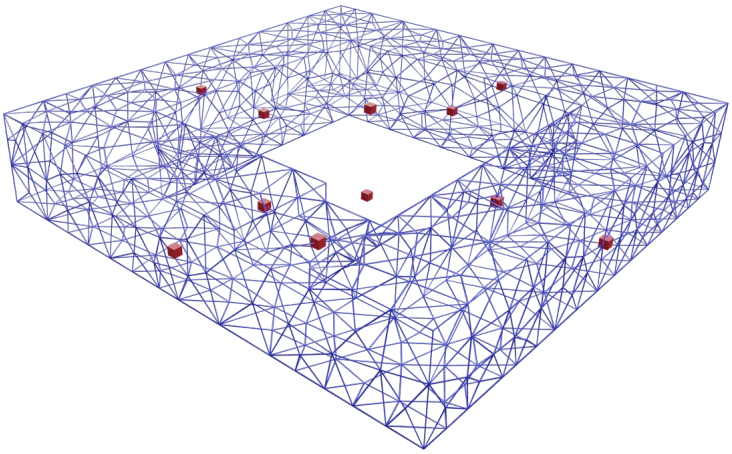


Figure 3.7: Detailed centers of the planes.

We need to project each one of the centers onto all it's parallel planes and obtain their projections (Figure 3.8). If we remember the configuration points from the EGR ratio, Section 3.1, we took the centers of the gaps. Here again we take the centers, but of our projections (Figure 3.9).

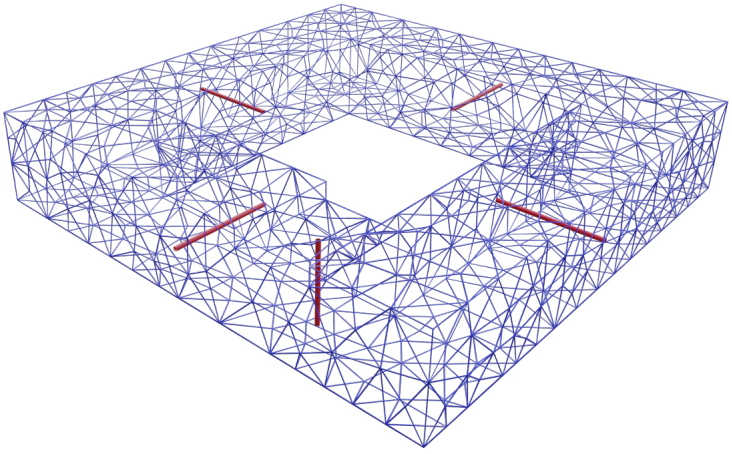


Figure 3.8: Filtered projections of each center onto the parallel plane.

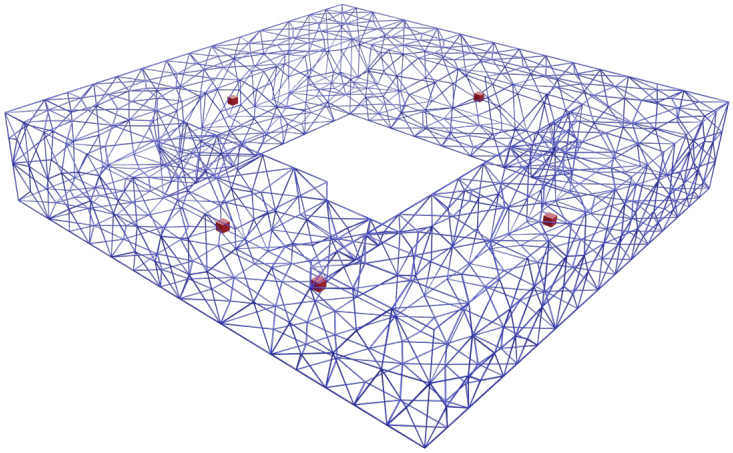


Figure 3.9: Final configuration points.

However, it immediately occurs that we create far more points than we are interested in when we make projections between all parallel planes. Nevertheless, we don't mind such points, because we are mainly interested in having the points there at all. It is important to have the center inside the cutout. The only disadvantage is an increase in time effort. On the other hand, we can help ourselves a little here by filtering out the projections whose centers are outside the body. Because from the assumption that notch is aligned with a gap the configuration points must always be inside the body.

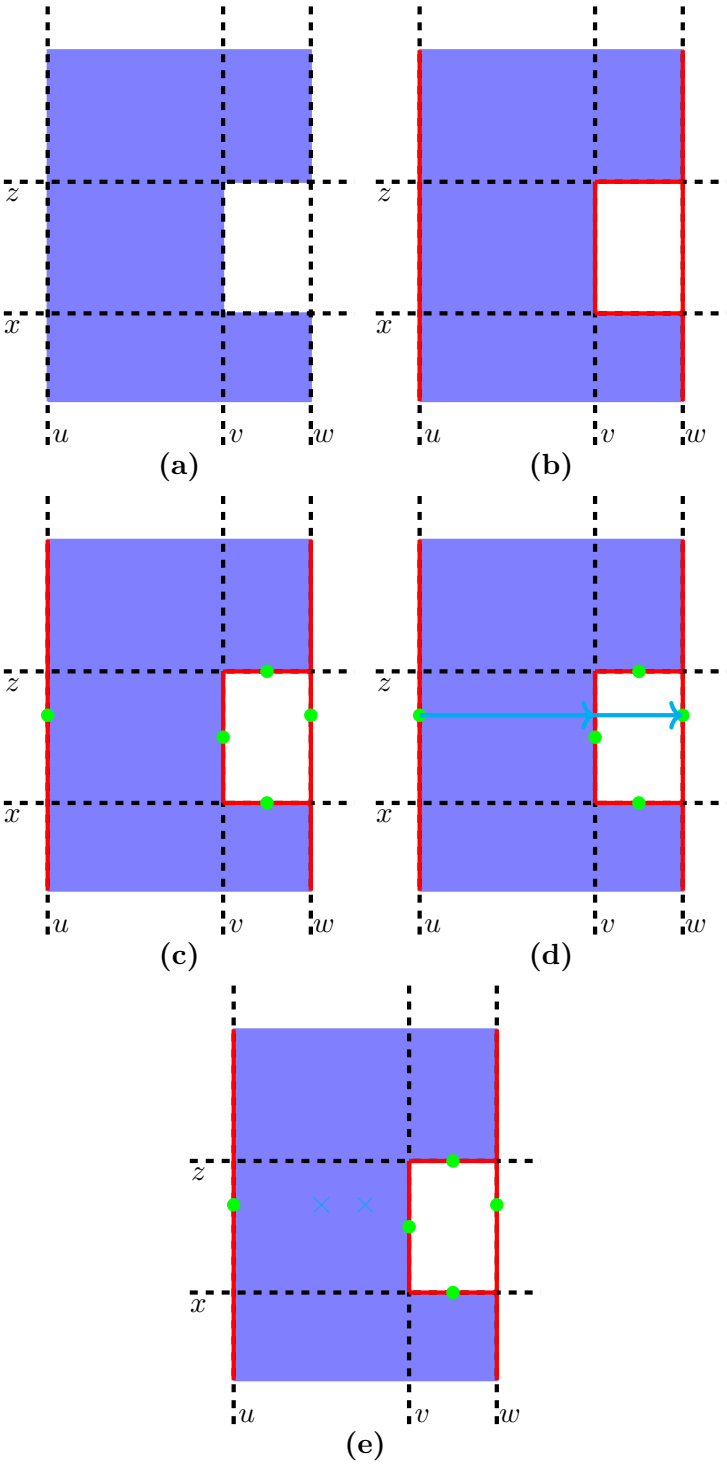


Figure 3.4: A simplified side view of the Duet puzzle, **(a)** focused on the notch. All parallel planes (u, v, w, x, y) are marked as a dashed line. It describes the procedure of finding the configuration points inside the notch. The procedure starts in the upper left picture as follows. Find all the parallel planes. Create their bounding box symbolizing the red trace **(b)**. We find their centers **(c)**, which we then project from each plane onto their parallel planes **(d)**. Finally, the configuration points lie in the middle of each projection, on the positions of blue crosses **(e)**.

Chapter 4

Approach

4.1 Configuration processing

With the knowledge about gaps, notches and planners, we can move on the algorithmic part. So let's assume we have the configuration points from the EGR ratio and the notch part. From these points we want to make key configurations including the rotations. As was said in Section 3, we presume that the narrow tunnels are either two gaps facing each other or a gap aligned with a notch. So the easiest way how to align such features is to take *Robot* part of the puzzle and translate it so they both with *Env* overlap in their configuration points. It is clear that once we move the *Robot* onto right place, we have to look for a collision-free rotations. Once we find such a state, we can declare this setting as a valid configuration. As a reminder, the configuration consists of a position and rotation which need to be collision-free.

It is likely that during the angle sweep we will find more correct rotations, see Figure 4.1. Here the question arises whether it is better to have less or more points. We can safely say that the more the points fill the angle sweep, the better the new sample points are connected. But for such a tree and finding the nearest neighbors, a lot of points might be too much time consuming. Therefore, we implemented and tested the following two ideas.

Let's have all possible rotations for one configuration point. In the description from our Figure 4.1, it means to take all angles marked as green free. Our aim is to connect all collision-free rotations together. This gives

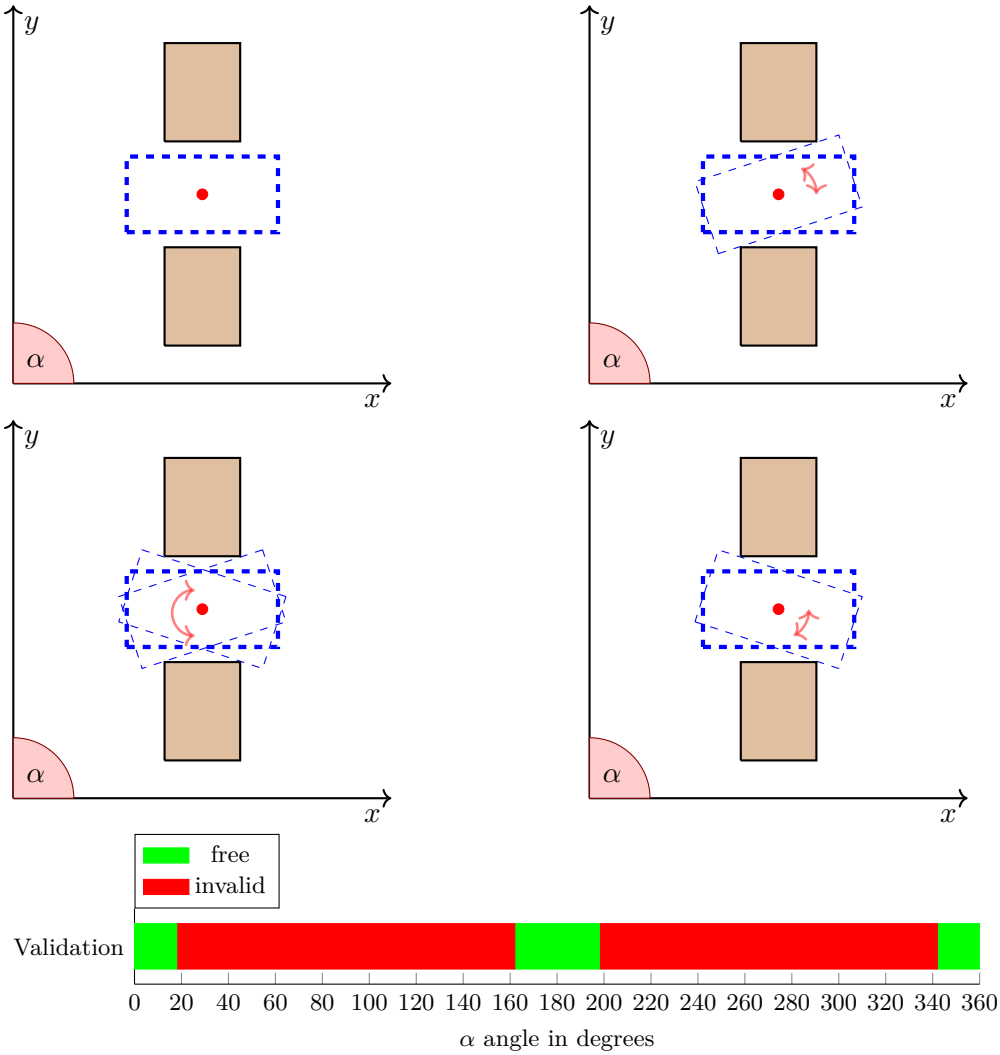


Figure 4.1: Illustration of a simple 2D angle sweep. Validation shows the group of collision-free angles.

us several independent regions full of rotations. The first (a) idea is to use K-means algorithm [22] to take only one point from each region that can be connected to as many points in the region as possible, or (b) to construct a tree from all points in the region. The procedure is then either to take as few points as possible, one point per region, or the number of trees per number of regions. Nevertheless, at the end of the procedure, we have either trees with one node each or trees with multiple nodes. Node is always considered as our key configuration. It is important to note here that for some configurations the angle sweep is very bulky and thus the tree size is very huge. For this reason we are testing both options, see results in Table 5.3.

Improvement. Suggestion from the article [23] says that gap-gap alignment has the maximum clearance between two pieces of puzzle when they are

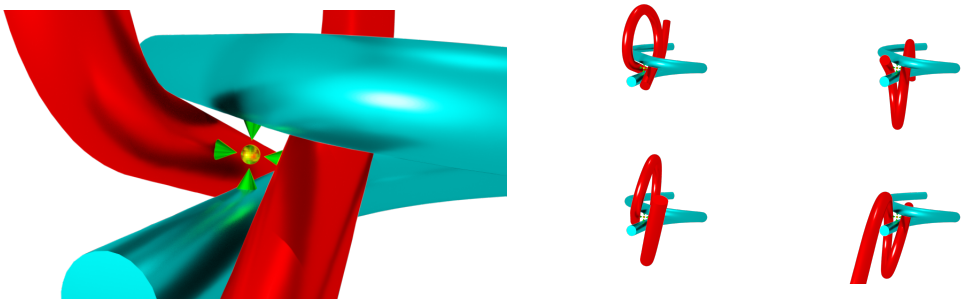


Figure 4.2: Gap-gap configuration focused on maximal clearance when orthogonal aligned. Visualization taken from the article [23].

orthogonal, illustration in Figure 4.2. On the other hand for alignment gap-notch when parallel. Therefore, during angle sweep, we can manually set up those static axis in advance. Because we use gap-notch alignment mainly on duet puzzles, we can set one angle right away either to 0, 90, 180 or 270 degrees. In order not to have to go through 360^3 possible angles during angle sweep.

4.2 Planner Selection

In the previous chapter we discussed puzzle's geometric features together with their implementation. We also already have knowledge about planning algorithms, so the last thing is to put it all together.

The best option will be to use some fast sampling algorithm which will connect the configurations that we obtain from our detection. The first option might be the Probabilistic roadmaps with sampling the whole C-space. But it's disadvantage here limits it very much. As already mentioned in Section 2.1, the probability of sampling the narrow passage is almost negligible.

The second possibility becomes Rapidly-exploring random tree that starts at one place and grows a tree until it finds a goal state. It is questionable how to use it. If we let RRT grow from each configuration, we will not know when to stop. One can say that it could stop when it hits other tree, but we have no guarantee and our experience has convinced us that we will not always find such a tree.

But that leads us to modification and that is the Rapidly-exploring dense tree.

As in the article [23] and described in Section 2.1, we also used this algorithm but with the following constraints. It stops growing the tree if the time of exploration exceeds user defined minutes or if the tree exceeds a maximum user-defined size. We have tried all the options that are recorded in the graphs in Subsection 5.2.1.

Because RDT algorithm is based on sampling and to avoid sampling from an infinite world, we create an imaginary box around each configuration, which corresponded to the longest side of the moving part of the puzzle in every direction. Each sample is just randomized right from this box. From our experience, this is very essential and important for a good sampling. Let's look more precisely on the sampling. When we sample from such a box, the probability of expanding a configuration tree is proportional to the size of its Voronoi region [13]. That means that the tree expands more often towards large areas, as illustrated in Figure 4.3. Therefore each root of tree is midpoint of such a box. This will leave an even distribution in the first iterations of RDT.

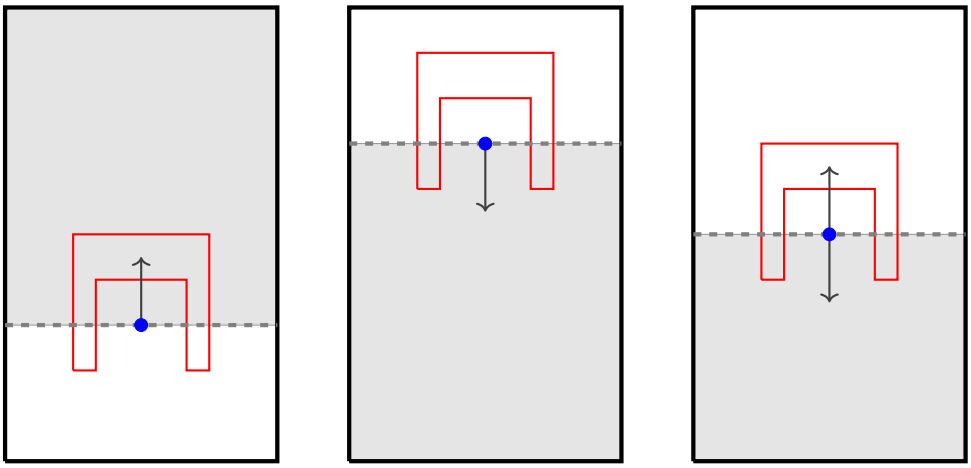


Figure 4.3: The black sampling box shows the importance of its position in relation to the blue configuration point. As you can see from the picture (*left*) it is much more likely to be sampled into the red wall, while in the picture (*middle*) to the space underneath. Picture on the right represents the ideal position, where the first samples will be equally distributed.

4.2.1 Implementation

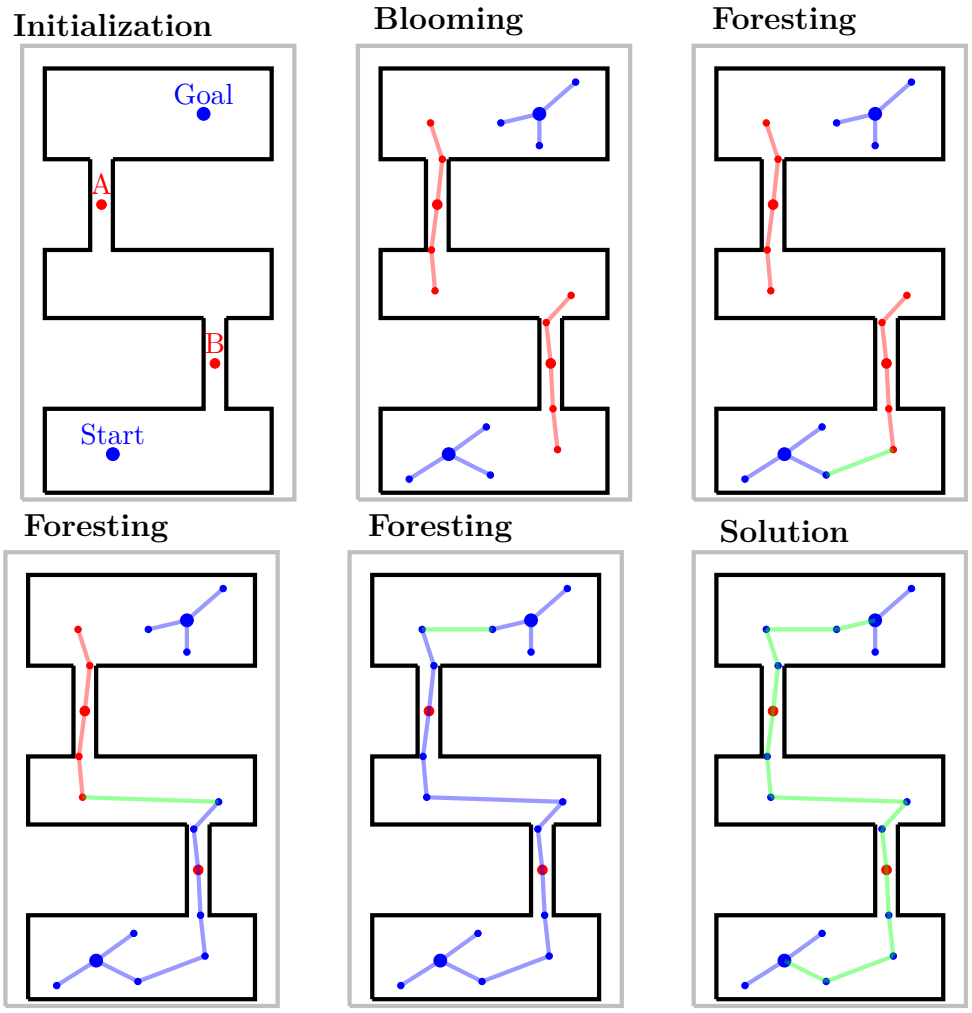


Figure 4.4: Steps of our implemented procedure. Initialize all configurations roots, bloom everyone till particular density, forest every tree, find path.

Let's have our configurations from both heuristics. Each of them constructed as roots of its own tree. We use a procedure described in Figure 4.4, followed in Algorithm 5, similar to the one in that article [23] divided into two steps. First one is called *Blooming*, where we mainly use the RDT planner. We iterate over those trees, trying to sample new point in C-space, that would be collision-free to its nearest neighbor. The aim is to make as many as possible within the user-defined time limit. We assume, that each new sampled point is normalized to the nearest neighbor to the distance of the surrounding world. That simply means, the distance between sample and nearest neighbor is one unit. The second one is called *Foresting*. We are still working with these large trees and trying to merge those trees together. We take every single tree

and iterate over its vertices and other trees looking for K-nearest neighbors. For the whole experiment, we used 8 closest neighbors. When we have these neighbors we look whether there is at least one collision-free route with that vertex. It there is, we merge that tree into the first one. We simply transfer all information and data from that tree into the first one. Important fact, each tree has to remember some identification, whether it contains nothing, goal or start root, because when we merge trees, the resulting tree has to get the higher priority identification. We say, such identification could be, for example, an enumeration type with values in order: *nothing*, *start/goal* and *both*, where the highest *both* has to be only in situations when tree contains both start and goal. So for situation, for example, when we merge *nothing* tree with *goal* tree, it has to results in *goal* tree.

After merging one tree with another tree, we delete nodes from the first one and move them to the other and leave just the unified one. We repeat this procedure till we merge all the trees or just trees including goal and start. It is important to realize that we do not have to merge all the trees in order to connect the goal and start. Eventually we find in that tree the path from start to goal.

When we iterate over every single vertex trying to connect to the other tree. We are able to simultaneously compare the distances of the connections of these two trees. In other words, we are able to guarantee the shortest connection between these two trees.

Algorithm 5 Solve Puzzle

```
1: function GET_PATH(M)   ▷ M represent the puzzle's triangulated mesh
2:     G, N, T = ∅;
3:     G ← GET_GAPS(M)                    ▷ configuration points from EGR
4:     N ← GET_NOTCHES(M)                 ▷ configuration points from Notches
5:     T = M.process(G, N);               ▷ configuration processing (Section 4.1)
6:     BLOOMING(T)
7:     if FORESTING(T) then               ▷ True if merged start and goal trees
8:         P ∈ C_space;                   ▷ P represent 3D points from C-space
9:         P = FIND_PATH(T);              ▷ graph search method [21], [20]
10:        return P;
11:    end if
12:    return ∅;
13: end function
```

Resources. During the planning phase, we also have to check collisions and memorize the whole tree structure. We had two libraries at our disposal. The first one was Rapid aka "Robust and Accurate Polygon Interference Detection" library version 2.01 [4] and it was used for collision detection. Basically each

part of the puzzle was constructed as RAPID_model on which collisions where detected. And the second one was MPNN aka "The Nearest Neighbor Library for Motion Planning" [1] which helped us constructing kd-tree and its findings.

Comparison. In order to have two functional procedures that we can eventually compare with each other, we also implemented the idea from the article [6] the Space-filling forest (SFF), but with modification to our RDT. Mainly we used its expansion for our *Blooming* part. Specifically, we randomly select a point from our tree, which we try to expand. Then, we sample a few new points in a unit sphere area and take the first one that meets following condition. If the new sample is closer to some other point in the tree than the one we have selected, we do not expand. This will ensure that we always expand towards unexplored area. This idea serves as an attempt to maximize the searched space as possible, but with fewer points. Eventually we have several modification we can test, see results in Tables 5.4, 5.5 and 5.6. Together with modifications from configuration processing (Section 4.1) we compare each different methods of searching the state space.

Improvements. Along with what we described in Section 2.1, the idea of improvement for our RDT might be to explore our configuration trees with some heuristics. It is therefore proposed to extend the basic RRT to RRT* [9]. For an illustration, a simple description. RRT* comes with two improvements. First, RRT* stores the knowledge about the distance from root to each vertex referred as the cost. This cost is recompute every time a new sample is joined to the closest neighbors. There comes the second difference, that RRT* rewires the vertices each time a vertex has been joined to the cheapest neighbor. That means neighbors are checked whether rewiring helps to decrease their cost. This feature makes the path more smooth and eventually even more optimal. This whole modification could be done on our RDT implementation as well.

4.3 Further Insight

Our implementation was done using the C++ programming language. All source code is included in the attachments of this thesis. In the course of our implementation we have encountered with several important findings that should be shared and described.

In the MPNN library, we used for constructing kd-tree and its search for

k-nearest neighbors, we came across the necessity to define the right amount of maximum points number for allocating the kd-tree. Together with the results generated with particular parameters, we came to conclusion that 400 thousands points for one tree is enough for both less demanding (Alpha) and more demanding (Duet) puzzles.

Particularly in header file multiann.h,

```
#ifndef MAXPOINTS
#define MAXPOINTS 400000
#endif
```

Collisions. In our implementation we solve collision checking as follows. We assume that the puzzle is in a certain position (**from**) and our goal is to get it to the next position (**to**). We define a set $\mathcal{P} = \{p_0, ..., p_n\}$ of uniform checkpoints in the trajectory, where we want to perform the collision check. There a problem arose, when we had points **from** and **to** close to each other, but their rotation angles where too different. It caused that we basically skipped the spot where they collided and declared them as free, illustrated in Figures 4.5 and 4.6.

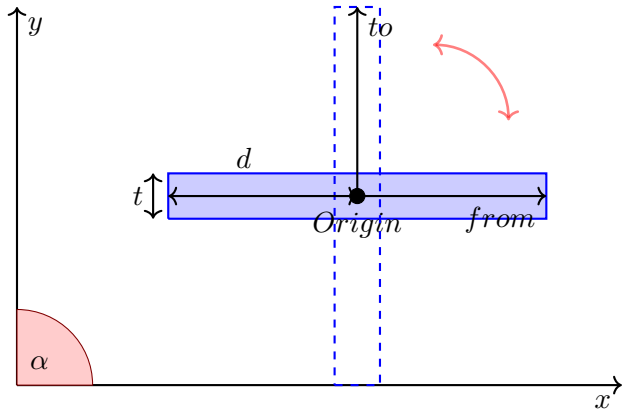


Figure 4.5: This picture together with Figure 4.6 illustrates the problem of checking collisions during rotation. The blue object represents the puzzle which stays on *Origin* and rotates over 90 degrees.

We came up with the solution, that determines how many checkpoints are we going to make. We always move the puzzle in a normalized distance (1 unit according to the surrounding world) and the puzzle itself has some thickness **t**. For our calculation we consider **t** as thickness in the thinnest place. We do not need to make any checkpoints at all when the distance of trajectory is smaller than the thickness. But there is still the rotation problem. We will determine the amount of checkpoints n by our equation:

$$n = \max(1, \frac{\frac{\alpha}{360} \cdot 2\pi \cdot d}{t}). \tag{4.1}$$

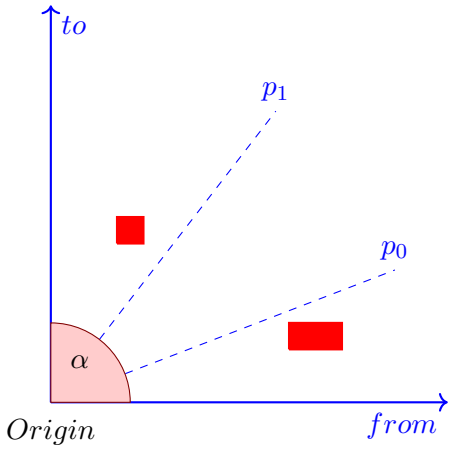


Figure 4.6: This picture together with Figure 4.5 illustrates the problem of checking collisions during rotation. Puzzle stays on *Origin* and it is just a rotation over α angle. It shows that even though it stays on the same position, it is not accurate and it will skip the red obstacles, if we made just two checkpoints.

It actually says how many times we can fit the thickness **t** of the puzzle into the rotation of α angle (in degrees). We declare that the distance **d** is the maximal length of puzzle from the origin of rotation into each direction. Thanks to this mechanism we are able to approach each rotation with a different number of collisions without having to generalize at all. As a result we are able to save a lot of checkpoints and at the same time we are sure that we do not miss any collision.

4.4 Parallelization

An essential part of programming is also the question whether the program can be parallelized. Here are several possibilities too. Among them we used the parallel calculation of the EGR ratio where both parts of the puzzle were computed separately, because these calculations do not depend on each other. But the biggest improvement could be done in *Blooming* part, where each tree can grow separately. We tried this approach as well, but because of the shared structures in the individual libraries Rapid [4] and MPNN [1], we had to use critical sections, which caused two things. The first is that they allowed us to process all trees concurrently, but we expanded far fewer points compared to the sequential solution. This means that the possibility is definitely there, but with the usage of different libraries. In our implementation both options are available, together with the results in Table 5.2.

Chapter 5

Summary

5.1 Testing with OMPL

Before we present our results within several particular parameters, we also had to test our implementation against other path planning algorithms. For this purpose we used the OMPL library [14] and its built-in planners. OMPL is a powerful interface, that provides high-level abstraction to make it easier to integrate your own planning system, whether you want to benchmark it with other high-level planners or develop one of your own.

In our case, we wanted to make a benchmark to be able to test it against other optimized planners including PRM, RRT, RRT-Connect, RRT*, EST, BIT and others. In order to leave the conditions as identical as possible for the other planners, we have handed over our collision checking library, RAPID [4], to OMPL to make the time spent on collisions as symmetric as possible. The same with the time constraints from our RDT, where we tried limitations on *Blooming* for 15, 20 and 25 minutes. But unfortunately none of them were able to solve even the basic Alpha puzzle, Figure 1.1.

5.2 Results

In this section we provide our results for testing several puzzles (Figure 5.1) taken from the article [23]. Among the results are described the individual input parameters within the solution and also the elapsed time of each procedure.

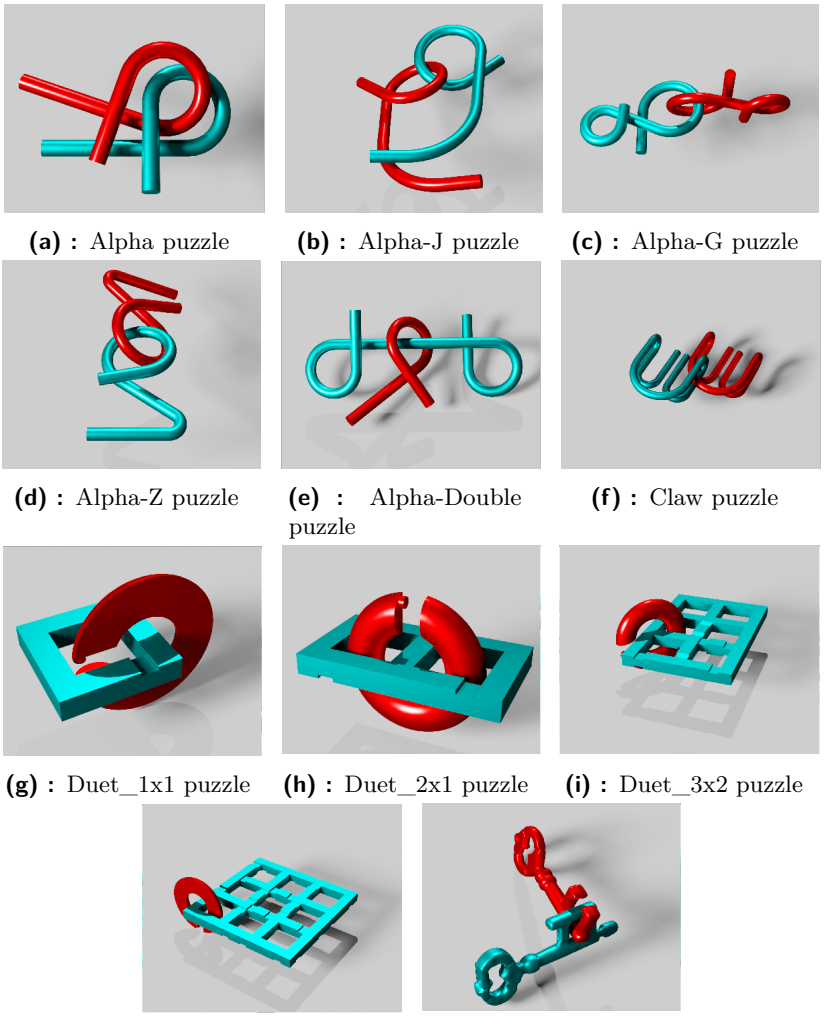
Before we go any further, we need to specify some of the parameters we talked about in the thesis. Firstly in gaps detection, Section 3.1, we did our procedure for user-defined number $N = 20$. During notches detection, Section 3.2, we talked about weighting the division in our equation by the threshold. In our implementation we chose 0.3 as the threshold. And eventually, when we described sampling from the sphere in SFF modification in Section 4.2.1, we chose 20 random samples.

Thanks to the possibility of testing on distributed machines, we were able to do several of these tests. For the comparison with SFF modification we used a 16-core machine with 4GB of RAM and for the rest of calculations an 8-core machine with the same memory. The following values in the graphs correspond to the average results after 100 iterations. Animations of solving these puzzles are included together with source code in the attachments of our thesis.

Unfortunately, due to zero success rates, we left out the most difficult Duet puzzle (Duet_3x3) together with Claw and Key, which were not solved by our heuristics due to their complex geometric shape.

For the most accurate results, we share puzzles configurations from heuristics in each run. That means one puzzle has the same configurations in each run, therefore we calculate the configurations via our heuristics in advance. This will help us to better define the quality of our planner. It is clear that for the Alpha puzzle and puzzles derived from it there is no need to compute notch configurations. In Table 5.1 are presented all configuration pairs[1].

[1]Pair, because for implementation purposes we need whole pair. Simply the main configuration point lies every time in the middle of this pair.



(a) : Alpha puzzle **(b) :** Alpha-J puzzle **(c) :** Alpha-G puzzle

(d) : Alpha-Z puzzle **(e) :** Alpha-Double puzzle **(f) :** Claw puzzle

(g) : Duet_1x1 puzzle **(h) :** Duet_2x1 puzzle **(i) :** Duet_3x2 puzzle

(j) : Duet_3x3 puzzle **(k) :** Key puzzle

Figure 5.1: Illustration of puzzles we tried to solve. Puzzles taken from the article [23].

	Alpha	Alpha-Double	Alpha-J	Alpha-G	Alpha-Z		
Configurations	5 pairs	4 pairs	5 pairs	2 pairs	4 pairs		
	Claw	Key	Duet_1x1	Duet_2x1	Duet_3x2	Duet_3x3	Duet_3x3a
Configurations	9 pairs	2 pairs	11 pairs	6 pairs	42 pairs	143 pairs	144 pairs

Table 5.1: First row includes puzzles, where EGR is enough for constructing configurations. The second row contains numbers of pairs including both from EGR and notch part.

5.2.1 General Results

The following Graphs 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8 and 5.9 represent general success rate and average elapsed time of solving each puzzle based on particular size of the *Blooming* tree with set constraint time. We used our implementation of RDT planner with parallelization. All configurations were used all the time (corresponding (b) idea from Section 4.1).

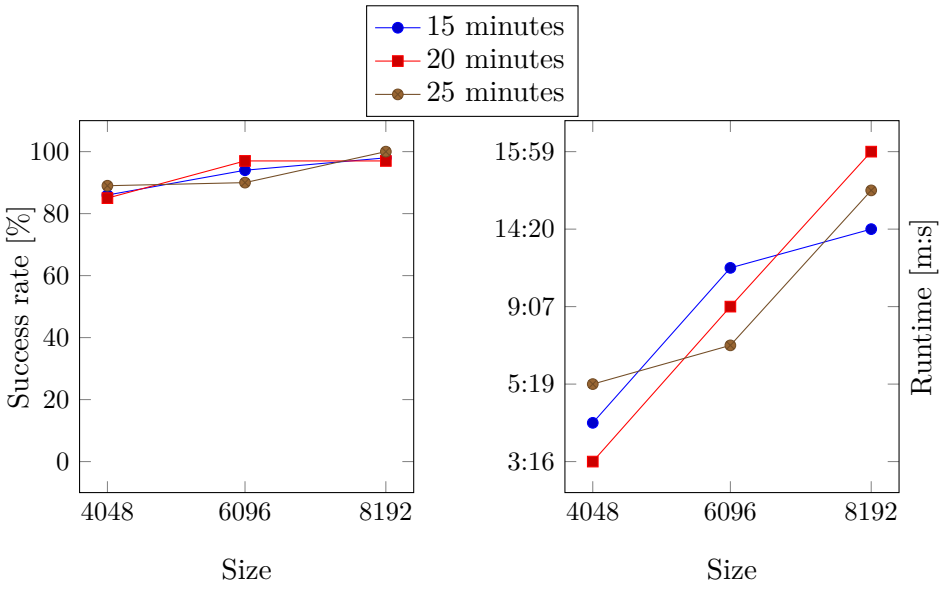


Figure 5.2: Alpha puzzle.

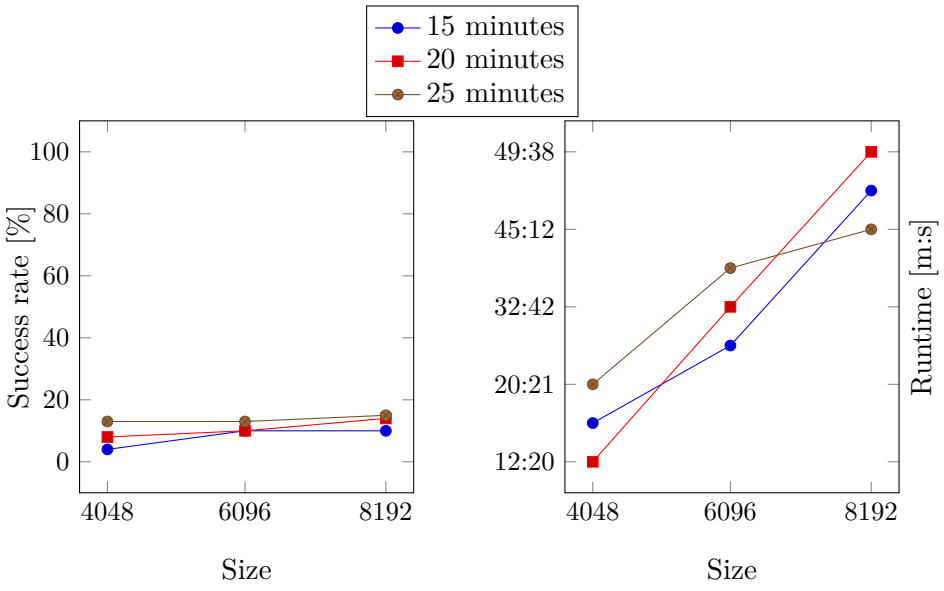


Figure 5.3: Alpha-Double puzzle.

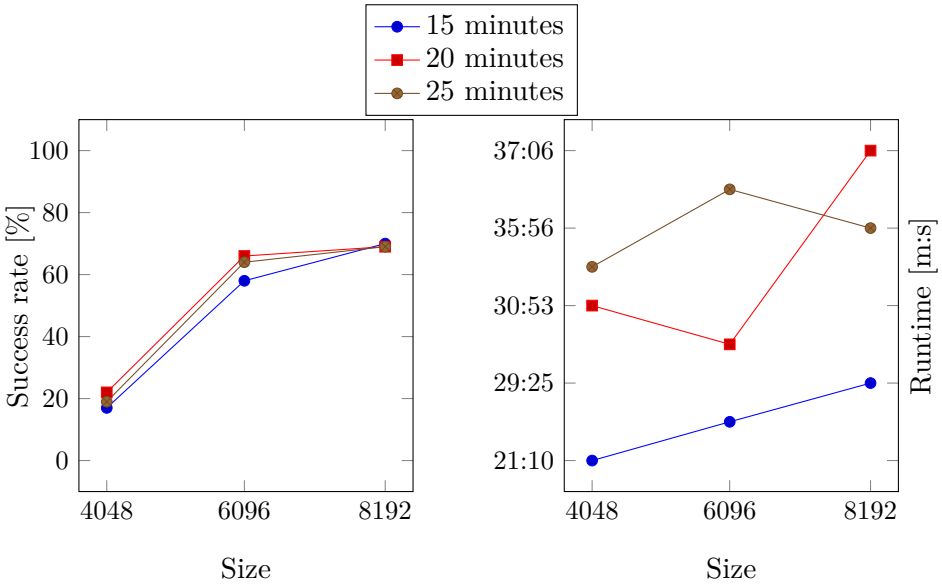


Figure 5.4: Alpha-J puzzle.

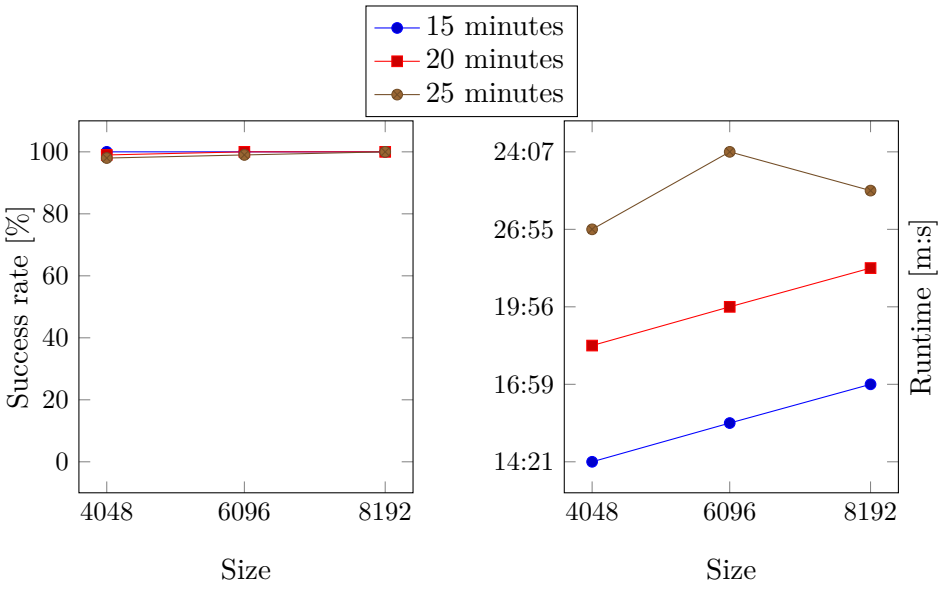


Figure 5.5: Alpha-G puzzle.

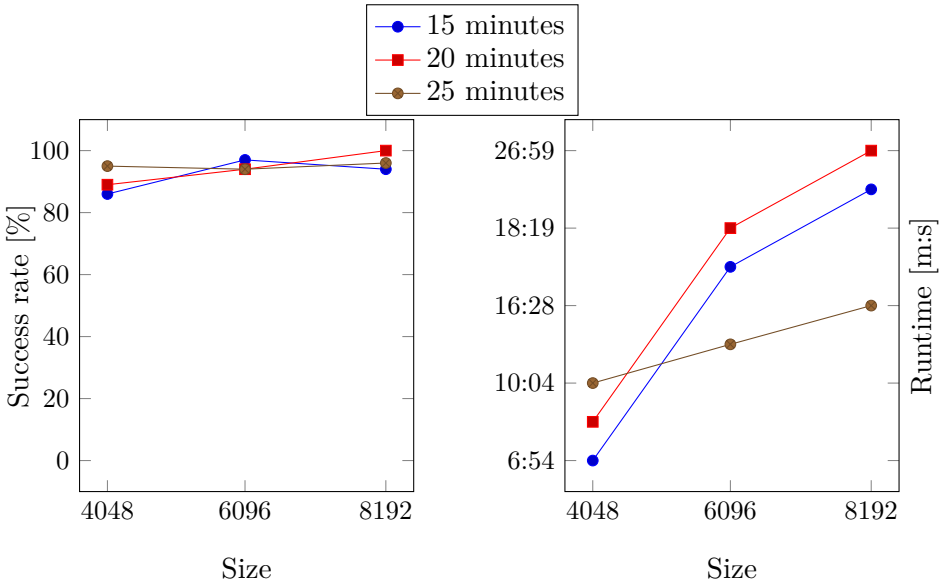


Figure 5.6: Alpha-Z puzzle.

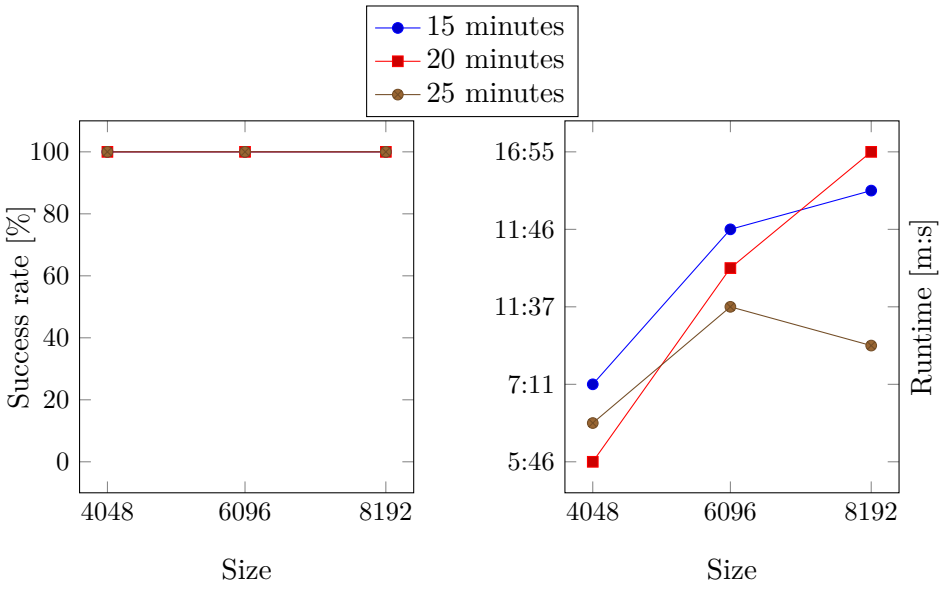


Figure 5.7: Duet_1x1 puzzle.

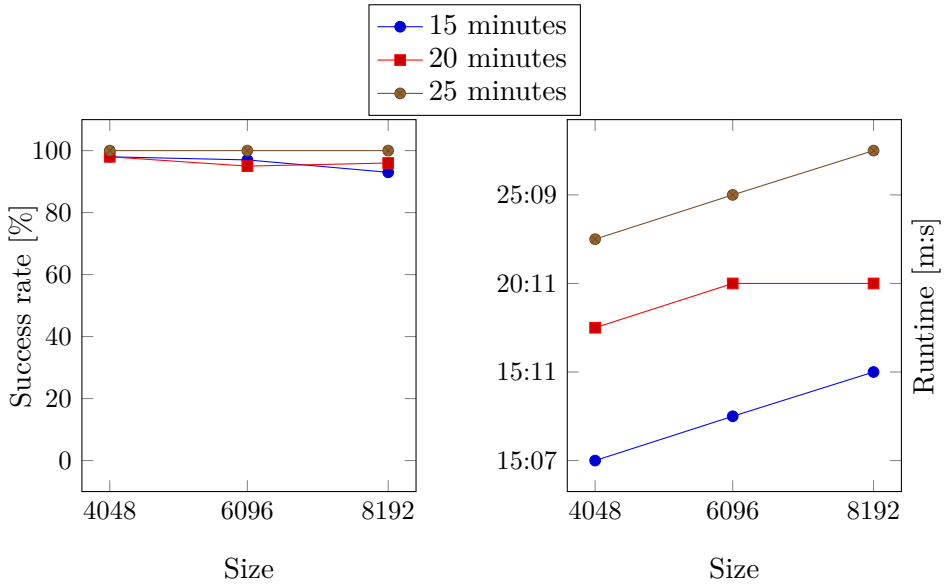


Figure 5.8: Duet_2x1 puzzle.

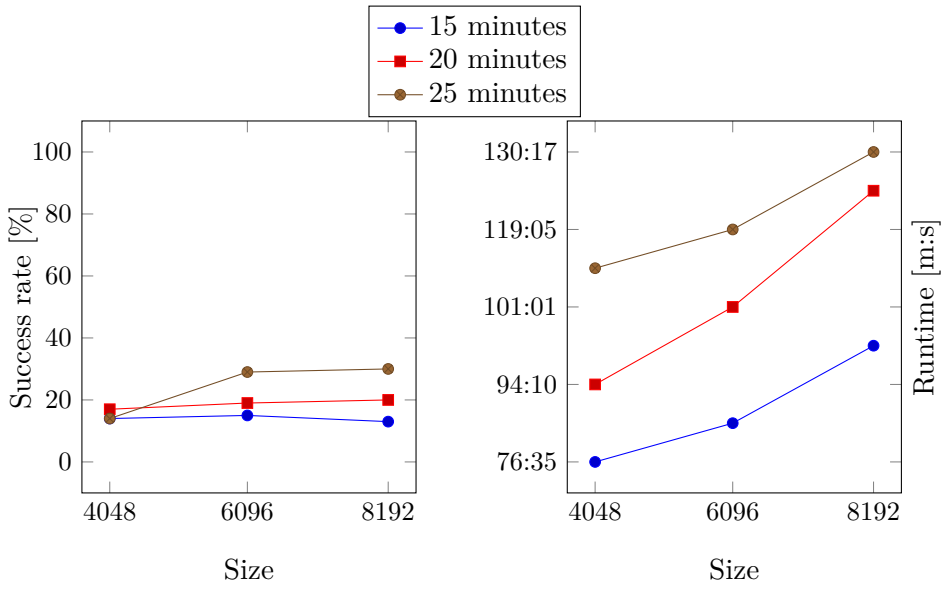


Figure 5.9: Duet_3x2 puzzle.

The results more or less show that the more points we provide for expanding and at the same time the longer we leave the possibility for expanding configuration trees, the better results we get. Of course, because we have the results after a hundred iterations, the trend can vary, e.g. for Duet_2x1 we even got a percent worse results for bigger size, but it has almost no effect on the average solution.

5.2.2 Comparison Parallel vs Sequential

Due to the mentioned problems with critical sections in the parallelized RDT implementation (Section 4.4), we have performed tests with sequential solution as well. The next table shows the differences with following settings. Both runs were made with 15 minutes per *Blooming* phase and 6096 size its trees. All configurations were used (corresponding (b) idea from Section 4.1).

	Parallel		Sequential	
Puzzle	Success rate	Runtime [m:s]	Success rate	Runtime [m:s]
Alpha	94%	9:15	90%	10:43
Alpha-Double	10%	32:19	11%	68:27
Alpha-J	58%	26:58	66%	66:17
Alpha-G	100%	16:01	99%	21:25
Alpha-Z	97%	18:12	93%	19:14
Duet_1x1	100%	11:46	100%	15:22
Duet_2x1	97%	15:08	100%	44:56
Duet_3x2	15%	80:34	36%	139:38

Table 5.2: The average success rate and average elapsed time between parallel and sequential RDT after 100 iterations. Constraints: 15 minutes per *Blooming* and 6096 as its size. All configurations were used.

From the table we can clearly see that the differences are very small and therefore the difference between sequential and parallel solutions is insignificant. On the other hand, if we want to save some time, a parallel version will certainly help us. In some cases more (e.g. Duet_2x1), in some cases less (e.g. Alpha).

5.2.3 Comparison of Configuration Processing

In Section 4.1, we described two ideas how to work with found configurations. We have mentioned that it is possible to use as many as possible or just particular ones. In the following table we present results of parallelized RDT with 15 minutes per *Blooming* and 6096 size its trees together with both options of handling configurations. The idea (a) is considered to be SINGLE and (b) as taken ALL.

	ALL		SINGLE	
Puzzle	Success rate	Runtime [m:s]	Success rate	Runtime [m:s]
Alpha	94%	9:15	93%	10:34
Alpha-Double	10%	32:19	11%	21:39
Alpha-J	58%	26:58	78%	34:19
Alpha-G	100%	16:01	100%	14:27
Alpha-Z	97%	18:12	99%	13:05
Duet_1x1	100%	11:46	100%	7:09
Duet_2x1	97%	15:08	98%	15:11
Duet_3x2	15%	80:34	35%	94:10

Table 5.3: The average success rate and average elapsed time of RDT between two different processes of handling configurations after 100 iterations. Constraints: 15 minutes per *Blooming* and 6096 as its size.

Although the differences in the results are small, it can be seen that if we take a few points, which are arranged in a good position by the K-means method, they lead to better results. Puzzles like Alpha-J and Duet_3x2 particularly. Unfortunately, we cannot say anything about the elapsed time, because our runtime differs in all cases.

5.2.4 Comparison with SFF modification

As we mentioned in Subsection 4.2.1, we also implemented the SFF modification into our RDT planner to compare two approaches of expanding our configuration trees. Unfortunately this modification needs considerably more time to grow the tree. Our experience is that if we limit the *Blooming* in RDT planner to 15 minutes, which in most cases is enough to fill 6096 points, the SFF modification will expand only a few tens of points in that time. For this possibility we have made a test with a limit of 120 and 180 minutes, which already gives better results. The following tables show success rate of SFF modification with both time limits and 6096 as size of its trees. The first table represents the processing of all (b) configurations, as described in Section 4.1, the second table otherwise (a).

	120 minutes		180 minutes	
Puzzle	Success rate	Runtime [m:s]	Success rate	Runtime [m:s]
Alpha	35%	24:50	40%	34:21
Alpha-Double	5%	44:00	6%	59:25
Alpha-J	0%	28:35	0%	42:34
Alpha-G	15%	24:24	25%	36:29
Alpha-Z	0%	27:44	9%	40:19
Duet_1x1	100%	15:03	100%	16:10
Duet_2x1	90%	24:01	97%	35:59
Duet_3x2	0%	54:30	0%	78:46

Table 5.4: The average success rate and average elapsed time of SFF modification after 100 iterations. Constraints: 120 and 180 minutes per *Blooming* and 6096 as its size. All configurations were used.

	120 minutes		180 minutes	
Puzzle	Success rate	Runtime [m:s]	Success rate	Runtime [m:s]
Alpha	5%	37:57	25%	45:01
Alpha-Double	0%	45:15	0%	51:48
Alpha-J	0%	60:53	0%	89:52
Alpha-G	10%	24:25	18%	36:29
Alpha-Z	10%	25:43	20%	25:58
Duet_1x1	100%	14:43	100%	18:02
Duet_2x1	80%	24:05	96%	36:04
Duet_3x2	0%	66:43	0%	88:22

Table 5.5: The average success rate and average elapsed time of SFF modification after 100 iterations. Constraints: 120 and 180 minutes per *Blooming* and 6096 as its size. Single configuration was used.

When we were describing the tree expansion in Subsection 4.2.1, we also mentioned taking 20 random points from the unit sphere. To enrich our results further, we performed a simple test on a selection of three particular puzzles with different sizes of that sphere. More precisely, we examined the sizes of half unit, two units and three units. One unit was selected by default and also tested in the previous results. Same constraints were retained and thus the 120 minutes per *Blooming* and 6096 as its trees size.

	Success rate			
Puzzle	Half unit	One unit	Two units	Three units
Alpha	0%	35%	10%	25%
Duet_1x1	100%	100%	100%	100%
Duet_3x2	0%	0%	0%	0%

Table 5.6: The average success rate of SFF modification after 100 iterations with various sizes of the sphere from which we take samples during *Blooming*. Constraints: 120 minutes per *Blooming* and 6096 as its size. All configurations were used.

From the previous tables it can be seen that for most of the puzzles the longer the SFF modification runs the better results it gives. On the other hand, compared to our RDT planner, the SFF modification lags far behind in both success rate and time consumption.

We also cannot draw any conclusion from the size of the sphere from which we sample, because it looks like the success rate does not depend on the size. However, it is obvious from the geometrical properties of the narrow passages that the size of the sampling sphere will not be clearly given. Simply put, if such a tunnel will have 2 units in the thickest place, we certainly cannot use a sphere of 3 units, etc. It basically depends on the construction of the puzzle. From our experience 1 unit is sufficient in our puzzles.

Chapter 6

Conclusion

This thesis presents a new perspective on how to use the geometric properties of the puzzles, based on narrow tunnels. For motion planning itself it is a challenging task. It was clear that from human view, we can see such a feature in a second, but not so far for a complex algorithm. The key features we dealt with were gaps and notches that predicted where the tunnels might be located. From those observations we simply created the configurations we used in the planning tree. We discussed advantages and disadvantages of simple planners and showed that even slightly enhanced sampling might be far better. We introduced the new RDT planner, which we then compared with the state-of-the-art planners from the OMPL library. None of the built-in planners from OMPL could not solve even the Alpha puzzle. Eventually, we presented a lot of results with several possible input parameters together with modification of introduced SFF planner. The results undoubtedly show that without any knowledge about environment it is very hard to solve even an easy puzzle.

6.1 Ideas of improvements

Throughout our thesis, we spoke about ideas of improvement and how to deliver better view on such problem. We still have to keep in mind that we have only dealt with puzzles consisting of pairs of two rigid moving pieces. Nonetheless, many other puzzles do not look like this and have to be handled differently. There is certainly room for improvement in the usage of geometric properties. As mentioned in Section 3.2 an approach through medial axis is

another option to search for configurations but it must be better than just an approximation. What we should not forget and it is very popular nowadays is the use of neural networks. In the article [23], they mentioned their own way how to find them in use.

Last but not least, one of the improvements that is quite crucial in the puzzle solving is how we know that the puzzle is solved. Specifically, when we say that one body is disantangled out of the other. One of the simplest solutions would be to create a box around both pieces of the puzzle and say that they are resolved precisely when these boxes do not overlap. We probably would not need the goal state, so we would be minus one configuration together with its tree simpler.

Bibliography

[1] Atramentov, A., and LaValle, S. Efficient nearest neighbor searching for motion planning. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)* (2002), vol. 1, pp. 632–637 vol.1.

[2] Borst, C., Ott, C., Wimbock, T., Brunner, B., Zacharias, F., Bauml, B., Hillenbrand, U., Haddadin, S., Albu-Schaffer, A., and Hirzinger, G. A humanoid upper body system for two-handed manipulation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation* (2007), pp. 2766–2767.

[3] Denny, J., Sandström, R., Bregger, A., and Amato, N. M. *Dynamic Region-biased Rapidly-exploring Random Trees*. Springer International Publishing, Cham, 2020, pp. 640–655.

[4] Gottschalk, S., Lin, M. C., and Manocha, D. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 171–180.

[5] Jaillet, L., and Porta, J. M. Path planning under kinematic constraints by rapidly exploring manifolds. *IEEE Transactions on Robotics 29*, 1 (2013), 105–117.

[6] Janoš, J., Vonásek, V., and Pěnička, R. Multi-goal path planning using multiple random trees. *IEEE Robotics and Automation Letters 6*, 2 (2021), 4201–4208.

[7] Jayasree, K. R., Jayasree, P. R., and Vivek, A. Smoothed rrt techniques for trajectory planning. In *2017 International Conference*

on Technological Advancements in Power and Energy (TAP Energy) (2017), pp. 1–8.

[8] Karaman, S., and Frazzoli, E. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research 30*, 7 (2011), 846–894.

[9] Karaman, S., Walter, M. R., Perez, A., Frazzoli, E., and Teller, S. Anytime motion planning using the rrt*. In *2011 IEEE International Conference on Robotics and Automation* (2011), pp. 1478–1483.

[10] Kavraki, L., Svestka, P., Latombe, J.-C., and Overmars, M. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation 12*, 4 (1996), 566–580.

[11] Kuffner, J., and LaValle, S. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)* (2000), vol. 2, pp. 995–1001 vol.2.

[12] LaValle, S. M. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[13] Lindemann, S., and LaValle, S. Incrementally reducing dispersion by increasing voronoi bias in rrts. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004* (2004), vol. 4, pp. 3251–3257 Vol.4.

[14] Şucan, I. A., Moll, M., and Kavraki, L. E. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine 19*, 4 (December 2012), 72–82. `https://ompl.kavrakilab.org`.

[15] The CGAL Project. *CGAL User and Reference Manual*, 5.3 ed. CGAL Editorial Board, 2021.

[16] Vega-Brown, W., and Roy, N. Task and motion planning is pspace-complete. *Proceedings of the AAAI Conference on Artificial Intelligence 34* (04 2020), 10385–10392.

[17] Vonásek, V., and Kozlíková, B. Tunnel detection in protein structures using sampling-based motion planning. In *2017 11th International Workshop on Robot Motion and Control (RoMoCo)* (2017), pp. 185–192.

[18] Wikipedia contributors. Medial axis — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-November-2021].

[19] Wikipedia contributors. Straight skeleton — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-November-2021].

[20] Wikipedia contributors. Breadth-first search — Wikipedia, the free encyclopedia, 2022. [Online; accessed 4-May-2022].

[21] Wikipedia contributors. Depth-first search — Wikipedia, the free encyclopedia, 2022. [Online; accessed 4-May-2022].

[22] Wikipedia contributors. K-means clustering — Wikipedia, the free encyclopedia, 2022. [Online; accessed 10-May-2022].

[23] Zhang, X., Belfer, R., Kry, P., and Vouga, E. C-space tunnel discovery for puzzle path planning. *ACM transactions on graphics 39*, 4 (2020), 104:1–104:14.