FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Autonomous Path Planning using probabilistic Maps

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Albin Frischenschlager BSc.
Matrikelnummer 0926427

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Mitwirkung: Univ.-Ass. Dipl.-Ing. Oliver Höftberger

Wien, 05.10.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Autonomous Path Planning using probabilistic Maps

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Albin Frischenschlager BSc.
Registration Number 0926427

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Assistance: Univ.-Ass. Dipl.-Ing. Oliver Höftberger

Vienna, 05.10.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Albin Frischenschlager BSc.
Rudolf Waisenhorn-Gasse 107, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all, I want to thank the advisers of my thesis Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu and Univ.-Ass. Dipl.-Ing. Oliver Höftberger, who gave me the chance to do this thesis and offered feedback and advise throughout my thesis.

Furthermore, I want to thank my girlfriend Natalie for her support during my studies and for proofreading this thesis. She encouraged me in frustrating moments, which helped me to finish this thesis and my studies.

Last but not least, I want to thank my family, which made my studies at the Vienna University of Technology possible. They also eased the time of my studies by supporting me in every possible way.

Abstract

To fulfil their tasks, autonomous robots have to be able to independently and safely reach a target location from their current position. Path planning enables a robot to do exactly this. Usually, this is done with a map containing the static obstacles (e.g., walls) of the environment. Since dynamic obstacles (e.g., humans) can be present in the environment, the static map alone is not sufficient. Thus, the robot additionally needs sensors to continuously observe the environment to detect dynamic obstacles and to avoid them. This approach, to avoid dynamic obstacles when the robot "sees them", can lead to highly suboptimal behaviour, since the robot will have to start, stop and change the direction frequently.

To remedy this problem, probabilistic maps are used in this thesis to enhance common path planning strategies. In contrast to static maps, a probabilistic map contains probabilistic information about the likelihood of encountering dynamic obstacles in certain areas of the environment. This information is used to avoid dynamic obstacles, such that the robot does not have to react to them.

In this thesis, the *navigation stack* from the *robot operation system* (ROS) is used to allow a Pioneer 3-AT (P3AT) robot to navigate autonomously. The ROS navigation stack, responsible for path planning, is modified to use probabilistic maps to avoid dynamic obstacles. Additionally, further adjustments and software components are developed to incorporate the limitations of the Pioneer 3-AT robot. Since the Pioneer 3-AT robot only has sonar sensors the ROS navigation stack is modified to use these sensors instead of a laser scanner.

With the help of experiments, suitable parameters for the new planning strategies have been found. Finally, the advantages of path planning using probabilistic maps compared to static maps are shown with further experiments.

Kurzfassung

Autonome Roboter müssen zur Erfüllung ihrer Aufgaben fähig sein, selbständig und sicher von einer Position eine andere Position zu erreichen. Pfadplanung erlaubt einem Roboter dieses Verhalten. Normalerweise wird dies mithilfe einer statischen Karte gemacht, welche statische Hindernisse (z.B. Mauern) der Umgebung enthält. Da in der Umgebung des Roboters auch dynamische Hindernisse (z.B. Menschen) vorkommen können, ist eine statische Karte alleine nicht ausreichend. Der Roboter benötigt zusätzlich Sensoren, um die Umgebung zu beobachten und somit dynamische Hindernisse zu entdecken und ihnen auszuweichen. Der Ansatz, dass der Roboter dynamisch Hindernissen ausweicht, wenn dieser "sie sieht", kann zu sehr suboptimalem Verhalten führen, da der Roboter dadurch oft starten, stoppen und Richtung wechseln wird.

Um dieses Problem zu umgehen, wurde in diese Arbeit mithilfe von probabilistischen Karten bestehende Pfadplanungsstrategien verbessert. Im Gegensatz zu statischen Karten, beinhaltet eine probabilistische Karte Informationen über die Wahrscheinlichkeit auf ein dynamische Hindernis in einem bestimmten Teil der Umgebung zu stoßen.

In dieser Arbeit wurde der *navigation stack* vom *robot operation system* (ROS) benutzt, um den Pioneer 3-AT (P3AT) Roboter autonom navigieren zu lassen. Der ROS navigation stack, welcher Pfadplanung durchführt, wurde so modifiziert, dass probabilistische Karten benutzt werden um dynamischen Hindernissen auszuweichen. Zusätzlich wurden weitere Veränderungen durchgeführt und Software Komponenten entwickelt um den Limitierungen des Pioneer 3-AT Roboters entgegenzuwirken. Da der Pioneer 3-AT Roboter nur über Sonar Sensoren verfügt, wurde der ROS navigation stack modifiziert, damit diese Sensoren anstelle eines Laser Scanners eingesetzt werden.

Sinnvolle Parameter für die neuen Planungsstrategien wurden mithilfe von Experimenten gefunden. Außerdem wurden die Vorteile von Pfadplanung mit probabilistischen Karten gegenüber von statischen Karten mit weiteren Experimenten gezeigt.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Nowadays, robots are increasingly present in our daily life and it is easy to predict, that more tasks will be accomplished by robots in the future, because of the advantages they bring with them. To name a few, these are their accuracy regarding repetitive tasks, ability to operate in terrains that are not accessible for humans or for reduced cost.

For every task, except the most basic ones, robots have to reason about actions they should execute and when they should carry these actions out. This reasoning is known as action planning. *The focus of this thesis is path planning, which is an important subcategory of action planning.* In path planning, the robot tries to safely reach a distant point, either supplied by a human user or by a high level task, with minimal cost. A high level task could be finding a missing person in difficult terrain.

Autonomously reaching arbitrary points is an important stepping stone for fulfilling various interesting tasks. Possible tasks could be searching for stuck/injured persons after an earthquake in difficult accessible- or dangerous areas, or finding a missing person in a forest. Both examples are classical urban search and rescue (USAR) duties, in which extensive research is done [79].

Another area of operation could be humanoid robot butlers, which help elderly or disabled people in the daily live. Tasks could be house cleaning or simple tasks like picking up dropped down things. This idea can be extended to use robots in healthcare, another rich research field [31].

A third possible task which is enabled through path planning is autonomously driving cars. Passengers just have to declare the desired goal and the car drives to the goal via an optimal route. An optimal route could be the fastest, the shortest, the most beautiful or the most fuel-efficient. No further human interaction is needed and the passengers can use the driving time for other activities like chatting, reading books or working.

1.2 Problem definition

Usually, to accomplish path planning, a static map of the environment is supplied. This map is used to localize the position of the robot and to avoid static obstacles (e.g., walls). Since in most environments dynamic obstacles, which are not contained in the map, exist, the static map alone is not sufficient to reach the goal in a safe manner. Such dynamic obstacles are for example humans in the robot's environment or cars on a road, which both change their position over time. To ensure safe traversal despite dynamic obstacles, the robot has to detect and avoid them. Unfortunately, avoiding dynamic obstacles can lead to high costs to reach the final goal. If, for example, the cost is measured as the time needed to reach the goal, movement through a place crowded with dynamic obstacles, where the robot has to start, stop and change the direction frequently because of these obstacles, will lead to a longer traversal time. A cheaper path with respect to the average travel time would use a probably longer, but less crowded route to reach the goal. The same is true for a street where traffic jams occur frequently. A longer alternative route, avoiding this street could lead to a shorter travel time.

Additionally, path planning suffers from the same problem as most other robotic applications: the inherent uncertainty of sensors. This is problematic, since it is not possible to infer the exact position of the robot and the obstacles. To overcome this fundamental problem, the information of various sensors, nowadays mostly modelled by probabilistic models, are combined and filtered to get more exact results. This idea of probability is transferred to static maps yielding probabilistic ones, that enable the planning algorithm to avoid regions with high probabilities of obstacles.

1.3 Methodological approach

In the first step an extensive literature study to collect existing navigation strategies and algorithms is done. Then the found strategies are evaluated for the possibilities for enhancement with probabilistic maps.

Based on this, a new path planning strategy is developed, which uses probabilistic maps. The idea of this new planning strategy is to avoid three types of areas. The first type are static obstacles, which every path planner tries to avoid. The second type of areas are crowded places like a busy road. At last, areas, which have the characteristic of infrequently used roads, which are in bad shape, are avoided, since only slow movement is possible on such roads. The avoidance of the last two areas is novel in path planning.

Additionally, the new path planning strategy is developed for the usage of sonar sensors, instead of the commonly used laser sensors. Since sonar sensors are a lot cheaper than laser scanners, this allows the usage of the new planning strategy for cheap autonomous robots.

The open source framework 'robot operating system' (ROS) [71] is used for implementing the new path planning strategy and for evaluation of the developed algorithms. Thus, distinct experiments are conducted to obtain suitable parameters for the new planning strategies. Finally, existing standard path planning algorithms are evaluated against the augmented strategies, with respect to cost and performance.

1.4 Structure of work

At first, Chapter 2 defines the path planning problem for static environments and introduces important concepts. After the presentation of state-of-the-art algorithms for discrete path planning, state of the art continuous path planning strategies for static environments are explained.

Chapter 3 defines the path planning problem for dynamic environments. Then it is shown, how to extend the strategies presented in Chapter 2 to yield state of the art path planning in dynamic environments.

After the theoretical background was laid out in the previous two chapters, Chapter 4 develops a new path planning strategy. First, probabilistic maps are used to enhance traditional path planning strategies. The differences to traditional path planning is analysed. After the comparison of sonar sensors with laser sensors, new algorithms are presented to use sonar sensors in path planning.

In Chapter 5 the new developed path planning strategies is implemented. First, the Robot Operating System (ROS), which is a central part in the implementation, is introduced. The main concept, features and principles of ROS are explained as well as how a component can be started in this system.

Then all ROS modules used in this thesis are presented in detail. It is shown how the navigation stack of ROS is adapted to use the new path planning strateges to yield more intelligent path planning, is presented in detail. The implemented algorithms and data structures are explained.

Chapter 6 shows how the adoptions changed the path planning of the ROS navigation stack, with the help of various experiments. First, the robot platform used for the experiments is presented. Then, the advantages of using probabilistic maps compared to static maps in path planning are shown.

Then, Chapter 7 discusses, how the implementation of this thesis can be extended to further increase the usefulness of probabilistic maps in path planning.

At last, Chapter 8 concludes this thesis.

CHAPTER 2

Static path planning

2.1 Introduction

As already mentioned at Section 1, motion planning and therefore *path planning* is a fundamental need in robotics [45]. Thus, it is not surprising that this is one of the most studied problem in robotics [17], which yielded various approaches and strategies to solve it. Most research for path planning was done in *static environments* [83], meaning all obstacles in the environment are known before the optimal path is computed. In contrast, in *dynamic environments* not all obstacles are known beforehand. Examples for dynamic obstacles are moving humans or just a chair, which was forgotten to put away and is now standing in an unexpected place.

In the following chapter, the path planning problem will be defined, followed by an overview of the general ways to solve static path planning. In Chapter 3 we will see how the static methods from this chapter can be adapted to solve path planning in dynamic environments.

2.1.1 Configuration space

The task of optimal path planning can be described as: Given a start and a goal location, the task of optimal path planning is to find a *collision free path* for a given moving entity (from here on called robot) with minimal cost.

To accomplish this in the most general form, the following attributes have to be known [83]:

- The geometry of the robot
- The geometry of the environment, in which the robot is operating (typically via a map)
- The degrees of freedom of the robot
- The start and goal location

With the help of the *configuration space*, introduced by Lozano-Pérez in [48], the path planning problem can be defined more formally. A *configuration* uniquely describes one possible

robot position in the world. For example a robot only capable of translation in a two-dimensional world can uniquely be described by two coordinates (x and y). Hence, a configuration in this example is two-dimensional. Note that the position of the same robot (only capable of translation in x and y direction) in a three-dimensional world can also be uniquely described by a two-dimensional configuration. Furthermore, a configuration of a robot which is capable of translation and rotation in a two-dimensional world is three-dimensional (x, y and the orientation of the robot theta). Hence, the minimal dimension of a configuration to uniquely describe a robot position in the world only depends on the degrees of freedom (DOF) of the robot [34], but not on the dimension of the world. The set of all possible configurations of a robot forms the configuration space $\mathcal{C}$. For a rigid robot, the maximum configuration space dimension can either be three (two-dimensional world) or six (three-dimensional world) [17]. If the robot consists of multiple moving bodies like an industrial manipulator arm or a robot with trailers, the maximum dimension of $\mathcal{C}$ can be arbitrary [17].

Naturally, static obstacles exist in the environment and the robot should not collide with them. Hence, all configurations in which a part of the robot collides with an obstacle are forbidden. Furthermore, all configurations in which the robot collides with itself are forbidden too, like for an industrial manipulator arm, where the end-effector may collide with the base of the robot [83]. All these forbidden configurations form the subset $\mathcal{C}_{coll}$ of the configuration space $\mathcal{C}$.

All other configurations are allowed and define the subset $\mathcal{C}_{free}$. Thus, $\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{coll}$ and $C_{free} \cap \mathcal{C}_{coll} = \emptyset$.

With the help of the configuration space, the notions of *path*, *collision free* and *minimal cost* can be defined more formally. In [83] Van den Berg defines the notions as follows: *"A path is defined as a continuous function $\pi : [0, L] \rightarrow \mathcal{C}$, parametrized by the length L of the path. The path planning problem is to find a (collision-)free path between a given start configuration $s \in \mathcal{C}$ and goal configuration $g \in \mathcal{C}$. Formulated in terms of the configuration space $\mathcal{C}$, that is finding a path π such that $\pi(0) = s$ and $\pi(L) = g$, and $\forall t \in [0, L] \colon \pi(t) \in \mathcal{C}_{free}$"*.

Note that in [55] it was shown, that in general the path planning problem is PSPACE-hard, which implies NP-hardness and that the runtime increases exponentially with every DOF of the robot, and thus with every dimension of the configuration space [83] [42].

Let $\mathcal{P}$ be the set of all paths satisfying the previous condition. We defined $c : \mathcal{P} \rightarrow \mathbb{N}$ as the cost function, which assigns a non-negative cost to every path. Since we are interested in the path with minimal costs (the optimal path), we search for $\min_{p \in \mathcal{P}} \{c(p)\}$.

We will see that we have to refrain from finding the optimal path when the dimension of $\mathcal{C}$ gets too high, and instead be content with any valid path.

Now one could assume, that it is unavoidable to construct $\mathcal{C}_{coll}$ to find such a path π. Later we will see that this assumption is not true and static path planning algorithms can be classified whether they need an explicit representation of $\mathcal{C}_{coll}$ or not. The former ones are known as *sampling based path planners* (see Section 2.3.2) the later ones as *combinatorial path planners* (see Section 2.3.1).

2.1.2 Frames

An important concept in robotics are *coordinate frames*, often just called *frames*. A *frame* is defined by the origin and the basis vectors of the coordinate system. One such frame is the *global frame* in which the position of the robot and all obstacles can be expressed. Another important frame is the *robot frame*, which can be imagined as a coordinate frame glued onto the robot. Thus, a part of a robot has always position p in the robot frame, regardless of the robot movement. Of course, this is not true for the position of the robot part in the global frame. When the robot moves, the position of the robot and thus of all its parts, changes in the global frame.

2.2 Shortest path algorithms

In this chapter the famous *Dijkstra algorithm* [14], which finds the *shortest paths* from a single start vertex to every other reachable vertex in a graph, will be introduced. As we will see, the Dijkstra algorithm solves the path planning problem in discrete worlds with special constraint robot geometry. The configuration space of a discrete world is a countable (in)finite set. Even though the real world is continuous, it is worthwhile to examine the algorithm, since most of the following solutions of optimal path planning in continuous worlds will be reduced to a discrete problem. After discussing the Dijkstra Algorithm, we will examine a second discrete shortest path algorithm, *A** [28], which extends the Dijkstra algorithm by using heuristics to speed up the process of finding the shortest path to the goal node.

2.2.1 Dijkstra

Assume a graph $G = (V, E)$, where V is the set of vertices and $E \subset V \times V$ is the set of edges that connect two vertices. Furthermore, we have a function $c : E \to \mathbb{N}$ assigning a non-negative cost to each edge $\{u, v\} \in E$. Given this ingredients the Dijkstra algorithm is able to find the shortest paths from a start vertex $s \in V$ to all reachable vertices in the graph G.

Depending on which parameter should be minimized, the cost function $c(\{u, v\})$ for all $\{u, v\} \in E$ can represent different costs. The most common one is the distance between u and v, but other possibilities, especially in path planning, could be travel time, clearance, safety or needed energy.

Algorithm 1 shows the pseudo code of the Dijkstra algorithm as in [83].

For every vertex v the algorithm stores the current optimal cost from the start vertex s to v, denoted by $g(v)$. Furthermore, a backpointer $bp(v)$ for every vertex stores the predecessor vertex from which the start vertex is reachable via the shortest path. Hence, by following this backpointers started by an arbitrary vertex v the shortest path between s and v can be determined.

At the beginning the true optimal costs from s to all other vertices are unknown, therefore $g(v)$ is set to infinity (line 2). Since reaching the start vertex s from itself has no cost, $g(s)$ is set to zero (line 4). To determine which vertex the algorithm should handle next, a queue Q sorted ascending by the current optimal cost g is maintained. The last initialisation step is to add s into Q (line 5).

After this initialisation, the algorithm repeats the next steps, until no further vertices are stored in the queue Q, and thus, the shortest paths from s to all other reachable vertices are

Algorithm 1 Dijkstra

```
1: for all v ∈ V do
2:     g(v) ← ∞
3: end for
4: g(s) ← 0
5: Insert s into Q
6: repeat
7:     v ← element from Q with minimal g(v)
8:     Remove v from Q
9:     for all neighbours u of v do
10:        if g(v) + c(v, u) < g(u) then
11:            g(u) ← g(v) + c(v, u)
12:            Insert or update u in Q
13:            bp(u) ← v
14:        end if
15:    end for
16: until Q ≠ ∅
```

found. The first vertex v is taken from Q (line 7 and 8) and for every neighbour u it is checked if u is reachable with less cost then before (line 10). The new cost to reach u is determined by the cost to reach v ($g(v)$) plus the cost to reach u from v ($c(v, u)$) If u is reachable with less cost, $g(u)$ gets updated and u gets either inserted into Q or updated if it is already present (line 11 and 12). To recover the shortest path later, the backpointer of u is set to v (line 13).

Hence, starting at s, the costs to reach a vertex gets propagated through the graph until every reachable vertex has a non finite cost and a valid backpointer. Note that the algorithm can easily be extended to work on directed graphs.

The following example will illustrate the behaviour of the Dijkstra algorithm. Figure 2.1 shows an example graph, where the number besides each edge represents the cost function c. With the help of the Dijkstra algorithm we compute the shortest path from s to every other vertex. Table 2.1 illustrates the steps of the Dijkstra algorithm by showing the queue Q and the current optimal cost g of every vertex at every step. Step 0 indicates the state of the algorithm after initialisation (after line 5 in Algorithm 1). Then the algorithm gets executed until the shortest paths, which are shown in Figure 2.2, are found.

The runtime of the Dijkstra algorithm depends on the implementation of the queue Q. If a binary sorted tree is used, the runtime is $O(E\ log\ V)$, whereas the optimal runtime of $O(V\ log\ V + E)$ can be achieved by using a Fibonacci heap [22] [83]. Furthermore, if only the shortest path from the start vertex to some specific vertex is needed, the algorithm can be terminated earlier when the shortest path to the goal vertex is found. The shortest path to a vertex is found, after the vertex is removed from Q and lines 9 to 15 are executed for this vertex. This is valid, since the current cheapest path is always pursued and when the mentioned condition is met, no cheaper paths to this vertex can exist. Unfortunately, this does not lower the upper bound of the runtime of the algorithm, since the goal vertex can be the last visited one, as in the

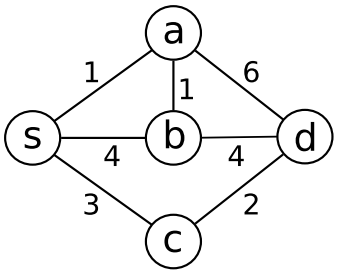


Figure 2.1: A graph for the Dijkstra example

step	Q	$g(s)$	$g(a)$	$g(b)$	$g(c)$	$g(d)$	$bp(s)$	$bp(a)$	$bp(b)$	$bp(c)$	$bp(d)$
0	s	0	∞	∞	∞	∞	null	null	null	null	null
1	a, c, b	0	1	4	3	∞	null	s	s	s	null
2	b, c, d	0	1	2	3	7	null	s	a	s	a
3	c, d	0	1	2	3	6	null	s	a	s	b
4	d	0	1	2	3	5	null	s	a	s	c
5	$\emptyset$	0	1	2	3	5	null	s	a	s	c

Table 2.1: Steps in the Dijkstra algorithm to find the shortest path in Figure 2.1

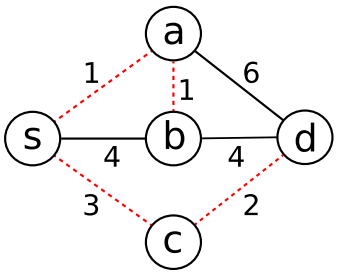


Figure 2.2: The dashed lines represent the shortest path from s to all vertices

example.

S

Assume a discrete world in which a robot is operating. Hence, the corresponding configuration space $\mathcal{C}$ is discrete too and can be transformed into a graph, where the costs of cells containing obstacles are either set to an invalid value or removed from the resulting graph. When the robot has only point-size, the Dijkstra algorithm solves the optimal discrete path planning problem by finding the cheapest collision free path. Collision free, since the robot will travel via cells visited by the Dijkstra algorithm. By construction, only cells which do not contain an obstacle can be visited by the Djikstra algorithm. At first sight the assumptions of a discrete world and a special robot geometry seem restrictive and not achievable in practice. But we will see later that most solutions for continuous worlds and arbitrary robot shape will be reduced to this approach.

2.2.2 A*

The Dijkstra algorithm is designed to find the shortest paths to all vertices in a graph from a given start vertex. In path planning only the cheapest path between a start node and a goal node is needed. This opens the question, if there are more suitable algorithms for this case. Indeed, the A* algorithm [28] finds the shortest path between two nodes in less or equal number of steps compared to Dijkstra. This is even true, when the Dijkstra algorithm is terminated after the shortest path to the goal is found. A* focuses the search in the graph by a heuristic function $h(v)$, which estimates the cost to go from vertex v to v_{goal}. When $h(v)$ is less or equal to the true cost for all $v \in V$ it is guaranteed that A* will find the optimal path [20] [19].

Algorithm 2 shows the pseudo code of A* as in [83].

Algorithm 2 A*

```
1:  for all v ∈ V do
2:      g(v) ← ∞
3:      h(v) ← lower-bound estimate of the distance between v and v_goal
4:  end for
5:  g(s) ← 0
6:  Insert s into Q
7:  repeat
8:      v ← element from Q with minimal (g(v) + h(v))
9:      Remove v from Q
10:     for all neighbours u of v do
11:         if g(v) + c(v, u) < g(u) then
12:             g(u) ← g(v) + c(v, u)
13:             Insert or update u in Q
14:             bp(u) ← v
15:         end if
16:     end for
17: until v = v_goal or Q ≠ ∅
```

A* is very similar to the Dijkstra algorithm, but there are two differences. First, the key for vertices sorted in queue Q is the sum of the current best cost $g(v)$ and the heuristic function $h(v)$. Second, when v_{goal} is removed from queue Q, the algorithm terminates. Both differences explain the reason why the heuristic has to return a value less or equal to the true cost to go from v to v_{goal}. If this is the case the keys are bound, which guarantees that nodes on cheaper paths will have smaller keys. Hence, when the goal is reached no cheaper path than the current one can exist.

Assume a graph embedded in the plane, where the cost function $c(\{u, v\})$ for all $\{u, v\} \in E$ expresses the Euclidean distance between u and v. A suitable heuristic for this example could be the Euclidean distance to the goal. This choice satisfies the requirements since in this scenario no path can be shorter than the Euclidean distance. The closer the return value of the heuristic is to the true cost to the goal, the better the performance of A* [45] [52]. When $h(v) = 0$ for all $v \in V$, A* is equal to the Dijkstra algorithm with early termination.

Figure 2.3 shows how many cells the Dijkstra- and the A* algorithm need to find the shortest path from the start (diagonal cross) to the goal (symmetric cross) in a simple example. It is assumed that every empty cell (white) has a cost of 1 and occupied cells (black) have infinite costs. Furthermore, it is assumed that the right cell gets added to Q before the bottom one. The number in a cell describes at which step the cell is visited. The heuristic $h(v)$ of A* returns the Manhattan distance, which is the length of the path with only left, right, up or down movements along the axis of the grid. In this example the heuristic returns the exact cost to go. This fact guarantees that A* does find the goal in minimum number of steps and no unnecessary cells are visited. Furthermore, note, that the Dijkstra algorithm terminates after it finds the shortest path to the goal, thus the last remaining cell is not visited.

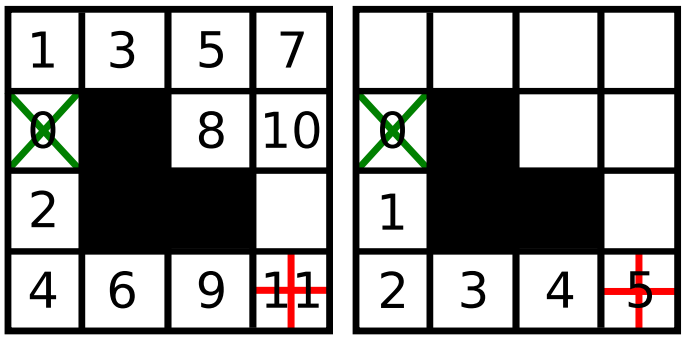


Figure 2.3: Cells visited in the Dijkstra- (a) and A* algorithm (b)

Best-first

The *best-first* algorithm is like A*, but Q only gets sorted by a *cost to go* heuristic $h(v)$. Furthermore, it is not required that $h(v)$ underestimates the true cost to go from v to the goal, which was important for A* to compute the optimal path. Thus, the solution of best-first will in general not be the path with minimal cost. The advantage of this - at first sight inferior - algorithm compared to A* is, that in many cases fewer vertices have to be visited until the goal is reached, which decreases the run time [45]. Unfortunately, this is not guaranteed and in the worst case the best-first algorithm needs longer than A*, since it is too greedy [45].

Later, we will see that for higher-dimensional problems only search for a valid path is relevant instead of an optimal path, due the complexity of path planning. Thus, best-first has its justification.

Up until now, only solutions for optimal path planning in discrete worlds were discussed. Unfortunately, the real world is continuous and not discrete. Thus, the next section will show algorithms to solve path planning in continuous worlds. Fortunately, the approaches for the discrete world can be reused.

2.3 Continuous path planning

Solutions for path planning in continuous worlds can roughly be categorized into two categories. *Combinatorial path planning* (Section 2.3.1) relies on the fact that $\mathcal{C}_{coll}$ is explicitly constructed. In contrast, *sampling based path planning* (Section 2.3.2) does not construct $\mathcal{C}_{coll}$ explicitly, but samples the configuration space $\mathcal{C}$ and uses fast collision checking methods to determine if a collision will occur.

2.3.1 Combinatorial path planning

Combinatorial path planners need an explicitly constructed $\mathcal{C}_{coll}$ to solve the path planning problem in continuous worlds. Unfortunately, constructing $\mathcal{C}_{coll}$ is a computationally expensive task. Thus, combinatorial path planners can only be used for low-dimensional configuration spaces [83].

On the other hand, combinatorial path planners are complete, as opposed to sampling based path planners. A path planner is complete, if it either finds a solution or it will correctly report that no solution exists [26].

Constructing $\mathcal{C}_{coll}$ explicitly

When $\mathcal{C}_{coll}$ is constructed explicitly, the robot can simply be seen as a point in the configuration space [83]. As long as the point stays out of $\mathcal{C}_{coll}$ no part of the robot will collide with an obstacle. If the robot is exactly at the border of $\mathcal{C}_{coll}$, the robot is "scratching" at the obstacle. To do this, the geometry of the robot has to be added to the obstacles in a prober way. Here, one can see that the requirement of point size robots for the shortest path algorithms is fulfilled for combinatorial path planners (see Section 2.2).

The easiest case for explicitly constructing $\mathcal{C}_{coll}$ is when the world is $\mathbb{R}^n$ for $n = 1$, 2 or 3 and the robot is only capable of translation. Under this assumptions $\mathcal{C}_{coll}$ can be constructed with the help of the *Minkowski difference*. For any two sets $X, Y \subset \mathbb{R}^n$, the Minkowski difference is defined as:

$$X \ominus Y = \{x - y \in \mathbb{R}^n \mid x \in X, y \in Y\}$$

in which $x - y$ is just vector subtraction in $\mathbb{R}^n$ [45] [48]. Let $\mathcal{O}$ be the union of all obstacles described as convex polyhedra (or polygons) and $\mathcal{A}$ be the convex polyhedra describing the robot geometry. Then we get $\mathcal{C}_{coll} = \mathcal{O} \ominus \mathcal{A}$. The statement is also true for non convex polyhedra, since they can be expressed by unions of convex polyhedra.

In [48] Lozano-Pérez gives an efficient algorithm to calculate $\mathcal{C}_{coll}$ using the Minkowski difference in a 2D world, which can be extended for 3D worlds. With the previously mentioned properties, the resulting $\mathcal{C}_{coll}$ is a polyhedral too. The idea of this algorithm shall be presented with the help of an example. The mathematical details and proofs can be found in [48].

Assume a single rectangular obstacle and a triangular robot (see Figure 2.4), where the arrows stemming from the black dot showing the robot frame . Each object is described by the edges forming the boundary and each edge e consists of two points $e.1$ and $e.2$. Since objects are polyhedra, each point of each edge coincides with exactly one point of another edge.

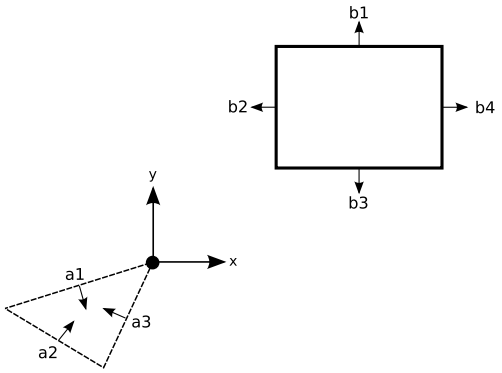


Figure 2.4: A triangular robot and a rectangular obstacle

The algorithm to calculate $\mathcal{C}_{coll}$ can be imagined as sliding the robot around the obstacle while keeping them both in contact (see Figure 2.5). It can be seen, that each edge of $\mathcal{A}$ and $\mathcal{O}$ forms a part of the resulting boundary of $\mathcal{C}_{coll}$. Thus, the algorithm has to determine the position of each edge, by computing its translation.

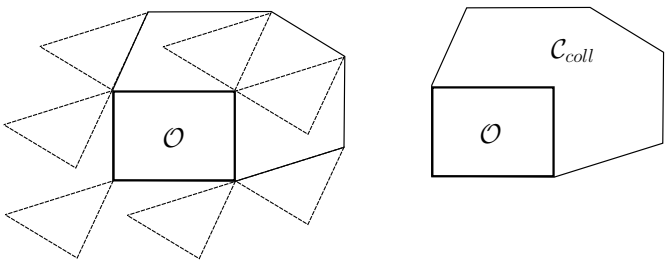


Figure 2.5: Sliding the robot around the obstacle creates $\mathcal{C}_{coll}$ for Figure 2.4

The algorithm uses the inward edge normals of the robot and the outward edge normals of the obstacle, sorted by the angle they make with the x-axis (see Figure 2.6 for a presentation in a circular fashion).

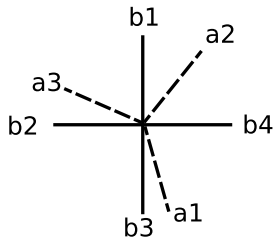


Figure 2.6: The normals of Figure 2.4 sorted in circular fashion

To determine the translation of every edge, the algorithm proceeds as follows. Starting at one edge normal v, it is checked if the other geometric object has an edge with the same normal w. If

this is the case, the edge e_v corresponding to v gets subtracted from the edge e_w corresponding to w. Since an edge is represented as two points, this subtraction is actually two subtractions, one for each point pair ($e_v.1 - e_w.1$ and $e_v.2 - e_w.2$).

If no such edge exists, the nearest normals x and y of the other geometric object to the left and right in the sorted data structure are selected. Since the normals are sorted and the object is polyhedral, the corresponding edges e_x and e_y of the two selected normals must be connected via a single point p. Hence, p is one point of the edge e_x and one point of the edge e_y. The sorting of the normals guarantees that p exists. For example, the edges corresponding to b1 and b2 are connected via a point forming the left upper corner of the obstacle. Consequently, in one way no other obstacle normal is between b1 and b2 in the sorted data structure. The edges corresponding to b1 and b3 are not connected via a single point. This is represented by the fact, that either b2 or b4 is between b1 and b3 in the sorted data structure. To gain the translation of edge e_v, subtract point p from the two points forming this edge ($e_v.1 - p$ and $e_v.2 - p$).

Note that the points forming the obstacle edges are in the global frame, and the the points forming the robot edges are in the robot frame. For example normal b2 has a1 and a3 as nearest normals from the robot. The corresponding edges of the normals a1 and a3 are connected via a point direct at the origin of the robot frame. Thus, the corresponding edge e_{b2} is not translated:

$$e_{b2}.1 - \begin{pmatrix} 0 \\ 0 \end{pmatrix} = e_{b2}.1$$
$$e_{b2}.2 - \begin{pmatrix} 0 \\ 0 \end{pmatrix} = e_{b2}.2$$

The same is true for normal b3. This procedure is repeated with the next normal in counter-clockwise direction until all normals have been handled once.

The algorithm has a worst case run time of $O(m+n)$ where m is the number of edges of $\mathcal{A}$ and n is the number of edges of $\mathcal{O}$. This runtime is only possible since the angles already appear in counter clockwise direction at $\mathcal{A}$ and $\mathcal{O}$.

Unfortunately, constructing $\mathcal{C}_{coll}$ gets more complicated when the robot is capable of rotation too. In the translation only case, the boundary of $\mathcal{C}_{free}$ is piecewise linear since translation is a linear operation [43] and the boundaries of $\mathcal{A}$ and $\mathcal{O}$ are polyhedral. Since rotations are not linear, this is not true for the case of a robot capable of performing rotation. Therefore, the previous piecewise linear representations are replaced by *semi-algebraic* representations. This means that the faces of $\mathcal{A}$, $\mathcal{O}$ and $\mathcal{C}_{coll}$ are represented by roots of multiple real valued polynomes. In geometry a face describes a part of the boundary surface of a solid obstacle. One consequence of the non-linearity of rotation is that the faces are curved and no longer polygonal [3] [48]. Constructing $\mathcal{C}_{coll}$ is still possible in near-quadratic run time [27]. Unfortunately, in general a combinatorial explosion occurs that produces too many facets [43] to use $\mathcal{C}_{coll}$ reasonably in subsequent path planners when the dimension of $\mathcal{C}$ gets too high.

General idea

Virtually all combinatorial path planners construct some sort of *roadmap*. Assume a graph $\mathcal{G} = (V, E)$, where each vertex is some configuration of $\mathcal{C}_{free}$ and each edge is some path

through $\mathcal{C}_{free}$ connecting two vertices. Graph $\mathcal{G}$ is called a roadmap when the following two requirements are met [41]:

1. **Coverage**: each configuration in $\mathcal{C}_{free}$ can easily be connected to $\mathcal{G}$

2. **Connectivity**: for all pairs of configurations in $\mathcal{G}$ between which a valid path exists in $\mathcal{C}_{free}$, a valid path through the roadmap exists as well. This requirement forces that solutions are not missed because $\mathcal{G}$ fails to map the connectivity of $\mathcal{C}_{free}$ [45].

After the construction of such a roadmap, it can be used for multiple path planning queries. If the start configuration of the robot, as well as the goal configuration is in $\mathcal{C}_{free}$, both can easily be added to $\mathcal{G}$ by the coverage property. Since a graph is constructed and the robot is only point size due to construction of $\mathcal{C}_{coll}$ the shortest path planning algorithms from Section 2.2 can be used to find the optimal path from start to goal. Thus, once such a roadmap is constructed the method is complete, since it will either find a solution or it will correctly report that no solution exists [83].

Various possibilities exist to construct a roadmap. Some of them are only possible in 2D configuration spaces or generate poor results in higher-dimensional ones.

Cell decomposition

One common way to construct a roadmap is by decomposing $\mathcal{C}_{free}$ into a finite number of cells. Then one configuration for each cell and one on each border between the cells are designated as sample points which serve as vertices of the roadmap. Which configuration gets selected is not of particular importance, so the cell centroids are good choices [43]. For each cell an edge from its sample point to the sample point on the borders is added to the roadmap. Each edge is a line-segment path between the sample points of the cells. See Figure 2.7 for such a constructed roadmap.

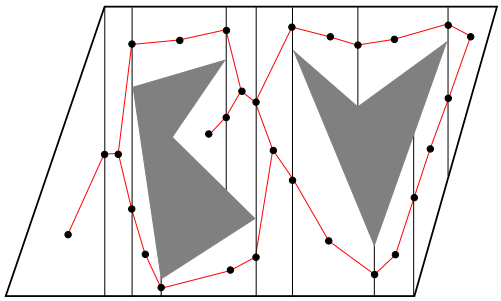


Figure 2.7: Roadmap resulting from the vertical cell decomposition

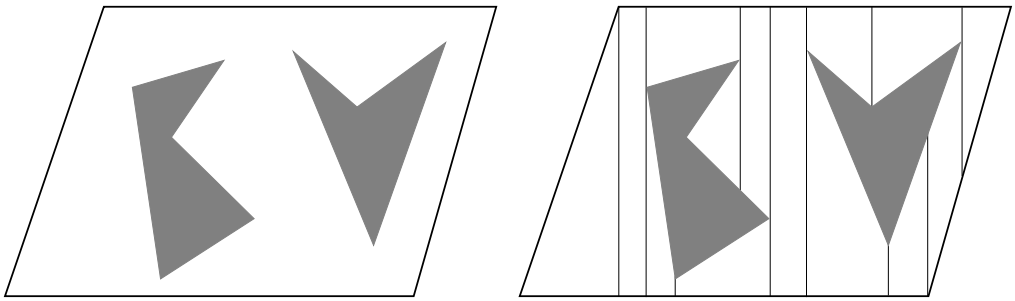
Again various different *cell decompositions* are possible, but [45] reports three properties a cell decomposition should satisfy:

1. Computing a path from one configuration to another configuration in a cell must be easy.

2. For every cell the adjacency information can easily be extracted to build the roadmap.

3. For a given start and end configuration, it should be easy to determine the cells which contain them.

One cell decomposition that satisfies all three properties is the *vertical cell decomposition*, which decomposes $\mathcal{C}_{free}$ such that every cell is either a trapezoid with vertical sides or a triangle (which is a degenerated trapezoid). Thus, this technique is also known as *trapezoidal decomposition*. This decomposition is also valid for configuration spaces with a higher dimension than two, as long as $\mathcal{C}_{coll}$ is polyhedral. To obtain this decomposition, try to extend rays upward and downward through $\mathcal{C}_{free}$ at every vertex which defines $\mathcal{C}_{coll}$, until $\mathcal{C}_{coll}$ is hit [45]. Figure 2.8 shows $\mathcal{C}$ before 2.8(a) and after 2.8(b) the vertical cell decomposition.



(a) (b)

Figure 2.8: Vertical cell decomposition

The resulting roadmap (see Figure 2.7) satisfies the coverage requirement because each cell is convex. Thus, every vertex can be connected to the corresponding cell sample point by a straight path. The connectivity requirement is satisfied too, since the roadmap is directly derived from the cell decomposition, which also preserves the connectivity of $\mathcal{C}_{free}$ [45].

With the help of the *plane-sweep* (or *line-sweep*) principle the worst case runtime to compute the decomposition is $O(n\ log\ n)$ where n is the number of vertices which define $\mathcal{C}_{coll}$. The line-sweep principle is an important idea from computational geometry and is the basis of many cell decompositions in combinatorial path planning [45]. The name stems from the fact, that some imaginary line sweeps across the space, stopping only where the underlying data features interesting properties. This stopping is called an event. At the vertical cell decomposition this imaginary line sweeps horizontally across the x-axis and stops at every vertex describing $\mathcal{C}_{coll}$. Firstly all of these vertices get sorted ascending by their x-coordinate. Furthermore, the algorithm maintains a list L, used to store some edges of $\mathcal{C}_{coll}$, implemented as a balanced binary search tree. Thus, it is possible to determine the edges above and below one vertex in $O(log\ n)$. At every halt of the line, edges get added or deleted from L, depending on whether the triggering vertex is at the beginning or the end of an edge. Adding and deleting elements to a balanced binary search tree can be done in $O(log\ n)$. Since the line stops n times, the runtime of the whole algorithm is $O(n\ log\ n)$. See Figure 2.9 and Table 2.2 for the execution steps of the line-sweep algorithm for the vertical cell decomposition.

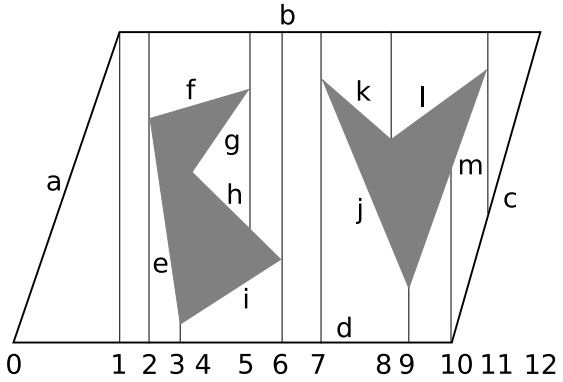


Figure 2.9: All events for the line-sweep algorithm

event	L	event	L
0	$\{d, a\}$	7	$\{d, j, k, b\}$
1	$\{d, b\}$	8	$\{d, j, l, b\}$
2	$\{d, e, f, b\}$	9	$\{d, m, l, b\}$
3	$\{d, i, f, b\}$	10	$\{c, m, l, b\}$
4	$\{d, i, h, g, f, b\}$	11	$\{c, b\}$
5	$\{d, i, h, b\}$	12	$\{\}$
6	$\{d, b\}$		

Table 2.2: Steps in the line-sweep algorithm

The last step to solve the continuous optimal path planning problem is to determine how to connect the start and stop configuration (s, g) to the roadmap. First, the cells which s and g contain have to be found, which should, by the requirement for a cell decomposition, be easy. If either s or g are not in $\mathcal{C}_{free}$, no cell will be found and the algorithm will correctly report that no solution is possible. Assume the cells C_s and C_g contain s respectively g. Then connect s to the sample point of C_s and g to the sample point of C_g. Again, this should by easy by the first requirement for a cell decomposition. Hence, we got a graph including start and goal which can be searched by the shortest path algorithms from Section 2.2 to find the optimal path. Figure 2.10 shows the result for a sample search.

The vertical cell decomposition offers a nice balance between the number of generated cells and computational efficiency [45]. In general it is preferable to generate less cells to reduce the number of vertices in the resulting roadmap. Unfortunately, it is difficult to optimize the number of cells, since determining the cell composition which produces the least number of convex cells in a polygonal $\mathcal{C}_{coll}$ with holes is NP-Hard [47].

Another possible cell decomposition is the *cylindrical cell decomposition* which is very similar to the vertical cell decomposition. The difference is that the lines spawning from the vertices do not stop at the border of $\mathcal{C}_{coll}$ but go all the way from $y = -\infty$ to $y = \infty$. Figure 2.11(a) shows the cylindrical cell decomposition of Figure 2.8(a). The cylindrical cell decomposition seems to be inferior to the vertical cell decomposition at first sight, since it generates more cells,

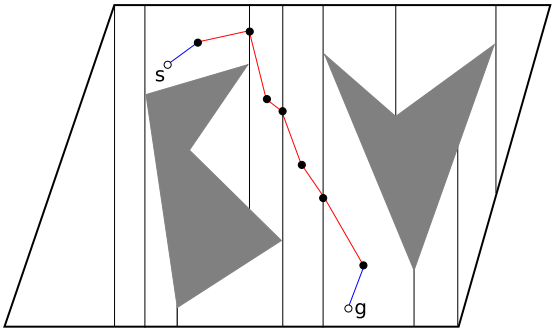
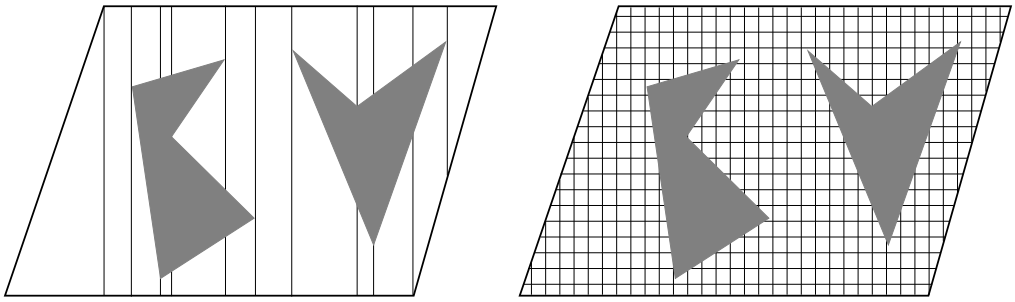


Figure 2.10: Result of a sample search

but it generalises nicely to any dimension and any C-space topology, in contrast to the vertical cell decomposition [45].



(a) (b)

Figure 2.11: Cylindrical and grid cell decomposition of Figure 2.8(a)

The easiest possible, but uncommon way to obtain a cell decomposition is to discretise $\mathcal{C}_{free}$ via a grid as shown in Figure 2.11(b). Only cells which do not contain an obstacle get a sample point assigned and instead of connecting each cell sample point with the border point, the sample points get directly connected to their neighbours cell sample point. Neighbouring cells are defined by the *k-neighbourhood*.

Lets name the set of neighbour cells for a cell the *neighbourhood*. [45] defines the *k-neighbourhood* for vertices in a graph. This definition can be easily adapted for cells in grids. Define the *1-neighbourhood* $N_1(v)$ for cell c as

$$N_1(c) = \{c + \Delta q_1, \ldots, c + \Delta q_n, c - \Delta q_1, \ldots, c - \Delta q_n\}$$

for an n-dimensional grid, where Δq_i for $i = 1 \ldots n$ are the grid resolutions of the axes. In a two-dimensional grid this yields four 1-neighbours, which are the cells directly to the left, top, right and bottom of c. In general, there are at most $2n$ 1-neighbours for a grid cell [45], since grid cells at the border have fewer neighbours. The 2-neighbourhood $N_2(c)$ for c can be defined as in [45]:

$$N_2(c) = \{c \pm \Delta q_i \pm \Delta q_j \mid 1 \leq i, j \leq n, i \neq j\} \cup N_1(c)$$

Similar to the 2-neighbourhood a *k-neighbourhood* can be defined for a positive integer $k \leq n$ [45].

The trade-off for this simple cell decomposition is obviously the huge number of cells and therefore a huge number of vertices in the roadmap $\mathcal{G}$. Thus, the burden of the cell decomposition gets transferred to the shortest path algorithms from Section 2.2. The grid cell decomposition can be applied to every configuration space, regardless of its dimension or topology, but one should keep in mind that at higher dimensions an explosion of cells and thus vertices will happen.

Visibility graph

There are ways to construct the roadmap $\mathcal{G}$ other than by decomposing $\mathcal{C}_{free}$ into cells. One such method is the *visibility graph* method described in [51]. Let every polygon vertex be a *reflex vertex* for which the interior angle (in $\mathcal{C}_{free}$) is greater than the number π. As long as no three consecutive vertices are collinear, every vertex of a convex polygon is a reflex vertex [45]. Let all reflex vertices which describe $\mathcal{C}_{coll}$ be the vertices of the roadmap $\mathcal{G}$. An edge between two vertices is added, if they are mutually visible. Figure 2.12(a) shows the visibility graph for our example configuration space.

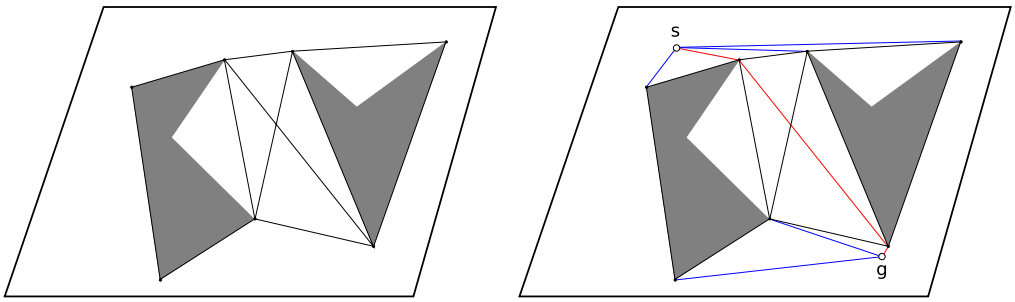


Figure 2.12: Shortest-Path roadmaps of Figure 2.8(a)

In fact, the resulting edges are the shortest possible paths between the vertices, which explains the alternative name of *shortest-path roadmap*. The start and the goal configuration of a query get connected to the roadmap via edges to all visible vertices. Again, the algorithms from Section 2.2 can be used to find the optimal path (see Figure 2.12(b)). When adopting the plane-sweep principle to *radial sweep* the visibility graph can be computed in $O(n^2\ log\ n)$ where n is the number of vertices of $\mathcal{C}_{coll}$. In [25] an algorithm which computes the visibility graph in $O(n\ log\ n + k)$ is presented, where k is the total number of edges in the roadmap.

Unfortunately, the visibility graph method has two flaws. First, a visibility graph encoding the shortest paths does not exist in configuration spaces with dimensions higher than two [83]. Second, the paths are directly on the border of $\mathcal{C}_{coll}$. Thus, the robot is constantly scratching at the obstacles on these paths. When the imperfect localisation and the inability of a robot to follow a path exactly is added, the resulting paths can not be used as solutions. To circumvent

this problem, the paths have to be adjusted so that they come close to $\mathcal{C}_{coll}$ but leave enough margin for localisation and path following errors.

2.3.2 Sampling based path planning

General Overview

Since $\mathcal{C}_{coll}$ is composed of an unwieldy number of facets [43] and due to the general complexity of path planning, combinatorial path planners are too slow to be used frequently in practice [13]. Thus, a different category of approaches was developed. In *sampling based path planning* the explicit construction of $\mathcal{C}_{coll}$ is avoided and instead $\mathcal{C}$ (not $\mathcal{C}_{free}$) is either deterministically or randomly sampled. To check if a sampled configuration is valid, a *collision detection* algorithm, which can quickly determine if two three-dimensional objects collide, is used. The restriction to three-dimensional objects is reasonable, since the underlying robot and obstacle geometry is at most three-dimensional even in higher-dimensional configuration spaces [83].

The drawback of sampling based path planners is that in contrast to combinatorial path planners they are not complete, since they do not guarantee to report whether a solution exists [45]. Instead, some weaker notions of completeness are introduced. Sampling based path planners using deterministic sampling can be *resolution complete*. This means that they will find a solution in finite time, when there is an existing solution, but they might search infinitely long when no solution is possible [45]. In contrast, a sampling based path planner using random sampling can be *probabilistically complete*, meaning that when a solution exists, the probability to find a solution converges to 1, as the run time approaches infinity [78].

In all presented combinatorial path planners, a special data structure (the roadmap $\mathcal{G}$) was constructed, which enables them to process multiple queries for a path between arbitrary start and goal configurations afterwards. The aim is to invest much time to build this roadmap, such that these queries can be processed easily and as fast as possible, which has significant advantages in real-time applications. All methods which follow this idea are called *multiple-query* planners. If this precomputation is either difficult or not reasonable (like in changing environments) *single-query* planners, which solve a query without expensive precomputations should be used.

Sampling based path planners for both approaches exist, but first a method is presented, which follows the single-query model. Most single-query sampling based path planners execute the following general steps [45]:

1. **Initialisation**: Create *search graph* $\mathcal{G}(V, E)$ with at least one vertex, typically s, g or both and no edges.

2. **Application of Vertex Selection Method (VSM)**: Choose a vertex $q_{curr} \in V$ for expansion.

3. **Application of Local Planning Method (LPM)**: Via sampling attain a configuration $q_{new} \in \mathcal{C}$ and create a path π between q_{curr} and q_{new}. Then check that π does not cause a collision with the help of the collision detector . If no collision free path could be created, go back to step 2.

4. **Insert an edge in the graph**: If q_{new} is not already present in V add it, then add the path π as edge from q_{curr} to q_{new} to E.

5. **Check for a solution**: Check if g can be reached, starting at s in search graph G.

6. **Return to Step 2**: Repeat all steps until a solution is found or a user defined termination condition is satisfied, in which the algorithm will terminate and return an error.

At step 3, Local Planning Method (LPM) a *local planner* tries to connect both configurations via a simple collision free path, in most implementations just a straight line [83] [24]. It is called local, since it does not try to solve the global path planning problem. It is expected that the local planner will fail often to connect two configurations and will only be successful in simple cases. This is the case since extensive experiments have shown that it is more important for a local planner to be fast than to be so powerful to have a high success rate [23]. The local planner will use the collision checker to identify if the path is valid. This is done by checking intermediate configurations on the path in an often pre-set resolution [45]. This parameter is critical since if the resolution is too fine, too much time is wasted on collision checking. On the other hand, if the resolution is too coarse a colliding configuration is overlooked, resulting in an accepted but invalid path.

Deterministic sampling

Again, the easiest way to implement a sampling based path planning algorithm is discretization of $\mathcal{C}$ (not $\mathcal{C}_{free}$ as in combinatorial path planning with a grid cell decomposition) as in via a grid. The resulting configurations (vertices of $\mathcal{G}$) are connected via paths with their respective neighbours as defined by the k-neighbourhood. Figure 2.13(a) shows such a grid using 1-neighbourhoods. When the start and stop configuration do not correspond with the grid's vertices, they need to be connected to the graph $\mathcal{G}$. Various possibilities exist, but as a general rule, the start and goal vertex should be connected to every vertex closer than the grid resolution [45] (see Figure 2.13(b)). Typically, obstacles are present, and therefore some grid points and connecting paths are not present in $\mathcal{G}$, since a robot at this locations would collide with the obstacles (see Figure 2.13(c)). After the start and goal vertex are connected to the graph, the shortest path algorithms from Section 2.2 can be used to find the optimal path (see Figure 2.13(d)).

There are two possible time-points to check for collisions, depending on whether a single-query or multi-query planner should be achieved. When the single-query philosophy should be applied, the algorithms from Section 2.2 are slightly modified to incorporate a collision checker to reveal the colliding vertices and paths "on the fly". On the other hand, when a multi-query planner is desired, all vertices and paths can be checked for collisions by a collision checker in a precomputing step, before the optimal path is searched. These differences reveal the strengths of both philosophies. If for any reason the graph is searched only once, a single-query planner using A* can save significant time, since the probability that not every grid vertex and path has to be checked for collisions is high. If on the other hand, the graph is used multiple times, all grid vertices and paths are checked exactly once which saves time in the long run.

One problem with this approach is to determine a good resolution. If the resolution is too low, a solution might not be found, although one exists. On the other hand, if the resolution is

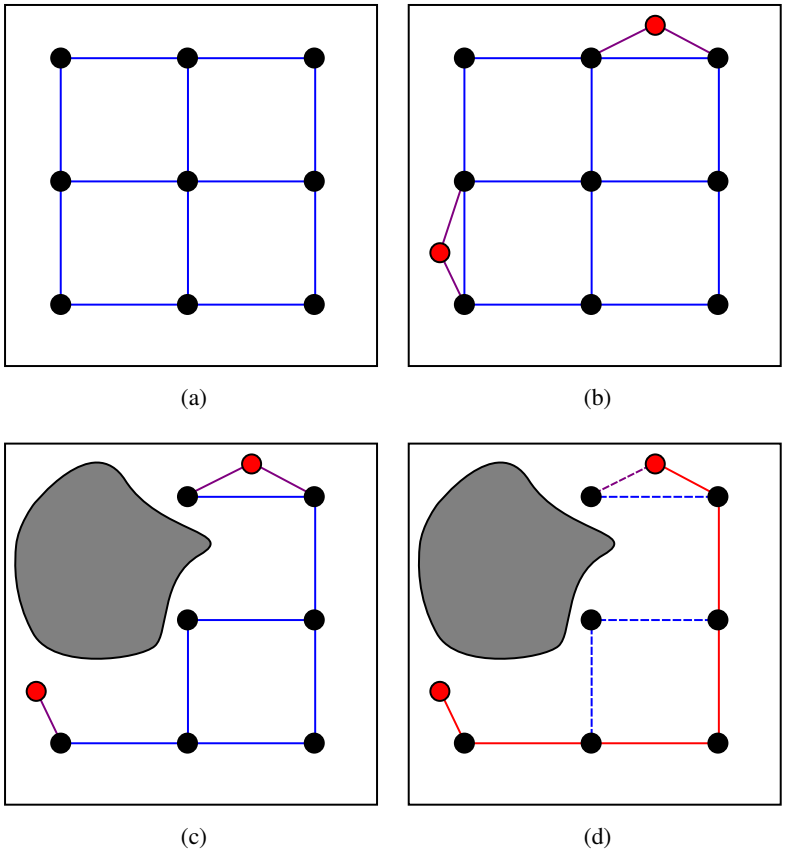


Figure 2.13: Grid Sampling Based Path Planner as depicted in [45]

too fine grained, the graph $\mathcal{G}$ gets too big, which is a problem, particularly in high-dimensional configuration spaces and the search becomes too slow. To solve this dilemma, the method gets adopted to iteratively refine the grid resolution until a solution is found, which results in interleaved sampling and searching. One approach to do this is to double the resolution after a search failed. Many of the vertices and edges can be reused and don't have to be checked again, but this profit diminishes rapidly in higher dimensions [45]. Assume a grid with 2^n points where n is the number of dimensions, resulting in two points per axis. Thus, after the first unsuccessful run the grid has 4^n points, after the second 8^n points and so on. The problem with this is, the bigger n is, the faster the growth of the number of grid points gets. For example, if $n = 10$ then 1024 vertices (grid points) are in $\mathcal{G}$ at the first iteration. In the next iteration it has already grown to more than 1 million vertices! A similar but superior method is to discard the information of the previous resolution by using grids with i^n grid points for iteration i. This yields 2^n, 3^n, 4^n points and so on which results in a better performance [45]. Discarding the information of the lower resolution grids seems reasonable, since the time saving of using them diminishes rapidly with higher dimensions.

Randomised potential fields

Using the shortest path algorithms from Section 2.2 in combination with a grid only works when the problem can be solved with a small number of points, to ensure that every vertex in $\mathcal{G}$ can be reached in reasonable time [45]. If the problem requires 50 points per dimension and has 10 dimensions, it is impossible to search all 50^{10} samples. In such a case a *best-first* heuristic can be used to find a solution without the need to visit most vertices in the graph. Unfortunately, for cases like shown in Figure 2.14, the best-first heuristic takes too long to reach the goal, since it has to explore all vertices in the semi-circle. This is the case, since the best-first heuristic prefers paths which will be blocked by the semi-circle.

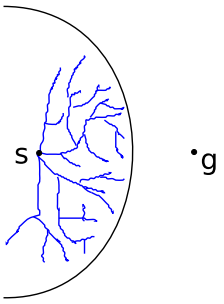


Figure 2.14: In such a semi-circle, best-first needs much time to reach the goal

To circumvent this behaviour, the *randomised potential field* approach [4] [6] uses *random walks* to escape situations where best-first is trapped at a local minimum. [45] reports that the randomised potential field approach was the first sampling based planner which used a special technique beyond classical discrete search in graphs and was very successful with it. The drawback is, that many heuristic parameters have to be set, depending on the problem [45].

The algorithm uses a *potential function* which is some form of cost function which tries to estimate the cost to go from the current node to the goal node, like the heuristic function h from A* (see Section 2.2.2). The difference lies in the fact that the potential function consists of an *attraction* term which draws the robot towards the goal and a *repulsion* term which repeals the robot from obstacles. Another analogy for a potential function are springs, drawing the robot to the goal but pushing it away from obstacles. Or imagine that the goal is on the bottom of a bowl and the obstacles are bumps. When the robot is a ball starting at the edge of the bowl, it will automatically roll towards the goal, avoiding obstacles. The spring and the bowl analogy represent stored potential energy and the robot tries to reach a lower energy state, which explains the name of the potential function. Furthermore, it is not required that h underestimates the true cost to go, as in A*.

The algorithm consists of three states and the VSM and LPM behaviour depends on the current state. Initially, a high resolution grid discretizes $\mathcal{C}$. The initial state of the algorithm is the best-first method, which is started at the start configuration. The VSM selects the newest added vertex $q_{curr} \in V$ via best-first and the LPM selects a vertex q_{new} in the neighbourhood of q_{curr} in a direction that minimizes h. This selection can either be done via random or deterministic sampling. If for some runs the best-first method was not able to decrease h, the algorithm

changes to the state *RANDOM WALK*, since best-first is stuck at a local minimum.

In the *RANDOM WALK* state, the algorithm tries to escape the local minimum via randomly changing the last added configuration (VSM). This is done by increasing or decreasing each component of the last configuration by the grid resolution depending on a fair coin flip (LPM). Via the collision checker it is checked if the obtained configuration is legal. When this check fails, the obtained configuration gets discarded. Otherwise it forms the basis of a new random walk. Random walks are repeated until either h gets decreased or a bound of maximum iterations is reached. The loop bound itself is determined by sampling a predetermined random variable. Then the algorithm state gets changed back to *BEST-FIRST*. Another bound K determines the maximum number of tries to enter *RANDOM WALK* when best-first fails ([5] reports $K = 20$ as reasonable value). When best-first fails after the *RANDOM WALK* state was entered K times, the algorithm changes to the *BACKTRACKING*- state. In *BACKTRACKING* a random vertex gets selected from all vertices obtained by *BEST-FIRST*, the counter gets reset and the state is changed back to *BEST-FIRST* to restart from the randomly selected vertex.

Due to the random walking, the resulting paths can be too complicated to follow for a robot. Fortunately, such paths can easily be transformed into smoother ones. This is done by iteratively connecting two randomly selected vertices from the path with a straight line if possible. This approach can be used for nearly all sampling based path planners.

The randomised potential fields approach is capable of escaping high-dimensional local minima and was able to solve problems up to 31 degrees of freedom which was a breakthrough in the early 1990s [45]. Unfortunately, a lot of parameters have to be tuned, which caused most people to switch to newer methods [45]. The two, perhaps most commonly used, planners are *rapidly exploring random tree* (a single-query planner) and *probability road map* (a multi-query planner) [17].

Rapidly exploring random trees

The *rapidly exploring random tree (RRT)* method introduced in [44] is an incremental sampling and searching algorithm which incrementally builds a search tree while gradually incrementing the resolution. It follows the single-query philosophy. Furthermore, RRT is resolution complete [44]. In contrast to the randomised potential field algorithm, no parameters are needed in general, but some implementation variants introduce different parameters. The basic idea is to get a random configuration α via sampling and to connect it to the nearest neighbour of the search tree usually via the shortest path.

Assume a graph with four vertices and three paths connecting them to a tree rooted at some vertex q_0 (see Figure 2.15(a)). When the nearest neighbour of the search graph is a vertex q_{curr} for the sampled configuration α as in Figure 2.15(b), both can be connected via a path. Alternatively, the nearest point can also be on an edge of the search graph as shown in Figure 2.15(c). In this case a new vertex q_{curr} is introduced at this point and the former path is split into two separate paths. The last possibility is that α is not reachable from the search graph, since it is either in $\mathcal{C}_{coll}$ or behind an obstacle blocking a possible shortest path. As in every sampling based planner this is checked by a collision detector. Independent of the reason why the sample is not reachable, a path is made from q_{curr} to the last possible collision free configuration q_{new} before the obstacle, as depicted in Figure 2.15(d).

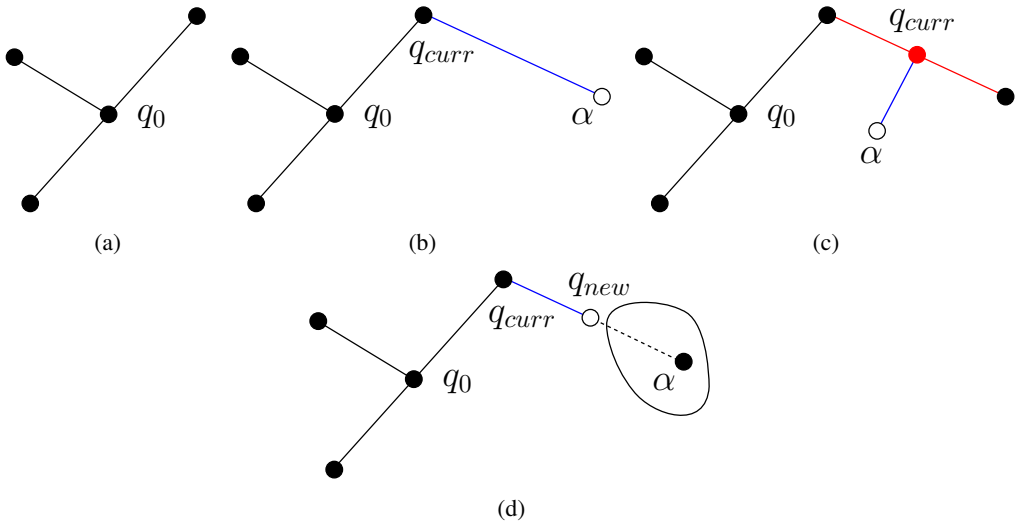


Figure 2.15: The initial situation and the different possibilities to connect q_{new} to the search tree as depicted in [45]

Algorithm 3 shows the pseudo code of the rapidly exploring random tree method as in [45]. In line 3, the nearest configuration in $\mathcal{G}$ to α gets selected by the function NEAREST. NEAREST basically is the VSM of the general framework presented before with a minor adaption. The adjustment lies in the fact, that q_{new} can also be a configuration which is not present in V beforehand, which happens when the nearest neighbour is on a path. Furthermore, it is assumed that the function NEAREST handles this case by splitting the path into two separate paths and adding a vertex at the splitting point. Also note that the NEAREST function does not take obstacles into account when the nearest neighbour is selected. In line 4, q_{new} is either set to the sampled configuration or to the nearest collision free configuration on the shortest path to α by the function STOPPING-CONFIGURATION. It is possible that $q_{curr} = q_{new}$ when q_{curr} is already the nearest collision free configuration in the direction of α. In the other case, q_{new} and the path between q_{curr} and q_{new} are added to the search tree $\mathcal{G}$ (lines 6 and 7).

Algorithm 3 Rapidly Exploring Random Tree

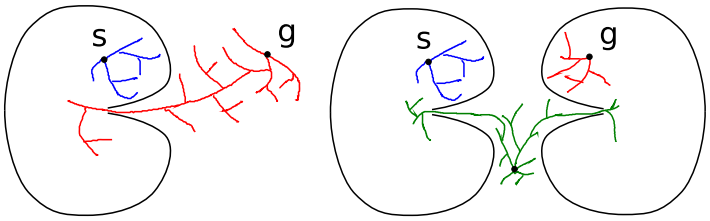
1: $\mathcal{G}$.unit(q_0)
2: **for** $i = 1$ to k **do**
3: $\quad q_{curr} \leftarrow$ NEAREST($\mathcal{G}$, α)
4: $\quad q_{new} \leftarrow$ STOPPING-CONFIGURATION(q_{curr}, α)
5: $\quad$ **if** $q_{curr} \neq q_{new}$ **then**
6: $\quad\quad \mathcal{G}$.add_vertex(q_{new})
7: $\quad\quad \mathcal{G}$.add_edge(q_{curr}, q_{new})
8: $\quad$ **end if**
9: **end for**

Figure 2.16 shows an example RRT taken from [45] where q_0 is in the centre of the graph. The earlier sampled configurations created some sort of main branches, since the initial node was the nearest neighbour. Later when more vertices and paths are present, smaller branches are added since the nearest neighbours are closer. Thus, it is easy to see that the tree densely fills the space, gradually increasing the resolution.

Figure 2.16: An example for a rapidly exploring random tree from [45]

Up until now, the presentation does not explain how to use RRT to solve a path planning query (s, g). As always in path planning, multiple possibilities exist. The easiest way is to use Algorithm 3 with s as initial node. Furthermore, the algorithm has to be modified to check if g can be added to the search tree $\mathcal{G}$ with a collision free path. If this is the case, a solution is found. The best way to do this is to use g instead of a random sample α when a biased coin flip shows head. [45] reports that the probability of 1/100 heads and 99/100 tails is a good value. If the probability for head is too high, the algorithm gets too greedy and the tree can't expand properly. On the other hand, if the probability is too low, the algorithm has no incentive to connect the tree to $\mathcal{G}$, resulting in unnecessary long run times. In this strategy, only one search tree is used, but situations exist, in which two or more search trees are beneficial. One such situation is depicted in Figure 2.17(a), showing a trap which is hard to leave for a tree rooted at s. A Tree starting at g might not have such difficulties.

Since the algorithm does not know apriori whether s or g is trapped, the idea is to use two search trees. $\mathcal{T}_a$ starts at s and $\mathcal{T}_b$ starts at g, which allows for much better performance [45]. One possibility to ensure that the trees meet while retaining their rapid exploring is to make the bidirectional search balanced,which ensures that both trees are the same size [45]. The pseudo code of this variant can be found in Algorithm 4, similar to the presentation in [83] and [45]. The way how $\mathcal{T}_a$ grows is exactly the same way as in Algorithm 3. When q_{new} gets added to $\mathcal{T}_a$, an attempt to extend $\mathcal{T}_b$ is made too. $\mathcal{T}_b$ extends the same way as $\mathcal{T}_a$, but instead of using the random sample α, q_{new} is used (line 9-14). Thus, $\mathcal{T}_b$ is effectively growing towards $\mathcal{T}_a$. When



(a) (b)

Figure 2.17: Various traps as depicted in [45]

both trees connect to the same vertex, a solution is found, since both trees merge to one (line 13). Line 14 is the reason why the algorithm is called balanced. The search gets focused to the smaller of the two trees by switching it to be the primary tree ($\mathcal{T}_a$). Therefore, $\mathcal{T}_a$ is not always the tree rooted at s. This is reasonable, since more energy should be invested into the tree which has problems at expanding, like in trap cases shown in Figure 2.17(a). Smaller, with respect to trees can reasonably be defined as number of vertices or the sum of all path lengths. When line 14 is changed to swap the trees every iteration, the algorithm uses unbalanced bidirectional search.

In situations when s as well as g are trapped, as shown in Figure 2.17(b), new trees in hard to reach parts of $\mathcal{C}$ can help to solve these difficulties. Using more trees introduces additional questions, as how to divide computation time between exploration and tree connection attempts. Furthermore, it is not clear which connection attempts should be performed. This considerations lead to the main idea of the planner of the next section. Every α gets added to the graph as a new component and then it is tried to connect it with neighbouring components. Here and subsequently, component is used in the graph theoretic sense. A subgraph H of graph G is called component, when for each vertex in H it holds, that it is connected to every other vertex in H via paths and it is not connected to every other vertex in G.

Probability road map

The *probabilistic road map (PRM)* approach is a probabilistically complete [45] multiple-query planner. Thus, significant time is invested to build a roadmap in the preprocessing phase to later answer multiple path planning queries fast and efficiently. The requirements for this roadmap are the same as for the combinatorial path planners presented in Section 2.3.1.

Various persons worked on PRM independently, so it is not possible to name an inventor unambiguously, but in most cases Kavraki and Latombe [36] or Overmars [53] are named.

The basic schema of PRM is shown in Algorithm 5 as presented in [45]. As in RRT, a configuration α from $\mathcal{C}$ gets randomly sampled and a collision checker tests whether it lies in $\mathcal{C}_{free}$ (line 2). Only if this check succeeds, α gets added to the roadmap $\mathcal{G}$ (line 3). Then it is attempted to connect α to nodes already present in $\mathcal{G}$ via collision free paths (lines 4-8), forming the edges of $\mathcal{G}$. To safe time, this connection attempt is only done with dedicated nodes out of $\mathcal{G}$. These nodes, called *neighbour set*, are computed by the function NEIGHBOURHOOD and

Algorithm 4 RRT BALANCED BIDIRECTIONAL

$\mathcal{T}_a$.unit(s)
$\mathcal{T}_b$.unit(g)
for $i = 1$ to k **do**
    $q_{curr} \leftarrow$ NEAREST($\mathcal{T}_a$, α)
    $q_{new} \leftarrow$ STOPPING-CONFIGURATION(q_{curr}, α)
    if $q_{curr} \neq q_{new}$ **then**
        $\mathcal{T}_a$.add_vertex(q_{new})
        $\mathcal{T}_a$.add_edge(q_{curr}, q_{new})
        $q'_{curr} \leftarrow$ NEAREST($\mathcal{T}_b$, q_{new})
        $q'_{new} \leftarrow$ STOPPING-CONFIGURATION(q'_{curr}, q_{new})
        if $q'_{curr} \neq q'_{new}$ **then**
            $\mathcal{T}_b$.add_vertex(q'_{new})
            $\mathcal{T}_b$.add_edge(q'_{curr}, q'_{new})
        end if
        if $q_{new} = q'_{new}$ **then**
            then return SOLUTION
        end if
    end if
    if $|\mathcal{T}_b| > |\mathcal{T}_a|$ **then**
        SWAP($\mathcal{T}_a$, $\mathcal{T}_b$)
    end if
end for
return FAILURE

we will see that there exist various possibilities to define this set. The function CONNECT, as usual a local planner using a collision checker, creates these collision free paths. These steps are repeated for a new sampled configuration, until a user defined stop-criterion is met. This can be a maximum number of repetitions or that predefined configurations are inter-connected.

Algorithm 5 Probability Road Map

repeat
    if $\alpha \in \mathcal{C}_{free}$ **then**
        $\mathcal{G}$.add_vertex(α)
        for all $q \in$ NEIGHBOURHOOD(α, $\mathcal{G}$) **do**
            if CONNECT(α, q) **then**
                $\mathcal{G}$.add_edge(α, q)
            end if
        end for
    end if
until some stop-criterion is met

Since all connection attempts for α can fail, $\mathcal{G}$ can consist of multiple components. This is

no problem, since when enough time is invested, enough configurations will be sampled to eventually connect all components (assuming no isolated holes). After the roadmap was constructed, it can be used for multiple path planning queries, as long as the environment does not change. To find a path from s to g, lines 2-9 in Algorithm 5 have to be executed twice, replacing α with s and g respectively. If both vertices are successfully connected to $\mathcal{G}$, the shortest path algorithms from Section 2.2 can be used to find a path. A path found in $\mathcal{G}$ directly corresponds to a path in $\mathcal{C}_{free}$. If it is not possible to connect s or g to $\mathcal{G}$, or the shortest path algorithm does not find a solution, PRM can not determine if no solution exists or just not enough time was invested to build $\mathcal{G}$. This stems from the fact that PRM is only probabilistically complete.

Various possibilities exist to define the *neighbour set*. One obvious possibility is to use the K nearest vertices to α. A typical value for K is 15 [45]. Another possibility is to use all vertices in $\mathcal{G}$ where the distance is at most d_{max}. The reason behind this is, that the probability to successfully connect two nodes decreases as the distance between the nodes increases [83]. To enhance this method, d_{max} could be reduced when the number of nodes in $\mathcal{G}$ increases. Additionally, an upper limit K can be introduced to combine this approach with the previous one. Finally, the connected components of $\mathcal{G}$ can be considered. Particularly it might not be useful to connect a node with nodes of the same component, since this would not increase the connectivity of the roadmap. Thus, the neighbour set could consist of the K nearest nodes of every connected component in $\mathcal{G}$. When $K = 1$, as recommended in [45] no cycle and thus no alternative routes will be present in $\mathcal{G}$.

Unfortunately, the standard PRM approach suffers from the *narrow passage problem*, which is the difficulty to connect two components of $\mathcal{G}$ through a narrow passage as shown in Figure 2.18 [83].

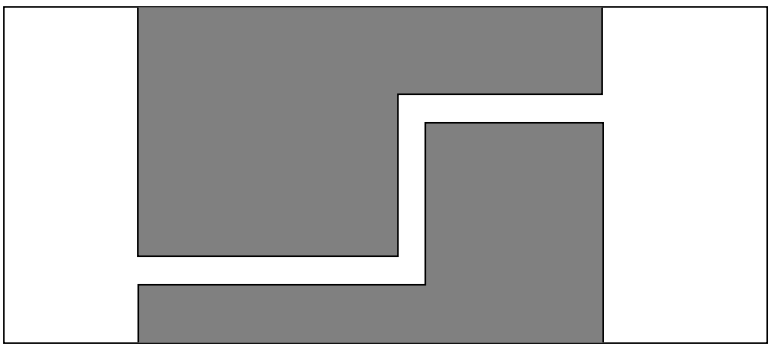


Figure 2.18: The narrow passage problem

When *uniform sampling* is used, many sampled configurations are located in open regions where no tight sampling is required to get a good coverage. On the other hand, only a small amount of samples will be located in the narrow passage, since the narrow passage is only a small fraction of $\mathcal{C}_{free}$. But especially there many samples are needed to get a good coverage, such that PRM is able to connect all components of $\mathcal{G}$. Thus, different sampling strategies were proposed to circumvent the narrow passage problem. Note, that the narrow passage problem only increases the time needed until a good connectivity of $\mathcal{G}$ is attained. Thus, the goal of every

new sampling strategy usually is to find ways to dramatically reduce the number or required samples [45].

Some strategies do not try to detect narrow passages directly, but generate more samples in the vicinity of obstacles, like *Obstacle-based PRM (OBPRM)* [1] [2] and *Gaussian PRM* [8]. In contrast strategies like *Bridge Test Sampling* [29] and *Watershed labelling* [82] find narrow passages to generate more samples in appropriate regions.

As expected, RRT is faster than PRM for single query problems, since no roadmap has to be build [17].

CHAPTER 3

Dynamic path planning

3.1 Introduction

Up to now, all presented path planning solutions assumed that obstacles are known beforehand and are stationary. Only the robot was allowed to move. A natural extension is to permit moving obstacles in the environment. Examples for moving obstacles are humans in a building, cars on a street or other robots. When moving obstacles exist, the robot operates in a *dynamic environment* with *dynamic obstacles*. As in the static case, *dynamic path planning* tries to find a collision free path between a given start and goal location, but dynamic obstacles also have to be taken into account. Thus, it is easy to see that the complexity of dynamic path planning is at least as high as in the static case.

How much the complexity of dynamic path planning increases depends on the available information about the dynamic obstacles. The easiest case is, when all dynamic obstacles and their behaviour is known beforehand. Planners which solve this case of dynamic path planning are presented in Section 3.2. The case, when the behaviour of dynamic obstacles is not known or can only be estimated for a short time, is significantly more compex [83]. Planning under this assumption is presented in Section 3.3.

One question that arises is, to which category obstacles which are stationary but which are not contained in the static map belong. An example could be a forgotten box in a building. On one hand, they do not move, hence, their behaviour is static. On the other hand, they are not marked in the map. Thus, a path produced from a planner presented in Chapter 2 could be unsafe. At the time when the map was created, the obstacle was not present and now it is. Thus, it makes sense to classify it as a dynamic obstacle, since it has some (very low) dynamic behaviour. So, static obstacles are all obstacles contained in the static map, dynamic obstacles are all other present obstacles, regardless of their current behaviour.

3.1.1 Configuration-time space

Define $T \subset \mathbb{R}$ as the *time-interval*, which can either be *bounded* or *unbounded* [45]. In the *bounded* case $T = [0, t_f]$, otherwise $T = [0, \infty]$, where 0 is the initial time and t_f is the final time. With the help of the time-interval, the configuration space $\mathcal{C}$ can be extended to the *configuration-time space* donated by $\mathcal{CT}$. The configuration-time space is formed as $\mathcal{C} \times \mathcal{T}$, consisting of pairs (q, t), where q is a configuration out of $\mathcal{C}$ describing a robot location, and scalar t out of $\mathcal{T}$ is a time-point [83]. Like the configuration space $\mathcal{C}$, the configuration-time space $\mathcal{CT}$ can be divided into $\mathcal{CT}_{free}$ and $\mathcal{CT}_{coll}$. When at time-point t the robot positioned at configuration q collides with an obstacle (static or dynamic), (q, t) is an element of $\mathcal{CT}_{coll}$. Otherwise it is an element of $\mathcal{CT}_{free}$. A configuration q, where the robot would collide with a static obstacle will be in $\mathcal{CT}_{coll}$ regardless of the time-point, since static obstacles do not change their position over time. Figure 3.1(a) [83] shows a configuration space with a static and a dynamic obstacle and Figure 3.1(b) [83] depicts the corresponding configuration-time space.

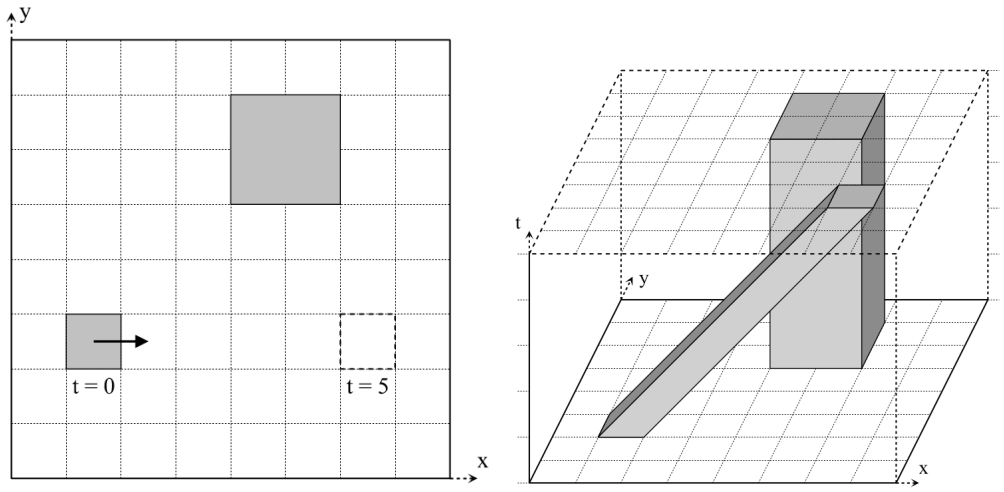


Figure 3.1: A configuration space with a static and a dynamic obstacle (a) [83]. The corresponding configuration-time space (b) [83]

As in static path planning, dynamic path planning using the configuration-time space has to be defined properly. In [83] Van den Berg defines: *"A path through a dynamic environment (in literature often called a trajectory) is defined as a continuous function $\pi : \mathcal{T} \rightarrow \mathcal{C}$, parametrized by time. The path planning problem in dynamic environments is to find a (collision-)free path between a given start configuration s and goal configuration g. Formulated in terms of the configuration-time space $\mathcal{CT}$, that is finding a path π such that $\pi(0) = s$ and $\pi(T) = g$, and $\forall t \in [0, T]$: $(\pi(t), t) \in \mathcal{CT}_{free}$"*.

The time-interval only adds one additional dimension to the configuration space and thus, planning directly in $\mathcal{CT}$ can be done with any of the static path planning methods from Chapter 2. However, one critical difference has to be observed, *time marches forward*. For example consider

a path which first reaches $(q_1, 5)$ and later $(q_2, 3)$, which is only possible with time travel [45]. Thus, paths should be monotonically increasing in the time component to prevent this unrealistic behaviour. The previous definition of a path enforces this constraint [83].

One last consideration concerns the goal. Up until now, a single configuration $g \in \mathcal{C}$ formed the goal. Considering the time-interval T the question arises, when g should be reached. Thus, when it is planned in $\mathcal{CT}$, the single goal is replaced by a *goal region*. This region can be expressed as $\{(g, t) \in \mathcal{CT}_{free} \mid t \in T\}$ [45]. Similar set definitions can be made when the goal should be reached before or after a specific time-point. The goal region forms a line (segment) parallel to the time axis in $\mathcal{CT}$. When g should be reached at a specific time-point t, the goal region only consists of state (g, t).

3.1.2 Online planning vs. offline planning

In dynamic path planning a fundamental problem is, that path planners themselves need time Δt to compute a solution, in which dynamic obstacles move, and thus, the environment changes. Hence, it is impossible to compute a path at $t = 0$, which should start at $t = 0$. The path would immediately be outdated. To overcome this problem, a computed path starts at $t_r + \Delta t$ when the plan was requested at time-point t_r [83].

The difficulty is to find a good value for Δt. If it is too small, the planner has too little time to find a path. On the other hand, if the value is to high, unnecessary and unacceptable delays for some applications arise [83]. Another problem is, that for sampling based planners (see Section 2.3.2) it is hard or impossible to predict the needed planning time beforehand [83]. To overcome this problem, so called *online-planners* were developed, which use approaches like *partial planning* [54] or *anytime planning* [84].

In *partial planning* the planner tries to find a plan in a given time window Δt. When the time is over, the planner returns the path that is considered to be the best. When too little time was available to find a complete path from start to goal, the closest path to the goal gets returned. While the robot is following the found path, time gets invested to extend or finalise it.

Anytime planners also use a predefined time Δt to find a path. The difference to partial planners is, that anytime planners do in general find a complete path from start to goal [83]. They use the remaining time to refine the initial potentially low quality plan until the time is over.

3.2 Planning in known dynamic environments

Path planning in *known dynamic environments*, i.e., when all obstacles and their corresponding behaviour is known, is the easiest form of dynamic path planning. When the path planners from Chapter 2 are adopted to yield monotonically increasing paths in the time component, they can be used to solve the path planning problem in the configuration-time space $\mathcal{CT}$.

3.2.1 Adapting combinatorial path planners

Combinatorial path planners are adapted by using a *directed roadmap*, where each edge is directed. Thus, for every two states (q_1, t_1) and (q_2, t_2), where $t_1 \neq t_2$ exactly one valid direction

exists for making a potential edge [45]. When $t_1 = t_2$ no edge between these two states can exist, because this would require the robot to teleport from q_1 to q_2 or vice versa. When all obstacles are polygonal and have *piecewise linear movement*, the resulting $\mathcal{CT}_{coll}$ is polyhedral [45]. An obstacle moves *linearly*, when the movement of an arbitrary point (x, y) on this obstacle can be described as $(x + c_1 \cdot t, y + c_2 \cdot t)$ for some constants $c_1, c_2 \in \mathbb{R}$. A *piece-wise linear* moving obstacle is allowed to change to other linear movements at finitely many points in time.

In a configuration-time space with polyhedral $\mathcal{CT}_{coll}$, the vertical cell decomposition can be applied to create a directed roadmap. First, the line-sweep principle can be used along the time-axis, stopping at the time-points where obstacles change their linear movement. The resulting cells can be further decomposed recursively. This procedure can be generalised to higher dimensional configuration-time spaces [45].

3.2.2 Adapting sampling based path planners

In single-query path planners, the local planning method (LPM) has to be adapted to create monotonically increasing paths in the time component. Usually a metric ρ for the configuration space $\mathcal{C}$ is used to reveal the distance of two configurations in LPM. Out of ρ a pseudometric $\hat{\rho}$ for the configuration-time space $\mathcal{CT}$ can be constructed, which penalises non-monotonically increasing path creation attempts. Let $\hat{\rho}$ for a pair of states $x = (q, t)$ and $x' = (q', t')$ be [45]

$$\hat{\rho}(x, x') = \begin{cases} 0 & \text{if } q = q' \\ \infty & \text{if } q \neq q' \text{ and } t' \leq t \\ \rho(x, x') & \text{otherwise.} \end{cases}$$

Hence, when LPM tries to connect two states via a path without forward time progress, the distance between these two points will be ∞ and the attempt will be aborted. With the help of $\hat{\rho}$, RRT guarantees to create paths which move forward in time, with a single search tree. More complicated is the case, when a bidirectional search should be utilised, because in general the goal is a region and not a single point. One possibility to solve the dilemma is root the goal tree at an entire time-invariant segment [45].

Multiple-query planners can easily be adapted by using a *directed roadmap* as define before, which already takes forward time progress into account.

3.2.3 Velocity-Tuning

As an alternative for using the methods from Chapter 2 directly, [35] splitted the problem into two parts. First, all dynamic obstacles are ignored and a path π gets computed with the help of methods from Chapter 2 for the static case. Then, the velocities of the robot get tuned to avoid dynamic obstacles crossing the precomputed path. This is done by constructing a new two dimensional space $\mathcal{X}$, where one dimension is the time and the second dimension is the position of an object on its trajectory. When, at a given time-point t, the robot at position s on its path collides with a dynamic obstacle, the state (t, s) is a colliding state in $\mathcal{X}$ (see Figure 3.2(b)).

$\mathcal{X}$ can be used to search for a safe path $\hat{\pi}$. As before, a valid path $\hat{\pi}$ in $\mathcal{X}$ has to be time-monotonic, but it can be non-monotonic in the position on the path axis. Thus, the robot can stop, accelerate, decelerate or even drive backwards on the original path π to avoid dynamic obstacles.

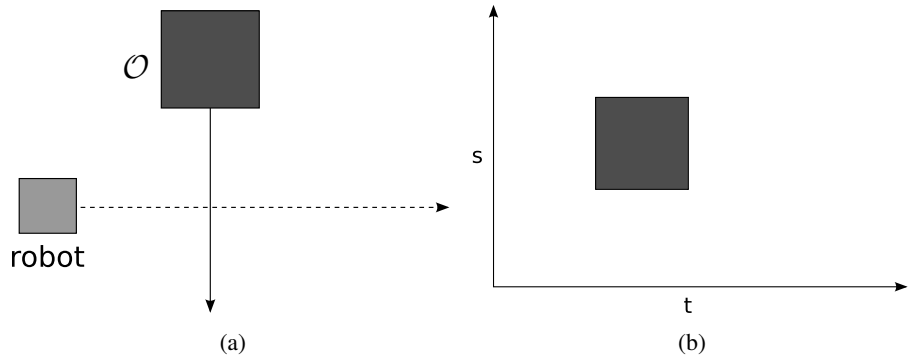


Figure 3.2: An example of velocity tuning. A moving obstacle crossing the precomputed path of the robot (a). The corresponding space $\mathcal{X}$ (b)

The advantage of the *velocity-tuning* method is, that $\mathcal{X}$ is always two dimensional [45]. This allows to use simple adapted combinatorial- or sampling based path planners to search for a path $\hat{\pi}$ in $\mathcal{X}$. Unfortunately, this method is not complete [83].

3.3 Planning in partially known or unknown dynamic environments

The previous section assumed that every obstacle and their behaviour is known in advance. In this chapter these assumptions are weakened, such that not all obstacles are known beforehand and the trajectory of obstacles can only be predicted for a *short time* or *not at all*. It is obvious that with such little knowledge it is impossible to create a path via an offline-planner once and follow this plan until the goal is reached. Thus, different approaches have to be taken.

When the future trajectory of obstacles should be predicted, a robot needs some sensors to observe the obstacles. When new obstacles appear or existing ones change their predicted behaviour, the plan can be updated. Thus, the paradigm changes from plan once to interleaved planning and sensing. How to predict future trajectories of obstacles is out of scope of this thesis. More information about this topic can be found in [7] and [85].

3.3.1 Planning in configuration space

The easiest way to solve the path planning problem in partially known or unknown dynamic environments is to avoid predicting the obstacle trajectories but use the methods of Chapter 2 repetitively. To be more precise, all obstacles marked in the map and observed by sensors are treated as static. Thus, the well known path planning algorithms can be used. Hence, the path planning is done in the configuration space $\mathcal{C}$.

When the robot follows the computed path, it periodically checks if the environment has changed with its sensors. For example, the assumed static obstacles changed their position or new obstacles are visible. Depending on the chosen strategy, upon changes, a new path is always

computed or only when the original path is blocked. The disadvantage of this approach is, that the plan always gets computed from scratch and the multiple-query planners, like PRM, are absolut useless.

Various extensions for PRM were proposed to efficiently repair the roadmap when the environment changes. In [32] the first step is to use the PRM method to create a roadmap while ignoring all dynamic obstacles. In the query phase a path is searched in the roadmap, while checking the edges for collision with observed dynamic obstacles. This is done in an efficient way, such that collision checks are postponed as long as possible to safe time. When no solution can be found, because some dynamic obstacles block an edge in the roadmap, a local RRT is applied to reconnect the broken edge. During path following it is periodically checked, if a collision with a dynamic obstacle will occur. When this might be the case, RRT is used again to create a local path which evades the obstacle while retaining portions of the old plan. If this is unsuccessful, a completely new plan is searched in the roadmap.

A similar approach to adapt PRM is presented in [34]. As in the previous approach, the first step is to create a roadmap with the help of PRM, while ignoring all dynamic obstacles. The main difference is, that in the next step the world gets discretised by a grid. In each grid cell the vertices and edges from the roadmap which are influenced by the cell are stored. Thus, when a grid cell changes its state, because a dynamic obstacle occupies or leaves a cell during the execution, the corresponding vertices and edges in the roadmap can easily be updated accordingly.

As we have seen, most path planners use the graph search algorithms from Section 2.2. Neither of them is capable of updating a path when parts of the graph change. Thus, they have to search for a path in the graph from scratch and can not reuse parts of the old path. To overcome this problem, in [76] Stentz presented the Dynamic A* algorithm (D*). D* uses some ideas from A*, but it allows that the edge weight changes during execution. For example, when new obstacles are sensed. Thus, D* is suitable for dynamic environments. The main idea is, that the search starts at the goal location and propagates to the start. When edge costs change, this will be in the vicinity of the robot. This is the case, since the sensors carried by the robot only have a limited range and field of view. Thus, often paths have to be repaired only in a small area and the majority of the path can be reused. With the help of the extension *Focused D** [77], a speedup in the order of up to two magnitudes can be achieved compared to repeated A* [40].

3.3.2 Planning in configuration-time space

The planners presented previously do not use methods to predict the future trajectory of obstacles. This simplifies the implementation, but the neglection of information like velocity or movement direction of an obstacle can result in highly suboptimal results [83]. [83] gives an example where such methods would have problems: *"For instance, imagine a robot trying to cross a road on which cars are driving. If the robot was to take a snapshot of the environment with its sensors and assume fixed positions for each obstacle, then it would be in serious risk of getting runover if it attempted to cross the road. In order to successfully accomplish this task, the cars really need to be modelled as moving obstacles so that it can be anticipated where they will be at future times."*

To do exactly this, planning has to happen in the configuration-time space $\mathcal{CT}$. Algorithms

using this $\mathcal{CT}$ space can incorporate the predicted trajectories in their search for a valid path. For example, all strategies for known dynamic environments (see Section 3.2) can be used. Unfortunately, they can only be used in low dimensional configuration-time spaces, since they require significant computation time [83].

A requirement, which is true for all algorithms used for planning in $\mathcal{CT}$ with an unknown dynamic environment is, that the runtime of the algorithm is shorter than the validity of the predicted obstacle trajectories. If this demand can not be fulfilled, because either the planner took too long or an obstacle unexpectedly changed its movement, the resulting path is immediately outdated. Thus, the planner has to start a new search with the changed environment. In case this happens too frequently, the robot will not make any, or only slow progress towards the goal.

To overcome this problem, an anytime planner for unknown dynamic environments was developed in [83]. The advantage of this approach is, that the available time for the anytime planner can be adjusted to the shortest valid trajectory prediction, and thus, that at every time a solution can be extracted [83]. During the execution of the path, the future parts get refined. Thus, the algorithm executes planning, execution, and observation in an interleaved manner.

In the initial phase, a roadmap is created while ignoring all dynamic obstacles with the help of the PRM. This discretises $\mathcal{C}$ to a graph with many cycles, and thus, many alternatives to reach a goal [83]. In the planning phase, a valid path from start to goal is searched with respect to the currently estimated obstacle trajectories. This is possible in the configuration-time space $\mathcal{CT}$. Thus, a time axis is added to the configuration space. Furthermore, the new axis gets discretised with step size Δt. Each state $\langle v, t\rangle$ in $\mathcal{CT}$ consists of one vertex v of the graph created by the PRM and one time-point t. The final search graph $\mathcal{G}$ is constructed by creating a vertex for every state $\langle v, t\rangle$. Edges are added for states $\langle v, t\rangle$ to states $\langle v, t + \Delta t\rangle$, allowing the robot to wait at its current location. Furthermore, edges are created for states $\langle v, t\rangle$ to states $\langle v', t + c_t(v, v')\rangle$, when v' is the successor of v in the PRM graph and where $c_t(v, v')$ is the time needed to traverse from v to v'. Thus, a discretisation of $\mathcal{CT}$ is achieved.

The actual search in $\mathcal{G}$ is done by a slightly modified *Anytime D* (AD*) algorithm* [46], an enhancement of the D* algorithm to be anytime capable. Like in the original D* algorithm, the search is done backwards, i.e., from the goal to the start. Similarly as in Section 3.1.1, in general the goal is a set of states in the configuration-time space $\mathcal{CT}$. Since it is not known in advance when the goal will be reached, the search is seeded with multiple goal states

$$\begin{aligned} GOALS = [&\langle v_{goal}, h_t(u_{start}, v_{goal})\rangle, \\ &= \langle v_{goal}, h_t(u_{start}, v_{goal}) + \Delta t\rangle, \\ &\vdots \\ &= \langle v_{goal},\ \text{max-arrival-time}\rangle] \end{aligned}$$

where v_{goal} is the goal vertex in the PRM graph and max-arrival-time is the maximum available time for the robot to reach the goal [83]. h_t is one of two used heuristics, to further increase the speed of AD*. $h_t(v_{start}, v)$ is set to the *minimum possible time* to reach vertex v from v_{start} and $h_c(v_{start}, v)$ is set to the *minimum possible cost* for the same vertices. Both heuristics can be computed in advance by any graph search algorithm and are used to increase the performance

of AD*. h_t is used to prune the search graph $\mathcal{G}$ and h_c controls the direction of the search of AD*.

During the execution of the path, the robot observes the environment while also continually improving the solution. When some changes occur, the previous solution can be repaired when necessary, as in ordinary D*. Additionally, this is done in an anytime fashion. First, the heuristics h_c and h_t have to be recomputed, which can be done quickly, since this has only to be done in the PRM created graph [83]. Then, all states which are affected by the changed environment are updated, which possibly triggers a replanning attempt, for example when the previously computed plan is blocked by an observed obstacle.

3.3.3 Obstacle avoider

A completely different approach to solve planning in partially known or unknown dynamic environments are so called *obstacle avoider*. In general, they do not rely on a map describing the environment, but are a *reactive sensor-based approach*. Furthermore, they do not try to predict the movement of obstacles. A new command is computed solely on the current sensor input and possibly the state of the robot [75]. This is computationally very efficient and thus, commands can be generated with a high frequency [75], which is ideal in highly dynamic environments. This justifies the decision to not predict obstacle trajectories.

Obviously, obstacle avoiders can not produce optimal solutions and they can easily be trapped in local minima like in U-shaped obstacles [75] [21]. Thus, they are also called *local approaches* since they only use a small part of the world (what the robot's sensors perceive) to generate movement commands. All previous methods where *global approaches*, since they use information of the whole world to find a path.

To solve this problem, obstacle avoiders are combined with global approaches. For example, in [80] and [12] a local planner gets intermediate points of a path planned by a global planner as goal. Thus, the local planner does not get stuck in local minima, but the ability to avoid obstacles is retained.

Obstacle avoider can roughly be separated into two categories. *Directional approaches* operate in Cartesian spaces or configuration spaces to create the direction in which the robot should move [39]. An example are *potential field* methods [37] [30] which use a potential function. Similarly to the randomised potential fields path planner from Section 2.3.2, a potential function consists of an *attraction* term which draws the robot towards the goal and a *repulsion* term which repeals the robot from obstacles. Another example for a directional approach is the *vector field histogram* method [9], which transforms multiple sensor readings into a histogram describing the unoccupied space. This histogram is then used to compute the motion commands.

The disadvantage of directional approaches is, that the dynamic constraints (maximum velocity/acceleration) of the robot are not taken into account [39]. Often these methods consist of two stages. First, the direction in which the robot should travel is picked, then the motion command to travel in the chosen direction is generated. This is problematic for robots with limited acceleration, since they have to take the impulse of the robot into account. [21]

Figure 3.3 depicts an example similar to [21] to show the problem. Assume the robot is driving in the direction of the arrow to reach the goal with high speed. When the robot detects the gap in the wall, the desired direction to move is obviously down (towards the goal) which

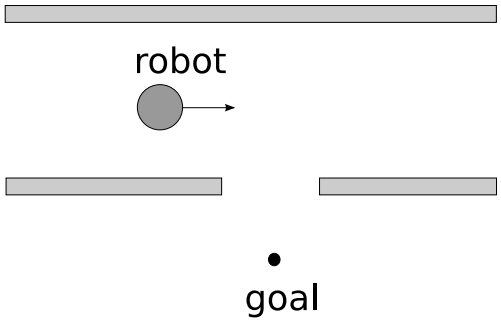


Figure 3.3: A situation where directional approaches have problems [21]

is a sharp right turn. When the robot is not able to decelerate fast enough, it will collide with the right wall segment. Thus, the obstacle avoider has to consider the robot dynamic limits to anticipate and prevent such situations

Velocity space approaches are capable to include the robot dynamics when the stearing command is chosen. As the name implies, all obstacle avoiders in this category search in the velocity space of a robot for a suitable trajectory. The velocity space is the set of velocities the robot can control [73]. For example, the velocity space for rigid robots with *synchro-* or *differential-drive* consists of two orthogonal dimensions. One dimension for the translation velocity (driving forward/backward in the robot frame) and one for the rotational velocity. Thus, each point in this velocity space corresponds to a trajectory with constant velocity in Cartesian space. The velocity space is limited by the robots maximum translational and rotational velocity. Furthermore, all points which would lead to a collision with an obstacle are forbidden. These forbidden points can efficiently be computed by collision avoiders.

One obstacle avoider which belongs to the velocity space category is the *Dynamic Window Approach (DWA)* [21]. To generate a trajectory which does not collide with obstacles for n time intervals, the DWA has to determine a velocity vector for each time interval. The problem is, that the search space for such a trajectory is exponential in n [21]. DWA uses three steps to reduce the search space, resulting in decreased runtime:

1. Only for the first time interval of a trajectory a velocity vector is searched. Constant velocity is assumed for the remaining ones. This is feasible, since after every time interval the search is repeated for the new "first" interval. This restriction reduces the search space to two dimensions.

2. Only admissible velocity vectors are permitted. A velocity vector is admissible when the robot can stop on the resulting trajectory before reaching an obstacle. This is done by limiting the maximum velocity of a velocity vector. The limit gets calculated by combining the distance to the closest known obstacle on the corresponding trajectory with the acceleration for breakage.

3. Only velocity vectors which are reachable in the next time interval are considered. Assume that at time interval t, the robot has a translational velocity v_a and a rotational velocity ω_a.

The dynamic window with accelerations $\dot{v}$ and $\dot{\omega}$ is:

$$\{(v, \omega) \mid v \in [v_a - \dot{v} \cdot t, v_a + \dot{v} \cdot t] \wedge \omega \in [\omega_a - \dot{\omega} \cdot t, \omega_a + \dot{\omega} \cdot t]\}$$

The dynamic window is centred at the current velocity and only velocity vectors inside this window are used for the search.

After these steps, the reduced down search space gets discretised and the velocity vectors out of the resulting finite set get evaluated by a scoring function. This function consists of three evaluation terms:

1. **Target Heading:** Measures the alignment of the robot to the goal. When the robot heads directly to the goal, this term is maximal.
2. **Clearance:** Velocity vectors resulting in trajectories which are farther away from obstacles are preferred.
3. **Velocity:** Velocity vectors with faster forward movement are preferred.

Then, the best valued velocity vector gets applied to move the robot for a short time and the whole procedure is repeated until the goal is reached.

In [10] the DWA scoring function gets modified to favour velocity vectors which stay in proximity of a previously planned global plan, to avoid local minima problem.

CHAPTER 4

Path planning with probabilistic maps and sonar sensors

All current strategies to solve path planning in partially or fully unknown dynamic environments (see Section 3.3) suffer from the same fundamental problem. Unknown obstacles can lead to highly undesirable behaviour. For example, imagine a crowded place which a robot tries to cross. In such a situation the robot frequently has to stop, restart and change directions, which is highly subpar. Another scenario could be a narrow corridor which is completely blocked by an obstacle with a very low dynamic. Thus, the robot has to backtrack to reach the goal via an alternative path. In the worst-case, when the robot can not drive backwards or not enough space is available for a rotation, the robot is stuck.

These problems stem from the fact, that not enough information about dynamic obstacles is available. It is simply not possible to add humans in a crowded place as obstacles in a conventional map. To circumvent this problems, *probabilistic maps* from [11] can be used.

With probabilistic maps it is possible to enhance path planning in partially or fully unknown dynamic environments. For example, reconsider the example where the robot has to cross a crowded place to reach its goal as fast as possible. With the additional information provided by a probabilistic map, the crowded place might be avoided by a longer but less cumbersome way. The distance covered by the alternative path is longer, but in average the goal is reached faster since no expensive evasion motions have to be executed.

In this chapter, a path planning strategy is presented, which uses probabilistic maps to enhance path planning in exactly this way. Furthermore, algorithms for using sonar sensors for environment observation in dynamic environments are shown.

4.1 Path planning with probabilistic maps

4.1.1 General idea

Carefully examining the information about *probabilistic maps* from [11] reveals how *probabilistic maps* can be used to enhance path planning. A *probabilistic map* is a pixel map, where the world is represented by a grid of pixels or cells. The special feature is, that for every cell the occupancy probability and the time until change is stored. The occupancy probability is the probability that an obstacle is present at the pixel's location. The time until change is the average time until the pixel changes it states from empty to occupied or vice-versa.

A cell contains a static obstacle like a wall, when the occupancy probability is high and the time until change is high. A busy road, for instance, is characterized by a high occupancy probability and a low time until change. The same is true for places crowded with moving people. An infrequently used road in bad shape, would have a low occupancy probability and a high time until change, since vehicles can only drive with reduced speed. On the other hand, rarely used roads are identified with a low occupancy probability and a low time until change. Note, that the roads are used as analogies to better see the impact of occupancy probability and time until change. The concrete meaning depends on the environment, which the map is representing. The words "high" and "low" have to be quantified by the probabilistic map path planning strategy and may depend on the context.

These four corner cases directly lead to the principle idea for the new probabilistic map path planning strategy. The robot should never travel through cells which have high occupancy probability and high time until change, since this is the worst case, a static obstacle. Furthermore, a robot should avoid cells with a low occupancy probability and a high time until change, since they represent roads where the robot can only drive very slow. At last, cells which behave like crowded places (high occupancy probability/low time until change) should be avoided by a robot, since it will have to stop, start and change direction frequently in such cells.

It is important to notice, that these three regions are not equally bad for a robot. Driving slow or starting and stopping frequently is certainly better than crashing into a static obstacle.

The last corner case (low occupancy probability and a low time until change) are the best possible cells for a robot. Since they are infrequently used roads, where the robot can drive with a high speed and obstacles will appear only with a low probability. Table 4.1 summarise theses results, where + expresses preferable-, - expresses forbidden- and ~ expresses suboptimal cells.

	high time until change	low time until change
high occupancy probability	-	~
low occupancy probability	~	+

Table 4.1: Assessment of the corner cases for the probabilistic map path planning strategy

The idea to avoid static obstacles is common to all path planning strategies. Novel is the idea to avoid crowded cells (high occupancy probability and low time until change) and roads in bad shape (low occupancy probability and high time until change).

4.1.2 N-ary configuration space

Additionally to static obstacles, the probabilistic map path planning strategy shall avoid crowded places and roads in bad shape before the robot "drives into them". Combinatorial- and Sampling based path planners, presented in Chapter 2 are capable of doing exactly this with static obstacles. Thus, it is obvious to adapt one of these path planners to be capable of avoiding crowded places and roads in bad shape.

For the selection of the right path planning strategy, it must not be forgotten, that a difference between static obstacles and the two other areas exist. Driving through a crowded place is undesirable, but not forbidden. Thus, when the only way to the goal is through a crowded place, the robot is allowed to pass through it. The same is true for roads in bad shape. On the other hand, the robot must never drive to an area in which a static obstacle is present, since this would lead to a crash. In other words, static obstacles reduce the allowed configurations for a robot, while the other two areas influences the cost of a path.

As consequences of this observation, the configuration space definition has to be extended. It is not enough to divide the configuration space into two sets $\mathcal{C}_{coll}$ and $\mathcal{C}_{free}$, since configurations which correspond to cells which have a crowded place characteristic, can not be member of either set. As noticed, they are no obstacles, thus they can not be in $\mathcal{C}_{coll}$. On the other hand, they can not be in $\mathcal{C}_{free}$, since all configurations in there are equal. However, there are configurations which are better for a robot than configurations which correspond to crowded cells.

To solve this problem, the *n-ary configuration space* for $n > 1$ is defined as follows:

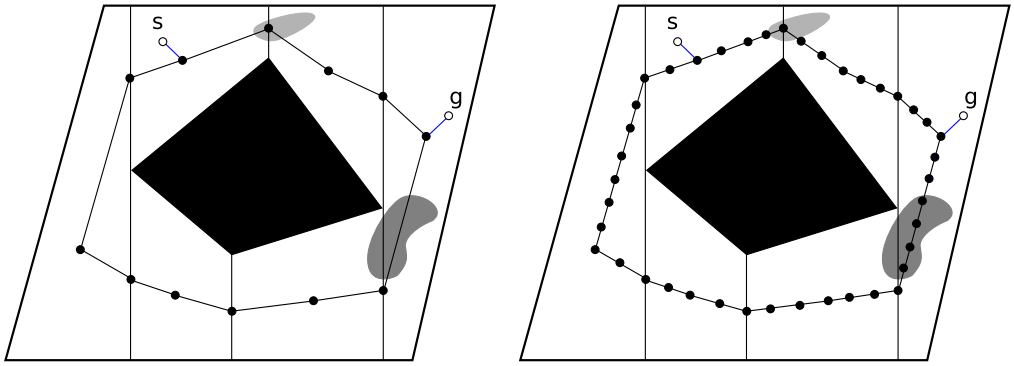
$$\begin{aligned} \bigcup_{i=1}^{n} \mathcal{C}_i &= \mathcal{C} \\ \bigcap_{i=1}^{n} \mathcal{C}_i &= \emptyset \\ \mathcal{C}_1 &= \mathcal{C}_{free} \\ \mathcal{C}_n &= \mathcal{C}_{coll} \end{aligned} \tag{4.1}$$

In this definition, the configurations space is divided into more than two sets, which is a generalisation of the configuration space definition presented in Section 2.1.1. The greater n, the more sets are available, and the finer the configurations can be distributed between the sets.

The set membership of each configuration directly translates to how attractive this configuration is for a robot. The lower the number of the set, the more attractive is the configuration for the robot. It is the task of the probabilistic map path planning strategy to distribute the configurations to the right sets. Note, that for $n = 2$ the well known "binary" configuration space is attained.

4.1.3 Choosing existing path planning strategies

To fully utilize the n-ary configuration space definition, $\mathcal{C}$ (for a sampling based path planner) or $\mathcal{C} \setminus \mathcal{C}_{coll}$ (for a combinatorial path planner) has to be discretised by a grid. To see this, examine Figure 4.1(a), where $\mathcal{C} \setminus \mathcal{C}_{coll}$ is discretised by a vertical cell decomposition (see Section 2.3.1).



(a) (b)

Figure 4.1: Vertical cell decomposition with dynamic areas (a) and more sample points to avoid unlucky sample selection (b)

In the configurations space, one static obstacle (black) and two dynamic regions (grey) are visible. The darker the grey of a dynamic region is, the less attractive it is for the path planner. One sample configuration on the shorter path between start and goal, lies in a dynamic region. When only the cost of the sample configurations are used to determine the best path from start to goal, the longer path on the bottom of the configuration space will be chosen. Unfortunately, this path is not only longer, but it also leads through the more unattractive dynamic region. To circumvent this, more points could be added on the paths created by the roadmap, shown in Figure 4.1(b). This would guarantee, that in this case the shorter path is chosen. In case additional dynamic regions are added to the top path, the bottom path will eventually get better, and will be chosen.

In general, this is not the desired result. A path planner should detect this situation and should avoid the dynamic region at the top, like depicted in Figure 4.2 with the dashed path.

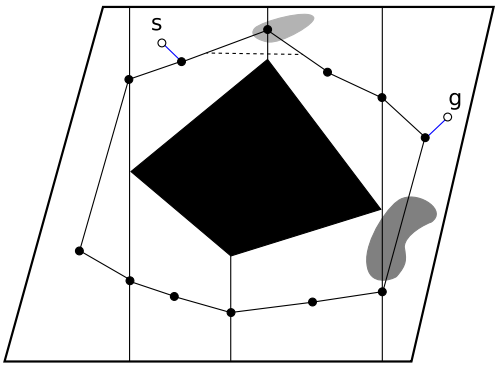


Figure 4.2: Illustrating wow the dynamic area should be avoided with the help of the dashed path

The problem of all path planners without a grid discretisation is, that they were not developed

for optimal path planning. Their sole purpose is, to find a valid path between start and goal in high dimensional configuration spaces. Thus, to realise the idea of avoiding dynamic regions, only the *combinatorial path planner with grid cell decomposition* (see Section 2.3.1) and the *grid sampling based path planner* (see Section 2.3.2) are suitable.

4.1.4 The probabilistic map path planning strategy

The task of the probabilistic map path planning strategy is, to determine the cost of every cell in the grid. The higher the cell cost, the less attractive the cell is for the planner. Every configuration in a cell has the same cost as the cell it belongs to. Thus, this is the same as determining the set membership for every configuration in a n-ary configuration space.

Whether this grid is then used with the *combinatorial path planner with grid cell decomposition* or the *grid sampling based path planner*, is not important. Both planners use a shortest path algorithm from Section 2.2 to find the cheapest path in this grid. They only differ in the way, how they detect cells with obstacles.

For now, assume that the probabilistic map has the same dimension (number of axes) as the n-ary configuration space. Hence, the cell cost of a grid cell can be determined by the *occupancy probability* and the *time until change* of its corresponding cell in the probabilistic map. Thus, the *occupancy probability* and the *time until change* have to be combined in a meaningful way.

The probabilistic map path planning strategy has to quantify the words high and low for the four corner cases presented before (see Table 4.1). Since these values depend on the application and the environment, two parameters have to be supplied by the user. *Forbidden_probability* specifies the point from which the occupancy probability is too high. For map cells with an occupancy probability equal or higher than *forbidden_probability*, the corresponding grid cells get a cost value of n (obstacle) assigned. Since the grid cell is marked as containing an obstacle, the robot will not drive through such cells. Similar, *forbidden_time_until_change* states, when a time until change is too low. Additionally, *forbidden_probability* and *forbidden_time_until_change* serve as weight factor of how much the occupancy probability and time until change influence the grid cell cost.

Special care has to be taken regarding the range of values of both map cell values. On one hand, the occupancy probability clearly has a range from 0 to 100. On the other hand, the time until change value can be between 0 and *dynamic_time_max*. *Dynamic_time_max* is contained in the probabilistic map. As consequence, forbidden_probability would have to be set in line with *dynamic_time_max*. For example, a *forbidden_time_until_change* of 5 in a map where *dynamic_time_max* is 100 is much more restrictive, than in a map where *dynamic_time_max* is 10. To circumvent this, the time until change is mapped to the same range as the occupancy probability (0 to 100). To map a time until change value to the occupancy probability range, it has to be multiplied with the constant $100/dynamic\_time\_max$. In principle, any other range could have been used. For the sake of the easier comparability to forbidden_probability, the range of occupancy probability is used. After this, the occupancy probability and the time until change have the same range and can be combined.

Equation 4.2 shows how to compute the rating of a map cell, by combining occupancy probability and time until change, following the previous ideas. The variable n is the maximal

cost value a cell can have and is identical to the n sets of the n-ary configuration space.

$$\begin{aligned} grid\_cell\_cost = & min\{n, max\{n/forbidden\_probability \cdot occupancy\_probability + \\ & n/forbidden\_time\_until\_change \cdot 100/dynamic\_time\_max \cdot \\ & time\_until\_change, min\_cost\}\} \end{aligned} \tag{4.2}$$

With the help of min_cost, each grid cell can have assigned a minimal cost. This allows the shortest path algorithms from Section 2.2 to include the path length into their search.

Figure 4.3 shows a plot of resulting cost values using Equation 4.2, with *forbidden_probability* set to 50, *forbidden_time_until_change* set to 75, n set to 200 and optional_term equals 0.

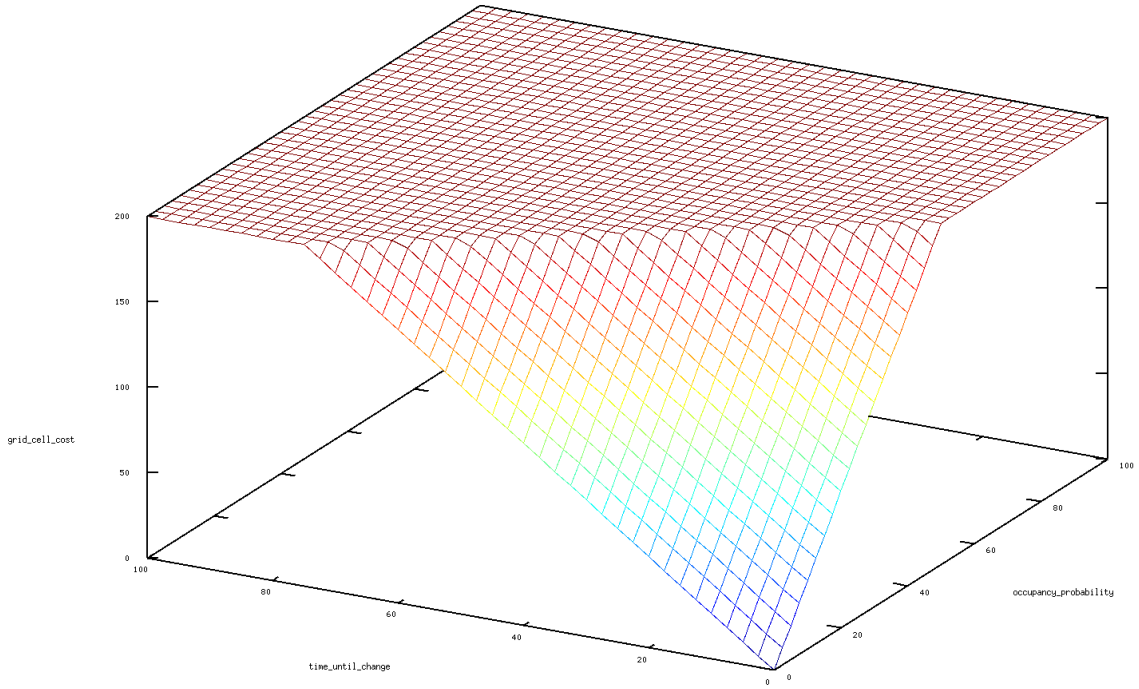


Figure 4.3: Plot of the resulting cost value using Equation 4.2, when *forbidden_probability* is set to 50, *forbidden_time_until_change* is set to 75, n is set to 200 and optional_term equals 0

In case the n-ary configuration space does not have the same dimension as the probabilistic map, one map cell determines the cell cost of multiple grid cells. For example assume the probabilistic map has only two dimensions (x and y), but the rigid robot can translate and rotate, forming a three-dimensional n-ary configuration space (x, y and θ). Thus, a map cell with coordinates (x, y) determines the cost of all grid cells which have x and y as part of their coordinate, independently of the rotation. This can be generalized for higher dimensional maps and configuration spaces.

4.1.5 Example

The effect of the probabilistic map path planning strategy is demonstrated with the help of an example. Figure 4.4(a) shows a two dimensional probabilistic map. The first value of each cell is the occupancy probability, the second value is the time until change. In this example the value range for time until change is [0, 20]. Region A is a crowded place, Region B has the characteristic of an infrequently used road in bad shape, cells in region C are static obstacles and region D is somewhere between the four corner cases shown in Table 4.1.

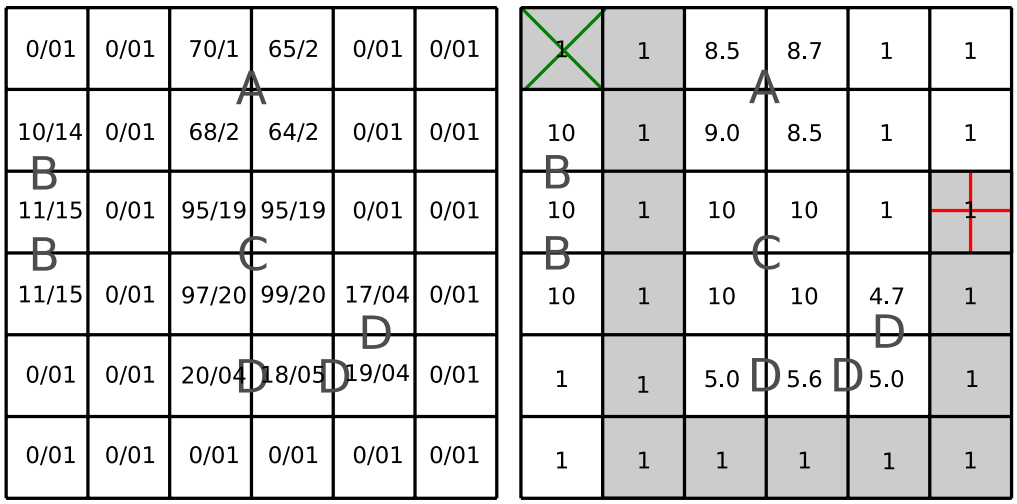


Figure 4.4: A probabilistic map, where the first value is the occupancy probability and the second value is the time until change (a). The computed grid cost values and the shortest path from start to goal (b)

The grid cell costs, computed with Equation 4.2 and the parameters $n = 10$, $forbidden\_probability = 90$, $forbidden\_time\_until\_change = 70$ and $min\_cost = 1$, are shown in Figure 4.4(b). Since time until change has a higher weight factor ($forbidden\_time\_until\_change$ is lower than $forbidden\_probability$), the grid cells in region D have relatively high cost values, although they have a low occupancy probability.

Assume a robot, only capable of translation in the directions up, left, down or right, wants to drive from start (diagonal cross) to the goal (symmetric cross). The algorithms from Section 2.2 would find the grey path, which indeed avoids the dynamic regions and the static obstacles.

Two major points have to be noted. First, the idea of the the probabilistic map path planning strategy is to avoid dynamic regions and thus, the impression could arise, that it is capable of avoiding dynamic obstacles. However, this is not true, since it is a pure global path planning strategy. Thus, the probabilistic map path planning strategy has to be combined with strategies presented in Chapter 3 to avoid dynamic obstacles.

Second, as discussed in Chapter 2, grid based path planners are not capable to find paths in higher dimensional configuration spaces, because of the complexity of the path planning

problem. Thus, it is to expect, that the probabilistic map path planning strategy will only work for rigid robots.

4.2 Dynamic path planning with sonar sensors

For dynamic obstacle avoidance, sensors are needed to observe the environment. Usually, laser scanners are used for the environment observation because of their precision. In this thesis, instead of laser scanners, sonar sensors are used for this task. The huge advantage of sonar sensors compared to laser scanners is, that they are significantly cheaper.

This section presents, how sonar sensors can be used for environment observation in dynamic path planning algorithms for partially known or unknown dynamic environments, as explained in Section 3.3. For this, an existing laser scanner environment observation technique is adapted to work with sonar sensors.

4.2.1 Laser vs Sonar

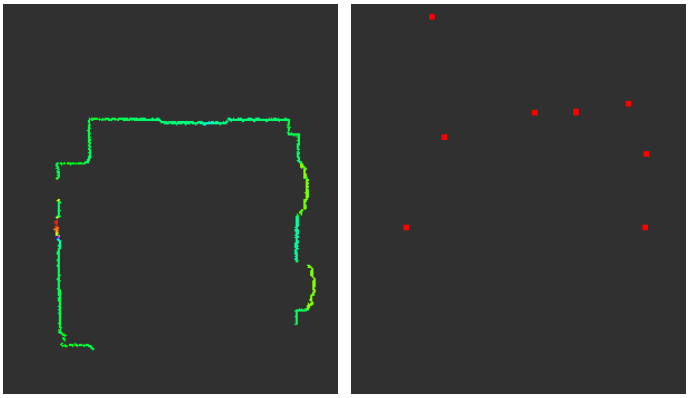
Before existing environment observation techniques for path planning can be modified, the general functionality and the differences between sonar sensors and laser scanners have to be examined. Sonar sensors emit *sound waves* with a specific frequency to detect obstacles. At obstacles the sound waves are reflected back to the sensor. The time between emitting and detecting a sound wave and the knowledge of the speed of sound can be used to calculate the distance to the obstacle. The functionality of a laser scanner is generally similar, but instead of *sound waves*, *laser beams* are emitted and the speed of light is used to calculate the distance. Additionally, the laser beam is steered, mostly with a mirror, to measure the distance in different angles.

These differences in the functionality lead to several shortcomings of sonar sensors compared to laser scanners. The first shortcoming is the amount of available information. Since the laser scanner can steer the laser beam, more measurements can be done per scan, compared to a single measurement a sonar sensor can do. For example, the SICK LMS100 laser scanner delivers 541 measured points (270° operation field with an angular resolution of 0.5°) in one scan. Figure 4.5(a) shows a measurement of the eight sonar sensors mounted on the Pioneer 3-AT robot platform and in Figure 4.5(b) the same measurement with a SICK LMS100 laser scanners is shown.

It is easy to see that the laser measurement has a huge quantitative advantage over the sonar sensor. The partial room layout is visible with one laser measurement. Additionally, laser scanners also have qualitative superiority. This is the case since sonar sensors suffer from three fundamental problems [16] [33]:

1. spatial resolution
2. specular reflections and cross talk
3. beam width

Sonar sensors have a lower *spatial resolution* than the laser system [16]. Thus, laser scanners produce more accurate range measurements than sonar based systems [18].



(a) (b)

Figure 4.5: The difference in the quantitative quality between the eight sonar sensors on the Pioneer 3-AT (b) and a SICK LMS100 laser scanners (a)

A more severe problem is, that sonar sensors suffer from *specular reflections* [33]. *Specular reflections* are incidents when the incoming sound waves do not get reflected back from the obstacle to the sonar sensors. Thus, the sonar sensors can not measure the time between emitting and detecting the sound wave. As a consequence, the sonar sensor does not detect the obstacle. Thus, only obstacles which are roughly perpendicular to the sound wave are detectable [15]. The specular reflection leads to another problem in multi sonar sensor environments. *Crosstalk* is the problem when one sonar sensor disturbs the measurement of another sonar sensor. This happens when an emitted sound wave of one sonar sensor gets reflected multiple times and reaches another sonar sensor during its current measurement. Thus, the receiving sonar sensor either measures the distance too short, or even worse, detects a non existent obstacle (a phantom obstacle).

Another problem is, that sound can not be focused as good as light. Thus, instead of a narrow beam, like in laser scanners, sonar sensors output a *broad cone*. This results in the inability to detect the exact position of the obstacle [38], since it can be everywhere on the circular arc created by the measured distance and the cone borders. To be on the safe side, it is assumed to be on the whole circular arc. This assumption creates the problem that obstacles are observed bigger than they are in reality. When an obstacle is detected right at one border of the cone, it gets extended until the other cone border. Figure 4.6 illustrates this problem, where the dashed lines represent the sonar cone and the solid line shows where the obstacle is assumed to be.

This is particularly problematic in narrow passages where the robot barely fits through, since the sonar sensors assume the gap at the left as well as at the right side to be smaller than it is in reality. Thus, such situations pose a serious problem for autonomously navigating robots which solely rely on sonar sensors.

When multiple sonar cones overlap, the possible locations of the obstacle can be potentially reduced. The possible obstacle locations are reduced by the sonar sensors, which have not detected the obstacle. Thus, the less sonar sensors detect the obstacle, the better the possible

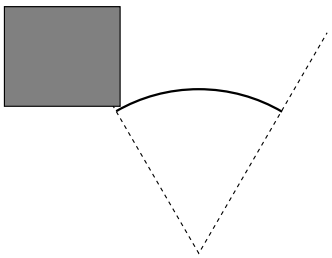


Figure 4.6: An example where the assumption that the obstacle is on the whole circular arc is problematic

location can be inferred. Figure 4.7 shows an example of such a situation.

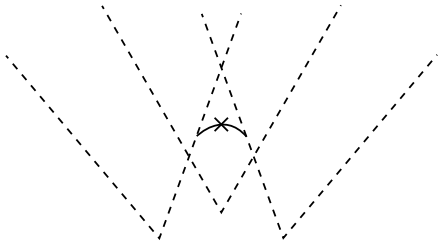


Figure 4.7: Overlapping sonar cones can potentially reduce the possible position of an obstacle

Since only the middle sonar sensor measures an obstacle at a range indicated by the cross, it can only be on the solid arc. Unfortunately, when all sonar sensors detect the obstacle, the possible obstacle locations can not be reduced.

Although sonar sensors suffer from these problems, there are reasons to use them nevertheless. First, sonar sensors are inexpensive. They are orders of magnitude less expansive than laser scanners [18]. Second, sonar sensors might not be optimal for map generation and detailed obstacle geometry recognition, but are good for pure obstacle detection [38]. Thus, if only the question, whether the space in front of the robot is clear or not, has to be answered, sonar sensors are a good choice.

Furthermore, laser scanners also have some problems. First, the advantage of a narrow beam can be a problem, since the focused light forms a narrow measurement plane. All obstacles under or above this measurement plane are not visible for the laser scanners [15]. Furthermore, obstacles with mirror or glass surfaces are problematic or impossible to detect for laser scanners.

4.2.2 Environment observation for dynamic path planning with laser scanners

The main task of environment observation for dynamic path planning is to provide a model of the environment in the vicinity of the robot. This model is used by the dynamic path planner to ensure that the robot is not colliding with obstacles, while driving to the goal.

The navigation stack of the robot operating system (ROS) [61] uses a laser scanner and a grid as environment model to fulfil this task. When a new laser scan is acquired, the first step

is to determine the grid cells which can not contain an obstacle. When a laser sensor detects an obstacle three meters in front of the robot, no obstacle can be present between the robot and the detected obstacle (at least not in the plane in which the laser is operating). This idea is implemented by raytracing from the laser sensor origin to each measured point in the point cloud. To compute the rays, the *Bresenham-algorithm* is used, which approximates a straight line between two given points with grid cells. Thus, the cell cost of each grid cell, which lays on a ray between the sensor origin and a measured point, is marked as empty. Figure 4.8 shows the line computed from the cell containing the sensor (lower left) to the cell containing the measured point (upper right).

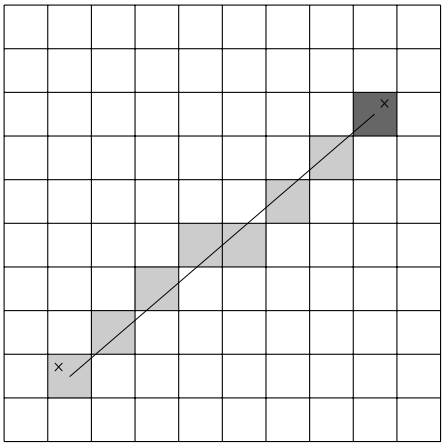


Figure 4.8: Raytracing a measurement to set grid cells as empty

After the raytracing step, the grid cells, which contain one or more measured points, are marked as containing an obstacle. In the example, every light grey grid cell would be marked as empty and the dark grey grid cell would be marked as containing an obstacle.

4.2.3 Environment observation for dynamic path planning with sonar sensors

The previously presented ideas are used to develop an environment observation for dynamic path planning with sonar sensors. The environment is modelled with the help of a grid, and raytracing is used to determine the occupied and free grid cells. Although the idea for determining which cells are occupied is presented before the idea to determine, which cells can not contain obstacles, the actual execution order is different. First, the cells are cleared and afterwards the obstacles are inserted into the grid. The presentation order was changed, since obstacle insertion for sonar sensors significantly influences how the cells are emptied. Furthermore, note that all algorithms assume, that the sonar sensor cones do not overlap.

Obstacle insertion

It is important to notice, that the semantics of sonar measurements are different from laser measurements. As explained in Section 4.2.1, due to the sonar cone, it is not possible to determine

the exact position of an obstacle in a sonar measurement. Thus, the obstacle has to be assumed on the whole circular arc which is created by the measured distance and the cone borders. To simplify this, the obstacle is instead assumed on a line between the point on the left cone border, which is exactly the measured distance away from the sonar sensor, and a similar point on the right cone border. "Left", "right" and "front" refer to the point of view of the sonar sensor. After the algorithm is presented, a justification for this simplification is presented.

The pseudo code of the environment observation for dynamic path planning is shown in Algorithm 6. Since the algorithm assumes no overlapping sonar sensors, it can be easily extended for multiple sonar sensors.

Algorithm 6 ProcessMeasurement($dist$)

```
1:  if dist ≥ MAX_DIST then
2:      return
3:  end if
4:  border_l.x ← dist · cos(sonar.θ + φ) + sonar.x
5:  border_l.y ← dist · sin(sonar.θ + φ) + sonar.y
6:  compute cell border_l_cell out of border_l
7:  border_r.x ← dist · cos(sonar.θ − φ) + sonar.x
8:  border_r.y ← dist · sin(sonar.θ − φ) + sonar.y
9:  compute cell border_r_cell out of border_r
10: clearCells(border_l_cell, border_r_cell)
11: doBresenham(border_l_cell, border_r_cell)
```

The algorithm gets as argument the measured distance of a sonar sensor. If the distance is greater or equal to a user defined *MAX_DIST*, the algorithm does nothing, since this indicates an invalid measurement (lines 1-3). In lines 4 and 5 the border point on the left cone border $border\_l$, which is exactly $dist$ away from the sonar sensor, is computed, where ϕ is the angle from one cone border to the cone's centre (see Figure 4.9).

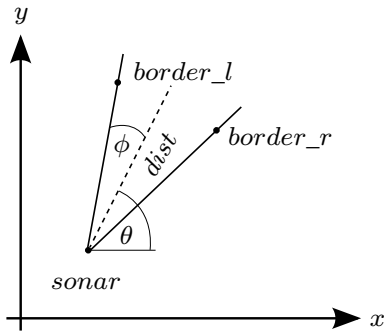


Figure 4.9: The variables used to compute the cone border points

Afterwards, the grid cell $border\_l\_cell$, which contains $border\_l$ is computed (line 6). In the same way, the grid cell $border\_r\_cell$, corresponding to the right cone border point is determined (lines 7-9). The function clearCells in lines 10 computes, based on the measurements,

the cells which can be marked as empty. How this is done is presented afterwards. The last step is to compute a straight line between $border\_l\_cell$ and $border\_r\_cell$ with the help of the Bresenham-algorithm and to mark the cells as containing an obstacle (line 11).

Since a straight line and not a circular arc is used to insert an obstacle into the occupancy grid, the obstacle is inserted nearer than the measurement is indicating. The error for an arbitrary distance is maximal in the centre of the cone. The farther away the sensed obstacle is, the bigger the error gets. The following formula can be used to calculate the error in the centre of the cone, where ϕ is again the angle from one cone border to the cone's centre:

$$max\_error = dist \cdot (1 - cos(\phi))$$

For example, assume the sonar sensor has an angle from one cone border to the cone's centre of roughly 7.5 degrees and a maximal range of 5 metres. Thus, the maximal error is 4.3 centimetres. This is not a problem, since it is inserted nearer than the obstacle is in reality. Hence, the robot will not collide with an obstacle, since it believes to be nearer than it really is. Furthermore, the error gets smaller when the robot is approaching the obstacle. At a distance of one metre, the error is only 0.9 centimetres. Both facts justify the used simplification.

Figure 4.10 depicts the functionality of Algorithm 6. The sonar sensor is located on the spot marked with the lower left cross. The dashed lines shows the cone's borders and the dotted line depicts the cone's centre. The angle from one cone border to the cone's centre (ϕ) is 20 degree. Assume that the sonar sensor measured an obstacle at a distance depicted by the cross on the dotted line. The algorithm calculates the points on the cone borders, which are the same distance away from the sonar sensor as the measured obstacle. The solid line in Figure 4.10 connects the cells, which contain the border points. With the help of the Bresenham-algorithm, the cells approximating this line are computed (grey cells). Every cell on this line is marked as containing an obstacle.

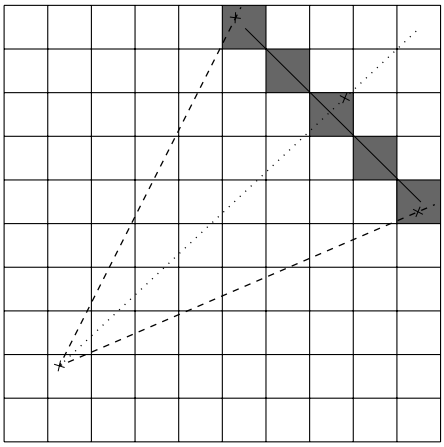


Figure 4.10: An example how Algorithm 6 inserts obstacles into the grid

Cell clearing

Cell clearing for sonar sensors uses the same idea as for laser scanners. When the sonar sensor measures an obstacle three meters away, no obstacle can be present between the sensor and the measured obstacle. The difference is, that not a single line but a triangle formed by the sonar cone borders and the straight line at the measured distance has to be cleared. All cells of this triangle have to be cleared.

Before the whole cell clearing function is explained, it is presented how the triangle's cells are cleared. With this knowledge, the whole cell clearing function is easier to understand. The idea is to clear all cells on a traverse through the triangle. Each column is traversed by starting at the border cell with the lowest y coordinate in this column. Then, neighbouring cells are visited by increasing the y coordinate until another border cell is reached. This is done for every column on which the triangle is present.

Since the traverse is always upwards, only two data structures describing the triangle are needed. The data structure $begin\_cells$ contains the cells, which approximate the lower side(s) of the triangle by the Bresenham-algorithm. Similarly, the data structure end_cells contains the cells of the top side(s) approximation. Figure 4.11 shows the lower side (marked with b for $begin\_cells$) and top sides (marked with e for end_cells) of an example triangle. Other possible orientations for a triangle are discussed later in this section.

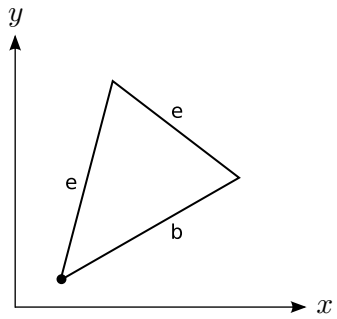


Figure 4.11: The top (e for end_cells) and bottom (b for $begin\_cells$) sides of a triangle

Algorithm 7 shows the pseudo code of function clearTriangle, implementing this ideas.

Algorithm 7 clearTriangle($begin\_cells$, end_cells)

```
1: for i = begin_cells.leftmostColIdx() to begin_cells.rightmostColIdx() do
2:     for j = begin_cells.col(i).smallestY() to end_cells.col(i).greatestY() do
3:         clear grid cell at (i, j)
4:     end for
5: end for
```

The outer loop iterates over the columns (x-axis), starting with the lowest (leftmost) one. In the inner loop, the column is traversed from the lowest cell to the highest cell (y-axis), clearing all cells on this traverse.

The clearCells function, is shown in Algorithm 8. The arguments are the previously computed border cells $border\_l\_cell$ and $border\_r\_cell$.

Algorithm 8 clearCells($border\_l\_cell$, $border\_r\_cell$)

```
 1: leftCells ← doBresenham(sonar_cell, border_l_cell)
 2: rightCells ← doBresenham(sonar_cell, border_r_cell)
 3: frontCells ← doBresenham(border_l_cell, border_r_cell)
 4: if border_l_cell.x < sonar_cell.x and border_r_cell.x < sonar_cell.x then
 5:     if border_l_cell.x < border_r_cell.x then                          ▷ 1.1
 6:         begin_cells ← leftCells
 7:         end_cells ← merge(rightCells, frontCells)
 8:     else                                                              ▷ 1.2
 9:         begin_cells ← merge(leftCells, frontCells)
10:         end_cells ← rightCells
11:     end if
12: else if border_l_cell.x > sonar_cell.x and border_r_cell.x > sonar_cell.x then
13:     if border_l_cell.x < border_r_cell.x then                          ▷ 2.1
14:         begin_cells ← rightCells
15:         end_cells ← merge(leftCells, frontCells)
16:     else                                                              ▷ 2.2
17:         begin_cells ← merge(rightCells, frontCells)
18:         end_cells ← leftCells
19:     end if
20: else
21:     if border_l_cell.x < border_r_cell.x then                          ▷ 3.1
22:         begin_cells ← merge(leftCells, rightCells)
23:         end_cells ← frontCells
24:     else                                                              ▷ 3.2
25:         begin_cells ← frontCells
26:         end_cells ← merge(leftCells, rightCells)
27:     end if
28: end if
29: clearTriangle(begin_cells, end_cells)
```

First, the cells forming the three borders of the sonar cone (triangle) are computed. Initially, a straight line between the cell $sonar\_cell$, containing the sonar sensor and the previously calculated $border\_l\_cell$ is computed with the help of the Bresenham-algorithm (line 1). This version of the Bresenham-algorithm delivers a data structure $leftCells$, containing the cells, which approximate the line. Line 2 computes a line from the sonar sensor cell to the right cone border cell and delivers a similar data structure $rightCells$ as before. In line 3, a line between the left and right cone border is drawn, forming the border of the triangle in front of the sonar sensor and delivering the data structure $frontCells$.

The next step is to determine which of the data structures $leftCells$, $rightCells$ and $front-$

$Cells$ form the data structure $begin\_cells$ respectively the data structure end_cells for the function clearTriangle. The decision depends on the orientation of the triangle in the grid.

Figure 4.12 shows the six triangle orientations which have to be distinguished by the function, to determine which data structure form the $begin\_cells$ and the end_cells. With them, every possible triangle rotation is covered. The black dot marks the position of the sonar sensor. Edges labelled with b have to be in $begin\_cells$ and edges labelled with e have to be in end_cells.

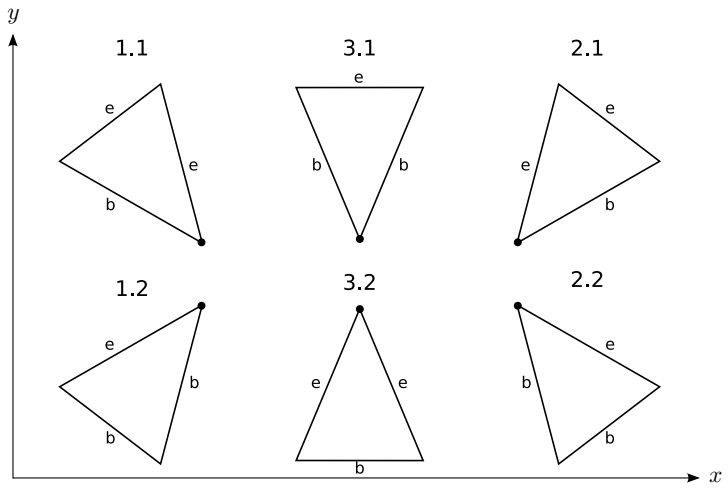


Figure 4.12: Six cases, which have to be distinguished by function *heading*

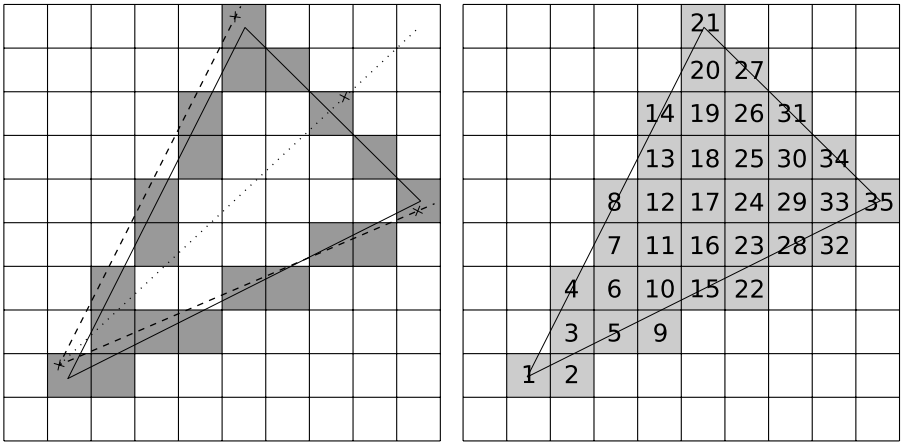
Case 1 is characterized by the fact, that the x cell coordinate of the left and right border points are smaller than the sonar position (line 4). If the x cell coordinate of the left border point is smaller than the right border point, the cone opens to the "top" (line 5, case 1.1). Thus, $leftCells$ forms $begin\_cells$ (line 6) and $rightCells$ combined with $frontCells$ form end_cells (line 7). If the left border point is not smaller than the right border point, case 1.2 applies. In this case, $leftCells$ and $frontCells$ form $begin\_cells$ (line 9) and $rightCells$ is end_cells (line 10).

Case 2 applies, when the x cell coordinate of both border points are higher than the sonar position (line 12). Lines 13-19 are similar to case 1, but are adapted to the changed rotation as shown in Figure 4.12. When the x cell coordinate of the sonar sensor is in between the left and right cone border x cell coordinate, case 3 has to be applied. Again, lines 21-27 are similar to case 1 with the necessary adaptations. At last, the function clearTriangle is executed with $begin\_cells$ and end_cells to clear the triangle.

4.2.4 Example

The environment observation for dynamic path planning with sonar sensors is concluded with an example, to see the effects of the algorithm. Figure 4.13(a) shows the sonar cone's borders as dashed lines and the centre of the cone with the dotted line. The cross on the dotted line depicts a measured obstacle, and the crosses on the cone's borders are the calculated border

points. The solid lines, connecting the cell containing the sonar sensor and the cells containing the border points, are approximated by the Bresenham-algorithm. The dark grey cells are in the data structures $leftCells$, $rightCells$ and $frontCells$ returned by the Bresenham-algorithm.



(a) (b)

Figure 4.13: The dark grey cells approximate the solid lines and are in the data structures $leftCells$, $rightCells$ and $frontCells$ (a) the ordering of how function clearTriangle (Algorithm 7) clears the cells of a triangle

Every light grey cell depicted in Figure 4.13(b) is cleared by the function clearTriangle (Algorithm 7). The numbers are the order in which the algorithm is traversing through the triangle, formed by the approximated lines. After the cells are cleared, the obstacle is inserted as shown in Figure 4.14. The dark grey cells are marked as containing an obstacle.

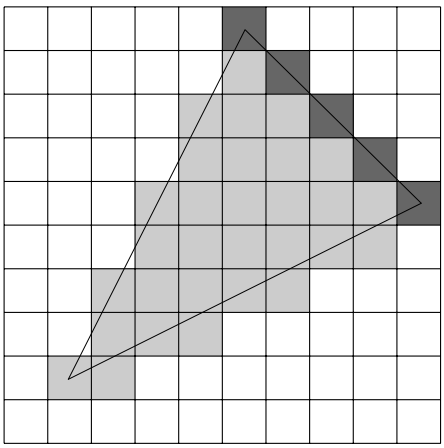


Figure 4.14: The result of clearing cells and inserting an obstacle based on a measurement

CHAPTER 5

Implementation

This Chapter shows, how the ideas of using probabilistic maps and sonar sensors for path planning, presented in Chapter 4 are implemented in the open source framework "robot operating system" (ROS) [71]. The goal is that a Pioneer 3-AT (P3AT) [49] from MobileRobots, supplied with a probabilistic map from [11], is capable of driving autonomously with its on-board sonar sensors to a goal.

5.1 Robot operating system (ROS)

The robot operation system (ROS) is an open source framework which tries to ease the burden of writing robust robot software. It contains tools, libraries and modules which address the most common tasks for autonomous robots. Although the name implies that ROS is an operating system, this is not the case. Functionalities a modern operating system offers, like memory management, process management or scheduling are not present. Actually, it runs on top of most common operating systems (e.g., Ubuntu, Debian, Windows). The core of ROS forms an interprocess communication facility with four capabilities [72]:

- **Message Passing:** Software in this system can communicate via anonymous publish/subscribe. Nodes subscribe at the master software node (ROS master noder) to one or more topics in which they are interested. When a message is published at a given topic, every node registered for this topic receives this message. This allows a very modular software structure. For example, one node is exclusively responsible to publish the measurement of a laser sensor. Other nodes can use this measurement to fulfil their tasks, without bothering with low level communication with the laser sensor. When the laser sensor is changed, only one node has to be updated accordingly.
- **Recording and Playback of Messages:** Since message passing in ROS is anonymous and asynchronous, messages can be saved to a file and replayed later, without changing any code in the modules. For example, laser sensor messages can be recorded and played back at a later time-point to test a module under development in an offline fashion.

- **Remote Procedure Calls:** When synchronous communication is required, instead of the asynchronous message passing, ROS provides remote procedure calls with so called services.

- **Distributed Parameter System:** Modules can share configuration information. This allows for easy module reconfiguration. Additionally, modules can change the configuration of other modules during the runtime to influence their behaviour.

ROS was developed around the idea of collaborative robotics software development. This manifests in the fact, that a lot of open source modules are present, which can be built upon and are supplied by the community are present. Thus, users can immediately start with the implementation of their specific idea and do not have to build needed, but for their interest unnecessary, features first.

ROS has three conceptual levels [65]:

- the Filesystem level

- the Computation Graph level

- the Community level

The following sections will present the major components of each level. A complete list can be found at [65].

5.1.1 Filesystem level

The Filesystem level mainly consists of resources on the disk such as:

- *Packages:* A package is the main unit of software management in ROS. It can consist of code, libraries, configuration files or other things which are logically connected.

- *Stacks:* Stacks consist of a collection of packages which solve a task together.

- *Message (msg) types:* Message types define the message structures which are sent in ROS.

- *Service (srv) types:* Service types define the request/response data structure used by services (remote procedure calls).

5.1.2 Computation Graph level

The Computation Graph is the communication facility of ROS and can be seen as virtual network through which different ROS processes communicate.

- *Nodes:* Nodes perform computations in the ROS system by executing packages. Since ROS is modular, many different nodes are active at the same time solving various tasks together.

- *Messages:* With the help of messages, nodes communicate with each other. A message is a data structure consisting of various fields of different types. The structure of a message is defined by the message type.

- *Topics:* Since communication in ROS follows the publish/subscribe pattern, topics are needed to control the message exchange. All nodes interested in a specific topic *subscribe* to it. A node sends a message by *publishing* it for a given topic and all nodes subscribed to this topic receive it. For a single topic multiple subscribers and publishers can exist. Furthermore, a node can publish and subscribe to multiple topics.

- *Services:* Services allow for synchronous communication via remote procedure calls, following the request/reply pattern. The message type of the request and the reply is defined by the service type.

- *Master:* The master stores the published/subscribed topics for each node. Thus, when a node subscribes to a topic, it asks the master for all registered publishers for this topic. With this information the node connects to the publisher nodes via standard TCP/IP sockets. Should a new node register as publisher, the master informs every subscriber node about this event via callbacks. This allows the subscriber nodes to connect to the new publisher node. Hence the master node is responsible that the nodes can find each other to exchange messages.

Since the low level communication between ROS nodes is done with TCP/IP, not all of them have to be located in the same system. Instead, they can be distributed between multiple machines. This enables different interesting scenarios. For example, a robot with a weak on-board computing system could send data to a strong system to outsource the more complex tasks. Another scenario could involve multiple robots which are able to access data from other vehicles to improve their local knowledge.

5.1.3 Community level

As already explained, ROS is built around the idea of code sharing. Thus, at the Community level ROS provides different resources for developers like a wiki or a mailing-list to share their developments.

5.1.4 Transformations

Another important concept in robotics is tht of *transformation*, which is handled by the tf package [70] in ROS. The need for transformations arises when a point in one frame (see Section 2.1.2) should be transformed to a point in another frame. For example, assume the robot frame's coordinate origin is at the centre of the robot. Furthermore, let a laser sensor be mounted, not in the middle of the robot, but displaced. A measurement of the laser sensor will be in the laser sensor frame. To know how far a possible sensed obstacle is away from the robot centre, the measurements have to be transformed into the robot frame. This is an example for a *static transformation*, since the origins of the robot frame (the middle of the robot) and the laser frame

(the middle of the laser sensor) do not move relative to each other. Thus, the transformation can be done with the help of a matrix multiplication, where the elements of the matrix are static and can thus be computed once.

More complicated are *dynamic transformations*. As the name implies, these transformations are time dependent. For example, consider again a robot with its frame centred at the origin and a map. Initially the robot stays in the centre of the map frame. When the robot is driving, the position of the robot changes in the map frame but the robot always stays in the centre of the robot frame. Hence, depending on the time, the position of the robot in the map frame changes and the origins of both frames move relative to each other. Again, to transform a point from one frame to another the point has to be multiplied with a matrix. The problem is that the elements of the matrix are not constant, since the relative position of the origins is not constant.

The tf package of ROS is capable of both transformations. The relationship of all frames is maintained in a buffered tree structure, the *transformation tree*, where the vertices are the existing frames. In the previous examples the map frame would be the root node with an edge to the robot frame. Since the laser sensor is mounted on the robot, the robot frame would be the parent of the laser frame. The static transformation from the laser frame to the robot frame can be published by a tool contained in the tf module. The dynamic transformation from the robot frame to the map frame is typically done by a localisation module. When all transformations are set up correctly, it is possible to transform points from any frame to any other frame by following the path between the two frames.

5.1.5 Starting a ROS node

There are two proper ways to start a ROS node. The easiest possibility is to use the command line tool *rosrun*, which allows to run an executable, where parameters may be given optionally, with the following syntax:

```
rosrun <package> <executable> _parameter:=value
```

In cases where multiple nodes with various parameters should be started, rosrun is impractical. A better way is to use the powerful tool *roslaunch*. It was developed to ease the starting of multiple local or remote nodes and setting the parameters of these nodes. The tool accepts a XML-file (with the .launch extension), which specifies which nodes should be started with which parameters [66]. Because of the number of needed nodes and its usefulness, roslaunch was used in this thesis. To use roslaunch to start nodes with a given launchfile, the following command is used:

```
roslaunch <file>
```

The most important tags for the launch file are:

- *<launch>:* It is the root element of a launch file and acts as a container for other elements.
- *<node>:* This tag specifies a ROS node which should be started. Note that roslaunch does not guarantee the starting order when multiple nodes should be started. As a consequence it must not be assumed that a specific node is already running, but a node has to be capable to wait until all required nodes are started. The following attributes are used in the thesis:

  - *pkg:* The ROS package in which the executable of the node is located.
  - *type:* The executable in the specified package which should be run.
  - *name:* The name of the node.
  - *output:* An optional argument. If output is set to screen, stdout/stderr from the node will be seen on the screen. If log is stated, stdout is directed to a log file in the ROS home directory and stderr will be printed on screen. The default argument is log.
  - *args:* The arguments which should be passed to the node.

  For a full list of attributes see [68].

- <*param*>*:* Used to set a single parameter on the parameter server. The variant used most often is to specify the tag inside of a <node> tag, in which case the parameter is set for this node. The following attributes occur in this thesis:

  - *name:* The name of the parameter.
  - *type:* The type of the parameter. Legal options are str, int, double and bool.
  - *value:* To which value the parameter should be set. Must conform to the specified type.

  For a full list of attributes consult [68].

- *rosparam:* This tag allows to load, dump and delete multiple parameters from the parameter server, by reading a rosparam YAML file. Alternatively, a parameter with a list type can be set with the help of rosparam. Again, the most common way is to use the tag inside of a <node> tag. The used parameters are:

  - *command:* An optional command which specifies the action. Valid values are load (the default), dump and delete.
  - *file:* The path to the YAML file, when the parameters should be read from a file. The file consists of *name: value* tuples.
  - *param:* The name of the parameter (when rosparam is used to set a parameter with a list as type).
  - *ns:* The namespace in which the parameter has its scope.

  Again, a full list of attributes can be found in [68].

- *include:* Allows to include another roslaunch XML file into the current one. Thus, it is possible to construct a complex launch file out of simple launch files. This mirrors the ROS architecture of complexity via composition [67]. The following attributes were used:

  - *file:* The path to the launch file.
  - *ns:* The namespace in which the file should be imported.

  Other attributes are listed in [68].

5.2 System Overview

Figure 5.1 shows the high-level ROS system architecture of the autonomous driving P3AT robot. All used components and their intercommunication can be seen. The boxes represent ROS packages and stacks respectively. A solid arrow represents a publish/subscribe communication, with the topic as label. The nodes at the arrow-ends are subscribed to the respective topic and consequently, nodes on the opposite ends publish messages to the topic. Dashed arrows represent ROS services. The nodes at the arrow-ends are the callees and the nodes at the opposite ends are the callers. Thus, the direction of the arrows illustrates the information flow.

The most important part, which forms the heart of the system, is the *navigation stack*. Although it is only depicted as a single package, it consists of various modules with complex interactions. It applies methods from Chapter 2 and Chapter 3 to safely move the robot from an initial position to a goal. The navigation stack subscribes to the */initialpose* topic and to the */move_base_simple/goal* topic to get the initial and goal location respectively. To publish both topics either a powerful GUI called *rviz* or the *navigation control module*, a slim command line tool, can be used.

The *probabilistic map server* reads a probabilistic map from a file and provides a static and a probabilistic map for other ROS modules. They get distributed either by the services *static_map* and *probabilistic_map* or via the topics */map* and */prob_map*. The navigation stack uses both maps and requests the static_map via the service *static_map* and subscribes to the topic */prob_map* to receive the probabilistic map.

The rosaria module allows other ROS modules to access functionalities of the P3AT robot. Rosaria publishes all sonar measurements to the */rosaria/sonar* topic. These are needed by the navigation stack for localisation and obstacle avoidance. Since the sonar measurements from rosaria contain systematic measurement errors, a *sonar calibration* module was developed to correct systematic errors. When a new sonar measurement is published on the */rosaria/sonar* topic the *sonar calibration* corrects it and publishes on the */sonar* topic.

The navigation stack creates velocity commands to drive the robot around. The navigation stack does this by publishing on the */cmd_vel* topic. The *safe navigation* module listens to this topic to determine if the movement command is safe. This is done by checking if the robot is currently touching an obstacle with its bumpers. The bumper information is published by rosaria to the */rosaria/bumper_state* topic. Depending on the bumpers, the safety module publishes commands to the */rosaria/cmd_vel* topic, which are used by rosaria to actually control the robots motors.

Not every module in Figure 5.1 was developed within the scope of this thesis. Modules with a white background colour were already existing and were not modified. Grey backgrounds indicate modules which were modified for this thesis. On the other hand, black modules were developed especially for this thesis.

The following sections will elaborate on the modules and explain the changes made.

5.2.1 Global transformation tree

Figure 5.2 shows the transformation tree of the whole system. The root frame of the transformation tree is the *map* frame, in which map positions are expressed. The *odom* frame, which

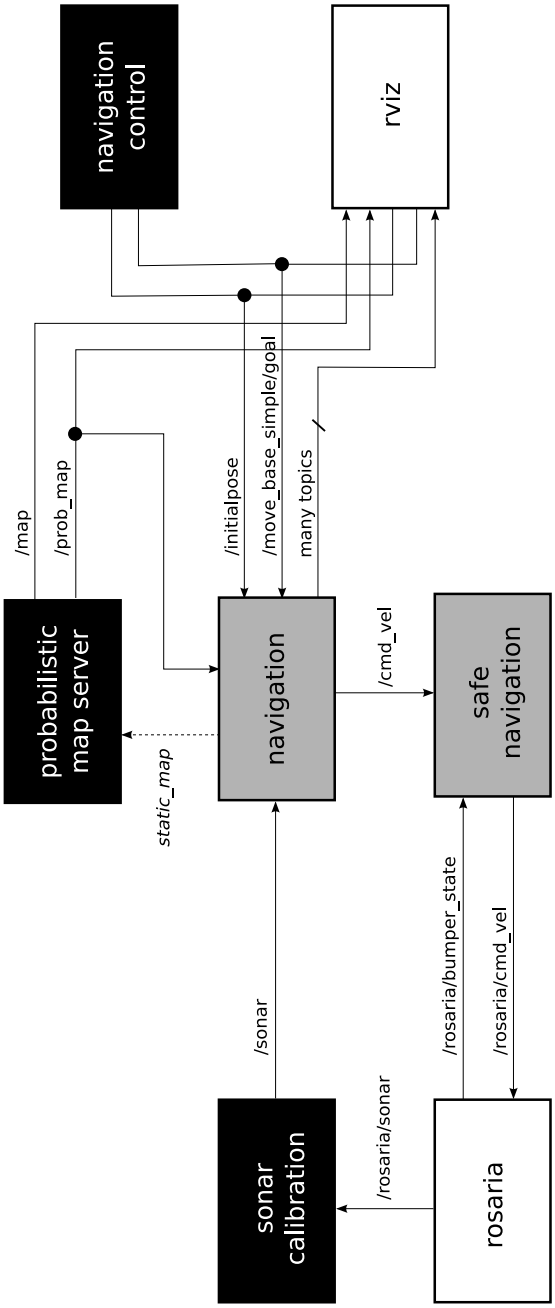


Figure 5.1: The high-level system architecture of the autonomous driving P3AT robot

is a child of the *map* frame, expresses the odometry information. The *AMCL* module (see Section 5.6.1) establishes the dynamic transformation between the *map*- and the *odom* frame. The *base_link* frame is the robot frame, where the coordinate frame is attached to the robot. The

centre of the robot is the origin of the *base_link*. *Rosaria* (see Section 5.3) manages the dynamic transformation between the *odom*- and the *base_link* frame. With this, it is possible to express the position of the robot in the *odom* frame or in the *map* frame. With the help of child frames of *base_link*, the position of robot parts like the sonar sensors can be expressed in arbitrary frames, as long as they are part of the transformation tree. Since the parts are fixed on the robot, transformations between them and *base_link* are static. Except for the *front_sonarX* frames, all those transformations are provided by the *p2os_urdf* package [63]. It contains a *Unified Robot Description Format (URDF)* model for the P3AT, which is an XML format to describe a robot layout. The static transformations from the *front_sonarX* frames to the *front_sonar* frame are provided by *static transformation publishers* [70]. The arguments to configure the static transformation publishers can be found in Appendix A.1.

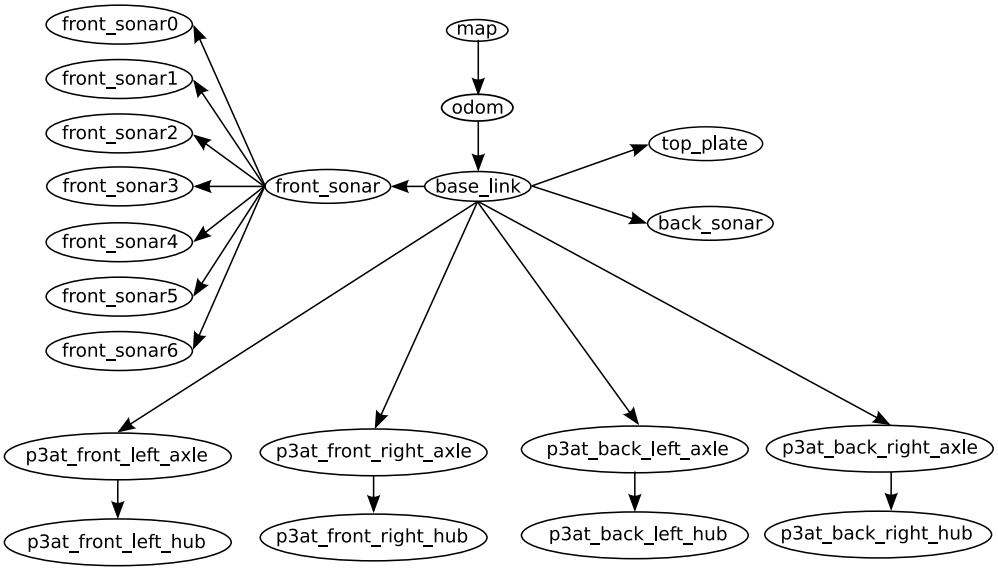


Figure 5.2: The ROS transformation tree, used in this thesis

5.3 Rosaria

The rosaria package [64] is needed to interact with the robot. It wraps the aria C++ drivers, such that they can be used in ROS. This allows other ROS modules to set velocity commands to drive the robot around and read odometry information, like estimated position. Furthermore, the front and back bumpers can be read to detect whether the robot is colliding with obstacles. Moreover, the eight front sonar sensors can be read to detect distant obstacles. In this thesis, rosaria publishes/subscribes the following topics, where the brackets indicate the message type:

- */rosaria/cmd_vel (geometry_msgs/Twist):* Allows to set velocity commands for the robot, which aria will maintains automatically. Hence, to obtain constant velocity, the command has to be sent only once.

- */rosaria/pose (nav_msgs/Odometry):* Delivers the pose of the robot relative to the starting position in the *odom* frame.
- */rosaria/bumper_state (rosaria/BumperState):* Delivers the bumper state of the front/back bumpers.
- */rosaria/sonar (sensor_msgs/PointCloud):* Delivers the sonar readings in the front_sonar frame.
- */tf (tf/tfMessage):* Rosaria publishes the position of the robot in the odom frame. This is done by providing the required informations for the transformation tree to transform points from the *base_link* to the map *odom* frame (and vice-versa). Thus, the position of the robot in an arbitrary frame can be requested from other ROS modules with the help of the tf module.

To deliver somewhat reasonable odometry results, three parameter values have to be set to calibrate the rotary encoders:

- *TicksMM (double):* The amount of read encoder ticks for an one millimetre tire rotation.
- *DriftFactor (double):* This value gets added to encoder reading of the left tire to compensate for tire circumference differences.
- *RevCount (double):* Specifies the amount of read encoder ticks for an 180 degree rotation.

Another important parameter is *port*. This has to be set to the device over which the computer communicates with the robot hardware. The correct port for the P3AT robot used in this thesis is */dev/ttyS0*.

No changes had to be made to rosaria to use it in this thesis. The values used for the odometry calibration can be found in Appendix A.1.

5.4 Probabilistic map server

The *probabilistic map server* reads a probabilistic map from the filesystem and provides it as a static and as a probabilistic map for other ROS modules. The probabilistic map server was developed especially for this thesis and uses different ideas and code snippets from dynamic mapping in [11] and the already existing static map server from the navigation stack [58].

The first step is to read the header-file of the probabilistic map. This file has to contain the following tags, which are explained in [11]:

- image
- resolution
- origin
- negate

- occupied_thresh
- free_tresh
- map_array_size
- dynamic
- dynamic_time_max

Afterwards, the map can be loaded from the filesystem and provided for other modules, which is done in two different ways. First, the map with and without dynamic information gets published on two topics, one for the static map and one for the probabilistic map. Additionally, the probabilistic map server publishes the map meta-data, like size and resolution on a third topic. The second method is to use ROS services. Two service callback methods are present to forward the static respectively the probabilistic map to the server caller.

In this thesis, the probabilistic map server publishes to the following topics and provides the following services:

- */map (nav_msgs/OccupancyGrid):* The topic on which the static map is published.
- */prob_map (dynamic_mapping/DynamicGrid):* The probabilistic map is published on this topic.
- */map_metadata (nav_msgs/MapMetaData):* The topic on which the map metadata is published.
- *static_map:* The service on which the static map can be requested.
- *probabilistic_map:* The service on which the probabilistic map can be requested.

The probabilistic map server has the following parameters:

- *file_name (string):* The path to the header file.
- *static_service_name (string):* The name of the service to acquire the static map.
- *static_map_topic (string):* The topic where the static map should be published.
- *static_frame_id: (string)* The frame of the static map.
- *probabilistic_service_name (string):* The name of the service to acquire the probabilistic map.
- *probabilistic_map_topic (string):* The topic where the probabilistic map should be published.
- *probabilistic_frame_id (string):* The frame of the probabilistic map.
- *map_metadata_topic (string:* The topic where the map meta-data should be published.

The parameter values used in this thesis can be found in Appendix A.1.

5.5 Sonar calibration

In this thesis, a sonar sensor message published by rosaria, contains eight sonar measurements. One measurement for every sonar sensor, where the ordering in the message coincides with the sonar sensors shown in Figure 6.2. Unfortunately, these sonar sensors readings delivered by rosaria are unaccurate. Figure 5.3 shows the same situation as Figure 4.5 but the sonar sensor readings from rosaria are overlayed with the laser readings.

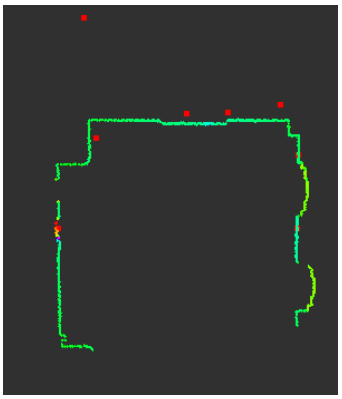


Figure 5.3: A sonar measurement compared to a laser measurement

It is easy to see that the sonar sensors do not measure the same distance as the laser sensor. Furthermore, extensive experiments showed, that the range difference of the measurement of the laser- and sonar sensor increases, the closer the obstacle is to the sensor. Additionally, the further the sonar sensors are on the side of the robot, the less the distance error is.

To overcome these problems, the *sonar calibration* module was developed. It subscribes to the topic on which rosaria publishes the sonar sensor measurement messages. The sonar calibration module calibrates the received measurements by using reference values and publishes the resulting sonar measurements on a second topic.

The eight sonar measurements in a published sonar measurement message are points in the front_sonar frame. The calibration is done by subtracting the estimated range error from the x and y component of the measurement of every sonar sensor. How much the estimated range error has to be reduced from x and y component depends on the angle of the sonar sensor to the x-axis in the front_sonar frame (ϕ). Thus, the following equations are used to compute the new point:

$$x_{new} = x_{old} - error \cdot cos(\phi)$$
$$y_{new} = y_{old} - error \cdot sin(\phi)$$

The range errors are estimated for every sonar sensor by its own linear function which were determined with the help of extensive experiments.

In this thesis, the sonar calibration module publishes/subscribes to the following topics:

- */rosaria/sonar (sensor_msgs/PointCloud):* On this topic the uncalibrated sonar measurements are published from rosaria.

- */sonar (sensor_msgs/PointCloud):* The calibrated measurements are published by the sonar calibration module on this topic.
- */tf: (tf/tfMessage):* The sonar calibration needs the position of each sonar sensor in the front_sonar frame, which is provided by the tf module on this topic.

The following parameters are used by the sonar calibration module:

- *linear_functions (list):* The eight linear functions, each represented by two parameters k and d. The first linear function is for the first (leftmost) sonar sensor, the second linear function for the second sonar sensor and so on.
- *sonar_sensor_topic_in (string):* The topic on which the uncalibrated sonar measurements are published.
- *sonar_sensor_topic_out (string):* The topic on which the calibrated sonar measurements are published by the sonar calibration module.

The parameter values used in this thesis can be found in Appendix A.1. Figure 5.4 shows the situation from Figure 5.3 with calibrated (white) sonar measurements. Note, that only the four middle sonar sensors are calibrated, since the side sonar sensors deliver reasonable measurements.

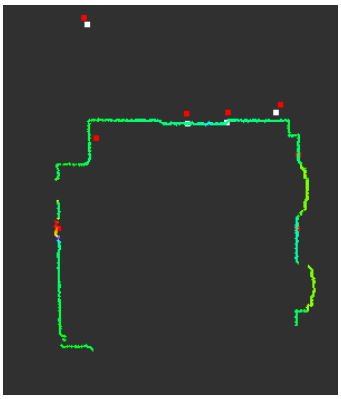


Figure 5.4: A calibrated sonar measurement compared to a laser measurement

5.6 Navigation stack

The *navigation stack* consists of various different ROS packages that solve together the task of autonomous robot navigation. The navigation stack has the following three hardware requirements [61]:

1. The robot has a differential or holonomic drive. Hence, the navigation stack produces velocity commands for x, y and theta.

2. A planar laser sensor is mounted on the robot. As already mentioned, only sonar sensors are used in this thesis. As a consequence, various parts of the navigation stack had to be changed to work with sonar sensors.

3. The robot's planar geometry should be (nearly) a square or circle. The navigation stack works for arbitrary shapes and sizes too, but it will have problems in narrow spaces like doors with large rectangular robots. The P3AT is not a square, thus problems in very narrow spaces are expected, especially in combination with sonar sensors (see Section 4.2.1).

In the navigation stack, path planning is implemented by combining a global path planning method with a local obstacle avoider, as presented in Section 3.3.3. Figure 5.5 shows an overview of the internal structure and dependencies of the navigation stack.

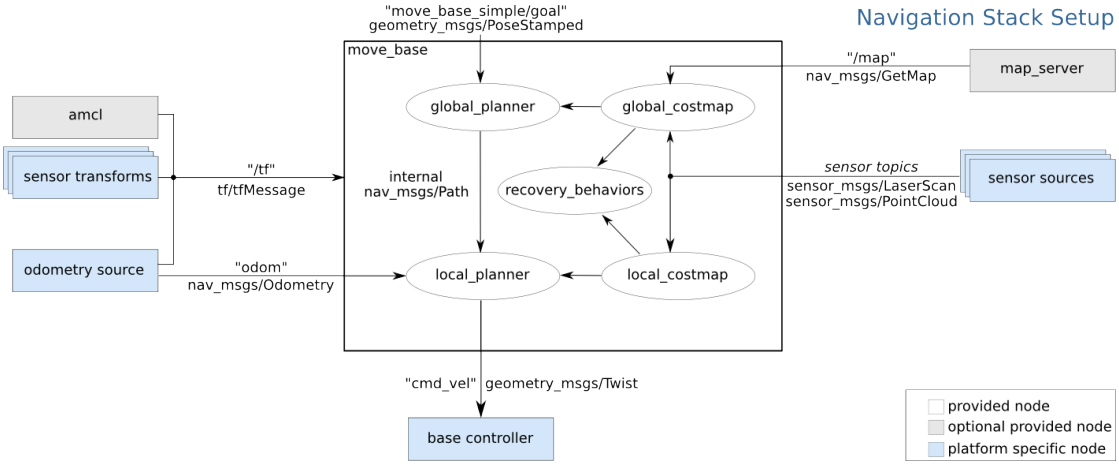


Figure 5.5: Overview of the internal structure of the navigation stack [62]

The heart of the navigation stack is the *move_base* module, consisting of various plug-ins. Thus, nearly every aspect can be influenced by writing and using new plug-ins, allowing for maximal flexibility. Move_base consists of exactly one *global-* and one *local_planner* plug-in, each with an instance of a *costmap*. Additionally, various *recovery_behaviour* plug-ins can be present to recover the robot in unexpected situations. The costmaps hold information about obstacles in the environment and are fed from sensors. Since only sonar sensors are used in this thesis, the information is provided by the sonar calibration module. Furthermore, the *global_costmap* gets supplied with a map of the environment. Obviously, the probabilistic map server module is responsible for this task. The velocity command produced by the local_planner is used by the *base controller* to drive the robot around. In this thesis, this is the safe navigation module which forwards the command to rosaria.

The left side hints the required transformations for the *move_base*. First, it has to know where the robot is located in the map. This is done by transforming the odometry frame (*odom*) to the map frame (*map*). Since this is a dynamic transformation, a module has to provide the appropriate information for the *tf* module. The *AMCL* module implements localisation and informs the *tf* module about the required transformations. The odometry source is the rosaria

module. The transformation between the sonar sensors and the robot frame (*base_frame*) is static. Thus, the required information can be supplied in advance and no further information is needed for these transformations. Additionally, the local_planner has to know the current speed, which is supplied by the rosaria module.

5.6.1 Adaptive Monte Carlo Localisation

The *Adaptive Monte Carlo Localisation (AMCL)* [56] module is responsible for *localisation*. *Localisation* is the task to determine the position of the robot in a known environment. This is done by combining observations of the environment, robot odometry information and a map to reason with the help of statistical methods about the robot's position. Therefore, the robot's movement and the sensor are modelled mathematically. With the help of a sensor model, the probability for a pose, given a measurement and the map, can be estimated. More concrete, the AMCL module implements a *particle filter*, namely the *adaptive (or KLD-sampling) Monte Carlo localisation approach* from [81], where also detailed information about how localisation is done can be found.

In this thesis, AMCL publishes/subscribes the following topics.

1. */sonar (sensor_msgs/PointCloud):* The sonar measurements, which are the previous "observation of the environment".
2. */tf (tf/tfMessage):* Tf is needed for two aspects. First, the odometry information maintained by rosaria is required for particle filtering. Second, the estimated position of the robot in the map frame is published. This is done by providing the required information for the tf module to transform points from the odom frame to the map frame (and vice-versa).
3. */initialpose (geometry_msgs/PoseWithCovarianceStamped):* Specifies roughly the current position of the robot in the map and AMCL will distribute particles for the Monte Carlo localisation around this position. The better this approximation is, the faster the robot can localise itself afterwards.
4. */amcl_pose (geometry_msgs/PoseWithCovarianceStamped):* On this topic, AMCL publishes the current estimated position of the robot in the map frame.
5. */particlecloud (geometry_msgs/PoseArray):* The topic on which AMCL publishes the particles of the filter for information or debugging purposes.

Since localisation is done with the help of previously perceived obstacles, a probabilistic map does not help to improve localisation. Thus, in this module no changes regarding maps had to be made. To get the static map from the probabilistic map server, the service *static_map* gets called.

Adaptations

Some adaptations had to be made on the AMCL module to involve the changed requirements. First, a sonar sensor model had to be developed. In the case of AMCL, the most severe dif-

ference between laser and sonar sensors in ROS is the message format of the measurement messages. The main difference between these message formats is that *sensor_msgs/LaserScan* messages from a laser sensor deliver the measurements in polar coordinates, whereas *sensor_msgs/PointCloud* messages from a sonar sensor deliver the measurement in cartesian coordinates. Since it is easy to convert between these representations the sonar sensor model is very similar to the original laser sensor model.

Since the other parts of both models are similar, most parts of the laser model in the *AMCLLaser* class are transferred into the abstract class *AMCLRange*. Among others, *AMCLRange* contains three pure virtual methods, which have to be implemented by the child classes *AMCLLaser* and *AMCLSonar*. These three methods express the difference between laser and sonar measurements. *computePolarCoordinates* and *computeCartesianCoordinates*, which are two out of these three methods, convert the measurement either to polar- or cartesian coordinates. Since sonar measurements are already in cartesian coordinates, in *AMCLSonar computeCartesianCoordinates* just returns the input. The same is true for *computePolarCoordinates* in *AMCLLaser*, since the laser measurements are already in polar coordinates. The last pure virtual method is responsible to check if the delivered measurement is invalid. A measurement is invalid, when the measured distance is greater or equal a predefined value. In both sensor models this can easily be checked. To control which sensor module is used and which message format to expect, the new parameter *range_sensor_type* was introduced. Furthermore, the parameters to influence the laser model were renamed from *laser_X* to *range_sensor_X* to express the possibility of other range sensor models.

Although, as before a static map is used for localisation and thus, nothing should change, the usage of probabilistic maps introduces a problem. As already mentioned, localisation is done with static obstacles. Thus, the internal map in AMCL only consists of three states, and each cell gets the following values assigned:

- -1: the cell is known to be empty (occupancy probability is 0)
- +1: the cell is known to be occupied (occupancy probability is 100)
- 0: the cell is neither empty nor occupied

The problem is, that due to implementation reasons, static obstacles in probabilistic maps from [11] can have an occupancy probability unequal to 100. Instead, even cells with a very high occupancy probability have to be treated as cells containing static obstacles. This is also true, when the dynamic part is removed from the probabilistic map to obtain a static one. The same problem arises with empty cells. Thus, cells with a very low occupancy probability have to be treated as empty cells too. To account for this problem, two new parameters were introduced to set the thresholds when a cell is accepted to be occupied (*occupied_threshold*) or empty (*empty_threshold*) respectively. When the occupancy probability is bigger or equal to *occupied_threshold*, the cell in the internal map is set to +1, when it is smaller or equal to *empty_threshold*, it is set to -1. As before, everything in between is set to 0.

Calibration

Unfortunately, the implemented particle filter and models have a lot of parameters, which were required to be tuned to compensate for the poor odometry and the shortcomings of the sonar sensors. The best parameter values were found by extensive experiments. The most important overall particle filter parameters are:

- *min_particles (int):* The minimum number of particles.
- *max_particles (int):* The maximum number of particles.
- *kld_err (double):* The maximum error between the estimated and true distribution.
- *kld_z (double): "Upper standard normal quantile for* $(1 - p)$*, where* p *is the probability that the error on the estimated distribution will be less than kld_err"* [56].

The parameters for the measurement model are the former parameters for the laser model and were renamed. These parameters are:

- *range_sensor_type (string):* Which sensor model is used. Valid values are *laser* and *sonar*.
- *range_sensor_model_type (string):* Which range measurement model is used.
- *range_sensor_topic (string):* The topic on which the measurements are published.
- *range_sensor_max_beams (int):* How many evenly-spaced beams in each scan to be used when updating the filter [56].
- *range_sensor_z_hit (double):* The weight of the "correct range with local measurement noise" component of the sensor model (consult [81] for more information).
- *range_sensor_z_short (double):* The weight of the "unexpected objects" component of the sensor model (consult [81] for more information).
- *range_sensor_z_max (double):* The weight of the "Failures" component of the sensor model (consult [81] for more information).
- *range_sensor_z_rand (double):* The weight of the "random measurement" component of the sensor model (consult [81] for more information).
- *range_sensor_sigma_hit (double):* Standard deviation, used in the 'correct range with local measurement noise" component of the sensor model (consult [81] for more information).
- *range_sensor_lambda_short (double):* Exponential decay parameter used in the "unexpected objects" component of the sensor model (consult [81] for more information).
- *range_sensor_max_range (double):* The maximum range of the sensor. Measured ranges greater equal this parameter are invalid.

The odometry model has the following parameters:

- *odom_model_type (String):* Which odometry model is used.
- *odom_alpha1 (double): "The expected noise in odometry's rotation estimate from the rotational component of the robot's motion"* [56].
- *odom_alpha2 (double): "The expected noise in odometry's rotation estimate from translational component of the robot's motion"* [56].
- *odom_alpha3 (double): "The expected noise in odometry's translation estimate from the translational component of the robot's motion"* [56].
- *odom_alpha4 (double): "The expected noise in odometry's translation estimate from the rotational component of the robot's motion"* [56].

At last, the previously mentioned thresholds for converting the received map to a reduced internal map representation can be set with the following parameters:

- *occupied_threshold (int):* Cells with cost value greater or equal this parameter are treated as obstacles.
- *empty_threshold (int):* Cells with cost value less or equal this parameter are treated as empty.

The values of the parameters can be found in Appendix A.1.1.

5.6.2 Costmap

Move_base uses the *costmap_2d* module for the local- and global_costmap. *Costmap_2d* provides a two-dimensional occupancy grid of the environment, on which planners can perform path planning. Each cell has one of 256 cost values, which are determined by previously created maps and sensor observations. Thus, the cost value of a cell can change when new observations are made. This allows to insert and delete observed obstacles. Furthermore, obstacles get inflated circularly by a user defined inflation radius. In this thesis, inflation is the process of increasing the size of an obstacle, by modifying the cell costs of cells in the vicinity of the obstacle. A cell with a cell cost of 255 is unknown, a cell cost of 254 represents an obstacle and and cells with cost 0 are empty cells. The cell values between 0 and 254 are used for obstacle inflation. More information about inflation can be found in Section 5.6.2. Furthermore, the occupancy grid gets published, which is mainly used for debugging and information purposes.

The cell manipulation is fully adaptable, since the logic which determines the cell costs is contained in plug-ins called layers. So for example, one layer is responsible to handle maps, and another one is capable of the previously mentioned inflation. It is easy to add new layers to extend the functionality of the costmap or adapt previously existing layers.

The costmap gets updated with a user defined frequency, which is configured via the *update_frequency* parameter. At every update cycle, each layer gets called to modify a passed

occupancy grid, which allows the layer to modify the cell costs. Since every layer gets the modified occupancy grid of the previous layers, the layer call order is important. The call order is influenced via the *plugins* parameter.

The costmap publishes/subscribes to the following topics, excluding the single layers:

- *costmap (nav_msgs/OccupancyGrid):* The occupancy grid gets published on this topic. The exact topic is either */move_base_node/global_costmap/costmap* or */move_base_node/-local_costmap/costmap*.
- */tf (tf/tfMessage):* The costmap subscribes to this topic to get a transformation link between the values of parameter *global_frame* and *robot_base_frame*. In this thesis, this links are provided by the AMCL module and the rosaria module respectively.

The parameters used by the costmap_2d module in this thesis, excluding parameters of the single layers, are:

- *global_frame (string):* The frame in which the costmap is operating. In the following module description, the term *global frame* always refers to the value of this parameter.
- *robot_base_frame (string):* The frame of the robot.
- *transform_tolerance (double):* The maximum allowed delay in transform (tf) data that is tolerable in seconds.
- *update_frequency (double):* Specifies the map update frequency, as explained before.
- *publish_frequency (double):* The frequency of publishing the map to visualize it with modules like rviz.
- *rolling_window (boolean):* Specifies if the rolling window version should be used. The rolling window version keeps the robot always in the centre of the costmap. Regions which were observed previously, but are out of bounds of the costmap, because the robot moved onwards, are dropped. This is useful when only information about the vicinity of the robot is required.
- *width (int):* The width of the occupancy grid in meters. This parameter can be overwritten by the probabilistic map layer.
- *height (int):* The height of the occupancy grid in meters. This parameter can be overwritten by the probabilistic map layer.
- *resolution (double):* The resolution of the occupancy grid in meters per cell. This parameter can be overwritten by a layer.
- *origin_x (double):* The origin of the occupancy grid in the global frame in meters on the x-axis. This parameter can be overwritten by the probabilistic map layer.
- *origin_y (double):* The origin of the occupancy grid in the global frame in meters on the y-axis. This parameter can be overwritten by the probabilistic map layer.

- *footprint (string):* The footprint of the robot, which is needed for path planning and inflation.

- *plugins (list):* Specifies which layers (name and type) are used. The order of this list is used as the call order for the layers.

When information of a point has to be extracted out of the costmap, two steps have to be performed. First, when the point is not already in the global frame of the costmap, it has to be transformed to this frame. Then, the cell in the costmap, corresponding to the point in the global frame has to be computed. In the subsequent algorithms, variables ending with _global are points in the global frame. Similarly, variables ending with _map are map cells.

Probabilistic_map_layer

The *probabilistic_map_layer* is responsible to insert the information of a probabilistic map into the occupancy grid. This layer was developed in this thesis to incorporate the dynamic information of probabilistic maps developed in [11]. It is based on the previously existing *static_layer* in the costmap_2d module, which is used to insert static maps into the occupancy grid. It implements the probabilistic path planning strategy presented in Section 4.1.

In the first step, the module subscribes to the topic on which the probabilistic map gets published from the *probabilistic map server*. The topic can be changed via the *probabilistic_map_topic* parameter. When the map is received, a temporary occupancy grid gets resized to match the size and the resolution of the map. Hence, afterwards the temporary occupancy grid and the probabilistic map have exactly the same number of cells. Then, the cell cost of each temporary occupancy grid cell is determined by the *occupancy probability* and the *time until change* of its corresponding cell in the probabilistic map with Equation 4.2 from Section 4.1.4. With the help of the two new ROS parameters *forbidden_probability* and *forbidden_time_until_change*, the corresponding values for Equation 4.2 can be set.

Algorithm 9 shows the pseudo code, which computes the cell cost for one temporary occupancy grid map cell. Lines 1-3 handle the case, when a map cell has an unknown cost value. The cell cost of the temporary occupancy grid cell is set to NO_INFORMATION (255 in costmap_2d). Line 4 computes the cell cost with Equation 4.2. Since LETHAL_OBSTACLE determines the maximal possible cost value, it is used as parameter n. Furthermore, min_cost is set to 0, since the shortest path algorithm used by the global_planner plug-in, already incorporates the path length into its search for the shortest path.

Algorithm 9 ComputeCellCost($occupancy\_probability$, $time\_until\_change$)

1: **if** $occupancy\_probability$ = unknown_cost_value **then**
2: **return** NO_INFORMATION
3: **end if**
4: **return** $min\{$LETHAL_OBSTACLE, $max\{$LETHAL_OBSTACLE$/forbidden_probability \cdot occupancy_probability +$ LETHAL_OBSTACLE$/forbidden_time_until_change \cdot 100/dynamic_time_max \cdot time_until_change, 0\}\}$

This algorithm is executed for every probabilistic map cell. When the probabilistic_map_layer gets called to perform its function, the temporary occupancy grid is copied into the passed occupancy grid. The probabilistic_map_layer subscribes to the following topic:

- */probabilistic_map (dynamic_mapping/DynamicGrid):* The probabilistic_map_layer subscribes to this topic to receive the probabilistic map from the probabilistic map server.

The probabilistic_map_layer has the following parameters:

- *probabilistic_map_topic (string):* The topic where the probabilistic map is published.
- *forbidden_probability (double):* Cells with an occupancy probability greater or equal this parameter are treated as obstacles.
- *forbidden_time_until_change (double):* Cells with a time_until_change greater or equal this parameter are treated as obstacles.
- *use_maximum (boolean):* Specifies how the temporary occupancy grid is copied into the passed one. When this parameter is set to true, the maximum of the temporary and the passed cell cost is used as new cell cost. When the parameter is false, the cell cost is overwritten with the cell cost of the temporary map.
- *unknown_cost_value (int):* The value of an unknown cell in the probabilistic map.

Obstacle_layer

The *obstacle_layer* changes the cell costs of the occupancy grid based on sensor inputs. Thus, it performs environment observation and models the environment with a two dimensional occupancy grid. The incoming measurement messages are stored in a measurement buffer until the *obstacle_layer* is called to perform its function. Then, based on the previously buffered measurement messages, cells get cleared (set cost value to 0) or marked as containing an obstacle (set cost value to 254). Afterwards, it is waited for the next call, during which incoming measurement messages are buffered again. The *obstacle_layer* is capable to handle multiple sensors in parallel.

The original implementation expects that the measurements are from laser sensors in either cartesian- or polar coordinates. Since the following steps require a point cloud, measurements expressed in polar coordinates are transformed to cartesian coordinates, which in fact are a point cloud. After this unification, the measurement can be stored in the measurement buffer.

The functionality is nearly the same as explained in Section 4.2.2. When a laser sensor detects an obstacle three meters in front of the robot, no obstacle can be present between the robot and the detected obstacle. Thus, raytracing the laser beams with the help of the Bresenham-algorithm is done to clear cells in the occupancy grid. Thus, the cell cost of each occupancy grid cell which lies on a ray between the sensor origin and a measured point is set to FREE_SPACE.

Special care has to be taken when a measured point is outside the occupancy grid. If this is the case, the ray is computed between the sensor origin and an endpoint, which lies on the border of the occupancy grid and on a straight line between the sensor origin and the measured

point. This case is shown in Figure 5.6. Since the measured point is outside of the occupancy grid, a helper point on the line between the sensor and the obstacle (dashed line in Figure 5.6) at the border of the map is computed. Then, the solid line between the cell containing the sensor and the cell containing the helper point is cleared with the help of the Bresenham-algorithm.

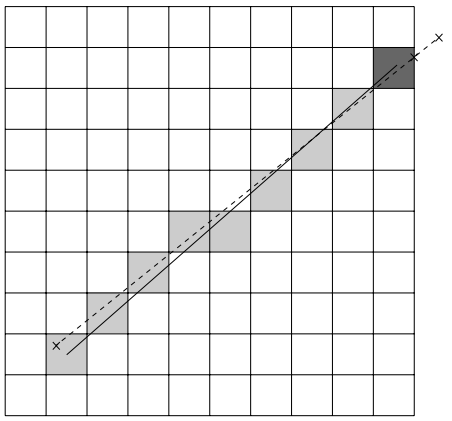


Figure 5.6: Raytracing for a measurement, which is outside of the occupancy grid

After the raytracing step, the measured points, or the helper points in case the measured point be outside of the grid, can be inserted into the occupancy grid. This is done by setting the cell cost value to LETHAL_OBSTACLE for each cell which corresponds to a measured point.

This thesis adapted the obstacle_layer to do environment observation with sonar sensors. Thus, the concepts presented in Section 4.2.3 were implementend. Algorithm 10 shows the implemented algorithm, which is basically Algorithm 6 from Section 4.2.3, with changes due to ROS.

The first difference are the arguments of the algorithm. Instead of a single distance delivered by a sonar sensor, a data structure containing one sonar measurement message M and a data structure $sonar$, containing the positions of every sonar sensor in the *front_sonar frame* is handed to the algorithm as parameter. A sonar measurement message in ROS consists of multiple points (not ranges), one for each available sonar sensor. These points are in the *front_sonar frame*. Algorithm 10 is executed for every buffered measurement message. Furthermore, note that the position of the sonar sensors in the *front_sonar frame* are static and can be precomputed once. Again, in the subsequent explanations, “left”, “right” and “front” always refer to the point of view of the sonar sensor.

Since multiple points are in M, for each point, cells are cleared and an obstacle is inserted. Since the measurement is delivered as point, it is converted in line 2 to a distance. Lines 6-15 compute the left and right cell containing the cone border points as in Section 4.2.3. The difference is, that the points $border\_l$ and $border\_r$ have to be transformed from the *front_sonar frame* to the points $border\_l\_global$ and $border\_r\_global$ in the *global frame* with the help of the *tf* module (Line 8 and 13). Furthermore, since $border\_l\_global$ and $border\_r\_global$ might be outside of the occupancy grid, they get scaled with an already existing function (line 9 and 14). A point outside of the occupancy grid is set to the border of the occupancy grid on a line

Algorithm 10 ProcessMeasurement(M, $sonar$)

1: **for all** $(x_i, y_i) \in M$ **do**
2: $dist \leftarrow \sqrt{(x_i - sonar.i.x)^2 + (y_i - sonar.i.y)^2}$
3: **if** $dist \geq$ MAX_DIST **then**
4: **continue**
5: **end if**
6: $border\_l.x \leftarrow dist \cdot cos(sonar.i.\theta + \phi) + sonar.i.x$
7: $border\_l.y \leftarrow dist \cdot sin(sonar.i.\theta + \phi) + sonar.i.y$
8: transform $border\_l$ from front_sonar frame to $border\_l\_global$ in the global frame
9: scale $border\_l\_global$
10: compute cell $border\_l\_cell$ out of $border\_l\_global$
11: $border\_r.x \leftarrow dist \cdot cos(sonar.i.\theta - \phi) + sonar.i.x$
12: $border\_r.y \leftarrow dist \cdot sin(sonar.i.\theta - \phi) + sonar.i.y$
13: transform $border\_r$ from front_sonar frame to $border\_r\_global$ in the global frame
14: scale $border\_l\_global$
15: compute cell $border\_r\_cell$ out of $border\_r\_global$
16: clearCells($border\_l\_cell$, $border\_r\_cell$, $sonar.i$)
17: doBresenham($border\_l\_cell$, $border\_r\_cell$, LETHAL_OBSTACLE)
18: **end for**

between the sonar sensor and the original point as shown in Figure 5.6.

Afterwards, the cells $border\_l\_cell$ and $border\_r\_cell$, containing the border points can be determined (Line 10 and 15). After the cells are cleared with function clearCells, which will be presented shortly, a straight line between $border\_l\_cell$ and $border\_r\_cell$ is computed, with the help of the Bresenham-algorithm. The cell cost value for each cell on the line is set to LETHAL_OBSTACLE (line 17). Thus, the obstacle is inserted as line between the cone border points.

Algorithm 11 implements Algorithm 8 from Section 4.2.3. The only difference is, that the position of the i-th sonar sensor in the *global frame* of the occupancy grid is determined with the help of the tf package (line 1). The sonar positions $sonar\_global$ in the global frame are not necessarily static, in contrasts to the *front_sonar frame*. Thus, no precomputation can be made. Lines 3-31 are Algorithm 8 from Section 4.2.3.

In this thesis, the obstacle_layer subscribes the following topics:

- */sonar (sensor_msgs/PointCloud):* The obstacle_layer subscribes to this topic, to receive the calibrated sonar measurements messages from the sonar calibration module.
- */tf (tf/tfMessage):* To receive the transformation tree, to transform various points to the global frame.

The original obstacle_layer is capable of processing measurements of multiple sensors operating concurrently. Thus, a future use-case could be to use laser- and sonar sensors concurrently to update the costmap. As a consequence, the adaptations explained previously, have to be included into the existing obstacle_layer. The obstacle_layer has just one list parameter *obser-*

Algorithm 11 clearCells($border\_l\_cell$, $border\_r\_cell$, $sonar.i$)

```
1: transform sonar.i from front_sonar frame to sonar_global in the global frame
2: compute cell sonar_cell out of sonar_global
3: leftCells ← doBresenham(sonar_cell, border_l_cell)
4: rightCells ← doBresenham(sonar_cell, border_r_cell)
5: frontCells ← doBresenham(border_l_cell, border_r_cell)
6: if border_l_cell.x < sonar_cell.x and border_r_cell.x < sonar_cell.x then
7:     if border_l_cell.x < border_r_cell.x then                          ▷ 1.1
8:         begin_cells ← leftCells
9:         end_cells ← merge(rightCells, frontCells)
10:    else                                                              ▷ 1.2
11:        begin_cells ← merge(leftCells, frontCells)
12:        end_cells ← rightCells
13:    end if
14: else if border_l_cell.x > sonar_cell.x and border_r_cell.x > sonar_cell.x then
15:    if border_l_cell.x < border_r_cell.x then                          ▷ 2.1
16:        begin_cells ← rightCells
17:        end_cells ← merge(leftCells, frontCells)
18:    else                                                              ▷ 2.2
19:        begin_cells ← merge(rightCells, frontCells)
20:        end_cells ← leftCells
21:    end if
22: else
23:    if border_l_cell.x < border_r_cell.x then                          ▷ 3.1
24:        begin_cells ← merge(leftCells, rightCells)
25:        end_cells ← frontCells
26:    else                                                              ▷ 3.2
27:        begin_cells ← frontCells
28:        end_cells ← merge(leftCells, rightCells)
29:    end if
30: end if
31: clearTriangle(begin_cells, end_cells)
```

vation_sources, which specifies the used sensors. The sub-parameters of *observation_sources* are:

- *topic (string):* The topic on which the sensor publishes the measurement messages.
- *data_type (string):* The type of messages published on *topic*. Legal values are *PointCloud*, *PointCloud2* and *LaserScan*.
- *sensor_frame (string):* The frame in which the measurement is expressed.

- *sensor_semantic (string):* Controls the raytrace and obstacle insertion. With *Narrow-Beam*, the original implementations are used, with *BroadBeam*, the new implementations are used.
- *cone_angle (double):* The angle of the cone from the centre to one sonar border.
- *observation_persistance (double):* How long a measurement should be buffered in seconds. A value of 0 stores only the latest measurement of this sensor.
- *marking (boolean):* When set to true, the measurements from this sensor are used to insert obstacles.
- *clearing (boolean):* When set to true, the measurements from this sensor are used to clear obstacles.

Probabilistic_inflation_layer

When the *probabilistic_inflation_layer* is called, it circularly inflates cells which got a cost value set by previous costmap layers. Thus, *probabilistic_inflation_layer* has to be the last layer in the call sequence to perform its function properly. The *probabilistic_inflation_layer* is based on the previously existing *inflation_layer* in the costmap_2d module, which only inflates obstacles (cells with cost value 254). The inflation of an obstacle works by setting proximity cells to a lower cell cost depending on the distance to the cell containing the obstacle. The cost of cells, which are farther away from the obstacle cell are set to a lower value, than cells in closer vicinity to the obstacle cell. Thus, inflation can be seen as a fadeout of cost values. Figure 5.7 shows the idea of obstacle inflation, where the Manhattan distance is used as distance metric.

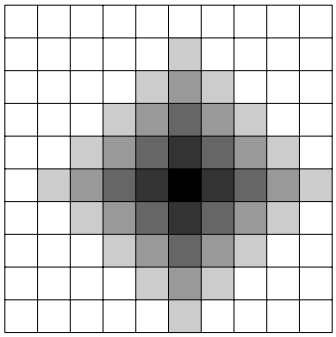


Figure 5.7: Inflating an obstacle cell

The inflation in both inflation_layers is influenced by three values. Cells which are not farther away from an obstacle then the *inscribed radius* of the robot are set to *INSCRIBED_INFLATED_OBSTACLE* (253). The *inscribed radius* is the radius of the biggest circle which fully fits inside of the robot's footprint (see Figure 5.8 for an example with a rectangular robot).

Thus, after inflating cells with obstacles, the occupancy grid can be seen as an almost explicitly constructed two-dimensional configuration space via a grid cell decomposition (see Chapter 2). The assumption is, that the robot geometry is a circle with the inscribed radius. Since the

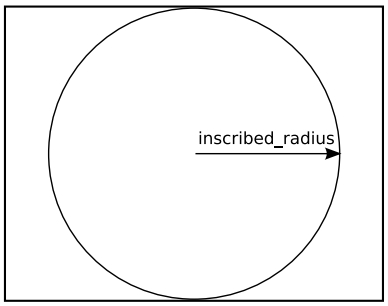


Figure 5.8: The inscribed radius of a rectangular robot

robot can rotate in place, the rotation of the robot can be ignored for configurations, which yields a two-dimensional configuration space. Unfortunately, the robot is not circular and depending on the rotation it might nevertheless collide with an obstacle, even if it is in a cell with a cost value smaller than 253. Hence, it is only an almost explicitly constructed two-dimensional configuration space. The usage of the inscribed radius for explicitly constructing the configuration space can be seen as underestimation of the robot's footprint.

A overestimation of robot's footprint is achieved, when the *circumscribed radius* is used. The *circumscribed radius* is the radius of the smallest circle, which fully circumscribes the robots footprint (see Figure 5.9 for an example with a rectangular robot).

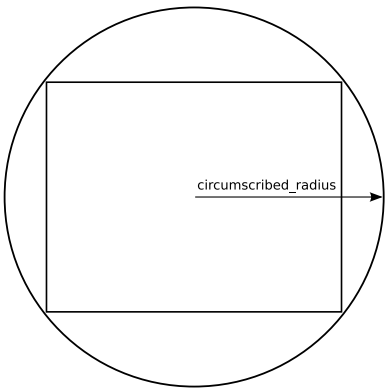


Figure 5.9: The circumscribed radius of a rectangular robot

Would the circumscribed radius be used for obstacle inflation and thus, explicitly constructing the configuration space, rotation would never cause collisions with static obstacles. The big drawback of using circumscribed radius is, that obstacles would be inflated too large. Paths which would be valid with the real robot footprint, are not found. Since this is not the case, when the inscribed radius is used, this radius is used for obstacle inflation. Furthermore, constant observation of the environment, prevent the robot to rotate into obstacles.

The two other values influencing the inflation is the maximal distance to which an obstacle is inflated (*inflation_radius*) and a weight factor how rapidly the cell costs should decline (*in-*

flation_factor). Another important property of inflation is,that it causes the robot to stay away from obstacles, since paths crossing cells with lower cell costs are preferred.

Since the dynamic areas inserted by *probabilistic_map_layer* can have arbitrary cell costs between 1 and 254 which shall be inflated too, the developement of the *probabilistic_inflation_layer* was necessary. The general idea is to insert every cell which is not empty and not unknown into a priority queue Q forming the "source" cells. Starting with the cell c in Q with the highest cell cost, the cell cost values of neighbouring cells are set. Additionally c gets removed from Q and the neighbouring cells are inserted into Q. This is repeated until every cell has got its appropriate cell value. Thus, a chain reaction starting from the source cells is launched. A record stored in Q has the following fields:

1. *pos:* The position of the cell in the occupancy grid, used to calculate the cell cost of neighbouring cells.

2. *src_pos:* The position of the source cell in the occupancy grid which caused the cell to be inserted into Q.

3. *cost:* The cost of the cell. This field is used to sort Q descendingly.

4. *src_cost:* The cost of the source cell.

The cell cost of one cell gets calculated by the function *enqueue* shown in Algorithm 12. The arguments pos and src_pos are the coordinates of the cells in the occupancy grid for which the cost value should be set (current cell), and the coordinates of the source cell which should be inflated respectively. The cell cost of this source cell is stored in argument src_cost.

To prevent endless loops, a global $seen$ array with the same size as the occupancy grid is used, to store the information if a cell was already visited. If this cell was already seen in this run, no further action is taken for this cell (lines 1-3). Then, the distance $dist$ in meters between the current cell and the source cell is determined (line 4). When the distance is bigger than the *inflation_radius*, the source cell does not influence the current cell and no further actions have to be taken. The idea to inflate obstacles with the help of the *inscribed radius* is in the *probabilistic_map_layer* the same as in the previously existing *inflation_layer*. Thus, when the source cell contains an obstacle and the distance to the current cell is smaller than the $inscribed\_radius$, new_cost is *INSCRIBED_INFLATED_OBSTACLE* (lines 8-10). Otherwise, new_cost is determined by the source cell's cost, the distance minus the inscribed_radius and the distance weight $inflation\_factor$ (line 12). If the source cell does not contain an obstacle, new_cost is determined like in line 12, but the distance is not decreased by the inscribed_radius (line 15). In line 17, the current cost of the current cell is stored in old_cost. If old_cost is smaller than new_cost, the current cell costs are updated and the cell is marked as derived (lines 19 and 20). Otherwise, new_cost is set to old_cost (line 22). Then, the cell is marked as seen (line 24) and inserted into Q.

Algorithm 13 manages the queue Q and it is executed when the *probabilistic_map_layer* gets called. Most parts of this algorithm are copied from the previously existing *inflation_layer*. It gets the coordinates of a rectangle segment inside the occupancy grid as arguments. The previous layers set the costs of cells only inside this rectangular sector.

Algorithm 12 enqueue(pos, src_pos, src_cost)

1: **if** $seen[pos.x][pos.y]$ = false **then**
2: **return**
3: **end if**
4: $dist \leftarrow \sqrt{(src\_pos.x - pos.x)^2 + (src\_pos.y - pos.y)^2} \cdot resolution$
5: **if** $dist > inflation\_radius$ **then**
6: **return**
7: **end if**
8: **if** src_cost = LETHAL_OBSTACLE **then**
9: **if** $dist \leq inscribed\_radius$ **then**
10: $new\_cost \leftarrow$ INSCRIBED_INFLATED_OBSTACLE
11: **else**
12: $new\_cost \leftarrow src\_cost - (dist - inscribed\_radius) \cdot inflation\_factor$
13: **end if**
14: **else**
15: $new\_cost \leftarrow src\_cost - dist \cdot inflation\_factor$
16: **end if**
17: $old\_cost \leftarrow occupancy\_grid[pos.x][pos.y]$
18: **if** $old\_cost < new\_cost$ **then**
19: $occupancy\_grid[pos\_x][pos\_y] \leftarrow new\_cost$
20: $derived[pos.x][pos.y] \leftarrow$ true
21: **else**
22: $new\_cost \leftarrow old\_cost$
23: **end if**
24: $seen[pos.x][pos.y] \leftarrow$ true
25: Insert $\{pos, src\_pos, src\_cost, new\_cost\}$ into Q

At line 1, every cell in $seen$ is set to false, indicating a new run. In lines 2-9, cells which are not empty, not unknown and which have not been set by the costmap in a previous run, are inserted into the queue Q, forming the "source" cells. After this initialisation, the record with the highest $cost$ is removed from Q. Then, in lines 10-25 the function *enqueue* gets called for each of the four possible neighbouring cells (top, right, bottom, left). This is repeated until Q is empty.

The probabilistic_inflation_layer has the following parameters:

- *inflation_radius (double):* The maximum distance in metres, to which an obstacle is inflated.

- *inflation_factor (double):* How fast the cell costs should decline with increasing distance to the source cell.

Algorithm 13 cellInflation(min_x, min_x, max_x, max_y)

set every field in $seen$ array to false
for $x = min\_x$ to max_x **do**
    for $y = min\_y$ to max_y **do**
        $cost \leftarrow occupancy\_grid.[x][y]$
        if $cost \neq$ FREE_SPACE and $cost \neq$ NO_INFORMATION and $derived[x][y]$ = false **then**
            Insert $\{(x, y), (x, y), cost, cost\}$ into Q
        end if
    end for
end for
while $Q \neq$ **do**
    $\{pos, src\_pos, cost, src\_cost\} \leftarrow$ element from Q with maximal $cost$
    Remove $\{pos, src\_pos, cost, src\_cost\}$ from Q
    if $x > 0$ **then**
        enqueue($(pos.x - 1, pos.y)$, src_pos, src_cost)
    end if
    if $y > 0$ **then**
        enqueue($(pos.x, pos.y - 1)$, src_pos, src_cost)
    end if
    if $x < size\_x - 1$ **then**
        enqueue($(pos.x + 1, pos.y)$, src_pos, src_cost)
    end if
    if $y < size\_y - 1$ **then**
        enqueue($(pos.x, pos.y + 1)$, src_pos, src_cost)
    end if
end while

5.6.3 Move_base

The *move_base* module [59] implements the path planning algorithm. It follows a two stage approach as presented in Section 3.3.3. When a new goal is published on the */move_base_simple/goal* topic, *move_base* uses the *global_planner* plug-in to compute a global plan. To fulfil this task, the *global_planner* uses the *global_costmap*, which represents the environment. When the *global_planner* plug-in finds a valid global plan, it gets forwarded to the *local_planner*. Otherwise, the recovery mode is triggered.

The *local_planner* is an *obstacle avoider* (see Section 3.3.3), which is responsible to follow the global plan and to avoid obstacles. Thus, it creates velocity commands which are published to actuate the robot until the robot gets within the tolerance to the goal or no valid velocity command can be created. In this case, the robot gets stopped and the *global_planner* is activated again. When a global plan is found again, it is forwarded to the *local_planner*. This procedure is repeated until the *local_planner* is able to create a trajectory or the last valid velocity command is longer than a user defined time ago. In this case, the recovery mode is triggered.

In the recovery mode the *recovery_behaviours* are executed sequentially. *Recovery_behaviours* influence path planning by changing execution or resetting data structures, when no path can be found. After every execution of a *recovery_behaviour*, the *global_planner* tries to find a path which gets forwarded to the *local_planner*. When both planners find a plan, the recovery mode is exited and normal operation continues. Otherwise, the next *recovery_behaviour* is executed. When a planner fails after the last *recovery_behaviour* was executed, the path planning is terminated and an error is displayed to inform the user that the goal can not be reached.

Move_base publishes/subscribes to the following topics:

- */move_base_simple/goal (geometry_msgs/PoseStamped):* Move_base subscribes to this topic to receive goal locations.
- */cmd_vel (geometry_msgs/Twist):* On this topic, move_base publishes velocity commands to actuate the robot.

In this thesis, the following parameters are used for the move_base module (see Appendix A.1.2 for the concrete values):

- *base_global_planner (string):* The plug-in, which should be used as global_planner.
- *base_local_planner (string):* The plug-in, which should be used as local_planner.
- *recovery_behaviors (list):* The list of recovery behaviours which should be used. The ordering of this parameter is the ordering in which the recovery behaviours get executed.
- *controller_frequency (double):* The execution frequency in Hz of the local_planner.
- *controller_patience (double):* How long move_base will wait in seconds for a valid trajectory from the local_planner, before recovery mode is entered.
- *conservative_reset_dist (double):* The radius around the robot in meters, which is used to clear the costmap at the first two recovery steps.

Global_costmap

The *global_costmap* is one instance of the *costmap_2d* module and is used by the global planner. Figure 5.10(a) shows the used layers of the *global_costmap*. Since the global planner needs information about obstacles and dynamic areas contained in a probabilistic map, the first layer of the *global_costmap* is the previously described *probabilistic_map_layer*. Because the global planner should be able to plan paths involving the observed environment, the second layer used is the *obstacle_layer*. The *probabilistic_inflation_layer* completes the layers used in the global_costmap. Since the *probabilistic_map_layer* is used, it is obvious that the *global_frame* of the *global_costmap* is set to */map*. The *robot_base_frame* is set to *base_link*. The values of the *costamp_2d* parameters for the *global_costmap* can be found in Appendix A.1.2.

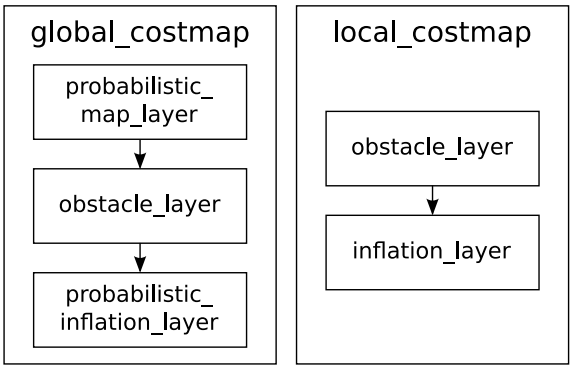


Figure 5.10: Layers of the global_costmap (a) and the local_costmap (b)

Local_costmap

The local_planner uses a second, independent instance of the *costmap_2d* module, called the *local_costmap*. Figure 5.10(b) shows the used layers of the *local_costmap*. As layers, the *obstacle_layer* and the *inflation_layer* are used. Since the local_planner is designed to create trajectories in close vicinity of the robot, the *rolling_window* parameter is set to true. Hence, the robot always stays in the centre of the costmap, and the costmap size can be reduced to 5 x 5 meters.

This is also the reason why the *probabilistic_map_layer* is not used. It would increase the size of the costmap to the size of the map, which would significantly increase the runtime of one local_planner cycle.

Since the costmap is centred around the robot, the *global_frame* is set to */odom*. As in the *global_costmap*, the *robot_base_frame* is set to *base_link*. The values of the costmap_2d parameters for the global_costmap can be found in Appendix A.1.2.

Global_planner

This thesis uses *navfn* [60] as global_planner plug-in to create global paths. Based on the global_costmap it creates a potential field. Then the Dijkstra algorithm and gradient descent is used to compute a global path from the current position of the robot to the goal location.

No changes were made to navfn in this thesis. Navfn publishes on the following topic:

- */move_base/NavfnROS/plan (nav_msgs/Path):* The last computed plan is published on this topic.

In this thesis, one nafvn parameter is used:

- *allow_unknown (bool):* Controls whether plans, which traverse unknown space are allowed.

Local_planner

As local_planner plug-in, the *dwa_local_planner* [57] module is used. As the name implies, *dwa_local-_planner* implements the DWA obstacle avoider, presented in Section 3.3.3.

To compute one velocity command, every part of the plan created by the global_planner , which is outside of the local_costmap, is removed. This trimmed down plan is called the *global plan* in the dwa_local_planner module. To avoid confusion with the plan created by the global-_planner, in this subsubsection *global plan* refers exclusively to the trimmed down plan. The end of the *global plan* forms the local goal. For each trajectory, the robot movement gets simulated, yielding a *local plan*. In the remainder of this section, trajectory and *local plan* is used interchangeably, since one can think of a data structure, containing both information.

The DWA implementation from [57] differs in how the sampled trajectories are scored. A trajectory in dwa_local_planner is scored by the following criteria:

1. **Target Heading:** Prefers plans, where at the endpoint the robot heads to the local goal.
2. **Clearance:** Plans which move over cells with lower costs are preferred. Since obstacles are inflated, the local planner tries to stay away from them.
3. **Global Path:** Prefers plans which are on the global plan.
4. **Progress:** Plans that go towards the local goal are preferred.
5. **Forward movement:** Prefer trajectories which drive forward.
6. **Velocity:** Prefer trajectories which speed matches to the surrounding. Hence, in free areas higher velocities are preferred. In dynamic areas or near obstacles a lower velocity is preferred.

Note, that the preferred forward movement is important for the P3AT, since it has no sonar sensors at the back. Thus, driving backwards is dangerous for the P3AT, as dynamic obstacles behind the robot will not be detected. Furthermore, the velocity criterion was developed especially for this thesis. For this, the local_planner needs access to the global_costmap. The dynamic areas are stored in the probabilistic map, but the local_costmap can not use the probabilistic_map_layer to utilise the map. Thus, a minor change was made to the move_base module in this thesis, such that the local_planner plug-in has access to the global_costmap.

The idea is to calculate the average cell cost of all cells in the surrounding of the path based on the global_costmap. Based on this average value, the optimal velocity can be calculated. The robot should move with maximum velocity, when the average is zero. An average of 254 (lethal obstacle) is the other extremum, where the robot should not move at all. In between these extrema, the cell cost averages are linearly mapped to velocities between zero- and maximum velocity. The difference between this optimal velocity and the velocity of the trajectory, which should be scored, is used to compute the score.

The pseudo code for this trajectory scoring idea is shown in Algorithm 14.

In the first three lines, needed variables are initialised. The average costs of the cells surrounding the local path is approximated. For sample points on the local path, the cell costs

Algorithm 14 velocityScore($traj$)

1: $sum \leftarrow 0$
2: $cell\_cnt \leftarrow 0$
3: $i \leftarrow 1$
4: **while** $i \leq traj.size$ **do**
5: transform $traj.i$ from the global frame of the local_costmap to pos_global in the global frame of the global_costmap
6: compute global_map coordinates pos_map out of pos_global
7: **for** $x = -max\_dist$ to max_dist **do**
8: $y \leftarrow |x| - max\_dist$
9: **while** $|x| + |y| \leq max\_dist$ **do**
10: $curr.x \leftarrow pos\_map.x + x$
11: $curr.y \leftarrow pos\_map.y + y$
12: **if** $curr \in global\_occupancy\_grid$ **then**
13: $sum \leftarrow sum + global\_occupancy\_grid[curr.x][curr.y]$
14: $cell\_cnt \leftarrow cell\_cnt + 1$
15: **end if**
16: $y \leftarrow y + 1$
17: **end while**
18: **end for**
19: $i \leftarrow i + 2 \cdot max\_dist + 1$
20: **end while**
21: $avg \leftarrow sum/cell\_cnt$
22: $vel\_opt \leftarrow vel\_max \cdot (1 - avg/\text{LETHAL\_OBSTACLE})$
23: **return** $e^{weight \cdot |traj.vel - vel\_obt|} - 1$

of neighbours, which are at most max_dist away (Manhattan distance) are summed up. The maximal distance can be set via the parameter *velocity_cost_max_dist*. As step size for these sample points $s \cdot max\_dist + 1$ is used. These sums can then be used to calculate the average cell cost. The most outer loop (lines 4-20) selects the sample points on the local path. Each sample point gets transformed from the global frame of the local_costmap to the global frame of the global_costmap (line). Then, the global_costmap map coordinates can be computed (line 6). Both loops in lines 7 to 18 sum up the costs of the sample point's cell and its neighbour cells.

In line 21 the average of all this summed up cells is calculated. Line 22 computes the optimal velocity based on the average cell costs of the surrounding of the path as described above. The velocity score for the trajectory is determined by the difference between the speed of the trajectory and the calculated optimal velocity, weighted by $weight$ (line 23). The $weight$ can be set via the *velocity_cost_weight* parameter. The exponential function ensures, that the more the trajectory speed deviates from the optimal speed, the worse (higher) the score for this trajectory is.

When a large dynamic object appears directly on, or very near to the plan created by the global_planner, the dwa_local_planner has problems to go round this obstacle. This happens, because the planner tries to stay close to the plan from the global_planner. The problem is, that

the dwa_local_planner is able to create trajectories, but with no overall progress, like moving forward and backwards repeatedly. So the global_planner is not activated to plan a path around the dynamic obstacle.

To circumvent this problem, it is checked, if the robot was able to move at least a minimum distance in a given time window. If this is not the case, it is assumed that the robot is stuck and the dwa_local_planner reports that it is not able to produce a valid velocity. This activates the global_planner, which might find a path around the blocking obstacle. This is done by storing the pose of the robot at every dwa_local_planner cycle and comparing the current position with the newest pose that is older than the current time + the time window length.

The dwa_local_planner publishes/subscribes to the following topics:

- */move_base/DWAPlannerROS/global_plan (nav_msgs/Path):* The trimmed plan the planner tries to follow.
- */move_base/DWAPlannerROS/local_plan (nav_msgs/Path):* The local plan, created by the best scored trajectory on the last cycle.
- */rosaria/pose (nav_msgs/Odometry):* The topic on which the odometry information is published, to obtain the current velocity of the robot.

Since the dwa_local_planner needs the dynamics of the robot, in the thesis a lot of parameters are used to calibrate the module:

- *acc_lim_theta (double):* The maximal rotational acceleration of the robot in rad/s^2.
- *max_rot_vel (double):* The maximum absolute rotational velocity of the robot in rad/s.
- *min_rot_vel (double):* The minimum absolute rotational velocity of the robot in rad/s.
- *acc_lim_x (double):* The maximum x acceleration of the robot in m/s^2.
- *max_vel_x (double):* The maximum x velocity of the robot in m/s.
- *min_vel_x (double):* The minimum x velocity of the robot in m/s.
- *acc_lim_y (double):* The maximum y acceleration of the robot in m/s^2.
- *max_vel_y (double):* The maximum y velocity of the robot in m/s.
- *min_vel_y (double):* The minimum y velocity of the robot in m/s.
- *max_trans_vel (double):* The maximum absolute translational velocity (sum of x and y velocity) of the robot in m/s.
- *min_trans_vel (double):* The minimum absolute translational velocity of the robot in m/s.
- *stop_time_buffer (double):* How much seconds the robot must stop before hitting an obstacle, to consider a trajectory as valid.
- *sim_time (double):* How long each trajectory is simulated forward in seconds.

- *sim_granularity (double):* The step size for the simulation between points on a trajectory in meters.
- *vx_samples (integer):* The number of samples to use in the x velocity space.
- *vy_samples (integer):* The number of samples to use in the y velocity space.
- *vtheta_samples (integer):* The number of samples to use in the theta velocity space.
- *path_distance_bias (double):* How much the distance to the path should be penalised.
- *occdist_scale (double):* How much the distance to obstacles should be penalised.
- *use_velocity_cost (boolean):* Controls if the velocity cost function should be used.
- *velocity_cost_max_dist (integer):* The radius of the circles used to compute the average in cells.
- *velocity_cost_weight (integer):* How much the difference between optimal and current velocity of the trajectory should be penalised.
- *min_travel_dist (double):* The minimal distance in meters, the robot has to move in the time window, such that the robot is not considered to be stuck.
- *storage_duration (double):* The length of the time window in seconds.

The concrete values of the parameters were found by extensive experiments and can be found in Appendix A.1.2.

Recovery_behaviours

The *recovery_behaviours* are executed, when either the global- or local_planner is not able to produce a plan over a time period greater than a user defined threshold. The following routines are used in this thesis, where the presentation order is the order of execution:

1. **clear_local_costmap:** Cells in the local_costmap, which are not farther away from the robot than the value of the move_base parameter *conservative_reset_dist*, are cleared.
2. **clear_global_costmap:** Cells in the global_costmap, which are not farther away from the robot than the value of the move_base parameter *conservative_reset_dist*, are cleared.
3. **clear_everything:** Every cell of the local- and global_costmap is cleared. Additionally, the buffer of the obstacle_layers, storing the sonar measurements is emptied.
4. **deactivate_obstacle_layer:** The obstacle_layer of the global_costmap is deactivated for exactly one global_planner cycle.

The *deactivate_obstacle_layer* recovery_behaviour was developed for this thesis, to overcome a problem with the sonar sensors and narrow passages. Due to the sonar sensor cone problematic, sometimes narrow passages are perceived smaller than they are in reality. When the robot drives nearer, the true size is revealed. When this wrong perception occurs during global path planning, the narrow passage could become too tight for the robot to fit through. This is especially problematic when the narrow passage is the only open door of a room, which should be entered or exited. Thus, as last resort, the obstacle_layer of the global planner is deactivated for one planning attempt, to ignore dynamic obstacles and plan only with the help of the probabilistic map.

5.7 Rviz

The *rviz* module [69] is a graphical user interface, to display information of ROS nodes and interact with them. This thesis uses the *rviz* module for two purposes. First, to display various information from the navigation stack, like the position of the robot and the costmaps. The second purpose of the *rviz* module in this thesis is, to set the current position of the robot and the goal location.

Figure 5.11 shows a picture of rviz, with the buttons to set the current position of the robot and the goal location encircled.

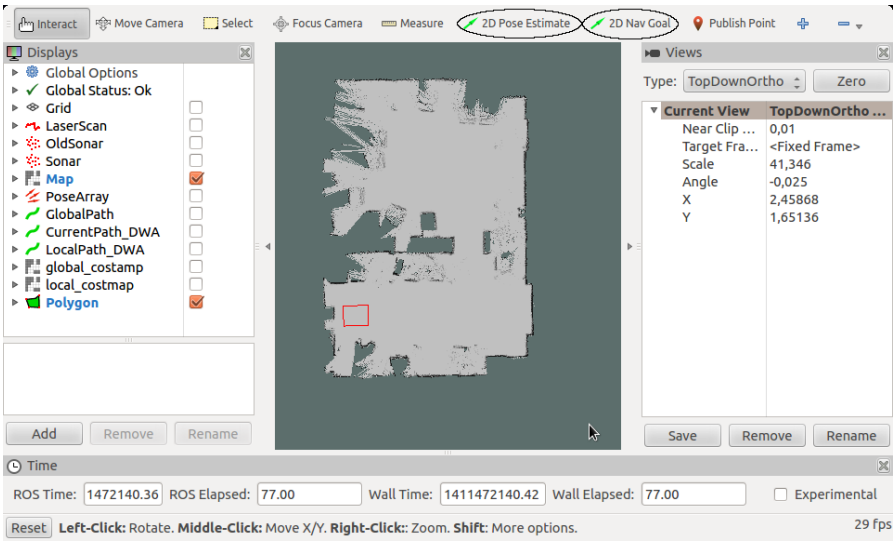


Figure 5.11: The rviz graphical user interface with encircled buttons to control the current position of the robot and the goal location

No modifications were made in the rviz module in this thesis.

5.8 Navigation control module

The *navigation control module* was developed especially for this thesis to control the initial position of the robot and to set a goal which the robot should try to reach. Via the command line, a user can select from predefined initial- and goal locations, and can thus, control the movement of the robot. The advantage of a predefined list is, that the user does not have to know the concrete coordinates of initial- and goal locations. This is especially useful in scenarios, where a robot has to reach predefined locations multiple times. An example is a service robot, which has to pick up and deliver objects from/to fixed locations.

The module gets supplied with two lists via the parameter server. The first list consists of pairs of name and initial position. The second list is similar, but instead of initial positions, goal locations are stated.

In the first step, the module reads both lists from the parameter server. Then for a name, supplied from the user via the command line, is waited. Depending whether an initial- or goal location was named, the module publishes the corresponding pose on the user defined initialpose or goal topic. Afterwards, for a new user input is waited.

The navigation control module publishes to the following topics:

- */initialpose (geometry_msgs/PoseStamped):* The topic on which the initial pose is published.
- */move_base_simple/goal (geometry_msgs/PoseStamped):* The topic on whith the goal pose is published.

The following parameters are used by the navigation control module:

- *initialposes (list):* The list of pairs consisting of name and initial position. An initial position consists of the coordinates (pose_x and pose_y) and the heading of the robot in quaternion (orientation_x, orientation_y, orientation_z and orientation_w).
- *goals (list):* The list of pairs consisting of name and goal location. The format is the same as for the initialposes list.
- *initialpose_topic (string):* The topic where a chosen initial pose should be published.
- *goal_topic (string):* The topic where a chosen goal location should be published.
- *frame (string):* The frame in which the initial- or goal location is located.

An example ROS launchfile can be found in Appendix A.2.

5.9 Safe navigation module

The sole purpose of the *safe navigation module* is to observe the bumpers of the Pioneer 3-AT robot platform published by rosaria (see Section 5.3). When a module tries to drive the robot forward but at least one forward bumper is pressed, the module commands the robot to stop. The same procedure is done with backward driving and the backward bumpers.

Basically the safe navigation module is a trimmed version of the safety module developed in [11]. That implementation used information from sonar- and laser sensors to avoid collisions and to slow down the robot. Since this task is done by the navigation stack, this features are redundant and thus, not needed. Furthermore it is computationally more efficient than the original version, since the previously unrestricted polling is capped to 100 Hz. This results in much less computational load for the processor.

The safe navigation module publishes/subscribes to the following topics in this thesis:

- */cmd_vel (geometry_msgs/Twist):* The safe navigation module subscribes to this topic to receive the velocity commands from the navigation stack.
- */rosaria/cmd_vel (geometry_msgs/Twist):* Safe velocity commands are published on this topic to actuate the robot via rosaria.
- */rosaria/bumper_state (rosaria/BumperState):* Rosaria publishes the state of the front and rear bumpers on this topic.

The safe navigation module has the following parameters:

- *cmd_vel_topic_in (string):* The topic on which other modules publish velocity commands to control the robots motors. The default value is */cmd_vel*.
- *cmd_vel_topic_out (string):* The topic on which the module which commands the robot base (in this thesis this is rosaria) is subscribed to receive velocity commands. The default value is */rosaria/cmd_vel*.
- *bumper_topic (string):* The topic on which the state of the bumpers is published. The default value is */rosaria/bumper_state*.

CHAPTER 6

Experiments and results

Various experiments have been conducted to test the implementations made in this thesis and to evaluate the effects of the improvements on the planned paths. The first experiment show, how the parameters to insert dynamic areas to the global costmap, influence the global path planning. The second experiments show the advantages of our path planning implementation, using probabilistic maps instead of static maps.

6.1 Pioneer 3-AT (P3AT)

For the experiments, the Pioneer 3-AT (P3AT) [49] from MobileRobots was used as robot platform, which has a size of 625 x 501 mm.

Figure 6.1 shows the sensors of the P3AT robot, which was used in this thesis:

1. Laser Scanner Sick LMS100
2. Camera Canon VC C50i
3. Front and rear bumpers
4. Eight front sonar sensors
5. RFID Scanner (white plate)

The laser, camera and RFID sensors are mounted extensions and no features of a basic P3AT. To stay compatible with other P3AT robots, this thesis only uses the front sonar sensors and the bumpers. The exact layout of the front sonar sensors can be seen in Figure 6.2.

The robot drives with a skid-steer, which is very similar to differential driving. Hence, the robot has two independent operating motors - one for each side. This allows the robot to determine its driving trajectory by different motor speeds on both sides. To drive a left curve, the speed of the left motor has to be lower than the right motor. When the speed of both motors are inverse, the robot rotates in place, which can be handy in narrow environments.

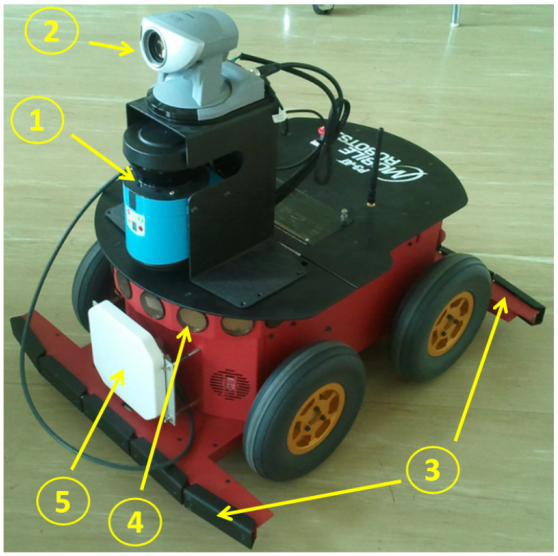


Figure 6.1: The Pioneer 3-AT (P3AT) with highlighted sensors [11]

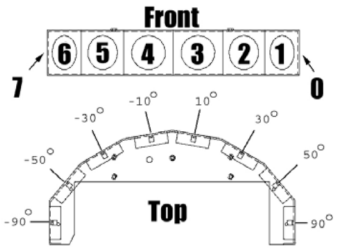


Figure 6.2: The sonar sensor layout at the Pioneer 3-AT [50]

The P3AT robot platform estimates its position by using rotary encoders at the wheels. Unfortunately, this information is poor, especially after driving many curves, since skid steering is known for its notoriously poor odometry [74]. The problem is that skidding depends highly on the underground, and since only the rotations of the wheels are tracked, the real movement of the robot might deviate from the estimated movement.

For the experiments, all the ROS modules, presented in Chapter 5, were used.

6.2 Experiment 1: determining the impact of forbidden_probability and forbidden_time_until_change

In this experiment, the effects of different values for the parameters *forbidden_probability* and *forbidden_time_until_change* are examined. *Forbidden_probability* is the weight factor for the occupancy probability of a cell in a probabilistic map from [11], used to calculate the cost value for the corresponding cell in the costmap. *Forbidden_time_until_change* is the weight

factor for the time until change of a cell in the same probabilistic map. Global plans from the global_planner are created on the global costmap, with different *forbidden_probability* and *forbidden_time_until_change* values. For each of these global paths, the path length in points and the cost is determined. Note, that the path cost is not the sum of all cell costs over which the path is travelling, but the costs created by the potential field used in the global_planner. Thus, those two metrics allow to compare the global paths, created with different values for the parameters.

Figure 6.3 shows the probabilistic map used for these experiments. During creation of the probabilistic map, a person was moving between a static box in the top right corner and a point in the bottom left corner. Thus, a dynamic corridor is formed between those two corners.

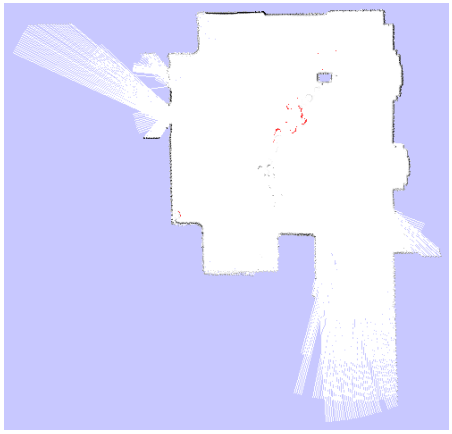


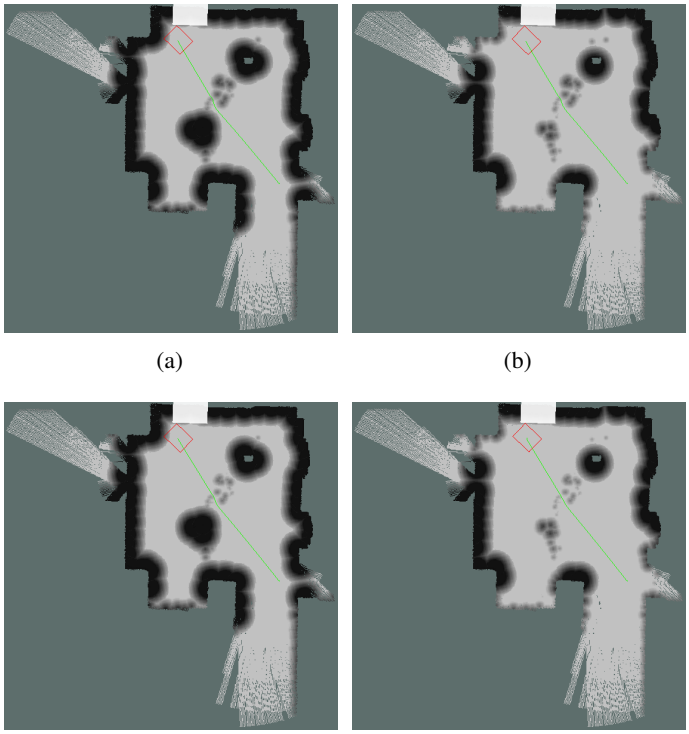
Figure 6.3: The probabilistic map used for the experiments to determine the impact of forbidden_probability and forbidden_time_until_change

To create situations with different time until change values, the time until change of the cells in the probabilistic map are multiplied with the factors 0.5, 1 or 2.

The first test is made with the multiplication factor of 0.5. Figure 6.4 shows the global plans created on costmaps with different forbidden_probability and forbidden_time_until_change values. The colour of each cell indicates the cost value of this cell in the costmap. Blue indicates a unknown cell (cost value 255). Black denotes a cell with an obstacle (cost value 254) and white cells are empty (cost value 0). The different grey gradients represents cost values between 253 and 1.

Note that the white square on the top of the picture is a display bug in rviz module, and is not present in the real costmap. Figure 6.4(a) shows the path planned with forbidden_time_until_change and forbidden_probability set to 50 (notated as 50/50). Figure 6.4(b), 6.4(c) and 6.4(d) show the paths with values 50/75, 75/50 and 75/75.

It is easy to see, how the forbidden_probability value influences the recognition of the walls as obstacles. The lower forbidden_probability is, the better the border between blue areas (unknown cells) and the white areas (empty cells) are separated by black areas (obstacles, in this case walls). Only when forbidden_probability is set to 50, the walls are recognized sufficiently as obstacles. Thus, it is advisable to set the forbidden_probability to a value lower or equal to



(a) (b)

(c) (d)

Figure 6.4: The paths created on the map with multiplication factor 0.5, where forbidden_time_until_change and forbidden_probability is set to 50 and 50 (a), 50 and 75 (b), 75 and 50 (c), 75 and 75 (d)

50 in this environment. As the time until change in the dynamic area is very low in this map, the value of forbidden_time_until_change does not influence the global path planning much. This is reflected in Table 6.1, showing the length and cost of the paths.

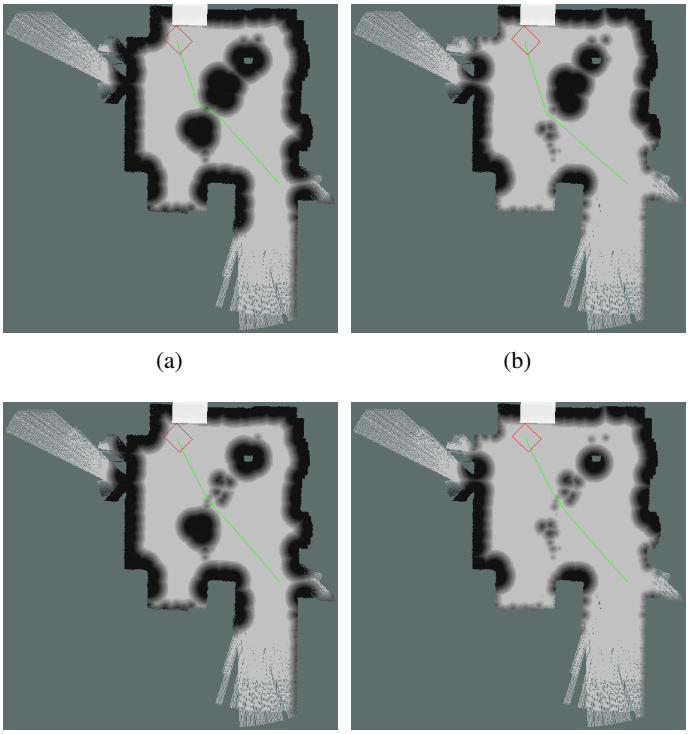
	50/50	50/75	75/50	75/75
path length	506	504	504	503
path cost	3385916	3316559	3297026	3272745

Table 6.1: Length and cost of the paths depicted in Figure 6.4

More interesting is the case, when the multiplication factor is 1, which results in the originally recorded map. Figure 6.5 shows the global paths with forbidden_time_until_change and forbidden_probability set to 50/50, 50/75, 75/50 and 75/75.

Here, forbidden_time_until_change heavily influences the path created by the global planner. Furthermore, in this setup, forbidden_probability only influences the costs of the path, but not the length of it. This can be seen in Table 6.2, showing the length and cost of the paths.

At last, Figure 6.6 shows the global paths, planned on the map with multiplication factor 2.



(a) (b)

(c) (d)

Figure 6.5: The paths created on the map with multiplication factor 1, where forbidden_time_until_change and forbidden_probability is set to 50 and 50 (a), 50 and 75 (b), 75 and 50 (c), 75 and 75 (d)

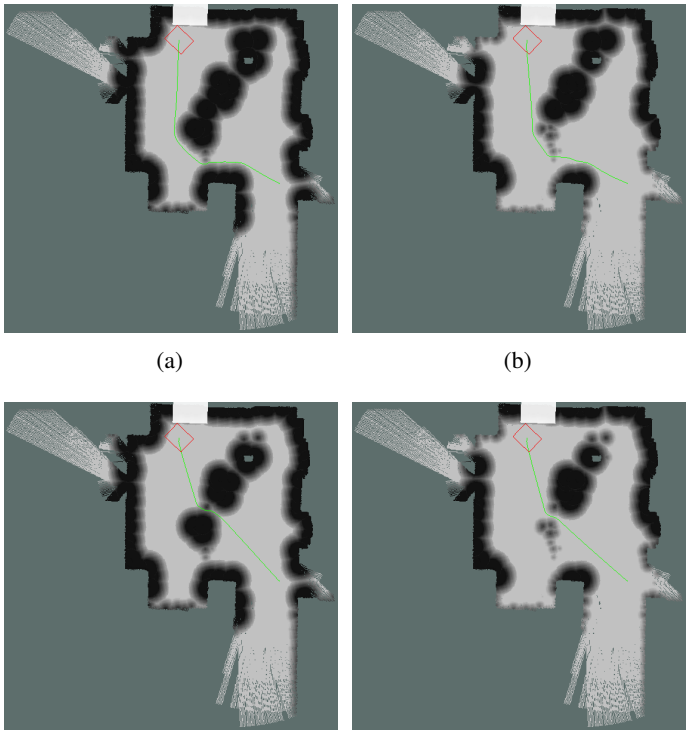
	50/50	50/75	75/50	75/75
path length	521	521	508	506
path cost	4124079	3542683	3429164	3385476

Table 6.2: Length and cost of the paths depicted in Figure 6.5

This results in very high time until change in the dynamic area. Thus, the value of 50 for forbidden_time_until_change ensures, that the global planner completely avoids the area where the person was walking. Of course, this is paid with a considerablly longer path, as shown in Table 6.3.

	50/50	50/75	75/50	75/75
path length	626	589	531	528
path cost	5087118	4491993	4412524	3634928

Table 6.3: Length and cost of the paths depicted in Figure 6.6



(a) (b)

(c) (d)

Figure 6.6: The paths created on the map with multiplication factor 2, where forbidden_time_until_change and forbidden_probability is set to 50 and 50 (a), 50 and 75 (b), 75 and 50 (c), 75 and 75 (d)

So, what are optimal values for forbidden_time_until_change and forbidden_probability? In general, no concrete answer can be given, since both values are highly dependent on the environment. When starting, stopping and replanning is very expensive, forbidden_time_until_change should be set to a high value. This allows the planner to find paths through regions with a low time until change. In areas with low time until change, few replanning attempts should be needed. The danger of such areas is, that when obstacles appear, they might stay there for a long time and the robot has to go back. When such reversals should be prevented, a low forbidden_time_until_change should be chosen. Then, the planner will avoid areas with a low time until change. Thus, should appear an obstacle on a path, found with a low forbidden_time_until_change, it should vanish soon.

The values derived from this experiment, which have been proven to be useful in practise for the used environment, are 75 for forbidden_time_until_change and 50 for forbidden_probability.

6.3 Experiment 2: travelling time of path planning using probabilistic- and static maps

The first experiment only showed how probabilistic maps influence the global path planning. The experiments in this section compare the needed time to reach the goal with probabilistic maps, in contrast to static maps. Thus, the robot was actually driving around and avoided static and dynamic obstacles in the dynamic areas. This was done ten times with the static version of the map and ten times with the probabilistic one, to gain a reasonable average of the travelling times.

6.3.1 Experiment 2a: Travel through a dynamic area

Experiment 2a is done on the map from the first experiment (see Figure 6.3), with multiplication factor 1 and forbidden_time_until_change and forbidden_probability set to 75 and 50, respectively. The robot has to drive from the left upper corner to the bottom right corner as shown by the planned paths in the first experiments, avoiding static and dynamic obstacles.

In the first test, no dynamic obstacles were placed inside the dynamic region. When the robot uses the static map, it drives straight to the goal through the dynamic region, since it has no knowledge about it. However, if the probabilistic map is used, the robot avoids the dynamic area by following the global path shown in Figure 6.5(c). Obviously, the curved path produced on the probabilistic map, is longer than the straight path planned on the static map. Thus, it is no surprise that the average travel time to reach the goal location from the start, is shorter with the static map than with the probabilistic map. The concrete values can be found in the first row of Table 6.4.

	avg. travel time stat. map	avg. travel time dyn. map
no obstacles	19,90 s	22,90 s
dyn. cylinder	36,47 s	24,65 s
static box and dynamic cylinder	48,75 s	24,73 s

Table 6.4: The average travel time of the robot in the first set of tests

In the second test, a cylindrical obstacle is placed inside the dynamic area after the global plan has been computed and the robot has started is movement, as shown in Figure 6.7. This is done to emulate a dynamic obstacle, which is unknown at the beginning.

The straight path planned on the static map is blocked by the dynamic obstacle. Thus, the robot following this path has to stop and completely change its direction to avoid the obstacle, which is time consuming. When the robot follows the path created by the global planner on the probabilistic map, the robot just has to slightly change the path to avoid the obstacle. This stems from the fact, that the obstacle hardly influences the global path created on the probabilistic map. The time difference can be seen in the second row of Table 6.4.

In the last test, a rectangular obstacle replaces the cylindrical obstacle, used in the second tests. This is done before the robot plans the global path, such that it can incorporate it into the

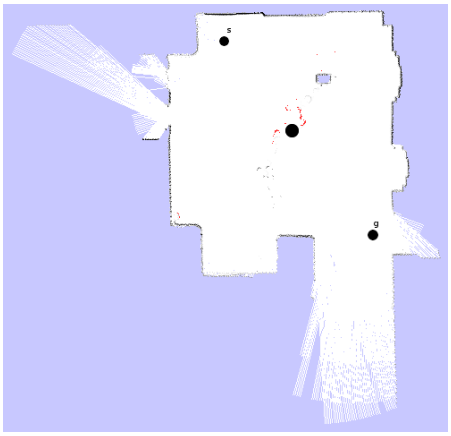


Figure 6.7: The cylindrical obstacle is placed after the robot started to move

plan. The cylindrical obstacle is placed between the rectangular obstacle and the box contained in the map, after the robot started to move (see Figure 6.8), to emulate a dynamic obstacle.

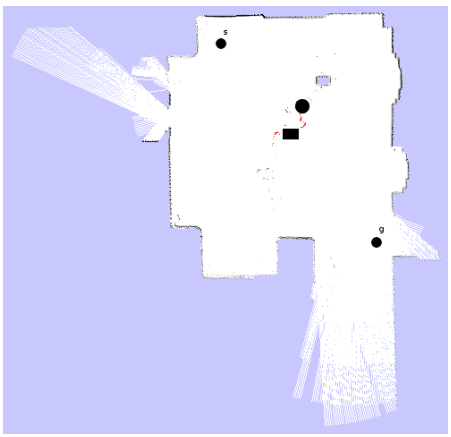


Figure 6.8: The cylindrical obstacle is placed between the rectangular obstacle (black) and the box contained in the map

The plan created on the static map runs between the rectangular obstacle and the box. When the cylindrical obstacle is placed, there is not enough space for the robot to drive between the rectangular obstacle and the box. Thus, the new shortest path is to drive at the bottom of the map to reach the goal, which is a huge time loss. A robot following the global plan created with the help of the probabilistic map is not influenced by the dynamic cylindrical obstacle. This is the case, since at the beginning the plan runs at the bottom of the map. The average time needed of both variants can be seen in the last row of Table 6.4.

6.3.2 Experiment 2b: Two ways to the goal

Experiment 2b is done on the map shown in Figure 6.9. There are two ways to reach the goal from the start. The upper way is shorter, than the lower one, but it has a dynamic region in it. In this environment, the parameters forbidden_time_until_change and forbidden_probability were both set to 50. Thus, the plan created on the static map uses the upper path, whereas the plan created on the probabilistic map uses the lower path.

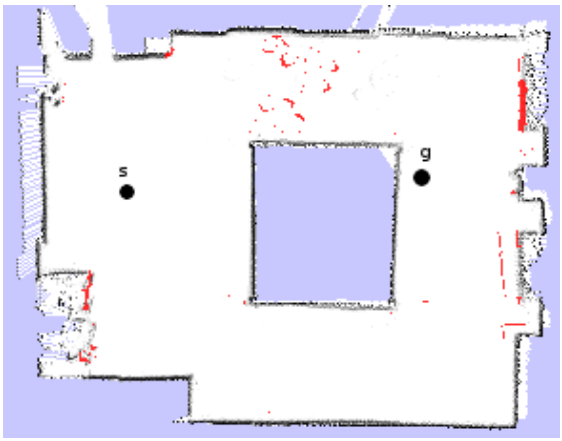


Figure 6.9: The environment used for the second set of tests

Again, in the first test no obstacles are placed inside the dynamic area. Since the plan produced on the static map uses the shorter way, it is not surprising that the average travel time of the robot following that plan is shorter, than the average travel time for the robot following the plan created with the help of the probabilistic map. The first row of Table 6.5 shows the corresponding average times.

	avg. travel time stat. map	avg. travel time dyn. map
no obstacles	35,50 s	48,75 s
obstacle at the border	77,00 s	48,75 s
obstacle in the middle	88,55 s	48,75 s

Table 6.5: The average travel time of the robot in the second set of tests

In the second test, a box is placed at the border of the dynamic area right besides the inner wall (see Figure 6.10), before the global plan is created. Since this narrows the corridor of the upper way, the robot following the plan created on the static map has a hard time to fit through this corridor. Thus, the average travel time is considerablly increased for the robot on this path (see the second row of Table 6.5). The average travel time for a robot following the plan created on the probabilistic map stays the same, sine it is not influenced by the obstacle on the upper way.

In the last test, a box is placed right in the centre of the dynamic area, blocking the upper path completely for the robot (see Figure 6.11). Thus, when the robot detects that the upper path

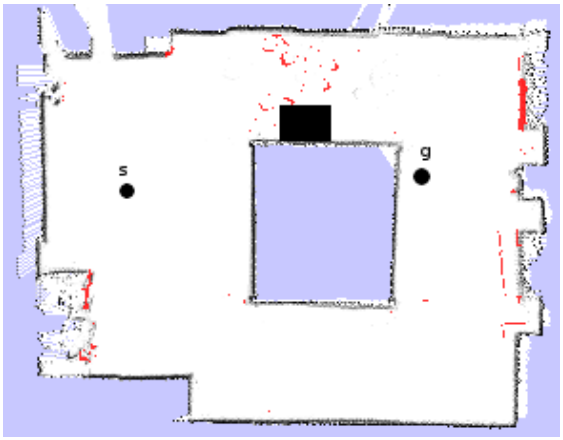


Figure 6.10: A box is placed at the border of the dynamic area

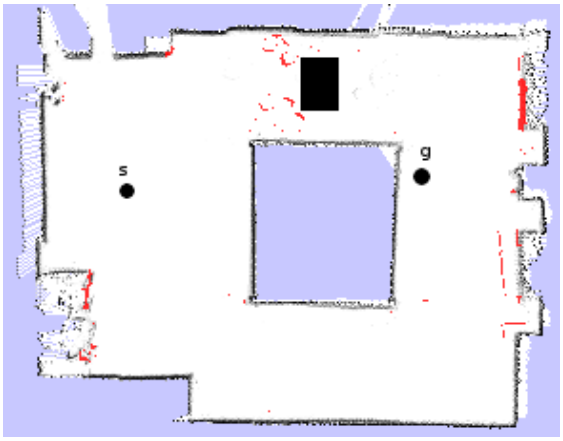


Figure 6.11: A box blocking the upper path

is blocked it has to drive back and take the lower path, which is time consuming. Again, the robot using the path computed with the probabilistic map is not affected. The last column of Table 6.5 shows the average travel time.

CHAPTER 7

Future work

7.1 Filter sonar sensors readings

A possible fask for future work could focus on the sonar sensor problems. One such problem is the specular reflection, creating phantom readings. These phantom readings are especially problematic, when they directly appear in front of the robot, since the robot believes to perceive an obstacle and tries to avoid it. A possible solution could be a module, which filters out such sporadic effects, while allowing valid sonar measurements.

Another problem is, that in some situations, sonar sensors perceive narrow passages smaller than they are in reality. Again, a module could use a heuristic, which detects such situations with the help of previous measurements and corrects them.

For both problems it is important, that this preprocessing is done with minimal time overhead. When the delay is to big, a fast driving robot will crash into obstacles before it got the measurements from the filtering modules.

7.2 Implement a new local_planner

The path planning implementation of this thesis could be enhanced by a local_planner plug-in for the *move_base* node, which is especially designed for the P3AT. This would increase the usefulness of the robot platform, since the current used *dwa_local_planner* module was designed for a broad range of robots. Unfortunately, it is not always capable to find valid trajectories for the P3AT, although the area around the robot is empty.

7.3 Introduce time into path planning

An interesting enhancement could be the introduction of time to path planning. This would allow to find fastest paths in contrast to the common shortest paths. Additionally, this would

allow the robot to wait for events, when the plan requires it. Combining this with probabilistic maps would allow for sophisticated path planners.

For example, assume that a room can be entered via two independent doors and a probabilistic map stores the dynamics of these doors. A path planner could decide to plan the path through the door which is farther away, but has a better combined rating of dynamics and probability that the door is open. The main difference to the current implementation is, that through introduction of time, the robot could wait in front of the possibly closed door, until it is opened. This would be feasible, since the known dynamics of the door allows to estimate a time window, in which the door should open.

CHAPTER 8

Conclusion

The focus of this thesis was to use the information of probabilistic maps to enhance the path planning of navigation algorithms, which are currently based on static maps. Probabilistic maps are capable of storing dynamic areas, by storing occupancy probability and time until change for every cell. A new path planning strategy was developed, which uses this dynamic areas, to decrease the average time the robot needs to reach its goal. This comes from the fact, that less expensive obstacle avoiding have to be done.

Additionally, instead of common used laser scanners, the developed path planning strategies uses significant cheaper sonar sensors. Thus, new ways to insert and delete observations into path planning algorithms were developed.

The developed path planning strategy was implemented by modifying the navigation stack of the robot operating system (ROS). Additional ROS modules were developed to support the autonomously driving robot.

To test the developed path planning strategy, a Pioneer 3-AT robot (P3AT) platform was used as autonomously driving robot. Various experiments were conducted to show how probabilistic maps influence the path planning of the navigation stack. Additionally, experiments with the P3A show the potential of avoiding dynamic areas to decrease the drive time of a robot. Compared to traditional path planning strategies, up to 45% less average time was needed to reach the goal with the new strategy.

At last, various ideas and possibilities to further enhance path planning are proposed.

APPENDIX A

ROS launchfiles

A.1 start_navigation.launch

```
<launch>
  <node name="rosaria" pkg="rosaria" type="RosAria">
    <param name="port" value="/dev/ttyS0" />
    <param name="TicksMM" value="149" />
    <param name="DriftFactor" value="0" />
    <param name="RevCount" value="33608" />
  </node>

  <include file="$(find p2os_urdf)/launch/pioneer3at_urdf.launch
     " />
  <node name="tf_sonar0" pkg="tf" type="
     static_transform_publisher" args="0.147 0.136 0 1.570796 0
     0 /front_sonar /front_sonar0 100" />
  <node name="tf_sonar1" pkg="tf" type="
     static_transform_publisher" args="0.193 0.119 0 0.872665 0
     0 /front_sonar /front_sonar1 100" />
  <node name="tf_sonar2" pkg="tf" type="
     static_transform_publisher" args="0.227 0.079 0 0.523599 0
     0 /front_sonar /front_sonar2 100" />
  <node name="tf_sonar3" pkg="tf" type="
     static_transform_publisher" args="0.245 0.027 0 0.174533 0
     0 /front_sonar /front_sonar3 100" />
  <node name="tf_sonar4" pkg="tf" type="
     static_transform_publisher" args="0.245 -0.027 0 -0.174533
     0 0 /front_sonar /front_sonar4 100" />
```

```
<node name="tf_sonar5" pkg="tf" type="
   static_transform_publisher" args="0.227 -0.079 0 -0.523599
   0 0 /front_sonar /front_sonar5 100" />
<node name="tf_sonar6" pkg="tf" type="
   static_transform_publisher" args="0.193 -0.119 0 -0.872665
   0 0 /front_sonar /front_sonar6 100" />
<node name="tf_sonar7" pkg="tf" type="
   static_transform_publisher" args="0.147 -0.136 0 -1.570796
   0 0 /front_sonar /front_sonar7 100" />

<node name="sonar_calibration_module" pkg="p3at" type="
   sonar_calibration_module">
  <rosparam param="linear_functions">
    - {k: 0.0, d: 0.0}
    - {k: 0,0, d: 0.0}
    - {k: -0.010, d: 0.218}
    - {k: -0.0172, d: 0.250}
    - {k: -0.0172, d: 0.250}
    - {k: -0.010, d: 0.218}
    - {k: 0.0, d: 0.0}
    - {k: 0.0, d: 0.0}
  </rosparam>
  <param name="sonar_sensor_topic_in" value="/rosaria/sonar"
     />
  <param name="sonar_sensor_topic_out" value="/sonar" />
</node>

<node name="safe_navigation" pkg="p3at" type="safe_navigation"
    />

<node name="probabilistic_map_server" pkg="dynamic_mapping"
   type="probabilistic_map_server">
  <param name="file_name" value="$(find dap_launch)/map.yaml"
     />
  <param name="static_service_name" value="static_map" />
  <param name="static_map_topic" value="/map" />
  <param name="static_frame_id" value="map" />
  <param name="probabilistic_service_name" value="
     probabilistic_map" />
  <param name="probabilistic_map_topic" value="/prob_map" />
  <param name="probabilistic_frame_id" value="map" />
  <param name="map_metadata_topic" value="/map_metadata" />
</node>
```

```
  <include file="$(find dap_launch)/amcl.launch"/>
  <include file="$(find dap_launch)/move_base.launch"/>
</launch>
```

A.1.1 amcl.launch

```
<launch>
<node pkg="amcl" type="amcl" name="amcl" respawn="false">
  <!-- overall filter parameters -->
  <param name="min_particles" value="500"/>
  <param name="max_particles" value="5000"/>
  <param name="kld_err" value="0.05"/>
  <param name="kld_z" value="0.99"/>
  <param name="update_min_d" value="0.1"/>
  <param name="update_min_a" value="0.2"/>
  <param name="resample_interval" value="1"/>
  <param name="transform_tolerance" value="0.2"/>
  <param name="recovery_alpha_slow" value="0.0"/>
  <param name="recovery_alpha_fast" value="0.0"/>
  <param name="gui_publish_rate" value="10.0"/>

  <!-- sonar model parameters -->
  <param name="range_sensor_type" value="sonar"/>
  <param name="range_sensor_topic" value="sonar"/>
  <param name="range_sensor_max_beams" value="8"/>
  <param name="range_sensor_z_hit" value="0.70"/>
  <param name="range_sensor_z_short" value="0.05"/>
  <param name="range_sensor_z_max" value="0.05"/>
  <param name="range_sensor_z_rand" value="0.30"/>
  <param name="range_sensor_sigma_hit" value="0.5"/>
  <param name="range_sensor_lambda_short" value="0.1"/>
  <param name="range_sensor_model_type" value="likelihood_field"
     />
  <param name="range_sensor_likelihood_max_dist" value="2.0"/>
  <param name="range_sensor_max_range" value="5.0"/>

  <!-- odometery model parameters -->
  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha1" value="0.5"/>
  <param name="odom_alpha2" value="0.1"/>
  <param name="odom_alpha3" value="0.3"/>
  <param name="odom_alpha4" value="0.4"/>
```

```
  <param name="occupied_threshold" value="90"/>
  <param name="empty_threshold" value="10"/>
</node>
</launch>
```

A.1.2 move_base.launch

```
<launch>
  <node pkg="move_base" type="move_base" name="move_base_node"
     required="true" output="screen">
    <rosparam file="$(find dap_launch)/global_costmap_params.
       yaml" command="load" ns="global_costmap" />
    <param name="base_global_planner" type="string" value="navfn
       /NavfnROS" />
    <param name="NavfnROS/allow_unknown" type="bool" value="
       false" />

    <rosparam file="$(find dap_launch)/local_costmap_params.yaml
       " command="load" ns="local_costmap" />
    <param name="base_local_planner" type="string" value="
       dwa_local_planner/DWAPlannerROS" />
    <rosparam file="$(find dap_launch)/dwa_local_planner_params.
       yaml" command="load" ns="DWAPlannerROS"/>

    <param name="conservative_reset_dist" type="double" value="
       3.0" />
    <param name="controller_frequency" type="double" value="4.0"
        />
    <param name="controller_patience" type="double" value="15.0"
        />

    <rosparam param="recovery_behaviors">
      - {name: local_costmap_reset, type:
         clear_local_costmap_recovery/ClearLocalCostmapRecovery}
      - {name: global_costmap_reset, type:
         clear_global_costmap_recovery/ClearGlobalCostmapRecovery
         }
      - {name: clear_everything, type: clear_everything_recovery/
         ClearEverythingRecovery}
      - {name: ignore_observations, type:
         ignore_observations_recovery/IgnoreObservationsRecovery}
    </rosparam>
```

```
  </node>
</launch>
```

global_costmap_params.yaml

```
global_frame: map
robot_base_frame: base_link
transform_tolerance: 0.2
rolling_window: false
footprint: [ [-0.3125, -0.2505], [-0.3125, 0.2505], [0.3125,
   0.2505], [0.3125, -0.2505] ]
update_frequency: 1.0
publish_frequency: 1.0

plugins:
  - {name: probabilistic_map, type: "costmap_2d::
     ProbabilisticMapLayer"}
  - {name: obstacles, type: "costmap_2d::ObstacleLayer"}
  - {name: probabilistic_inflation, type: "costmap_2d::
     ProbabilisticInflationLayer"}

probabilistic_map:
  probabilistic_map_topic: /prob_map
  forbidden_time_until_change: 75
  forbidden_probability: 50
  use_maximum: false
  unknown_cost_value: -1

obstacles:
  obstacle_range: 2.5
  raytrace_range: 3.0

  observation_sources: point_cloud_sensor
  point_cloud_sensor: {
    sensor_frame: front_sonar,
    data_type: PointCloud,
    topic: sonar,
    sensor_semantic: BroadBeam,
    cone_angle: 7.5,
    observation_persistence: 2.0,
    marking: true,
    clearing: true
  }
```

```
probabilistic_inflation:
  inflation_radius: 0.55
  inflation_factor: 750.0
```

local_costmap_params.yaml

```
global_frame: odom
robot_base_frame: base_link
transform_tolerance: 0.2
rolling_window: true
footprint: [ [-0.3125, -0.2505], [-0.3125, 0.2505], [0.3125,
   0.2505], [0.3125, -0.2505] ]
update_frequency: 1.0
publish_frequency: 1.0
resolution: 0.02
width: 5.0
height: 5.0

plugins:
  - {name: obstacles, type: "costmap_2d::ObstacleLayer"}
  - {name: inflation, type: "costmap_2d::InflationLayer"}

obstacles:
  obstacle_range: 2.5
  raytrace_range: 3.0

  observation_sources: point_cloud_sensor
  point_cloud_sensor: {
    sensor_frame: front_sonar,
    data_type: PointCloud,
    topic: sonar,
    sensor_semantic: BroadBeam,
    cone_angle: 7.5,
    observation_persistence: 2.0,
    marking: true,
    clearing: true
  }

inflation:
  inflation_radius: 0.55
  cost_scaling_factor: 10.0
```

dwa_local_planner.yaml

```
odom_topic: /rosaria/pose

### parameters for speed_cost_function ###
use_velocity_score_function: true
velocity_score_penalty: 30
velocity_score_perimeter: 2

### physical roboter parameters ###
acc_lim_theta: 0.8
max_rot_vel: 0.50
min_rot_vel: 0.10

acc_lim_x: 0.12
max_vel_x: 0.25
min_vel_x: -0.10

acc_lim_y: 0
max_vel_y: 0
min_vel_y: 0

max_trans_vel: 0.25
min_trans_vel: 0.10


### simulation parameters ###
stop_time_buffer: 1.0

sim_time: 1.5
sim_granularity: 0.05
vx_samples: 10
vy_samples: 0
vtheta_samples: 15

path_distance_bias: 60
occdist_scale: 0.75

### roboter is stuck parameters ###
min_travel_dist: 0.30
storage_duration: 15.0
```

A.2 navigation_control_module.launch

```
<launch>
  <node name="navigation control module" pkg="navigation control
      module" type="navigation control module" output="screen">
    <rosparam param="initialposes">
      - {name: ia,
        pose_x: 0.520615816116,
        pose_y: 2.68255114555,
        orientation_x: 0.0,
        orientation_y: 0.0,
        orientation_z: -0.020884441603,
        orientation_w: 0.999781896265
       }
      - {name: ib,
        pose_x: 3.7177541256,
        pose_y: 2.67154550552,
        orientation_x: 0.0,
        orientation_y: 0.0,
        orientation_z: 0.017499173739,
        orientation_w: 0.999846877736
       }
    </rosparam>
    <rosparam param="goals">
      - {name: ga,
        pose_x: 0.520615816116,
        pose_y: 2.68255114555,
        orientation_x: 0.0,
        orientation_y: 0.0,
        orientation_z: -0.020884441603,
        orientation_w: 0.999781896265
       }
      - {name: gb,
        pose_x: 3.7177541256,
        pose_y: 2.67154550552,
        orientation_x: 0.0,
        orientation_y: 0.0,
        orientation_z: 0.017499173739,
        orientation_w: 0.999846877736
       }
    </rosparam>

    <param name="initialpose_topic" value="/initialpose" />
```

```
    <param name="goal_topic" value="/move_base_simple/goal" />
    <param name="frame" value="/map" />
  </node>
</launch>
```

Bibliography

[1] Nancy M. Amato, O. Burchan Bayazit, Lucia K. Dale, Christopher Jones, and Daniel Vallejo. Obprm: An obstacle-based prm for 3d workspaces. In *Proceedings of the Third Workshop on the Algorithmic Foundations of Robotics on Robotics : The Algorithmic Perspective: The Algorithmic Perspective*, WAFR '98, pages 155–168, Natick, MA, USA, 1998. A. K. Peters, Ltd.

[2] N.M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 113–120 vol.1, Apr 1996.

[3] F. Avnaim, J.D. Boissonnat, and B. Faverjon. A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles. In *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, pages 1656–1661 vol.3, Apr 1988.

[4] J. Barraquand and J.-C. Latombe. A monte-carlo algorithm for path planning with many degrees of freedom. In *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 1712–1717 vol.3, May 1990.

[5] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: controllability and motion planning in the presence of obstacles. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 2328–2335 vol.3, Apr 1991.

[6] Jérîme Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. *Int. J. Rob. Res.*, 10(6):628–649, December 1991.

[7] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Learning motion patterns of persons for mobile service robots. In *In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3601–3606, 2002.

[8] V. Boor, M.H. Overmars, and AF. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1018–1023 vol.2, 1999.

[9] J. Borenstein and Y. Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *Robotics and Automation, IEEE Transactions on*, 7(3):278–288, Jun 1991.

[10] O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 1, pages 341–346 vol.1, 1999.

[11] Stephan Brugger. Integrating probabilistic information of dynamic environment into maps for enhanced action planning. Master's thesis, Vienna University of Technology, 2014.

[12] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an interactive museum tour-guide robot. *Artif. Intell.*, 114(1-2):3–55, October 1999.

[13] John F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, USA, 1988.

[14] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[15] A Diosi and L. Kleeman. Advanced sonar and laser range finder fusion for simultaneous localization and mapping. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 1854–1859 vol.2, Sept 2004.

[16] G. Dudek, P. Freedman, and IM. Rekleitis. Just-in-time sensing: efficiently combining sonar and laser range data for exploring unknown worlds. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 667–672 vol.1, Apr 1996.

[17] M. Elbanhawi and M. Simic. Sampling-based robot motion planning: A review. *Access, IEEE*, 2:56–77, 2014.

[18] A Elfes. Sonar-based real-world mapping and navigation. *Robotics and Automation, IEEE Journal of*, 3(3):249–265, June 1987.

[19] Henri Farreny and Henri Prade. Heuristics—intelligent search strategies for computer problem solving, by judea pearl. (reading, ma: Addison-wesley, 1984). *International Journal of Intelligent Systems*, 1(1):69–70, 1986.

[20] Steven Fortune and Gordon Wilfong. Planning constrained motion. *Annals of Mathematics and Artificial Intelligence*, 3(1):21–82, 1991.

[21] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *Robotics Automation Magazine, IEEE*, 4(1):23–33, Mar 1997.

[22] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[23] R. Geraerts and M.H. Overmars. Reachability analysis of sampling based planners. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 404–410, April 2005.

[24] Roland Geraerts and Mark H. Overmars. A comparative study of probabilistic roadmap planners. In *IN: WORKSHOP ON THE ALGORITHMIC FOUNDATIONS OF ROBOTICS*, pages 43–57, 2002.

[25] Subir Kumar Ghosh and D.M. Mount. An output sensitive algorithm for computing visibility graphs. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 11–19, Oct 1987.

[26] Ken Goldberg. Completeness in robot motion planning, 1993.

[27] Dan Halperin and Micha Sharir. A near-quadratic algorithm for planning the motion of a polygon in a polygonal environment, 1995.

[28] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[29] D. Hsu, Tingting Jiang, J. Reif, and Zheng Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 3, pages 4420–4426 vol.3, Sept 2003.

[30] Y.K. Hwang and N. Ahuja. A potential field approach to path planning. *Robotics and Automation, IEEE Transactions on*, 8(1):23–32, Feb 1992.

[31] IEEE Spectrum. http://spectrum.ieee.org/robotics/medical-robots. Accessed: 2014-05-10.

[32] L. Jaillet and T. Simeon. A prm-based motion planner for dynamically changing environments. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 1606–1611 vol.2, Sept 2004.

[33] Klaus-Werner Jörg. World modeling for an autonomous mobile robot using heterogenous sensor information. *Robotics and Autonomous Systems*, 14(2–3):159 – 170, 1995. Research on Autonomous Mobile Robots.

[34] M. Kallman and M. Mataric. Motion planning using dynamic roadmaps. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 5, pages 4399–4404 Vol.5, April 2004.

[35] K Kant and S W Zucker. Toward efficient trajectory planning: The path-velocity decomposition. *Int. J. Rob. Res.*, 5(3):72–89, September 1986.

[36] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration for fast path planning. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 2138–2145 vol.3, May 1994.

[37] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, volume 2, pages 500–505, Mar 1985.

[38] Kyung-Hoon Kim and Kyung Suck Cho. Range and contour fused environment recognition for mobile robot. In *Multisensor Fusion and Integration for Intelligent Systems, 2001. MFI 2001. International Conference on*, pages 183–188, 2001.

[39] Nak-Yong Ko and R.G. Simmons. The lane-curvature method for local obstacle avoidance. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1615–1621 vol.3, Oct 1998.

[40] S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 1, pages 968–975 vol.1, 2002.

[41] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[42] J.P. Laumond. *Robot motion planning and control*. Lectures Notes in Control and Information Sciences 229. Springer, N.ISBN 3-540-76219-1, 1998.

[43] S.M. LaValle. Motion planning. *Robotics Automation Magazine, IEEE*, 18(1):79–89, March 2011.

[44] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[45] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[46] Maxim Likhachev, David Ferguson , Geoffrey Gordon, Anthony (Tony) Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[47] Andrzej Lingas. The power of non-rectilinear holes. In Mogens Nielsen and ErikMeineche Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 369–383. Springer Berlin Heidelberg, 1982.

[48] T. Lozano-Perez. Spatial planning: A configuration space approach. *Computers, IEEE Transactions on*, C-32(2):108–120, Feb 1983.

[49] MobileRobots Inc. http://www.mobilerobots.com/researchrobots/p3at.aspx. Accessed: 2014-05-10.

[50] MobileRobots Inc. *Pioneer 3 Operations Manual with MobileRobots Exclusive Advanced Robot Control & Operations Software* , 2006.

[51] Nils J. Nilsson. A mobile automation: An application of artificial intelligence techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 509–520, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[52] Nils J. Nilsson. *Principles of artificial intelligence*. Tioga Publishing Co., Palo Alto, CA, 1980.

[53] Mark H. Overmars and Mark H. Overmars T. A random approach to motion planning. Technical report, 1992.

[54] S. Petti and T. Fraichard. Safe motion planning in dynamic environments. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 2210–2215, Aug 2005.

[55] J.H. Reif. Complexity of the mover's problem and generalizations. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 421–427, Oct 1979.

[56] Robot Operating System. http://wiki.ros.org/amcl. Accessed: 2014-09-04.

[57] Robot Operating System. http://wiki.ros.org/dwa_local_planner. Accessed: 2014-09-25.

[58] Robot Operating System. http://wiki.ros.org/map_server. Accessed: 2014-09-01.

[59] Robot Operating System. http://wiki.ros.org/move_base. Accessed: 2014-09-22.

[60] Robot Operating System. http://wiki.ros.org/navfn. Accessed: 2014-09-23.

[61] Robot Operating System. http://wiki.ros.org/navigation. Accessed: 2014-09-04.

[62] Robot Operating System. http://wiki.ros.org/navigation/tutorials/robotsetup. Accessed: 2014-09-04.

[63] Robot Operating System. http://wiki.ros.org/p2os_urdf. Accessed: 2014-09-22.

[64] Robot Operating System. http://wiki.ros.org/rosaria. Accessed: 2014-08-18.

[65] Robot Operating System. http://wiki.ros.org/ros/concepts. Accessed: 2014-08-16.

[66] Robot Operating System. http://wiki.ros.org/roslaunch. Accessed: 2014-08-21.

[67] Robot Operating System. http://wiki.ros.org/roslaunch/architecture. Accessed: 2014-08-21.

[68] Robot Operating System. http://wiki.ros.org/roslaunch/xml. Accessed: 2014-08-21.

[69] Robot Operating System. http://wiki.ros.org/rviz. Accessed: 2014-09-28.

[70] Robot Operating System. http://wiki.ros.org/tf. Accessed: 2014-08-18.

[71] Robot Operating System. http://www.ros.org. Accessed: 2014-08-15.

[72] Robot Operating System. http://www.ros.org/core-components. Accessed: 2014-08-16.

[73] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 4, pages 3375–3382 vol.4, Apr 1996.

[74] D.J. Spero and R.A Jarvis. Towards exteroceptive based localisation. In *Robotics, Automation and Mechatronics, 2004 IEEE Conference on*, volume 2, pages 822–827 vol.2, Dec 2004.

[75] C. Stachniss and W. Burgard. An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 1, pages 508–513 vol.1, 2002.

[76] A Stentz. Optimal and efficient path planning for partially-known environments. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 3310–3317 vol.4, May 1994.

[77] Anthony Stentz. The focussed d* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1652–1659, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[78] Petr Svestka. On probabilistic completeness and expected complexity of probabilistic path planning, 1996.

[79] TEDUSAR - Technology and Education for Search and Rescue Robots. http://www.tedusar.eu/cms/. Accessed: 2014-05-10.

[80] Sebastian Thrun, Arno Bücken, Wolfram Burgard, Dieter Fox, Thorsten Fröhlinghaus, Daniel Hennig, Thomas Hofmann, Michael Krell, and Timo Schmidt. Artificial intelligence and mobile robots. chapter Map Learning and High-speed Navigation in RHINO, pages 21–52. MIT Press, Cambridge, MA, USA, 1998.

[81] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[82] J.P. van den Berg and M.H. Overmars. Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 1, pages 453–460 Vol.1, April 2004.

[83] Jur Pieter van den Berg. *Path Planning in Dynamic Environments*. PhD thesis, Utrecht University, 2007.

[84] J. Vannoy and Jing Xiao. Real-time adaptive and trajectory-optimized manipulator motion planning. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 1, pages 497–502 vol.1, Sept 2004.

[85] Dizan Vasquez and Thierry Fraichard. Motion prediction for moving objects: a statistical approach. In *In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2004.