Czech Technical University in Prague
Faculty of Electrical Engineering

DIPLOMA THESIS

Bc. Maroš Vaňo

Qualitative Reasoning for Robotic Topological Map Building

Department of Cybernetics
Supervisor: **Ing. Karel Košnar, Ph.D.**

Prague, 2013

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Maroš V a ň o

Studijní program: Kybernetika a robotika (magisterský)

Obor: Robotika

Název tématu: Kvalitativní uvažování pro stavbu robotických topologických map

Pokyny pro vypracování:

1. Diplomant se seznámí s metodami kvalitativního prostorového uvažování a s metodami tvorby a udržování topologických map prostředí.
2. Diplomant navrhne vhodné metody pro stavbu topologické mapy pro mobilní robot z dat z laserového dálkoměru a metody pro kvalitativní uvažování nad těmito mapami.
3. Navržené metody naimplementuje a
4. otestuje v úloze mapování neznámého prostředí s uzavíráním smyček,
5. navrhne, provede a vyhodnotí experimenty s reálnými roboty systému SyRoTek.

Seznam odborné literatury:

[1] Jan Oliver Wallgrun: Qualitative Spatial Reasoning for Topological Map Learning. Spatial Cognition & Computation, 10:207-246, 2010

[2] Margaret E. Jefferies, Wai-Klang Yeap (Eds.): Robotics and Cognitive Approaches to Spatial Mapping, Springer-Verlag Berlin Heidelberg, 2008

Vedoucí diplomové práce: Ing. Karel Košnar, Ph.D.

Platnost zadání: do konce letního semestru 2013/2014

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 7. 11. 2012

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Maroš V a ň o

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Diploma Thesis: Qualitative Reasoning for Robotic Topological Map Building

Guidelines:

1. Student will learn methods for qualitative spatial reasoning and methods for building and maintaining of topological maps.
2. Student will design methods for building topological map from laser range-finder data suitable for mobile robots and methods for qualitative reasoning over these maps.
3. He implements designed methods and
4. tests these methods on the task of the mapping of unknown environment with loop-closing and
5. designs, realizes and evaluates experiments with real robots of the SyRoTek platform.

Bibliography/Sources:

[1] Jan Oliver Wallgrun: Qualitative Spatial Reasoning for Topological Map Learning. Spatial Cognition & Computation, 10:207-246, 2010

[2] Margaret E. Jefferies, Wai-Klang Yeap (Eds.): Robotics and Cognitive Approaches to Spatial Mapping. Springer-Verlag Berlin Heidelberg, 2008

Diploma Thesis Supervisor: Ing. Karel Košnar, Ph.D.

Valid until: the end of the summer semester of academic year 2013/2014

L.S.

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, November 7, 2012

Prehlásenie autora práce

Prehlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky použité informačné zdroje v súlade s Metodickým pokynom o dodržovaní etických princípov pri príprave vysokoškolských záverečných prác.

V Prahe dňa 9.5.2013 .....

.....
podpis autora práce

Abstrakt

Diplomová práca sa zaoberá problémom topologického mapovania neznámeho prostredia mobilným robotom, ktoré obsahuje objekty pravouhlého tvaru. Všetko čo má robot k dispozícii sú len lokálne dáta z odometrie a laserového diaľkomeru. Počas procesu mapovania robot vyhľadáva významné miesta v prostredí, z ktorých následne buduje hypotézy. Hypotéza predstavuje odhad skutočného stavu prostredia, ktorej reprezentácia je vo forme orientovaného grafu. Čím dlhšie robot prostredie prechádza, tým viac nesprávnych hypotéz je odstránených. Prostredie je prehľadávané dovtedy, kým nie sú všetky významné miesta navštívené. Na konci procesu mapovania má robot k dispozícii aspoň jednu platnú hypotézu zodpovedajúcu tvaru prostredia. V práci sú položené teoretické základy metódy generovania hypotéz. Tieto metódy sú následne implementované a otestované pre rôzne prostredia použitím rôznych plánovačov. Na záver sú dosiahnuté výsledky zdokumentované a zhodnotené.

Abstract

This thesis deals with problem of topological mapping of an unknown environment using a mobile robot. Everything the robot has are only odometry data and data from laser range-finder available. During the mapping process robot searches the most significant places in the environment which are then used for building of hypotheses. Hypothesis represents an estimate of the actual environment state in the form of bidirectional graph. The longer the robot passes the environment, the more false hypotheses is removed. The environment is traversed until all significant places are visited. At the end of mapping process the robot has at least one valid hypothesis corresponding to the shape of the environment available. In the thesis are introduced theoretical fundamentals of method of generating hypotheses. These methods are then implemented and tested on various environments using various planners. Finally, achieved results are documented and evaluated.

I would like to thank my supervisor Ing. Karel Košnar, Ph.D. for his continual support and advice throughout this work.
This work is dedicated to my parents who supported me during my studies.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Robotics is the sector of technology that deals with construction, operation, application and design of robots. It deals with automated machines that can take the place of humans in manufacturing processes, medical processes, dangerous environments, everyday activities, or resemble humans in appearance, behavior and cognition. The beginnings of robotics sometimes relate to early water clocks discovered in China in the 6th century BC and with a stop-watch for imposing a time limits on clients' visits in Athenian brothels used by Clepsydra in the Greco-Roman world in the 4th century BC. History of robotics continues through centuries with the notable record in the 15th century, where the first designs of a humanoid robot representing a mechanical knight in armor which was able to sit up, wave its arm and move its head and jaw, was sketched by Leonardo da Vinci. Then in 1898 Nikola Tesla built and demonstrated a remote controlled robot boat at Madison Square Garden. Today's form of robotics dates from the half of the 20th century when the first programmable robot was designed by George Devol in 1954.

Nowadays, robots are quickly moving from the factories and laboratories into offices, households, and streets. Mobile robots have to explore and navigate large-scale environments with only partial assumptions about their structure. Thus, they sense the environment, construct its representation, reason over this representation and perform the actions to reach specified goals. The robots also need to communicate in some way with humans for transferring gathered information and achieved goals.

1.1 Motivation

Let us have the following situation: A robot is moving through graph-like environment like the one shown in Figure (1.1). The robot does not have any information about this environment. Everything he has are only its own sensors for sensing the environment, motors for acting in this environment and computing capacity for reasoning over sensed data. The goal for this robot is to create its own map of this environment which will correspond to the real world state. This task can be compared to a situation, when a human is in an unknown environment and needs to orientate in it.

The robot will, similarly like the human, try to find most significant places in this environment, store information about their look and position, and it will try to estimate its position according to the deployment of this places in the environment.

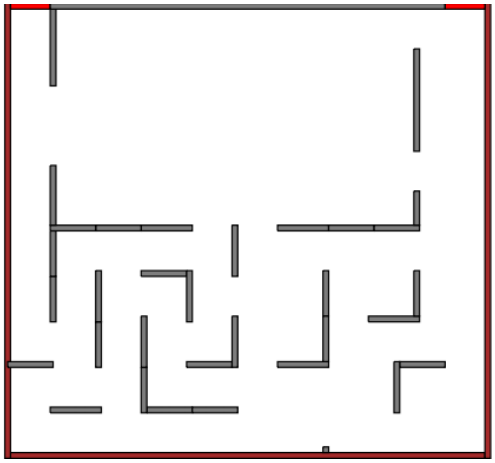


Figure 1.1: An example of graph-like environment

The problem of mapping is one of the most fascinating problems in the robotics area. Though, it is also one of the hardest task connected with solving a lot of uncertainties, unpredictable events and errors.

1.2 Thesis outline

This thesis is organized into three parts. The fist part consists from this introduction, Chapter (2) which defines the problem of robotic mapping and introduces the goals of the thesis, and Chapter (3) discusses the theory in the areas of the robotics, the robot and its sensors, a robot control architectures, building the map of the environment, multi-hypothesis topological mapping, and robotic frameworks.

The second part of this thesis is a description of a solution. Chapter (4) discusses the solution and used methods for dividing the problem into logical parts and solving these parts. The last part consists of experiments in Chapter(5) where the used approach was tested in detail and results are summarized. The thesis concludes in Chapter (6).

Chapter 2

Problem definition

This thesis deals with the problem of building maps in a large environment. A mobile robot must have an ability to move through any environment for fulfilling specified tasks. Then it needs to have a model of the environment which would be efficient, robust and the move in it must be repeatable.

The *large-scale* space is environment which can not be sensed at once. The robot must navigate through this environment and build a map. The map is integrated from local observations gathered from different places. Note, that large-scale definition does not tell about physical size of the environment, but it defines "large" in manner of perceptual limitations.

The problem of *mapping* is defined as a process of learning and maintaining a spatial model of an unknown environment. This spatial model is called a *map*. A mobile robot is moving through an unknown environment and collects local information, called observations. This information is then integrated into overall coherent model which reflects a section of the real world. The integration process can be either *incremental* (on-line) when new information is immediately incorporated into the current model or *built from the history* (off-line) when all the input data are available at once and the process of their integration can be repeated.

The mapping process is very hard and challenging because the sensed information is imprecise, ambiguous and erroneous. The main problem while mapping a large-scale environment is the similarity of various objects. There is no way how to distinguish these places using only the local observation. The problem is also to get the correspondences between the current observation and the stored entities in the robot's internal model of the environment.

2.1 Thesis goals

This thesis aims to solve the problem of mapping and unknown environment using only **local information**. Human operator specifies only initial parameters and lets the robot map the environment. At the end of mapping process a map corresponding to this environment must be produced. Thus, the map has to be stored in an appropriate data structure also understandable for people.

To reach this complex problem, the goals of this thesis have been determined as:

1. Learning methods for qualitative spatial reasoning and methods for building and maintaining of topological maps.
2. Design of methods for building topological map from laser range-finder data suitable for mobile robots and methods for qualitative reasoning over these maps.
3. Implementation of designed methods.
4. Testing of designed methods on the task of mapping of an unknown environment with loop-closing.
5. Designing, realizing and evaluating experiments with real robot of the *SyRoTek* platform.

2.2 Conditions

The mapping problem is very hard in general. Thus, a subset of this complex problem was chosen to solve, where the following limitations have to be assumed. The environment consists only from obstacles which have **perpendicular geometric shape** with defined borders and dimensions. Then the environment must be **stable** meaning that all the changes of environment does not influence its structure. The last required condition is the **static** environment. The obstacles and object in this environment cannot change their position in a time.

The environments are supposed to be perpendicular, stable and static in the rest of the thesis.

Chapter 3

Theory

3.1 Robots

A robot describes any construct with input and output that automates some behavior. In terms of the autonomy the robot can be **teleoperated** when it is operated by a human, **semi-autonomous** when it is executing only partial goals assigned by a human, or **autonomous** if goals assigned by a human or computer system are autonomously executed. The highest level of the autonomy is **cognitive** robot which has its own generator of goals for realizing given strategy. Note that the word **robot** was popularized by Czech writer Karel Čapek in his play R.U.R. (Rossum's Universal Robots) in 1921.

A **mobile robot** is an automatic machine that is capable of movement in given environment. Typically it has these basic functionalities:

- *sensing* - collecting the information from the environment via sensors,
- *perception* - processing and converting sensor information into internal representation,
- *planning and reasoning* - choosing a method for achieving goals and generating the trajectory, and
- *action* - acting the environment with physical realization of the plans.

How the robot senses the world is discussed in the following Section. Then methods of creating internal representation of this world are introduced in Section (3.1.2). Finally, reasoning over this representation and planning the trajectory in it is described in Section (3.2).

3.1.1 Robot sensors

The robot uses sensors for gathering information about the environment and its local state. This state is then used for building the internal space model of the environment. The sensors can be divided into two basic categories, *interoceptive* and *exteroceptive*. The *interoceptive* constitutes all sensors that measures the current state of the robot, like optical encoders. The *exteroceptive* sensors determine the measurements of objects relative to a robot's frame of reference. These sensors include contact sensors, range sensors and vision sensors.

The next basic categorization of sensors is *passive* or *active*. *Passive* sensors does not emits any energy to get properties of the environment, like camera or compass. The opposite group are *active* sensors transmitting some energy to the environment to get its properties. An example of active sensors is sonar or laser range-finder.
In the following sections, the most used sensors in mobile robotic will be described.

3.1.1.1. Optical encoders

Optical encoders are mostly used for registering wheel revolutions of the robot. These encoders works on principle where a focused beam of light aimed at a matched photodetector is periodically interrupted by a coded opaque/transparent pattern on a rotating intermediate disk attached to the shaft of interest. The rotating disk may take the form of chrome on glass, etched metal, or photoplast. This encoder is considered as low-cost reliable package with good noise immunity.
Two basic types of optical encoders are known: *absolute* or *incremental*. The absolute model measures angular position and deduces velocity while the incremental version measures rotational velocity and can deduce a relative position. Errors which can potentially occur can be divided into two categories: *systematic errors* and *non-systematic errors*.
Systematic errors: unequal wheel diameters; average of actual wheel diameters differs from nominal wheel diameter; actual wheelbase differs from nominal wheelbase, misalignment of wheels; and finite encoder resolution and sampling rate.
Non-systematic errors: travel over uneven floors; travel over unexpected object on the floor; wheel-slippage due to slippery floors; over-acceleration and so on [2].

Absolute optical encoders Absolute encoders are typically used for slower rotational velocities where the positional information is required even if power interruption occurs. Discrete detector elements in a detector array are individually activated according to broken beam with concentric encoder tracks as can be seen in Figure (3.1).

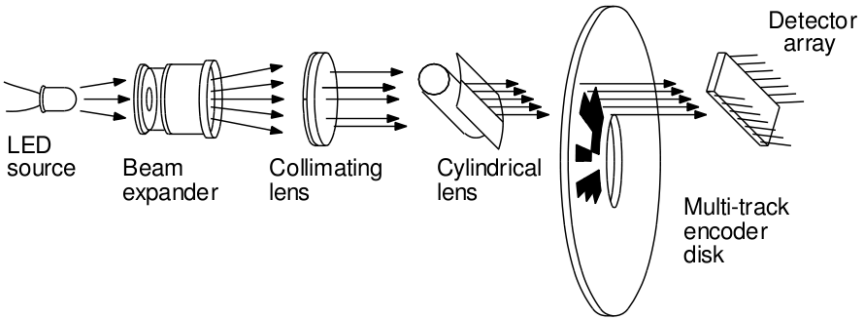


Figure 3.1: A line source of light passing through a coded pattern of opaque and transparent segments on the rotating encoder disk results in a parallel output that uniquely specifies the absolute angular position of the shaft. [1]

This type of encoders provide a parallel output with a unique code pattern for each quantized shaft position. The most used coding schemes are binary-coded decimal, natural binary, and Gray code which can be seen in Figure (3.2).

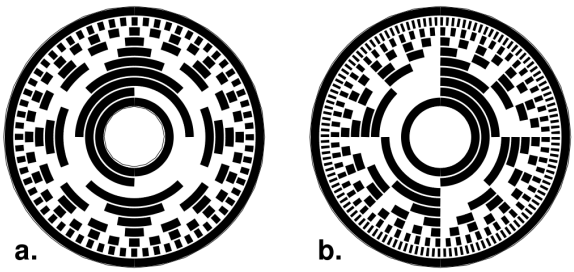
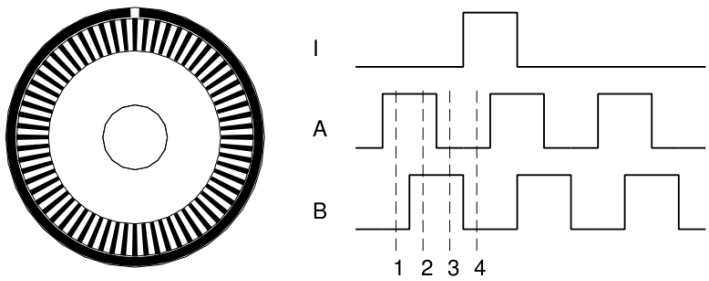


Figure 3.2: Rotating an 8-bit absolute Gray code disk.
a. Counterclockwise rotation by one position increment will cause only one bit to change.
b. The same rotation of a binary-coded disk will cause all bits to change in the particular case (255 to 0) illustrated by the reference line at 12 o'clock. [4]

Incremental optical encoders The well known simplest type of incremental encoder is a single-channel *tachometer encoder*, essentially a mechanical light chopper that generates a certain number of sine- or square-wave pulses for each shaft revolution. For higher resolution of the unit, more pulses need to be added. When using single-channel tachometer encoders, the direction of rotation cannot be detected, thus they cannot be used as position sensors. These problems overcome *Phase-quadrature incremental encoders* by adding a second channel, displaced from the first channel. The next problem with these encoders is at extremely slow velocities, where noise and stability problems can occur due to quantization errors [1]. For applications, where continuous 360° is needed, most encoders incorporate as a third channel that is set high once for each complete revolution of the shaft (see Figure (3.3)). Then intermediate shaft positions are specified by the number of encoder up/down counts from this known position. The only disadvantage of this approach is that the relative position information is lost when the power is interrupted.



State	Ch A	Ch B
S_1	High	Low
S_2	High	High
S_3	Low	High
S_4	Low	Low

Figure 3.3: The observed phase relationship between Channel A and B pulse trains can be used to determine the direction of rotation with a phase-quadrature encoder, while unique output states S1 - S4 allow for up to a four-fold increase in resolution. The single slot in the outer track generates one index pulse per disk rotation [4]

3.1.1.2. Range sensors

Many of today's range sensors use the *time-of-flight* (TOF) method, therefore are classified to the group of active, exteroceptive sensors. This sensors work on principle that the energy is emitted in a rapid sequence of short bursts aimed directly at the object being ranged. The time t to travel the round-trip distance d_r is measured and used to calculate distance d to the target, based on the speed of the light c (roughly $0.3m/ns$) or speed of sound in air (roughly $0.3m/ms$). Note that t is time required for a given pulse to reflect off the object and return, thus d_r is measured twice and needs to be reduced by half. This calculation is given by the following equations [2]:

$$d_r = vt \tag{3.1a}$$

$$d = \frac{d_r}{2} \tag{3.1b}$$

The returned signal has essentially the same path back to a receiver which is located coaxially with or in close proximity to the transmitter. Potential errors for TOF systems include the following [2]:

- "Variation in the speed of propagation, mostly in the case of acoustical systems,
- Uncertainties in determining the exact time of arrival of the reflected pulse,
- Inaccuracies in the timing circuitry used to measure the round-trip time of light, and
- Interaction of the incident wave with the target surface."

Sonar range-finder This range-finder emits acoustic waves with frequency $50kHz$ to the environment and measures the period until these waves return back. Range of this type of sensor is typically in $3\text{-}6m$ with accuracy in $3\text{-}5cm$. An disadvantage of the sonar range-finder is focusing its emitting diagram what leads to false echo and

subsequently to the problem with data interpretation. This problems can be partially solved using the model of the sonar sensor which has the probability model of data interpretation. The scheme of this sensor is shown in Figure (3.4).

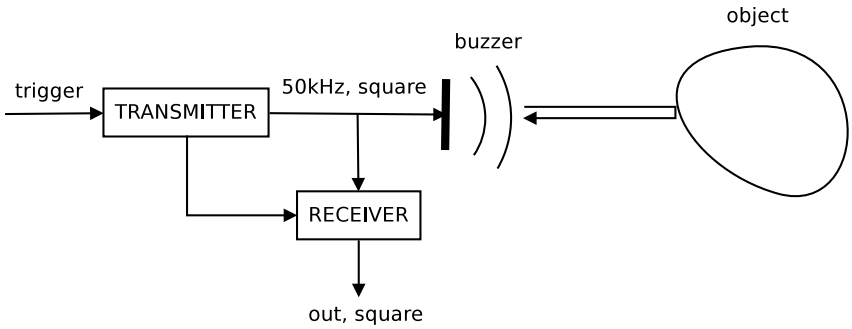


Figure 3.4: Scheme of TOF sonar range-finder

Laser range-finder This type of sensor uses the light as the source of emitted energy. It is more precise than sonar range-finder, because its emitted beam has very narrow dispersion. Laser range-finder is used in applications, where the high accuracy is required. Its measurements are often averaged internally by the laser hardware. Its range is typically in *5-100m*, accuracy is in *10-30mm* and resolution in azimuth is in *0.3-5°*. The scheme of this range-finder can be seen in Figure (3.5).

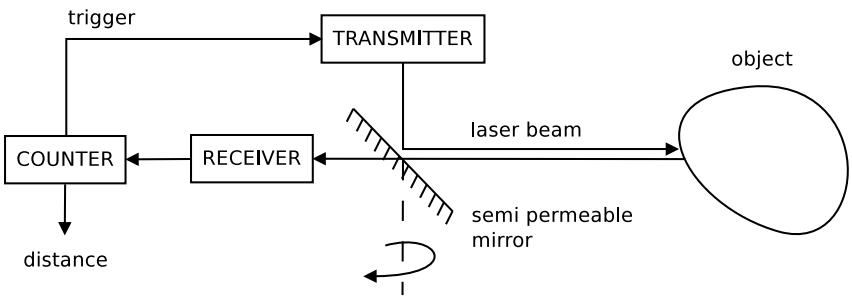


Figure 3.5: Scheme of TOF laser range-finder

3.1.1.3. Camera

Camera is a passive, exteroceptive sensor often used in robotics because it provides complex view of the environment at once. But recognition and obstacle classification is one of the hardest problem to solve. Firstly, the camera must be calibrated to ensure the widest possible range of measured information. Then the perspective view of the camera must be transformed into the coordinate system of the robot. After this process, the camera is prepared for the image procession. This procession lies in recognizing the borders, shapes, significant points and objects in the image. Note that for measuring depth of objects in the image two cameras are needed to fulfill the triangular rule.

3.1.2 Map building

The robot builds a map to have an internal information about the world it is moving in. This information is used in reasoning and planning process. There are two fundamental approaches of modeling, the **metric**, and the **topological**. The *metric* approach has the most commonly used subset, the *grid-based* approach which is represented by evenly-spaced grids. These grids, called also *occupancy grids*, are multidimensional random fields that contains estimates of the occupancy state of certain area stored in cells. Occupancy grids, are easy to build and to maintain in large-scale environments. The position of the robot is estimated incrementally, based on odometric information and sensor scans taken by the robot. Whereas the position of a mobile robot can be tracked accurately, different positions for which sensors measurements look similarly can be naturally eliminated. A major disadvantage of grid-based approach is their enormous space and time complexity. The resolution of a grid must be fine enough to capture every detail in the environment which leads to this enormous complexity.

Topological approach represents robot environments as a graph. Nodes in this graph correspond to significant places, situations, or landmarks and they are connected by edges if the direct path between them exists. This approach determine the position of the robot relative to the model, mainly based on landmarks or significant current sensor scans. For instance, when the robot traverses two places which looks alike, this approach often has a problem with determining if these places are the same or not (especially if these places have been reached via different paths). On the other side, key advantage of topological representation of the environment is its compactness since its resolution corresponds directly to the complexity of the environment. The next advantages of this approach are: their representation permits using of symbolic planners and problem-solvers, fast planning is permitted and they provide more natural interface for human (for example *"go to place A"*). Moreover, topological approach does not require the exact determination of the position of the robot.

Summarization of both methods and their strengths and weakness according to [13] can be seen in Table (3.1).

3.1.2.1. Grid-based maps

Grid-based maps described here are discrete, two-dimensional occupancy grids where each grid cell $\langle x, y \rangle$ in a map has stored value that measures the subjective belief about occupancy of this cell.

When building metric maps, sensor scan data must be converted into occupancy values $o_{x,y}$ for each cell $\langle x, y \rangle$. These values can be in interval $\langle 0, 1 \rangle$ where value *1* indicates a occupied cell, value *0* indicates a free cell and value *0.5* indicates maximal uncertainty.

Sensor interpretations are integrated over time to get a single, consistent map. For t-th sensor reading s_t, the *probability* of occupancy of a grid cell $\langle x, y \rangle$ is: $P(o_{x,y}|s_t)$. A map is constructed by integrating these probabilities for all sensor readings $s_1, s_2, ..., s_T$ where for each grid cell $\langle x, y \rangle$ the probability is: $P(o_{x,y}|s_1, s_2, ..., s_T)$. A commonly used approach is to estimate this quantity applying *Bayes's rule*. For

Table 3.1: Advantages and disadvantages of grid-based and topological approaches to map building

Grid-based (metric) approaches	**Topological approaches**
+ easy to build, represent, and maintain	+ permits efficient planning, low space
+ recognition of places (based on geometry) is non-ambiguous and view point independent	+ does not require accurate determination of the robot's position
+ facilitates computation of shortest paths	+ convenient representation for symbolic planner/problem solver, natural language
− planning inefficient, space-consuming (resolution does not depend on the complexity of the environment)	− difficult to construct and maintain in large-scale environments if sensor information is ambiguous
− requires accurate determination of the robot's position	− recognition of places often difficult, sensitive to the point of view
− poor interface for most symbolic problem solvers	− may yield suboptimal paths

this, $P(o_{x,y}|s_t)$ must be independent of $P(o_{x,y}|s_{t'})$ if $t \neq t'$. Then the desired probability with prior probability $P(o_{x,y})$ can be computed in the following way [13]:

$$P(o_{x,y}|s_1, s_2, ..., s_T) = 1 - \left(1 + \frac{P(o_{x,y}|s_1)}{1 - P(o_{x,y}|s_1)} \prod_{\tau=2}^{T} \frac{P(o_{x,y}|s_\tau)}{1 - P(o_{x,y}|s_\tau)} \frac{1 - P(o_{x,y})}{P(o_{x,y})}\right)^{-1} \tag{3.2}$$

The formula can be updated by Bayes's rule and the conditional independence assumption. According to Bayes's rule:

$$\frac{P(o_{x,y}|s_1, s_2, ..., s_T)}{P(\neg o_{x,y}|s_1, s_2, ..., s_T)} = \frac{P(s_T|o_{x,y}, s_1, s_2, ..., s_{T-1})}{P(s_T|\neg o_{x,y}, s_1, s_2, ..., s_{T-1})} \frac{P(o_{x,y}|s_1, s_2, ..., s_{T-1})}{P(\neg o_{x,y}|s_1, s_2, ..., s_{T-1})} \tag{3.3}$$

which can be simplified by assumption of the conditional independence to:

$$\frac{P(o_{x,y}|s_1, s_2, ..., s_T)}{P(\neg o_{x,y}|s_1, s_2, ..., s_T)} = \frac{P(s_T|o_{x,y})}{P(s_T|\neg o_{x,y})} \frac{P(o_{x,y}|s_1, s_2, ..., s_{T-1})}{P(\neg o_{x,y}|s_1, s_2, ..., s_{T-1})} \tag{3.4}$$

Induction over T gives:

$$\frac{P(o_{x,y}|s_1, s_2, ..., s_T)}{P(\neg o_{x,y}|s_1, s_2, ..., s_T)} = \frac{P(o_{x,y})}{1 - P(o_{x,y})} \prod_{\tau=1}^{T} \frac{P(o_{x,y}|s_\tau)}{1 - P(o_{x,y}|s_\tau)} \frac{1 - P(o_{x,y})}{P(o_{x,y})} \tag{3.5}$$

The update of equation is now obtained by solving $P(o_{x,y}|s_1, ..., s_T)$, using the fact that $P(\neg o_{x,y}|s_1, ..., s_T) = 1 - P(o_{x,y}|s_1, ..., s_T)$.

The accuracy of the metric map depends on the alignment of the robot with its map where a slippage and drift can have negative effect on the estimation of the robot position. Thus, correcting a slippage and drift is important there. When the

robot process an actual sensor reading, it constructs a local map. Then, the correspondence of the local and the global map is measured. Let we have $\langle x, y\rangle$ denoting the coordinates of a cell in the global map and let we have $\langle x', y'\rangle$ denoting th corresponding coordinates in the local map. Let we have $\langle x_i, y_i\rangle$ for $i = 1, ..., 4$ denoting the coordinates of the four grid points in the local map that are nearest to $\langle x', y'\rangle$. Then for local occupancy grid $lo_{x,y}$ the global occupancy $o_{x,y}$ is matched with the local occupancy value using the following interpolation [13]:

$$\frac{\sum_{i=1}^{4} |x - x_i||y - y_i| lo_{x_i,y_i}}{\sum_{i=1}^{4} |x - x_i||y - y_i|} \tag{3.6}$$

Using equation (3.6), the coordinate of a global grid cell is projected into the local robot's coordinates.

Exploration in grid-based maps lies in moving the robot on a minimum-cost path to the nearest unexplored grid cell. This cost is determined according to occupancy value. The minimum-cost path is calculated using a modified version of *value iteration* $V_{x,y}$ in these steps [13]:

- "*Initialization*: $V_{x,y} = 0$ if $\langle x, y\rangle$ is unexplored and $V_{x,y} = \infty$ if $\langle x, y\rangle$ is explored. Grid cells are considered as *explored* if their occupancy value $P(o_{x,y})$ has been updated at least once, otherwise they are considered as *unexplored*.

- *Update loop*: $V_{x,y} = min\{V_{x+\psi,y+\xi} + P(o_{x+\psi,y+\xi})\}$ where $\psi = -1, 0, 1$ and $\xi = -1, 0, 1$ is done for all explored grid cells $\langle x, y\rangle$. Values of all explored grid cells are updated by the value of their best neighbors plus the cost of the path to these neighbors. The cost is represented by probability $P(o_{x,y})$ meaning that grid cell $\langle x, y\rangle$ is occupied. Then, update rule is iterated until it converges, where each value $V_{x,y}$ measures *cumulative cost* for moving to the nearest unexplored cell.

- *Determine motion direction*: The robot generates a minimum-cost path what is done by steepest descent in V, starting at the actual robot position.

- *Selective reset phase*: Every time when the map is updated, values $V_{x,y}$ that are too small are found and reset. This is done by the following loop:

$$\forall\ explored\ \langle x, y\rangle\ V_{x,y} = \infty\ if\ V_{x,y} < min\{V_{x+\psi,y+\xi} + P(o_{x+\psi,y+\xi})\} \tag{3.7}$$

  where $\psi = -1, 0, 1$ and $\xi = -1, 0, 1$.

- *Bounding box*: a rectangular bounding box $\langle x_{min}, y_{min}\rangle$ x $\langle x_{max}, y_{max}\rangle$ containing all grid cells in which $V_{x,y}$ may change is maintained. Because of this, value iteration focuses only on a small fraction of the grid, thus it converges much faster."

3.1.2.2. Topological maps

Topological maps are built either directly from the sensor data or from the grid-based maps.

In the first case, landmarks in an unknown environment must be found. Landmarks are usually significant places in the environment, like corners, junctions, and so on. These places are usually detected using image recognition when obtained from the camera, or using laser scan procession when obtained from laser range-finder. Then, these processed data are stored in a specified form as a *vertex* of a graph. A reliable detection is always the biggest problem when constructing the topological maps directly from the sensoric data.

An *edge* is a path connecting two vertices usually represented by a hallway. Orientation of the edge in the graph determines, from which vertex the next one was reached.

In the other case of building topological maps, grid-based maps are decomposed into a small set of regions separated by narrow passages (also called *critical lines*) such as doorways. These critical lines are found by detecting a skeleton of the environment. The partitioned map is then mapped into a graph, where vertices correspond to regions and edges connect neighboring regions. This map is built in the following steps [13]:

- "*Thresholding*: each occupancy value $o_{x,y}$ in the occupancy grid is thresholded. Cells with value below given threshold are considered as free-space C and other cells are considered as occupied $\bar{C}$.

- *Voronoi diagram*: $\forall$ *point in*$\langle x, y\rangle \in C$ there is at least on *nearest point* in the occupied space $\bar{C}$. These points are called *basis points of* $\langle x, y\rangle$, and the distance between $\langle x, y\rangle$ and its basis points *clearance of* $\langle x, y\rangle$. Then Voronoi diagram is the set of points in free-space having at least two equidistant basis-points.

- *Critical points*: are points on the Voronoi diagram that minimize clearance locally. Each critical point $\langle x, y\rangle$ has these two properties: a) it is a part of the Voronoi diagram and b) the clearance of all points in ϵ-neighborhood of $\langle x, y\rangle$ is not smaller.

- *Critical lines*: are obtained by connecting each critical point with its basis points. They have exactly two basis points.

- *Topological graph*: this partitioning is mapped into the graph."

Planning in the topological map is easy and straightforward, since its abstraction allows using the well known algorithms for finding a path between two vertices. These algorithms include A*, Dijkstra's, Breadth-first search, Depth-first search and so on.

3.1.3 Robot control architectures

Control architecture determines, how the robot performs its actions and act the environment according to inputs. The inputs are represented by data from sensors and the environment is acted by motors of the robot. There are three basic robot architectures used in robotics, **reactive architecture**, **functional decomposition** and **deliberative architecture**.

A *paradigm* is a technique characterizing the approach to a problem classification [11]. Paradigms of appointed architectures can be seen in Figure (3.6).

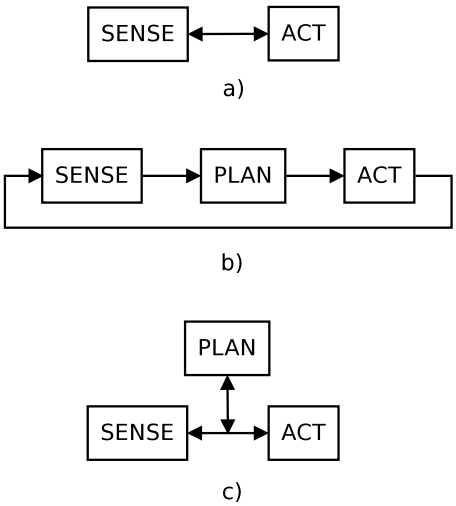


Figure 3.6: Robot paradigms: a) reactive, b) hierarchical and c) hybrid deliberative/reactive.

In the following sections, description of every architecture will be provided.

3.1.3.1. Reactive architecture

In reactive architecture, the basic modules of the robot in the lowest layer are defined. From these modules, the more complex structures are constructed. In this case, a behavior based approach is used. Modules have access to direct sensory data and overall behavior is computed from all outputs of the modules. Reactive architecture does not model the environment, but it directly reacts to stimulus from the environment. *Reactive paradigm* used in this architecture is **SENSE-ACT** (see Figure (3.6)(a)). It assumes that the input to an **ACT** will always be the direct output of a sensor, **SENSE**.

This architecture is suitable for easier tasks. Its advantage is in the high degree of robustness and an ability to deal with unexpected changes in the dynamic environment. A disadvantage is the deadlock in the local extreme of control functions.

3.1.3.2. Functional decomposition

Functional decomposition divides more complex tasks into simpler subtasks. The subtasks are then solved by particular modules, where the next module uses input from the previous one. Thus, not every module works with the sensory data, but every module creates its own model. Overall action is then influenced only by the last module in the chain.

Paradigm of functional decomposition is the *hierarchical paradigm* (see Figure (3.6)(b)). Under this paradigm, the robot senses the world, plans the next action and acts in the environment in manner **SENSE-PLAN-ACT**. At each step, the robot explicitly plans the next move. An advantage of this method is that it can solve the most complex tasks. For solving these tasks, algorithms of theory of the graphs, artificial intelligence or math can be used. A disadvantage is an error distribution. For instance, if the error occurs in the first module, all the following modules will be affected.

3.1.3.3. Deliberative architecture

Deliberative architecture is an mixture of both described methods. It tries to address deficiencies and exploit the advantages of both methods. Most todays robotic systems use this approach, where more complex tasks can be decomposed and solved with functional decomposition approach with model of the environment or with reactive approach.

This architecture uses *hybrid paradigm*, known as **PLAN, SENSE-ACT** (see Figure (3.6)(c)). The planning is done at one step, then sensing and acting are done together in the next step.

3.2 Multi-hypothesis topological mapping

Let we have a graph-like environment like the one shown in Figure (3.7). The environment consists of junctions connected by **straight hallways**. For every discovered junction, the robot stores *junction observation* J_i consisting of a set of leaving hallways $\langle l_1^{[J_i]}, l_2^{[J_i]}, ..., l_n^{[J_i]} \rangle$ and spatial description consisting of spatial relations over the set of observed leaving hallways. For instance, for junction observation J_1 in Figure (3.7) the spatial description could be $\{180°, 270°\}$.

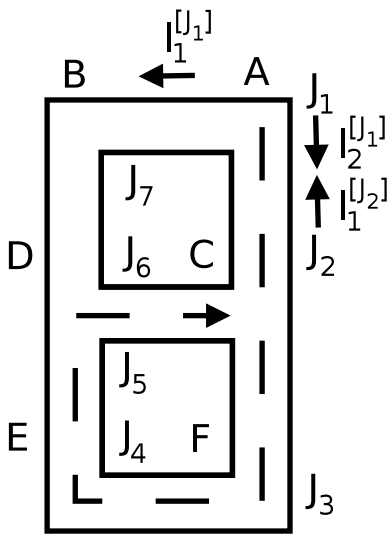


Figure 3.7: An example of graphlike environment

Junction observations are connected by *hallway traversal actions*, which consists of leaving the current junction via one of the observed leaving hallways and arriving at the next junction via one of the leaving hallways belonging to the next junction observation. For instance, traversing the hallway connecting A and C, where $l_1^{[J_2]}$ is observed leaving hallway leading north in J_2 is $l_2^{[J_1]} \rightarrow l_1^{[J_2]}$. The *history* of one specific exploration run through the graph environment is formed by a list $\langle J_1, T_1, J_2, T_2, ..., T_{n-1}, J_n \rangle$ of alternating junction observations J_i and hallway traversals T_j.

The goal of a topological mapping algorithm now is to incrementally process the history of observations and actions and for each step determine all *route graph hypotheses* describing different topological structures of the environment that can be considered as valid explanations of the information traversed so far.
A route graph hypothesis **H** consists of the following [14]:

- a bidirectional graph $G_H = (V_H, E_H)$, where the nodes V_H represent the junction of the environment and the edges E_H represent the hallways
- combinatorial embedding of G_H into the plane (two-dimensional space)
- the starting position S_H of the robot at the beginning of the exploration run

An assumption, about which observed leaving hallways correspond to the same physical hallway and which junction observations correspond to the same physical junction is made by route graph hypothesis. Junction observations assigned to the same node in the hypothesis need to be compatible, meaning that the spatial relations over leaving hallways match (this condition will be further discussed in the next Section). For the given combinatorially embedded graph G_H and the starting position S_H of a route graph hypothesis H, a history clearly induces a corresponding walk through G_H, which directly gives the nodes and edges associated with each junction observation and hallway traversal. It is suitable to store the current position C_H of the robot by recording the node, which corresponds to the last processed junction observation and the edge corresponding to the last hallway traversal. When we want to depict a route graph hypothesis, we have to choose one way of the infinitely many possible ways how to draw the graph into the plane. If it is possible, we do it in a way that retains the spatial relations contained in the junction observations. A route graph hypothesis does not specify a geometric embedding into the plane, it only *restricts* possible geometric embeddings via its combinatorial embedding and the associated spatial relations. There are two ways, how to model the route graph hypothesis, either to model only those junctions that have been visited and perceived, or also make predictions about how are perceived, but not traversed leaving hallways connected [14].

A hypothesis that has been valid so far during exploration, may turn out to be invalid, when the next junction observation is processed. Thus, instead of committing to a single hypothesis, our approach is to keep all valid hypotheses simultaneously. It means, the hypotheses we consider during the mapping process form a tree, like the one shown in Figure (3.8).

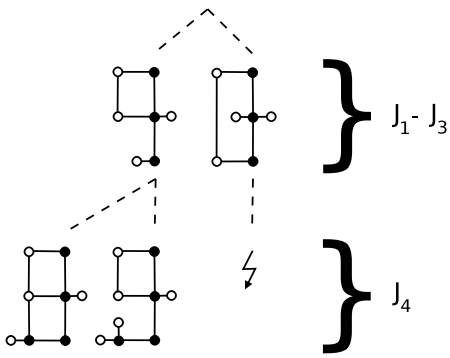


Figure 3.8: Hypotheses generation process, where black nodes are observed junctions and white nodes are junctions, which have not been observed yet

When a new junction observation J_{i+1} and a new hallway traversal action T_i are processed, successor hypotheses are generated for every hypothesis H_j corresponding to the position in the search tree by performing the following steps [14]:

1. "The current position of the robot C_{H_j} within the graph G_{H_j} is updated in accordance with T_i

2. J_{i+1} is matched with the node in E_{H_j} that now corresponds to the updated position. Information about new hallways in J_{i+1} is used to update G_{H_j} by adding new edges. The fact that there may exist multiple ways of how the new edges can be connected to existing nodes in G_{H_j} (or to a completely new node), means that there can be multiple successor hypotheses to H_j in the search tree. In addition, if no way exists to match J_{i+1} witch the current position of the robot, there will be no successor hypotheses and H_j becomes a dead branch in the search tree."

There is an important assumption that the junction observations are given in terms of qualitative cardinal direction calculus, north, west, south, and east [7]. In practice, where the environment itself is not often graph-like, the framework will have to be combined with a certain topological representation approach, which realizes the abstraction from the real environment to the discrete graph structure of the corresponding topological map so that information about perceived nodes and edges can be used with this mapping framework. One approach, how to realize the abstraction is to use *Voronoi diagram*, from which generalized Voronoi graph can be generated.

3.2.1 Minimal route graph models

In this thesis we will interpret the *simplest* hypothesis as hypothesis that contains a minimal number of nodes. For the number of nodes $|V_H|$ in a route graph hypothesis H, for the set of all plausible hypotheses $\mathcal{H}_E$ and for a given exploration history E, we can define the subset $\mathcal{H}_E^*$ as [14]:

$$\mathcal{H}_E^* = \{H \in \mathcal{H}_E \mid \forall H' \in \mathcal{H}_E : |V_{H'}| \geq |V_H|\} \tag{3.8}$$

We will call the elements of $\mathcal{H}_E^*$ *minimal route graph models.*

Here we can use one of two approaches of hypotheses building. One approach consider only the minimal route graph model using $\mathcal{H}_E^*$. The other approach consider all current valid hypotheses from which successor hypotheses are built. But there we need to take into account a fact that the number of nodes in the graph hypotheses grows monotonically with increasing depth in the search tree, because every time new nodes and edges will be added, but never removed.

3.2.2 Valid route graph models

From background knowledge of the route graph models, there are additional constraints which need to be satisfied. Using these kind of constraints is crucial for finding the minimal model algorithm to counteract the exponential growth of the search tree with the length of the exploration history and high degree of ambiguity. As discussed above, the goal is to investigate map learning based on coarse, but reliably observable spatial information and qualitative direction information about

the leaving hallways. Hypothesis is considered as valid, when three conditions are satisfied [14]:

1. "The sequence of actions specified in the history within the hypothetical route graph yields a sequence of node degrees identical to the original sequence of leaving hallway numbers (*structural constraint*)

2. There must be a way how to draw the hypothetical route graph into the plane without crossing edges that is in accordance with the specified combinatorial embedding (*planarity constraint*)

3. There must exist a drawing satisfying condition 2 that at the same time also reproduces the direction relations provided by the original junction observations when repeating the actions in the given exploration history (*direction constraints*)"

3.2.2.1. Structural constraint

The two junction observations J_i and J_k can only correspond to the same node in a hypothesis, if the perceived hallways of J_i and J_j match (with some small tolerance). According to the used approach, the compared junction observation can come either from the history, or from current state.

3.2.2.2. Planarity constraint

A *planar drawing* of a graph is a depiction of the graph to a plane with the vertices at distinct locations with no crossing edges (except at their common vertex endpoints). A graph is *planar* if a planar drawing exists, and a planar embedding is an equivalence class of planar drawings described by the clockwise order of the neighbors of each vertex. There exists the proof, introduced by Kuratowski that a **non-planar graph** must contain a subgraph homeomorphic to either K_5 or $K_{3,3}$, shown in Figure (3.9).

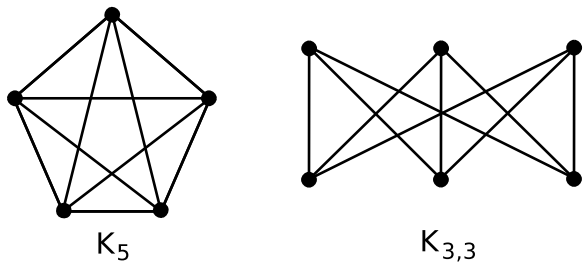


Figure 3.9: The planar obstructions [3]

Indication of a Kuratowski subgraph (a subgraph homeomorphic to K_5 or $K_{3,3}$) provides a simple test of non-planarity, as a planar embedding provides a simple test of planarity. For some applications, finding a Kuratowski subgraph is the first step in eliminating crossing edges in the graph.

While preserving planarity, the edge is the fundamental unit of addition to the partial embedding. *Edge addition* method uses the fact that subgraphs can become biconnected when adding a single edge. It eliminates the partial planarity condition testing of the previous vertex addition approaches supporting a few localized decisions of a path traversal process. Non-planarity is detected, when at the end of step v a back edge from v to a descendants was not embedded [3].

3.2.2.3. Direction constraints

When deciding, whether a given hypothesis is valid or not, we need to determine, if a drawing that satisfies the planarity constraint as well as the direction constraints exists. Such a drawing exists, only if we can assign coordinates to the junctions in such way that all constraints are satisfied. For instance, the drawing in Figure (3.10) is planar, but the positions assigned to the nodes do not reproduce the direction information (refer in next Section) correctly, because there exists a hallway that is supposed to lead east from J_2 and arrive at J_4 from the west. Hence, J_4 should be to the east of J_2. However we have another knowledge that the hallway connecting J_2 with J_3 leads south and the hallway connecting J_3 with J_4 leads leads to the west. Therefore, it can be concluded that J_4 has to be somewhere to the southwest of J_2. As a result we know that no drawing satisfying the direction constraints for this hypothesis can exists, because direction information is inconsistent.

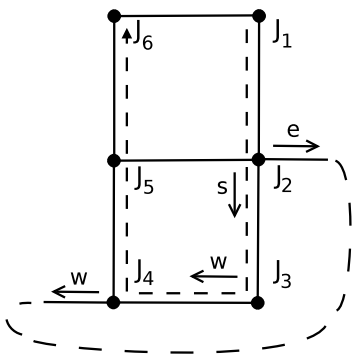


Figure 3.10: An example of invalid graph hypothesis

3.2.2.4. Direction information

Direction information is a calculus, which relates two point objects in the plane. The calculus can be either absolute or relative. The absolute cardinal direction calculus distinguishes nine base relations, the eight direction relations *north, northwest, west*, etc. and the *equal* relation for the case, when both points are equal. This relations can be seen in Figure (3.11)(a). Absolute direction information uses these three conditions [7]:

1. When adding a new edge to a node A, it can only be inserted between those edges, which are in accordance to cardinal directions (*valid direction orderings*)

2. When a new junction observation J is associated with some node A, there need to be correspondences between the leaving hallways of J and edges of A i.e. the direction relation of corresponding leaving hallways of J and edges of A are compatible (*valid junction matchings*)

3. There needs to be a way, how to assign coordinates to the nodes in way that all direction constraints are satisfied (*global consistency*)

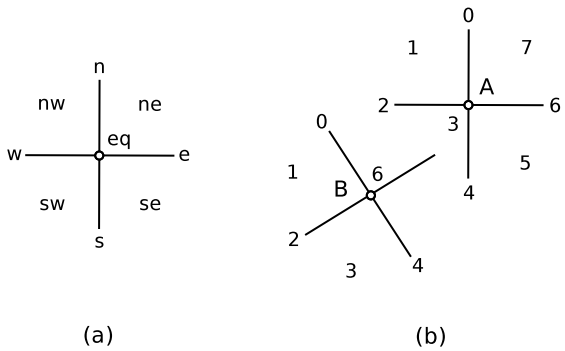


Figure 3.11: Direction information: (a) the absolute cardinal direction calculus and (b) the $\mathcal{OPRA}_2$ calculus

The relative $\mathcal{OPRA}_2$ calculus uses approach that for each of the two objects the plane is divided into eight sectors, as illustrated in Figure (3.11)(b). Each relation states which sector of A contains B and which sector of B contains A. For instance, the relation A $\angle_6^3$ B means that B lies in sector 3 of A and A lies in sector 6 of B. In this case, valid direction ordering cannot be employed, because there is no inherent cyclic order between the base relations. However, other conditions like valid junction matching and global consistency are still usable [10].

3.2.3 Breadth-first search

Breadth-first search is a traversal through a graph that visits all of the vertices reachable from selected source vertex. In general, the order of the traversal is such that the algorithm will explore all neighbors of a vertex before proceeding on to neighbors of its neighbors. Expansion of breadth-first search can be imagine like a wave spreading from a stone dropped into a pool of water. Vertices, which are in the same "wave" have the same distance from the source vertex. A vertex is discovered when it is encountered by the algorithm for the first time. If all neighbors of a vertex was explored, the vertex is marked as finished. An example of the traversal can be seen in Figure (3.12), where s represents the source vertex [9].

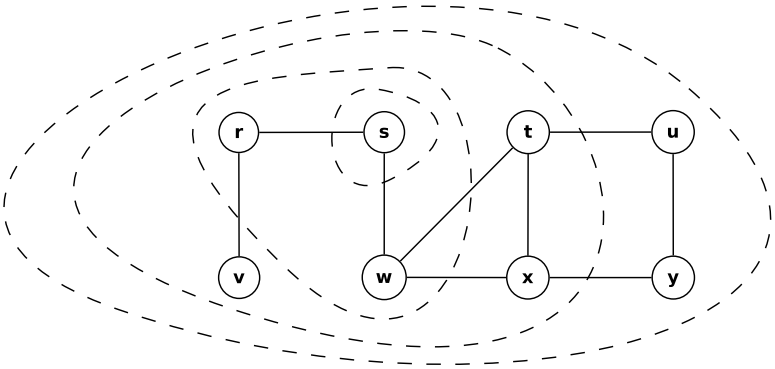


Figure 3.12: Expansion of breadth-first search algorithm

3.3 Robotic frameworks

Robots are complex systems using various types of hardware and software. The approach of programming robots "from scratch" is therefore no longer used. The trend is to divide this system into separable modules which will communicate with each other. An advantage of using modules is that they can be easily changed with no effort. For this purpose, robot frameworks was designed. These frameworks include module and hardware support, and various tools useful for debugging of the system. In the following sections a short description of three mostly used robotic frameworks will be provided.

3.3.1 ROS - Robotic Operating System

Ros is an open-source, multi-platform operating system with hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, writing, building and running code across multiple computers. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a parameter server. ROS is not a real-time framework, though it is possible to integrate ROS with real-time code [6].

ROS has two levels of concepts: the *file-system level* and the *computation graph level*. These levels will be described in the next sections.

3.3.1.1. ROS file-system level

The filesystem level contains ROS resources, where the main resources are the following [12]:

- *Packages*: are the main units in ROS. A package may contain ROS runtime processes, libraries, datasets and so on.
- *Manifests*: provide metadata about package.
- *Message types*: custom message descriptions which define the data structures for messages sent in ROS.
- *Service types*: custom service descriptions which define the request and response data structures for services.

3.3.1.2. ROS computation graph level

The computation graph is the peer-to-peer network of ROS processes that are processing data together. Its basic concept include [12]:

- *Nodes*: are processes that perform computation. They are designed to support modularity of the system,
- *Master*: provides name registration and lookup to the rest of the computation graph.
- *Messages*: data structures comprising typed fields. Standard primitive types such as integer, floating point, and other are supported.
- *Topics*: are transport systems where the messages are routed with publish/-subscribe semantics.
- *Services*: is request/reply system for transporting the messages.

The ROS master stores topics and services registration information for ROS nodes. Nodes communicate with the master to report their registration information. As these nodes communicate with the master, they can receive information about other registered nodes and make connections as appropriate. The master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run. The communication process between two nodes can be seen in Figure (3.13).

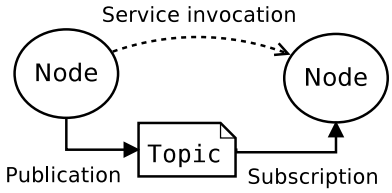


Figure 3.13: ROS nodes communication concept [12]

3.3.2 The Player/Stage project

The project provides the *Player* robot device server and the *Stage* multiple robot simulator, plus supporting tools and libraries. Player provides a clean and simple interface to the robot's sensors and actuators over a network. Client protocol programs talk to Player over a Transmission Control Protocol (TCP), reading data from sensors, writing commands to actuators and configuring devices on the fly. Player supports a variety of robot hardware and provides implementations of sophisticated sensing and control algorithms, such as landmark tracking and probabilistic localization.

Stage provides a population of simulated robots and sensors operating in a two-dimensional bitmapped environment. The devices are accessed through Player, as if they were real hardware. Stage aims to be efficient and configurable rather than highly accurate.

Player and Stage run on many UNIX-like platforms and they are released as free software under the GNU General Public License [5].

3.3.2.1. Player

Player is a socket-based device server that allows control of a wide variety of robotic sensors and actuators. Player executes on a machine that is physically connected to a collection of such devices and offers a TCP socket interface to clients that wish to control them. Clients connect to Player and communicate with the devices by exchanging messages with Player over a TCP socket. Player can support multiple clients connected currently, each on a different socket [5].

3.3.2.2. Stage

Stage simulates a population of mobile robots, sensors and environmental objects. It has two main purposes: to enable rapid development of controllers that will eventually drive real robots and to enable robot experiments without access to the real hardware and environments.

Stage was specifically designed to support research into multi-robot systems. There are several aspects of Stage's design that make it suitable for multi-robot systems [5]:

- *good enough fidelity*: Stage provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity
- *linear scaling with population*: All sensor models use algorithms that are independent of population size.
- *configurable, composable device models*: Various sensors and actuators are provided, including sonars, scanning laser range-finders, visual color segments, fiducial detectors, and a versatile mobile robot base with odometry. The models are often more general and flexibile than any specific piece of hardware, so each model is configured to approximate the (real or imagined) target device.
- *player interface*: all sensor and actuator models are available through Player's standard interfaces.

3.3.3 YARP - Yet Another Robot Platform

YARP is open software robot framework written particularly for humanoid robots. It has a set of libraries, protocols and tools to keep modules and devices decoupled. The framework supports building a robot control system as a collection of programs communicating in a peer-to-peer way supporting connection types like TCP, UDP, multicast, local and others. It also contains flexible interfaces of hardware devices.

This framework has two main parts, *YARP ports for communication* and *YARP devices*. These parts will be shortly described in the following sections [8].

3.3.3.1. Ports for communication

Communication in *YARP* is based on the *Observer* design pattern. A *port* is a module containing special objects delivering messages to observers (other ports), processes, distributed across machines, using several communication protocols. Every port belongs to a process and every connection can use a different protocol and/or physical network. Processes can run on different machines and operating systems using different programming languages [8].

3.3.3.2. Devices

The devices in *YARP* implements specific drivers for particular devices and defines interfaces for device families. Device interfaces should be generalized because of minimizing impact of changing the device. Then the device should implement network wrappers for interface which give the further flexibility for scaling computer cluster, isolating hardware devices that does not play well together. A driver in *YARP* is *C++* class, interface are abstract base classes, and network wrappers are special cases of devices that use network resources to satisfy their interfaces [8].

Chapter 4

Solution

4.1 Selected resources

The resources used in our implementation was chosen to get the best results with the smallest effort. As a robot framework was selected *ROS* because of its capabilities, robustness and performance. Also strong documentation and support by wide community played a big role in decision process.

As a programming language was chosen *C++* because of its availability of abstraction and speed of compiled program. Then we have selected the BOOST *C++* library as a method for representing the graph. This library has many features, tools, variabilities and implemented algorithms what makes it suitable for fulfilling our needs.

4.2 Modules

The complex problem stated in Section (2) is divided into three separate parts. These parts are implemented as modules: *Junction detector*, *Navigator* and *Planner*. The division was designed to have the ability to replace *Junction Detector*, when using other sensors, or to replace *Navigator*, when using another robot hardware. An advantage of this approach is that the modules are absolutely separated and every module is responsible only for one activity, communicating with other modules via specified messages. The modules represent *ROS* nodes using subscribers to get input messages and publishers to provide output messages. The topology of these modules with their connection and transferred messages can be seen in Figure (4.1):

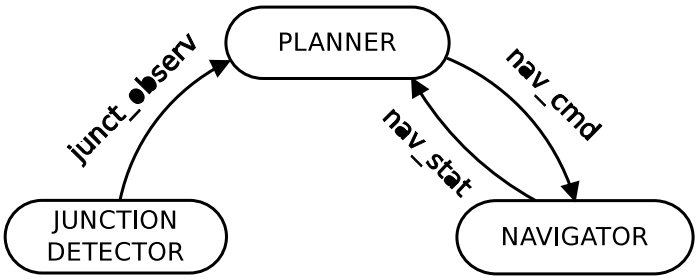


Figure 4.1: The topology of the modules

The *Planner* is the highest layer in the topology using other two subordinated layers the *Junction detector* and the *Navigator*. The highest layer acquires some information about current junction from the *Junction detector* and updates its imagine about how the environment looks like. Then planes the trajectory of the robot and acts the environment using the *Navigator*, which also gives a feedback to it.

In the next Sections, every module will be described in detail.

4.3 Junction detector

The junction detector is responsible for detection of junctions and it is done with finding free hallways, which are wide enough for the robot to pass through. The detection is done **on-line** from the current robot position. Every time, when data from *Hokuyo laser* are prepared, they are passed into this module via *base link* message. The data are processed and the result is then published **continuously** regardless of any change.

The junction detector module was written as *C++ class* having no special properties. Instead, it has methods which process the data from the laser and produce the output. These methods will be discussed in the next Sections.

4.3.1 Initializing the junction detector

Since this module does not have special properties, the initialization process lies in subscribing this module for taking of the laser data received from *ROS topic base_link*. Then the publisher *junct_observ* is registered as the new *ROS topic*.

4.3.2 Getting free middle angles

As described in Section (2.2), the environment we are dealing with is **perpendicular**. Thus, we can use this fact in finding free middle angles. Data from the laser contain a scanned distances from obstacles in all available angles. The scan range is divided into a set of segments $S = \{s_0, s_1, ..., s_n \mid s_i \in \mathbb{R}\}$, where segment s_0 has the middle angle $\phi_m = -90°$, segment s_n has the middle angle $\phi_m = 90°$. The step between the middle angles of two following segments is 90°. Each segment has an angle range $\Phi_s = \langle \phi_1, \phi_2 \rangle$ where the widest hallway is searched. Note that $\forall\ s_i, s_j \mid s_i \cap s_j = \emptyset$ is needed, because we want to eliminate a situation, when one

hallway at the end of the segment is found and the same hallway at the start of next segment is found. Threshold between angle ranges of two following segments, where the hallway is not detected was set to FREE_AREA_THRESHOLD (refer Section (5)). Then, the algorithm for finding the middle angles of hallways for every segment is the following:

Algorithm 1: Finding the middle angles of hallways for every segment

Data: S, Φ
Result: a list r containing the middle angles of hallways

```
initialize r;
foreach s ∈ S do
    φ1 = φm − 40°;
    φ2 = φm + 40°;
    initialize area_max;
    foreach φ ∈ Φs do
        check a hallway width in φ;
        find area_max;
    if area_max found then
        store the middle angle of area_max to r;
return r;
end;
```

4.3.3 Checking width of a hallway

Let us assume an angle ψ_m, in which the width of a hallway is being checked and let us assume a strip containing an interval of angles $\Psi = \langle \psi_1, \psi_2 \rangle$ and its ranges $R = \langle r_1, r_2 \rangle$, where ψ_1 is angle $\psi_m - 90°$ and ψ_2 is angle $\psi_m + 90°$. In fact, the strip represents the set of positions of the robot on the line of angle ψ_m starting from the robot current position and ending at maximal sense range MAX_SENSE_RANGE (refer Section (5)). Then, the hallway width is calculated by the following algorithm:

Algorithm 2: Calculation of the width of a hallway

Data: ψ_m, $range_{max}$, and $width_{min}$
Result: true, if a hallway has minimal width, false otherwise

```
ψ1 = ψm − 90°;
ψ2 = ψm + 90°;
foreach ψ ∈ Ψ do
    r = rψ;
    if r < rangemax then
        d = |r sin(ψ − ψm)|;
        if d < widthmin then
            return false;
            end;
return true;
end;
```

Situation described by above algorithm can be seen in Figure (4.2).

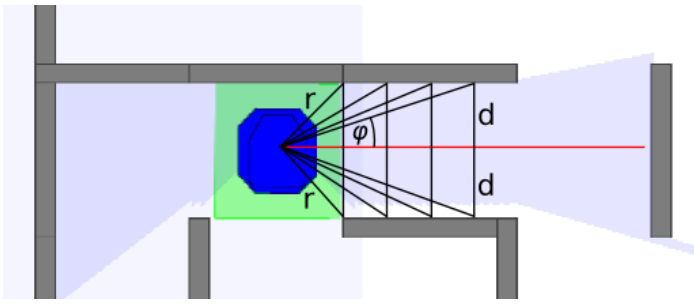


Figure 4.2: Calculation of the width of the hallway

4.3.3.1. Laser data callback

When the data from the laser are prepared and passed to this callback, the method for getting the free middle angles is called. This method, as was described above, ensures that the relevant list containing these data will be returned. Then, this result is stored in the *ROS topic message*. The message has layout of the standard message *Float32MultiArray*, where the data part contains the middle angle of every segment. Listing of the message has the following format:

```
MultiArrayLayout   layout        # specification of data layout
float32[]          data          # array of data
```

4.4 Navigator

The navigator module is responsible for moving and rotating the robot through an environment. It has a mechanism for avoiding collisions with obstacles and mechanism for following the wall on one side, which is currently sensed. If no wall is within a sensed range, the robot maintains the last trajectory. This module does not do any action itself, it only executes commands received from higher layer - Planner (refer the Section (4.5)). Similarly like the module *Junction detector*, it uses data from *Hokuyo laser* but moreover, it also uses the data from the robot odometry for local localization and publishes the velocity of the navigator. The module was written as a *C++ class* with these properties:

- *current angle* - an absolute angle of the robot rotation
- *parameters of the rotation* - speed of the rotation and a desired angle
- *parameters of the movement* - speed of the move, starting position, and distance
- *correction state* - a reference to a state machine correcting the trajectory of the robot
- *navigator local parameters* - wall angles, distance from walls and the minimal range in the front of the navigator
- *navigator status* - set of actions the robot is currently performing

There are also methods, which work with these parameters. Their function will be discussed in the next sections.

4.4.1 Initializing the navigator

In the initialization process, the current angle of the navigator is set to 0° and parameters of the movement and of the rotation are reset. The navigator status and correction state are set to default values. Then the subscribers for the laser, the commands from the *Planner*, and the odometry are attached to *ROS topics base_link, nav_cmd,* and *odom*. Finally, the publisher for velocity commands is registered as common *ROS topic cmd_vel* and the status is registered as the new *ROS topic nav_stat*.

4.4.2 Laser data callback

Every time, when the data from the laser are prepared and delivered via message, the local parameters of the navigator are updated. These parameters include: *right/left wall angle, a distance from right/left wall, and the minimal range in the front of the robot*. The right wall angle ϕ_r and the left wall angle ϕ_l is obtained from a scan on a particular side. These angles are used for measuring of the distances d_r and d_l of the robot from the walls. Now, the minimal range from the obstacle r_{min} in the front of the robot is searched. If r_{min} is less than MIN_OBSTACLE_RANGE

(refer Section (5)), the navigator is stopped. In the next step, a delta angle $\Delta\phi_r$ on the right side, $\Delta\phi_l$ on the left side, a delta distance Δd_r from the right side, and Δd_l from the left side are computed for the both sides using the following equations, where d_m is the followed distance from the wall:

$$\Delta\phi_r = \frac{\pi}{2} - \phi_r \tag{4.1a}$$

$$\Delta\phi_l = \phi_l - \frac{\pi}{2} \tag{4.1b}$$

$$\Delta d_r = d_m - d_r \tag{4.1c}$$

$$\Delta d_l = d_l - d_m \tag{4.1d}$$

If the right wall is in the sense distance range, the parameters on the right side are considered as $\Delta\phi = \Delta\phi_r$ and $\Delta d = \Delta d_r$. If the left wall is detected, the parameters on the other side are $\Delta\phi = \Delta\phi_l$ and $\Delta d = \Delta d_l$. Otherwise, the correction process is not used, because there is no reference. $\Delta\phi$ and Δd are then passed to the correction state machine. This state machine ensures that the robot will follow the wall in distance FOLLOW_DISTANCE (refer in Section (5)) if the wall is detected. When the robot is closer to a wall on the right or the left side, it rotates away from the wall, moves to the desired distance and rotates back. The states of the correction state machine are shown in Figure (4.3).

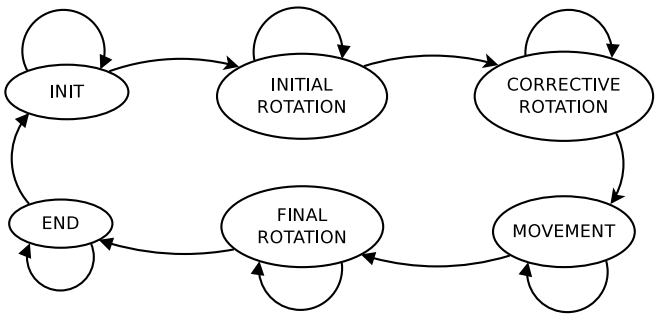


Figure 4.3: The correction state machine

In the following paragraph, this state machine will be described.

- *INIT* - Δd is compared with the maximal delta range Δd_{max}. If is greater than Δd_{max}, the state machine is activated and the next state will be *INITIAL ROTATION*. If not, then the correction is not needed and no next action is performed.

- *INITIAL ROTATION* - the angle $\Delta\phi$ is compared with the maximal delta angle $\Delta\phi_{max}$. If is greater than $\Delta\phi$ the robot is first rotated to have the wall angle. This action is important because greater distance from the wall could be caused only by a bad rotation. The navigator remains in this state until the rotation is finished. Then the next state is *CORRECTIVE ROTATION*.

- *CORRECTIVE ROTATION* - Δd is again compared with Δd_{max}. If Δd is less than Δd_{max}, the robot is on the right position and correction process is ended with passing to state *END*. If not, the correction angle ϕ_C is calculated according to the difference between Δd and R_R using:

$$\phi_C = \arcsin \frac{\Delta d}{d_c} \tag{4.2}$$

  if Δd is less than R_R, where d_c is the correction distance, or using:

$$\phi_C = \arcsin \frac{R_R}{d_c} \tag{4.3}$$

  if Δd is greater than R_R. This state is held until the rotation is finished. Then the next state is *MOVEMENT*.

- *MOVEMENT* - This state ensures that the robot will move at the angle ϕ_C until Δd is greater than Δd_{max}. Then, the next state will be *FINAL ROTATION*.

- *FINAL ROTATION* - Now, the robot is in the desired distance from the wall, but it is still rotated to the angle ϕ_C. Thus, the robot is turned to have the wall angle. The state is again held until the rotation is finished. After that, the next state is *END*.

- *END* - The correction process is now finished and the state machine is returned to the state *INIT*.

4.4.3 Command callback

The *Navigator* is controlled by the *Planner* using the command message. The robot can be commanded only if it is not doing any action, i.e. its status is *READY*. However, there is the one exception that the movement of the robot can be stopped even its status is not *READY*. A list of available commands with their description is the following:

- *STOP* - stops the robot by sending the zero velocity. This is applicable only for movement.

- *MOVE* - moves the robot forward with given speed. Note that this command is not necessary to send periodically, because the robot will move until the *STOP* command is received. The move is either continuous or short-time, depending on the parameter.

- *ROTATE* - rotates the robot to the absolute angle specified by the parameter. In this case the angular speed around the axis z is sent, where this axis is leading up from the robot. The rotation process can not be stopped.

- *ROTATE RELATIVELY* - rotates the robot relatively by an angle specified by the parameter.

The commands are transferred through modules via custom message. The message has the following form:

```
uint8 code            # the command code, where:
                      # STOP = 1, MOVE = 2, ROTATE = 4,
                      # ROTATE RELATIVELY = 8
float32 param         # used only for commands ROTATE and
                      # ROTATE RELATIVELY
```

4.4.4 Odometry callback

The odometry data message is periodically published by the *ROS* odometry topic of the robot. The message contains the current pose and the current twist of the robot. The orientation we are interested in, is in a quaternion form. Thus it needs to be converted to a regular angle ψ_C. For this purpose the *ROS transform "tf"* is used. Then if the rotation and its desired angle ψ_D is required, ψ_C is compared with ψ_D and the robot is rotated publishing the angular velocity until $\psi_C \approx \psi_D$ is reached. Note that there we need to take into account a fact that the accuracy of the odometry is not ideal, therefore reaching ψ_D is within MIN_ROTATION_DANGLE (refer Section (5)). If the movement of the robot is required, then the linear velocity is published via specified message. If a distance for the movement is specified, then the robot measures traveled distance comparing the start pose and the current pose P obtained from the odometry message. During this process, velocity message is published. Finally, the robot status is published according to the current action. The robot has these status types:

- *EMERGENCY STOPPING* - if the robot was stopped because of an obstacle standing in critical distance in the front of the robot,
- *READY* - if the robot is not performing any action,
- *MOVING CONTINUOUSLY* - if the robot is performing continuous move,
- *MOVING* - if the robot is moving in desired distance,
- *ROTATING* - if the robot is changing its orientation according to specified angle.

In addition to this status action the robot also measures its distance to the closest obstacle in the front and in the back. This process is described in detail in Section (4.4.8).

The custom message containing the robot status has the following form:

```
uint8 code                  # the status code, where:
                            # EMERGENCY STOPPING = 1, READY = 2,
                            # MOVING = 4, MOVING CONTINUOUSLY = 8,
                            # ROTATING = 16
float32 param               # the absolute current angle
float32 frontDistance       # distance to the closest obstacle in
                            # the front of the robot
float32 backDistance        # distance to the closest obstacle in
                            # the back of the robot
```

4.4.5 Getting a wall angle

A wall angle is angle between the x-axis of the robot and the wall. Let denote the angle of the side S by γ, range in this angle by r, range in the front of this side by a and range in the back of this side by b. Then the wall angle in the front side β_f is calculated using the law of cosines:

$$c_f = \sqrt{r^2 + a^2 + 2rb\cos\gamma} \tag{4.4a}$$

$$\beta_f = \arccos\frac{r^2 - a^2 + c_f^2}{2rc_f} \tag{4.4b}$$

and the wall angle in the back side β_b is calculated using:

$$c_b = \sqrt{r^2 + b^2 + 2rb\cos\gamma} \tag{4.5a}$$

$$\beta_2 = 90^\circ - \arccos\frac{r^2 - b^2 + c_b^2}{2rc_b} \tag{4.5b}$$

Finally, the wall angle of S is considered as valid if $|\beta_f - \beta_b| < \epsilon$ where ϵ is less than MAX_WALL_DANGLE (refer Section (5)).

4.4.6 Getting a wall distance

When the wall angle β_w on the side S is detected, then the calculation of the distance from the wall on S is simple, since β_w is perpendicular to the wall. In this case the distance can be found simply with getting the range at the angle β_w of the scan.

4.4.7 Getting the minimal range

The minimal range from the obstacle is checked at the front part of the robot to avoid the collision. Firstly, the angle range of the front part must be intended. Let us assume the angle range Ψ, the minimal range from the obstacle r_m and the radius of the robot R_R. Then Ψ is calculated using:

$$\Psi = \arctan \frac{R_R}{r_m} \tag{4.6}$$

Finally the minimal range r_{min} is searched in the scan L by the following simple algorithm:

Algorithm 3: Finding of the minimal range

Data: Ψ and L
Result: the minimal range r_{min}

```
set r_min to maximal range of L;
foreach ψ ∈ Ψ do
    r := get range in ψ;
    if r < r_min then
        r_min := r;
return r_min;
end;
```

4.4.8 Getting the minimal distance

The minimal distance is mainly used for measuring the distance from the obstacle in the front or in the back of the robot. These distances are measured for the left and the right side of the robot. For the left side, the side angle ψ_S is set to 90°, for the right side, ψ_S is set to −90°. The range Ψ is for the back distance given by BACK_DISTANCE_CHECK_RANGE and for the front distance given by FRONT_DISTANCE_CHECK_RANGE with sign set according to the side it is measured in. Then $\forall\ \psi \in \Psi$ distance from the obstacle is calculated. Finally, the minimal distance d_{min} is returned.

Algorithm 4: Finding of the minimal distance

Data: Ψ and L
Result: the minimal distance d_{min}

```
set d_min to MAX_SENSE_RANGE;
foreach ψ ∈ Ψ do
    r := get range in ψ;
    d := r|sin(ψ − ψ_S)|;
    if d < d_min then
        d_min := d;
return d_min;
end;
```

Note that d_{min} can be 0 if the closest obstacle is in angle ψ_S. This distance is considered as invalid and no considered in further calculation process. Finally, the minimal distance of the left or the right side is marked as overall minimal distance.

4.5 Planner

The planner has a task to build a map of an environment, localizing in this map and planning a new trajectory for the robot. This module is considered as the highest layer in the module topology using services of modules *Junction detector* and *Navigator*. The new trajectory is planned every time when a junction is reached. While the robot is moving across an unknown environment, hypotheses of its appearance are created and destroyed. The planner module was written as a *C++ class* having the following properties:

- *step* - a number of visited junctions
- *maximal ID of graph hypothesis* - a number of last created hypothesis
- *previous junction observation* - a list of middle angles of free hallways
- *leaving hallway* - an angle of the current leaving hallway
- *robot state* - a reference to a robot state machine
- *robot parameters* - a list of parameters used within a state
- *graph hypotheses* - a priority queue containing all the current hypotheses sorted by number of the nodes
- *navigator status* - a state published by module *Navigator*

Then, the planner class has methods which operate among described properties. Bigger part of the program logic is stored in callback methods. These methods will be discussed in the next Sections.

4.5.1 Initializing of the planner

In the initialization process, properties *step* and *maximal ID of graph hypothesis* are set to zero meaning that no hypothesis has been created yet. The robot state property is there set to default value. Then, the subscribers for the junction observation and the navigator status are attached to *ROS topics junct_ observ* and *nav_ stat*. From this, every time when data of any publisher will be prepared, the corresponding callback will be called. Finally, we need to register new *ROS topic*, for publishing commands of the *Navigator*, *nav_ cmd*.

4.5.2 Junction observation callback

This function is called every time when the data from the module *Junction detector* was published. In this case, the junction observation O contains the middle angles of all the free hallways. O is now relatively rotated according to the direction of the robot, but our approach is to store O so that the angle $0°$ always leads the **east**. An advantage of this approach is then simpler comparison of two junction observations. Thus O is rotated to the absolute position and the rotated junction

observation O_r is then processed by the planner state machine. The planner state machine contains all states used when mapping the environment. These states can be seen in Figure (4.4).

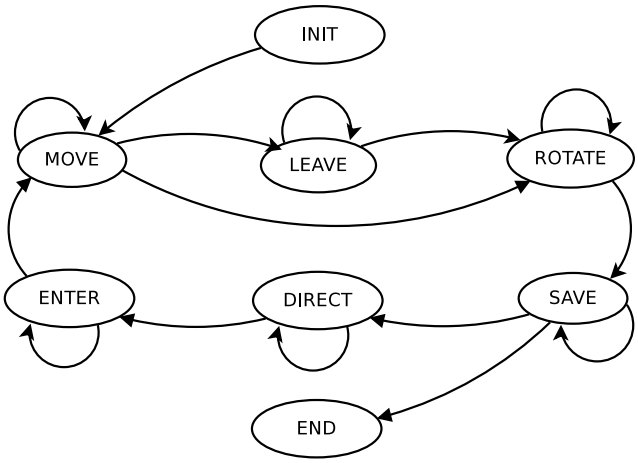


Figure 4.4: The planner state machine

In the following lines, this state machine will be described. Note that before entering to any state of the state machine the robot status is checked because the robot must be prepared to execute any action required by a state. If the robot is waiting for acceptance of desired command or if it is still doing some action requested from the previous state, the state machine is in the closed loop until these actions are finished.

- *INIT* - the previous junction observation O_p is cleared and the leaving hallway l is set to 0°.
- *MOVE* - the change of O_r is monitored. If O_r has changed, R is immediately stopped and the next state is *LEAVE*. If R reached dead end, it is automatically stopped and the following state is *ROTATE*. Skipping of state *LEAVE* in the other case is needed because there is no free space in the front of the robot. The state is active until one of these situations occurs.
- *LEAVE* - leaves the current hallway by traversing a distance obtained from the *Navigator* status message and parameter *back distance* if measured. The state machine remains in this state until movement is finished. The movement is done using the navigator *MOVE* command message with specified distance.
- *ROTATE* - at the first, this state ensures that the current angle, l and O_r as O_s is stored. Then, the robot is rotated by 90° in direction according to a junction. If the junction has free hallways on the right side from R, it is rotated to the right. Otherwise R is rotated to the left. The rotation is performed using the navigator *ROTATE* command message. This state is held until the rotation is finished. Then the next state is *SAVE*.

- *SAVE* - merges O_s with the current junction observation O_c. The merge process lies in searching the difference between O_s and O_c. The missing part of O_s contained in O_c is added to O_s. After that, all collected information are saved and graph hypotheses are generated. After this process, the minimal path to a undiscovered node is calculated (refer the Section (4.5.7)). If the path was found, R is rotated to the closest leaving hallway and the next state will be *DIRECT*. Then this state is active until the rotation is finished. If no path was found, the state machine will end in state *END*, because there is no undiscovered leaving hallway. This case occurs when the environment was completely mapped.

- *DIRECT* - rotates the robot to the middle angle α_{l_f} of the first leaving hallway l_f in the path. It is done by getting the current junction measurement searching a hallway with the middle angle α_l related with α_{l_f} as $|\alpha_l - \alpha_{l_f}| < MAX\_HALLWAY\_DANGLE$. The rotation is again performed using *ROTATE* command message. The state machine holds this state until the rotation is finished.

- *ENTER* - if the measurement of distance to the closest hallway in the front is available from *Navigator* status message, the movement by distance specified in parameter *front distance* is performed. Otherwise R is moved by distance equal to its diameter. This is considered as the minimal entering distance. This state is then held until movement is finished. After that, the planner state machine passes to state *MOVE*.

- *END* - the last state which is held until the end of the program.

4.5.3 Navigator status callback

The navigator status callback function is called periodically when the data are published by the module *Navigator*. The purpose of this function is only to save the navigator status used in the robot state machine.

4.5.4 Graph hypotheses

Graph hypotheses submodule is the main part of this work, responsible for generating and managing the graph hypotheses while the robot is moving through environment. It does not use any ROS node, or topic, but it uses the submodule *Direction information* (refer in the Section (4.5.5)) and it is used by the *Planner*.

As discussed in the Section (3.2), the graph hypothesis consists of a graph having specific features. According to these features and the features resulting from our needs, the graph hypothesis was written as a *C++ class* with these properties:

- *id* - a unique number of the hypothesis
- *step* - a number of visited junctions (when the hypothesis was created)
- *the last node* - a reference to the last inserted, or connected node
- *the graph* - the BOOST adjacency list with bidirectional edges, where the edges and the vertices have specific properties and both of them are stored in the standard *C++ vector*

The edge property consists of the angle of the **leaving hallway** and the **direction information**. The vertex property consists of the **name** of the vertex (used when visualizing the graph) and the **junction observation**, where the middle angles of all free hallways are stored.

The class also contains a methods, which operate among these data. There are the main methods and methods, which just support the functionality. The main methods will be described in the following sections.

There is an important note that certain parts we have implemented in different way as described in the article [14]. The first part is that we have chosen another way of building the graph hypothesis H like discussed in the Section (3.2.2). An approach according to the article is to build H from the *history*. Our approach is to do it **incrementally**. This brings several benefits, the speed at the first place. In original implementation, H (and its history) is completely rebuilt at every history addition, but in our implementation we build and extend only the current hypotheses. Another benefit of our approach is that we do not need to check the *structural constraint* when creating a new node, because H is built incrementally. The detailed implementation of this approach will be discussed in the next Sections. The next approach we have done in different way is the representation of the *direction information*. For the relative calculus was originally used the $\mathcal{OPRA}_2$, but we have developed a simpler calculus, using two-dimensional space coordinates. Our calculus will be discussed in the Section (4.5.5).

4.5.4.1. Creating a new hypothesis

When creating the new graph hypothesis, the properties *id, step, and a source graph hypothesis* are available from the *Planner*. The first two properties are simply stored and the other property is used as a reference for the new graph hypothesis. It means that all edges, vertices, and its properties from the source hypothesis must be

copied, because the new graph hypothesis is expanding the old one. Finally, we also need to copy the reference to the last inserted or connected node from the source hypothesis.

4.5.4.2. Creating a new node

For creating the new node N_n, the junction observation O_n and the leaving hallway l_n must be available. These data are stored as the node property P_e. The item *name* of the node property is set according to the number of the vertex in the graph. When connecting N_n with the last node N_l (if available), the constraints have the following meaning:

- *structural constraint* - not needed, because we are building H incrementally
- *planarity constraint* - not needed, because we can place N_n so that no crossing edge exists
- *direction constraints* - only one edge can lead from N_l to N_n in the specified angle

Then whole creation process can be described by the following algorithm:

Algorithm 5: The process of creating the new node

Data: O_n, l_n, N_n, N_l, P_e and G
Result: true, if an addition is valid, false otherwise

```
add N_n to G;
if N_l exists then
    if l_n ∉ O_n then
        return false;
        end;
    if ∃ connection from N_l in l_n then
        return false;
        end;
    create connection between N_l and N_n;
store P_e;
N_l := N_n;
return true;
end;
```

4.5.4.3. Connecting with an existing node

When connecting with an existing node, the last node N_l is a reference to the last inserted or connected node and it needs to be connected with another target node N_t. The junction observation is then a new junction observation O_n of N_t. The old junction observation O_t is stored in the property of N_t. As the robot was moving and stopped at this junction, the leaving hallway l_n is known and a direction information

d_{l_n} for l_n is calculated. This information is then stored in edge property.
When connecting N_l with N_l all constraints must be fulfilled. Meaning of the constraints in this situation is the following:

- *structural constraint* - O_n and O_t must match
- *planarity constraint* - the graph without crossing edges
- *direction constraints* - the direction information through all nodes in the path between N_l and N_t must be zero

As discussed above, direction constraints must be fulfilled, but there is a problem that the BOOST graph does not consider a direction information. Thus, the planarity check will be true even for some graph although it is clear that no depiction without crossing edges exists. An example of such graph is shown in Figure (4.5) where the nodes N_1 and N_9 are trying to be connected.

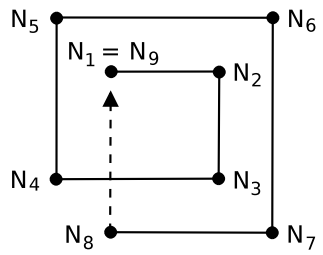


Figure 4.5: An example of non planar graph

Hypothesis in Figure (4.5) will be considered valid, which is not so big problem, since on every step the only minimal hypothesis is expanded. The algorithm describing the connection process is the following:

Algorithm 6: The process of connection with an existing node

Data: O_n, N_t, N_n, l_n and G
Result: true, if a connection is valid, false otherwise

```
1  if structural constraint for N_t, O_n, l_n is not fulfilled then
2      return false;
3      end;
4  if direction constraints for N_t, l_n are not fulfilled then
5      return false;
6      end;
7  if ∄ edge between N_l and N_n then
8      if ∃ connection from N_l in l_n then
9          return false;
10         end;
11     create connection c between N_l and N_n;
12     if planarity constraint is not fulfilled then
13         return false;
14         remove connection c;
15         end;
16 N_l := N_n;
17 return true;
18 end;
```

4.5.4.4. Checking the structural constraint

As discussed in the Section (4.5.4), we have implemented a different approach of building the graph hypothesis H. Thus our structural constraint has a quite different function since it is checked only when connecting the two existing nodes. In this case, the source node N_s (known as the last node), the target node N_t, the source junction observation O_s, the target junction observation O_t, the new target junction observation O_n, and the leaving hallway l_n is known. The main idea of our structural constraint is only to check, whether O_t and O_n match. The algorithm ensuring this functionality is the following:

Algorithm 7: Checking the structural constraint

Data: O_t, O_n, O_s, N_s, N_t, l_n, and G
Result: true, if the constraint is fulfilled, false otherwise

```
l_n^R := l_n + π;
get O_s from G at N_s;
get O_t from G at N_t;
if l_n ∉ O_s then
    return false;
    end;
if l_n^R ∉ O_t then
    return false;
    end;
if O_t ≉ O_n then
    return false;
    end;
return true;
end;
```

Where the symbol $\not\approx$ means that the angles from both junction observations differ more than given threshold MAX_HALLWAY_DANGLE (refer Section (5)).

4.5.4.5. Checking the planarity constraint

For the planarity check we have used *Boyer-Myrvold planarity testing/embedding* described in the Section (3.2.2.2.). The BOOST library offers the function with the same name testing if the given graph is planar or not. Moreover, a planar embedding can be constructed for planar graph, or for non planar graph the minimal set of the edges that form a *Kuratowski subgraph* can be found. We are using only the main functionality of this function what is the planarity check.

4.5.4.6. Checking the direction constraints

The direction constraints are tested mainly when connecting the two nodes in the graph. In this case the constraints lie in the check that the sum of all direction information in a path between the source node N_s with the leaving hallway l_t leading to N_t and the target node N_t is zero. By other words, the direction constraints check if the robot has passed a closed loop. For getting the path the breadth-first search algorithm is used (refer in the Section (3.2.3)). This algorithm searches a path between N_s and N_t and records predecessors P for all nodes in the path. Then the path is traversed and the sum d_s of all direction information is calculated. The situation is described by the following algorithm:

Algorithm 8: Checking the direction constraints

Data: N_s, N_t, l_t, P, d_s, and G
Result: true, if the constraint is fulfilled, false otherwise

```
l_t^R := l_t + π;
d_s := l_t^R;
record P in breadth-first search from N_t;
p := N_s;
while p ≠ N_t do
    get direction info d_p for p from G;
    d_s := d_s + d_p;
    p := P[p];
if d_s = ∅ then
    return true;
else
    return false;
end;
```

4.5.4.7. Getting a path

In this work, we have implemented two types of algorithms used for getting the path between the last (current) node and an undiscovered node. The undiscovered node is node which has only the one leaving hallway known from its adjacent node. The first type of used approach is **Depth-first search** algorithm and the other type is **Breadth-first search**. These algorithms differ in the way of finding the first undiscovered node from the current node, where the first visits all the nodes in a branch until the one or end is finished, and the other visits all nodes in a layer starting from the root of the graph. The detailed description of both variants is on the following paragraphs.

Getting the path using DFS algorithm This search is firstly started at the last node N_l. If N_l has at least the one leaving hallways undiscovered, the path is found and the middle angle of the leaving hallway is returned. If not, then the breadth-first search algorithm is used for getting all the adjacent nodes of N_l, and recording distances D and predecessors P of all nodes. Then for every distance N_t is searched. If found, then the path from N_l to N_t is calculated. If not, the next N_t is searched and this process is repeated until any N_t is found. Note that there can occur a situation when no N_t can be found. Then is clear that all the nodes have been discovered which implies that the environment was completely mapped. The whole process can be described by the following algorithm:

Algorithm 9: Finding the path using DFS algorithm

Data: N_l, P, D, and G
Result: a list r containing the path

```
clear list r;
search the free leaving hallway l_f in N_l;
if l_f found then
    add l_f to r;
    return r;
    end;
record P and D in breadth-first search from N_l;
foreach d ∈ D do
    foreach node N_t in d do
        search the free leaving hallway l_f in N_t;
        if l_f found then
            add l_f to r;
            p := N_t;
            while p ≠ N_l do
                get the leaving hallway l_p for p from G;
                add l_p to r;
                p := P[p];
            reverse r;
            return r;
            end;
return r;
end;
```

Note that if the path was found for some N_t, r must be reversed since we have got P recorded from N_l. According to the result r we can recognize if the path exists and ultimately, we know if the environment was completely mapped or not.

Getting the path using BFS algorithm The undiscovered node is there searched from the node N_0 representing the root of the graph using breadth-first algorithm with recording distances D of all nodes. The search process of a undiscovered hallway is done in layers for $\forall\ d \in D$, starting from N_0. In every layer, all the nodes with distance d are searched. Then $\forall\ N_t \in d$, a free leaving hallway l_f is checked. If found, the path C between N_l and N_t is calculated and C is added into priority queue Q sorted by the number of nodes in the path. All paths between nodes N_t having l_f and N_l are added to Q and at the end of search process node with the minimal path is returned as the nearest undiscovered node.

Algorithm 10: Finding the path using BFS algorithm

Data: N_l, P, D, and G
Result: a list r containing the path

```
clear list r;
record P in breadth-first search from N_0;
foreach d ∈ D do
    foreach node N_t in d do
        search the free leaving hallway l_f in N_t;
        if l_f found then
            C := get the path between N_l and N_t;
            store C in Q;
    if Q ∉ ∅ then
        r := the top of Q;
        return r;
        end;
return r;
end;
```

4.5.5 Direction information

The direction information calculus in our case is used in different way like discussed in the Section (4.5.4). We use the x, y coordinates to relate the two objects in the plane. Let us assume an angle α for which the direction info d_α is being calculated. Let us assume the number of segments n representing the division the angle range 2π into n equal parts p_i of range r and its half range r_h so that $\forall\ i \in n \mid p_i = ir + < \frac{-r}{2}, \frac{r}{2} >$. Then we calculate the half segment s_h using:

$$s_h = \frac{\alpha}{r_h} \tag{4.7}$$

and calculate the full segment s_f using:

$$s_f = \frac{\alpha}{r} \tag{4.8}$$

From s_f and s_h only the integer part is considered and s is calculated using:

$$s = (s_h - s_f) \bmod n \tag{4.9}$$

Now the x, y coordinates are calculated in this way:

$$x = \cos sr \tag{4.10a}$$

$$y = \sin sr \tag{4.10b}$$

where again only the integer part of x, y are considered.

4.5.6 Generating the hypotheses

As discussed in the Section (4.5), the graph hypotheses are stored in the *C++* priority queue Q_H sorted by number of the nodes. The graph hypothesis is a class containing the functionality described in the Section (4.5.4). When generating the hypothesis the junction observation O, the leaving hallway l and the step s are available. If no hypothesis exists in Q_H, the first hypothesis and its new node is created. This hypothesis is saved in the new vector Q_H^*. If more hypotheses exist in Q_H, then every hypothesis $H \in V_H$ is expanded in manner of new node being created and the last added or connected node N_l of H is tried to be connected with each other nodes of H providing O and l. O in this case represents the new junction observation of a node and l represents the leaving hallway from N_l. Note that H with the new node, or with the connected node is generated as the new hypothesis H_n at step s with new unique ID. Then the validity of H_n is checked. If H_n is valid, it is stored in Q_H^* and the maximal ID of graph hypothesis ID_{max} is incremented. If is not valid, H_n is destroyed and no longer considered in the expansion process. Finally, Q_H is replaced with Q_H^* which ensures that the old hypotheses are deleted and on the next turn only new hypotheses will be expanded. This process is described in detail by the following algorithm:

Algorithm 11: The process of generation the new hypotheses

Data: Q_H, Q_H^*, s and ID_{max}
Result: Q_H^* containing the new valid hypotheses

```
clear Q*_H;
if Q_H ∉ ∅ then
    foreach hypothesis H ∈ Q_H do
        foreach node N in H do
            create H_n at s with ID_max;
            connect with the existing node N with O and l;
            if H_n is valid then
                add H_n to Q*_H;
                ID_max := ID_max + 1;
            else
                destroy H_n;
            create H_n at s with ID_max;
            create the new node with O and l;
            if H_n is valid then
                add H_n to Q*_H;
                ID_max := ID_max + 1;
            else
                destroy H_n;
else
    create H_n at s with ID_max;
    create the new node with O and l;
    if H_n is valid then
        add H_n to Q*_H;
        ID_max := ID_max + 1;
    else
        destroy H_n;
s := s + 1;
Q_H := Q*_H;
end;
```

4.5.7 Getting a minimal path

The minimal path is acquired from the minimal hypothesis H_M. Since we are using priority queue Q_H sorted by number of the nodes, getting H_M is very simple because it is on the top of Q_H. Then for H_M the path to the nearest undiscovered node is calculated and returned. The process of calculation of the path of a hypothesis is described in the Section (4.5.4.7.).

Chapter 5

Experiments

In this chapter, the experimental results of proposed planning methods are evaluated. The results are divided into the groups according the environments, they have been performed in.

The experiments could not be done using *SyRoTek* and the real robots because the arena cannot be configured to provide sufficient number and variance of the experimental environment. These results are not affected by their origin, i.e. if we would had the real arena with the same shape like arena in the simulator, then we would obtain the same results. Thus, the test was done using *ROS 1.6.8* and *Stage 3.2.2* simulator on the computer with processor *Intel Celeron dual-core 1.7Ghz* and RAM memory of size *2GB*.

For the experiments in simulator, the parameters of individual modules referred in Chapter (4) were set with these values:

Table 5.1: Parameters used in experiments

Parameter name	**Value**
FREE_AREA_THRESHOLD	32°
MAX_SENSE_RANGE	36cm
MIN_OBSTACLE_RANGE	11cm
FOLLOW_DISTANCE	16cm
MIN_ROTATION_DANGLE	1°
MAX_WALL_DANGLE	8°
MAX_HALLWAY_DANGLE	44°
BACK_DISTANCE_CHECK_RANGE	90°
FRONT_DISTANCE_CHECK_RANGE	90°

As discussed in Section (3.2.1), the successor hypotheses can be generated either from the last valid minimal hypothesis, or from all last valid hypotheses. Since have wanted to test generation of hypotheses with omitting some constraint for various environments, there can occur a case, when the current valid minimal hypothesis can probably become invalid in the next steps. If we would have used the first approach,

we would never had a valid map at the end of generation process. Therefore, we have chosen the other approach when all the current valid hypotheses are considered.

5.1 Planning methods

When generating graph hypotheses, there are several methods how to plan the trajectory of the robot. The planner is based either on finding the path from the current hypothesis (where the path to the nearest undiscovered hallway is searched) or on reacting to the current environment state. The methods that have been tested using two deterministic planners and one non-deterministic planner. A list of tested planners is the following:

- *Breadth-first planner*: planner finds the hallway using the BFS algorithm described in Section (4.5.4.7.).
- *Depth-first planner*: planner finds the hallway using the DFS algorithm described in Section (4.5.4.7.).
- *Random planner*: this planner randomly selects the hallway from the current junction observation. There is not used any logic of choosing the path, or using the information stored in the graph hypothesis.

Deterministic planners **always** give the same results. Non-deterministic planner gives various results which needs to be averaged. Thus, for calculation of trend and performance of the Random planner four measurements was executed.

There were designed **two types of experiments**. The first type has a task of examining the performance of planned algorithms to the number of hypotheses and steps in different environments. The subtask is to find out which planner is the most suitable for the environment in general containing closed and open loops. The task of the other experiments is to examine the influence of omitting individual constraints used in generation of hypothesis (refer Section (3.2.2)) to a number of hypotheses and steps in hypothesis validation process. The omitting constraints includes **direction constraints** and **planarity constraint**. Structural constraint was retained because it is considered as the basic rule of rejecting generated hypotheses. Other constraints were examined because their omitting is not so crucial and we wanted to find out, if it is possible to get the minimal valid hypothesis at the end of generation process and how many steps and hypotheses does it take.

All of these types of the planners and constraints have been tested in **closed-loop** or **open-loop** environments. The closed-loop environment is environment where the number of edges N_E and the number of vertices N_V is related as: $N_E \geq N_V$.

The open-loop environment is environment where the number of edges and vertices is related as: $N_E \leq N_V$.

In the following sections the results for both kinds of the environments are shown. Environments depicted there have shape containing black or white color, where the **black color** represents **obstacles** and the **white color** represents **free space** where the robot can pass. The symbol "S" in circle denotes the start position of the robot.

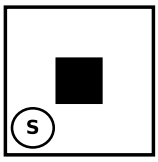
If more start positions depicted, the tests from the various start positions was performed.

Tables shown in the next sections contain the number of graph hypotheses according to used planer. The symbol "x" means that the final minimal hypothesis does not correspond to the real map of the environment and the symbol "/" means that specified case was not tested. The resulting maps generated by the planners using all constraints can be seen in Section Appendix (A).

5.2 Closed-loop environments

The performance of the BFS and the DFS planner was measured on seven closed-loop environments which differs in size N_E. The smallest environment has 4 edges and subsequently environments are incremented by 3 edges up to size of 20 edges. The Random planner was tested in environments up to 10 edges because of its time and memory consumption.

The first environment has $N_E = 4$ and $N_V = 4$. Its shape and the performance results can be seen in Figure (5.1), and the generated maps are shown in Appendix (A.1).



(a) Arena

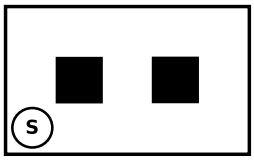
conditions/planner	BFS	DFS	Random
Number of hypotheses			
no direction constraints	10	6	10
no planarity constraint	10	6	9
all constraints	10	6	9
Number of steps			
no direction constraints	9	5	9.75
no planarity constraint	9	5	8.75
all constraints	9	5	8.75

(b) Performance results

Figure 5.1: Closed-loop arena with 4 edges and performance results of the planners

Using the BFS and the DFS planner with planarity or direction being switched off had no effect for this kind of environment. Note that since this kind of environment the DFS planner generates approximately twice less hypotheses in twice less steps than the BFS planner.

The second environment has $N_E = 7$ and $N_V = 6$. Its shape and the performance results can be seen in Figure (5.2), and the generated maps are shown in Appendix (A.2).



(a) Arena

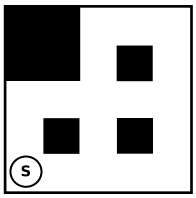
conditions/planner	BFS	DFS	Random
Number of hypotheses			
no direction constraints	24	14	39.5
no planarity constraint	24	14	32.25
all constraints	24	14	9
Number of steps			
no direction constraints	16	9	9.75
no planarity constraint	16	9	8.75
all constraints	16	9	8.75

(b) Performance results

Figure 5.2: Closed-loop arena with 7 edges and performance results of the planners

Here again using the BFS and the DFS planner with switched off planarity or direction constraints had not any affect.

The third environment has $N_E = 10$ and $N_V = 8$. Its shape and the performance results can be seen in Figure (5.3), and the generated maps are shown in Appendix (A.3).



(a) Arena

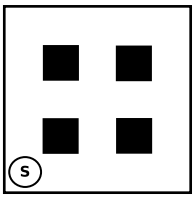
conditions/planner	BFS	DFS	Random
Number of hypotheses			
no direction constraints	41	24	605
no planarity constraint	56	24	134.25
all constraints	56	24	117.5
Number of steps			
no direction constraints	21	12	40
no planarity constraint	23	12	44.25
all constraints	23	12	43

(b) Performance results

Figure 5.3: Closed-loop arena with 10 edges and performance results of the planners

In this case the BFS planner with switched off direction constraints generated less amount of hypotheses and steps than the same planner with switched off the planarity constraint or with using all constraints. This is caused by the shape of the environment, which has the two identical junctions on the upper left border. Then the minimal hypothesis in each step influenced the behavior of the planner. Note that the complexity of the random planner starts to grow markedly.

The fourth environment has $N_E = 12$ and $N_V = 9$.
The shape of the environment and the performance results can be seen in Figure (5.4), and the generated maps are shown in Appendix (A.4).



(a) Arena

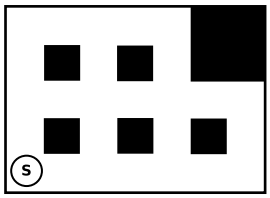
conditions/planner	BFS	DFS
Number of hypotheses		
no direction constraints	135	54
no planarity constraint	135	54
all constraints	135	54
Number of steps		
no direction constraints	29	16
no planarity constraint	29	16
all constraints	29	16

(b) Performance results

Figure 5.4: Closed-loop arena with 12 edges and performance results of the planners

From this result is notable that the number of hypotheses and the number of steps is the same for all kinds of the constraints. It is caused by the environment which does not have any two identical junctions in the absolute position. Then the number of hypotheses generated by the planner in individual types of constraints starts be striking.

The fifth environment has $N_E = 15$ and $N_V = 11$. Its shape and the performance results can be seen in Figure (5.5), and the generated maps are shown in Appendix (A.5).



(a) Arena

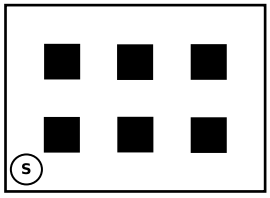
conditions/planner	BFS	DFS
Number of hypotheses		
no direction constraints	934	178
no planarity constraint	433	128
all constraints	432	128
Number of steps		
no direction constraints	35	21
no planarity constraint	37	20
all constraints	37	20

(b) Performance results

Figure 5.5: Closed-loop arena with 15 edges and performance results of the planners

Using the BFS planner in this type of environment with no direction constraints resulted in less steps than with other kinds of constraints, but generated much more hypotheses.

The sixth environment has $N_E = 17$ and $N_V = 12$. Its shape and the performance results can be seen in Figure (5.6), and the generated maps are shown in Appendix (A.6).



(a) Arena

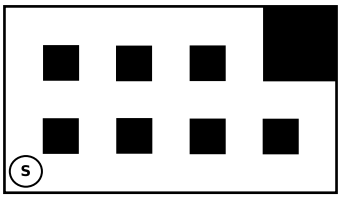
conditions/planner	BFS	DFS
Number of hypotheses		
no direction constraints	4,106	279
no planarity constraint	1,393	170
all constraints	1,386	170
Number of steps		
no direction constraints	42	22
no planarity constraint	42	22
all constraints	42	22

(b) Performance results

Figure 5.6: Closed-loop arena with 17 edges and performance results of the planners

In this situation is clear that using no direction constraints for the BFS planner has a big affect to the amount of generated hypotheses. Although the environment has more junctions of the same type, the number of steps in all kinds of constraints for the BFS and the DFS planner is the same. It is caused by the start position of the robot.

The last environment has $N_E = 20$ and $N_V = 14$. The performance of the random planner was not tested in any case because of its requirements. Its shape and the performance results can be seen in Figure (5.7), and the generated maps are shown in Appendix (A.7).



(a) Arena

conditions/planner	**BFS**	**DFS**
Number of hypotheses		
no direction constraints	88,849	17,069
no planarity constraint	5,171	308
all constraints	5,108	308
Number of steps		
no direction constraints	49	37
no planarity constraint	50	24
all constraints	50	24

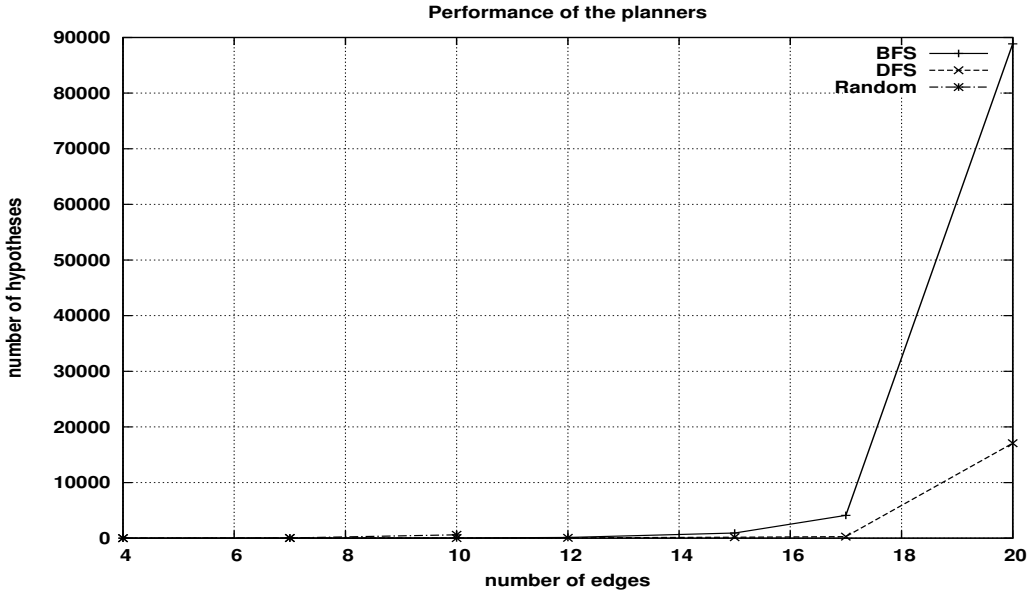
(b) Performance results

Figure 5.7: Closed-loop arena with 20 edges and performance results of the planners

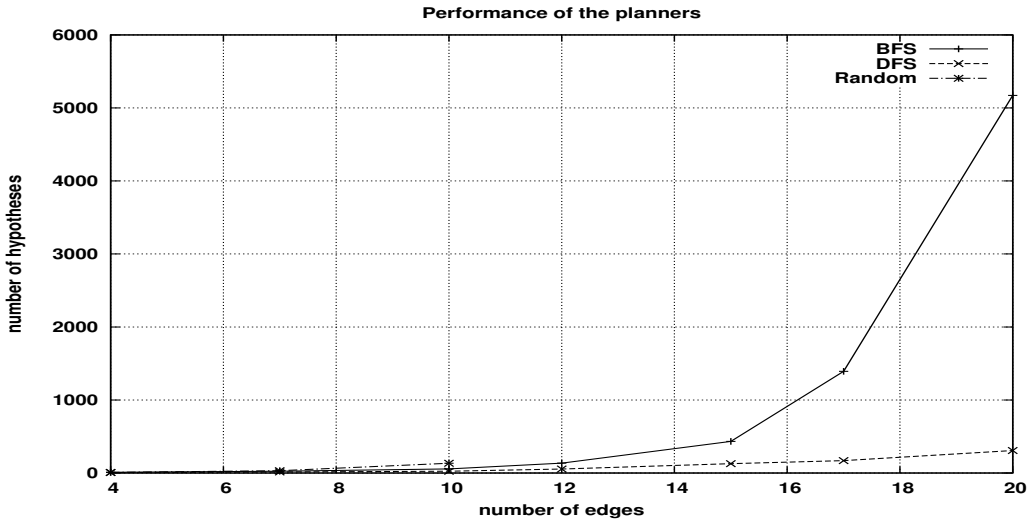
From the last environment we can conclude that using no direction constraints for any planner has considerable consequences in the number of generated hypotheses and also in its memory requirements. Here, for the first time using no planarity constraint had a small effect in reducing the hypotheses of the BFS planner.

5.2.1 Results

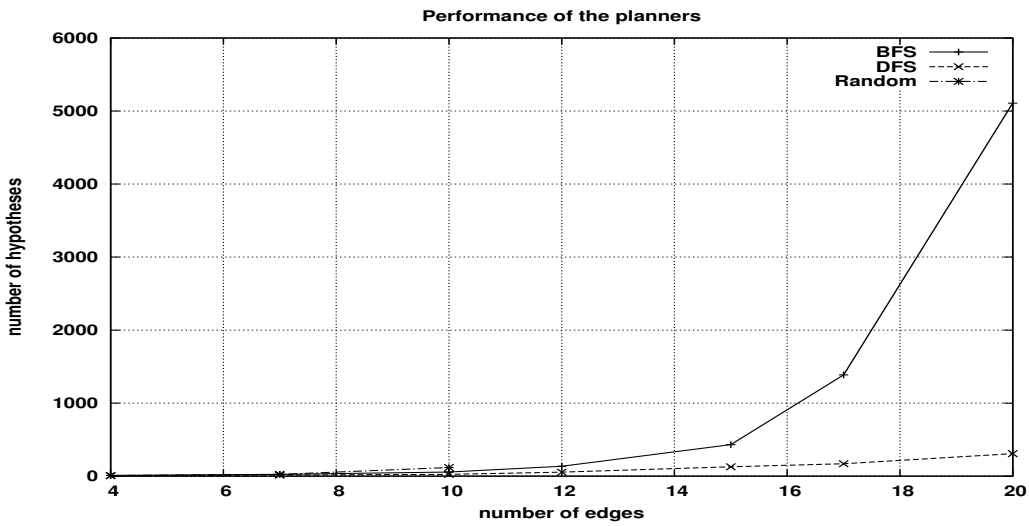
Measured data was summarized for depicting the dependency of the number of hypotheses/steps on the number of edges of closed-loop environments. Results can be seen in the following Figures:



(a) no direction constraints

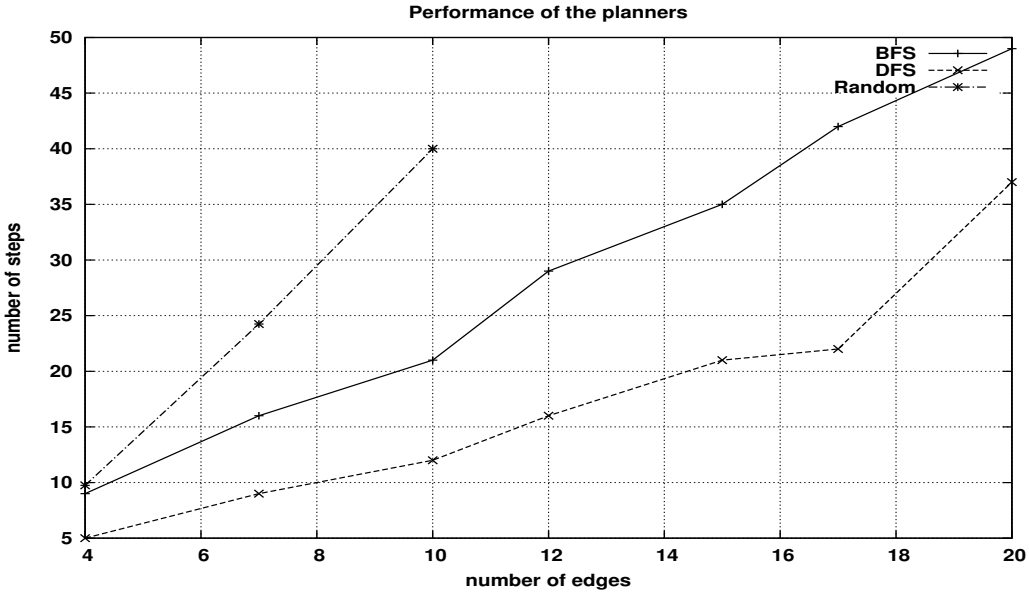


(b) no planarity constraint

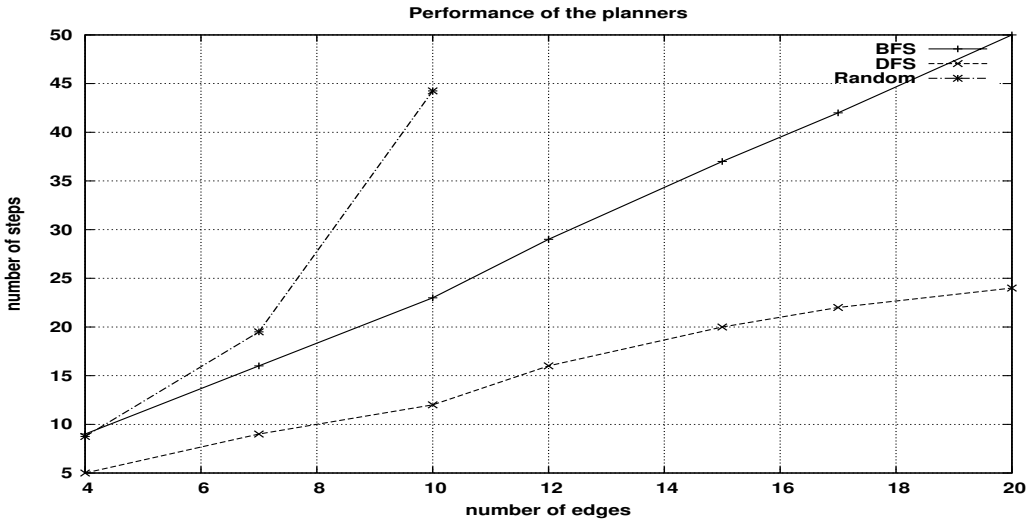


(c) all constraints

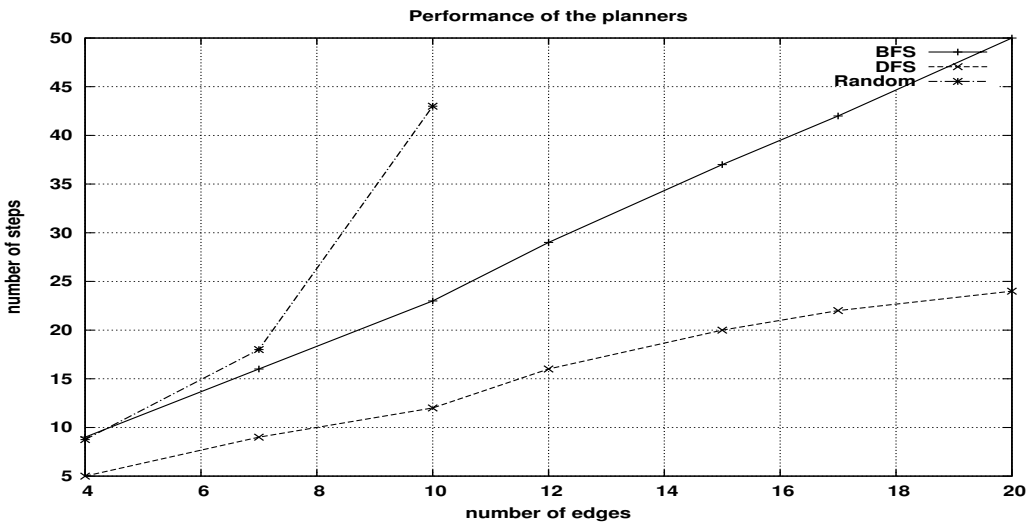
Figure 5.8: Dependency of the number of hypotheses on the number of edges for closed-loop environments



(a) no direction constraints



(b) no planarity constraint



(c) all constraints

Figure 5.9: Dependency of the number of steps on the number of edges for closed-loop environments

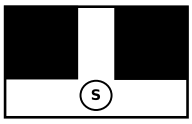
According to the results we can conclude that the **DFS planner** has **the best performance** over all planners. It **always** generates **less amount of hypotheses** in **less steps**. The more edges the environment has, the higher difference is between the performance of the DFS planner and other planners. The second best is BFS planner which is suitable for use in environments with less amount of the edges. For more than 15 edges in environment, the number of hypotheses generated by this planner starts to grow significantly. The worst is the Random planner which is usable only for environments with 8 edges. With more than 8 edges the complexity of this planner is significant.

From the graphs can be seen that using no **direction constraints** had the **biggest affect** on the number of generated hypotheses and steps. It is caused by the similarity of several junctions in the environment. If not using this constraints, a node can be connected with all nodes, whose junction observation is the same like in new node. On the other side, using no planarity constraint had not big influence to the number of generated hypotheses and steps. Finally, when using described planner with all constrains, as expected, produced the least amount of generated hypotheses and steps.

5.3 Open-loop environments

For measuring a performance of planners in open-loop environment, there was again designed seven types. The smallest environments have 3 and 4 edges and subsequently environments are incremented by 3 edges up to size of 19 edges. Random planner was tested in open-environments up to 7 edges because of its time and memory consumption.

The first environment has $N_E = 3$ and $N_V = 3$. Its shape and the performance results can be seen in Figure (5.10), and the generated maps are shown in Appendix (A.8).



(a) Arena

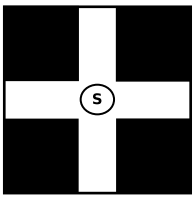
conditions/planner	BFS	DFS	Random
Number of hypotheses			
no direction constraints	6	6	8.5
no planarity constraint	6	6	11.5
all constraints	6	6	13.5
Number of steps			
no direction constraints	6	6	8.5
no planarity constraint	6	6	11.5
all constraints	6	6	38

(b) Performance results

Figure 5.10: Open-loop arena with 3 edges and performance results of the planners

For this simple environment, the results of the BFS and the DFS planners was the same in all types of constraints, because there are four different types of junctions connected by only the one common junction.

The second environment has $N_E = 4$ and $N_V = 5$. Its shape and the performance results can be seen in Figure (5.11), and the generated maps are shown in Appendix (A.9).



(a) Arena

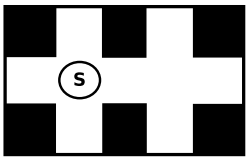
conditions/planner	BFS	DFS	Random
Number of hypotheses			
no direction constraints	8	8	18.5
no planarity constraint	8	8	22.5
all constraints	8	8	14.5
Number of steps			
no direction constraints	8	8	18.5
no planarity constraint	8	8	22.5
all constraints	8	8	14.5

(b) Performance results

Figure 5.11: Open-loop arena with 4 edges and performance results of the planners

Here again the results of the BFS and the DFS planners are the same for all kinds of constraints. The reason is the same like in the previous environment.

The third environment has $N_E = 7$ and $N_V = 8$. Its shape and the performance results can be seen in Figure (5.12), and the generated maps are shown in Appendix (A.10).



(a) Arena

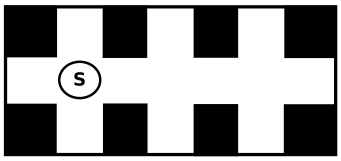
conditions/planner	BFS	DFS	Random
Number of hypotheses			
no direction constraints	15	14	29.75
no planarity constraint	15	14	32.25
all constraints	15	14	38
Number of steps			
no direction constraints	15	14	29.75
no planarity constraint	15	14	32.25
all constraints	15	14	38

(b) Performance results

Figure 5.12: Open-loop arena with 7 edges and performance results of the planners

In this environment, the difference between the complexity of the BFS and the DFS planner becomes noticeable. But there is also the same amount of generated hypotheses not depending on the kind of the constraint for these planners. Since there are some two junctions the same, the planning method has not allowed generation of a map which does not correspond to its shape. This is not the case of Random planner.

The fourth environment has $N_E = 10$ and $N_V = 11$. Its shape and the performance results can be seen in Figure (5.13), and the generated maps are shown in Appendix (A.11).



(a) Arena

conditions/planner	BFS	DFS
Number of hypotheses		
no direction constraints	26	x
no planarity constraint	22	20
all constraints	22	20
Number of steps		
no direction constraints	24	x
no planarity constraint	22	20
all constraints	22	20

(b) Performance results

Figure 5.13: Open-loop arena with 10 edges and performance results of the planners

There is notable the result of the DFS planner using no direction constraints which produces the invalid minimal map, even the robot has started from different positions. This map can be seen in Figure (5.14).

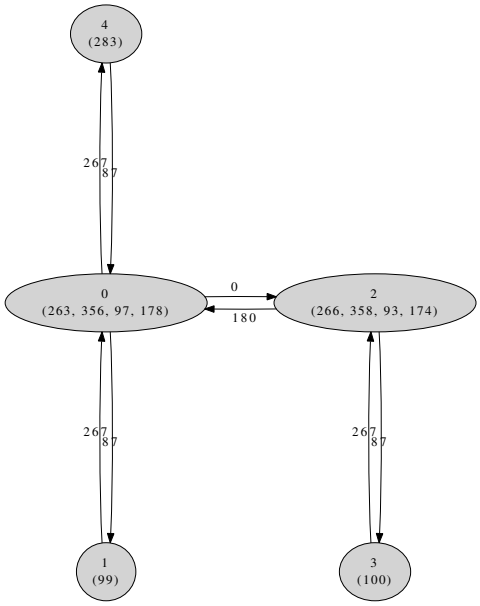
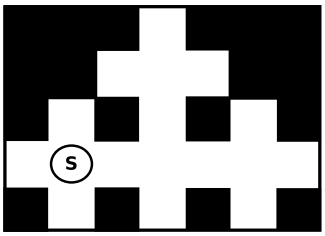


Figure 5.14: The invalid map of the environment with 10 edges produced by the DFS planner using no direction constraints

The fifth environment has $N_E = 13$ and $N_V = 14$. The shape and the performance results can be seen in Figure (5.15), and the generated maps are shown in Appendix (A.12).



(a) Arena

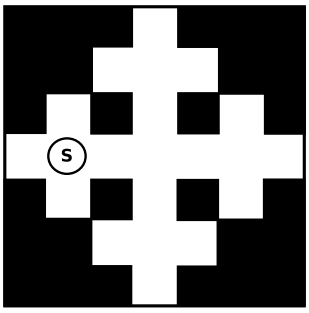
conditions/planner	BFS	DFS
Number of hypotheses		
no direction constraints	39	30
no planarity constraint	28	26
all constraints	28	26
Number of steps		
no direction constraints	30	25
no planarity constraint	28	26
all constraints	28	26

(b) Performance results

Figure 5.15: Open-loop arena with 13 edges and performance results of the planners

The results show that the difference between using no direction constraints and other kinds of constraints becomes noticeable, but only in small manner. It is caused by the shape of the environment.

The sixth environment has $N_E = 16$ and $N_V = 17$. Its shape and the performance results can be seen in Figure (5.16), and the generated maps are shown in Appendix (A.13).



(a) Arena

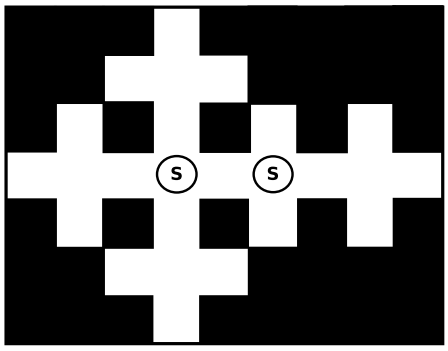
conditions/planner	BFS	DFS
Number of hypotheses		
no direction constraints	74	45
no planarity constraint	36	32
all constraints	36	32
Number of steps		
no direction constraints	42	31
no planarity constraint	36	32
all constraints	36	32

(b) Performance results

Figure 5.16: Open-loop arena with 16 edges and performance results of the planners

From this situation and all situations of the open-loop environment described above can be seen that the number of steps and the number of hypotheses does not differ so much. It is so because there is many junctions with only one leaving hallway.

The last environment has $N_E = 19$ and $N_V = 20$. Its shape and the performance results can be seen in Figure (5.17), and the generated maps are shown in Appendix (A.14).



(a) Arena

conditions/planner	BFS	DFS
Number of hypotheses		
no direction constraints	135	x
no planarity constraint	46	36
all constraints	46	36
Number of steps		
no direction constraints	56	x
no planarity constraint	46	36
all constraints	46	36

(b) Performance results

Figure 5.17: Open-loop arena with 19 edges and performance results of the planners

All the maps produced by the planners were correct except one, when using the DFS with no direction constraints. With these conditions the planer produced the minimal map that does not correspond to the shape of the environment even for various start positions of the robot. This map can be seen in Figure (5.18).

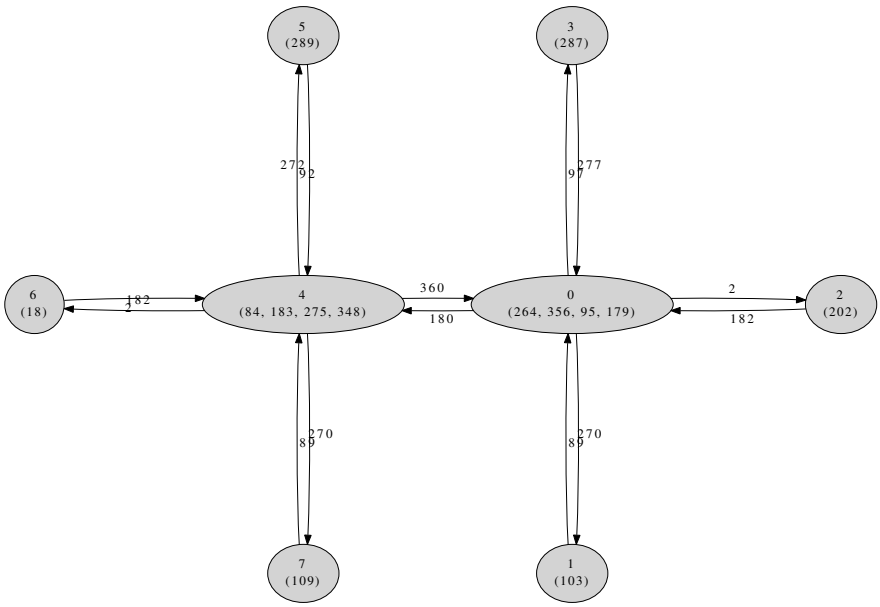
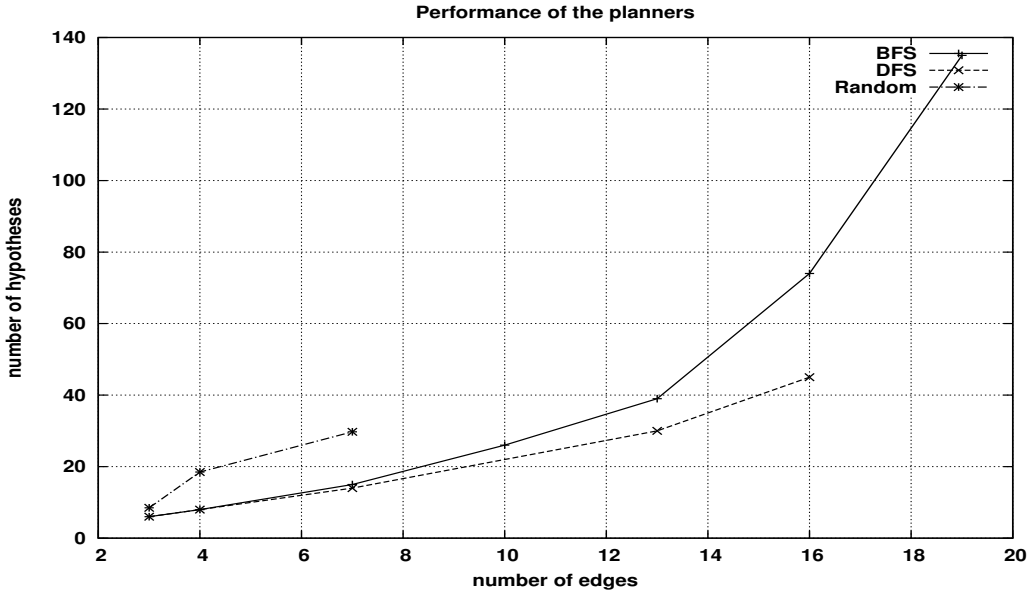


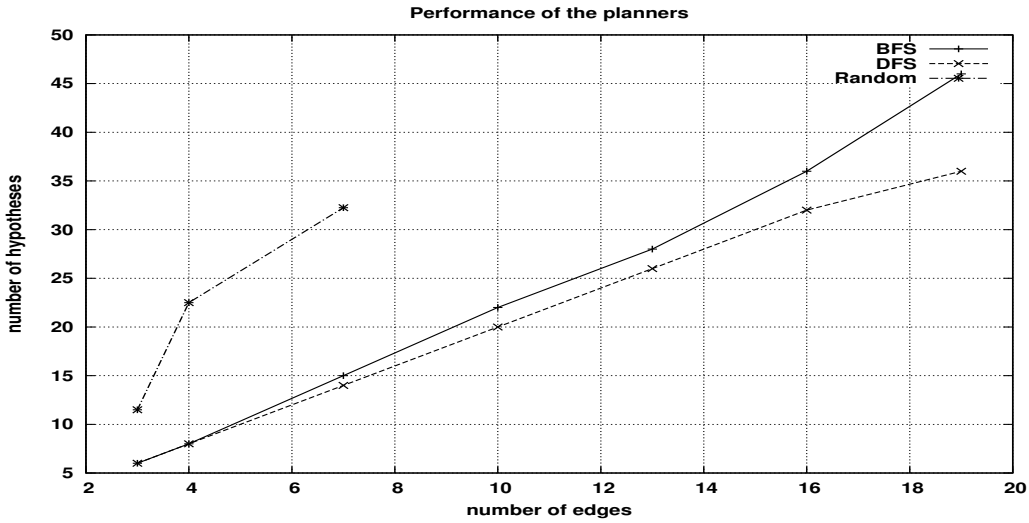
Figure 5.18: The invalid map of the environment with 19 edges produced by the DFS planner using no direction constraints

5.3.1 Results

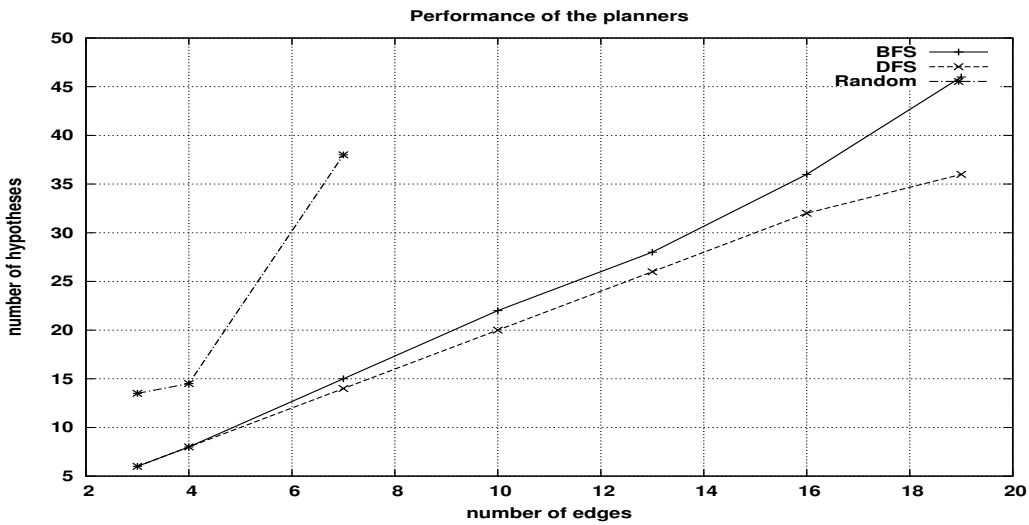
The overall results of the open-loop environments and their graphs showing dependency of the number of hypotheses/step on the number of edges can be seen on the following Figures:



(a) no direction constraints

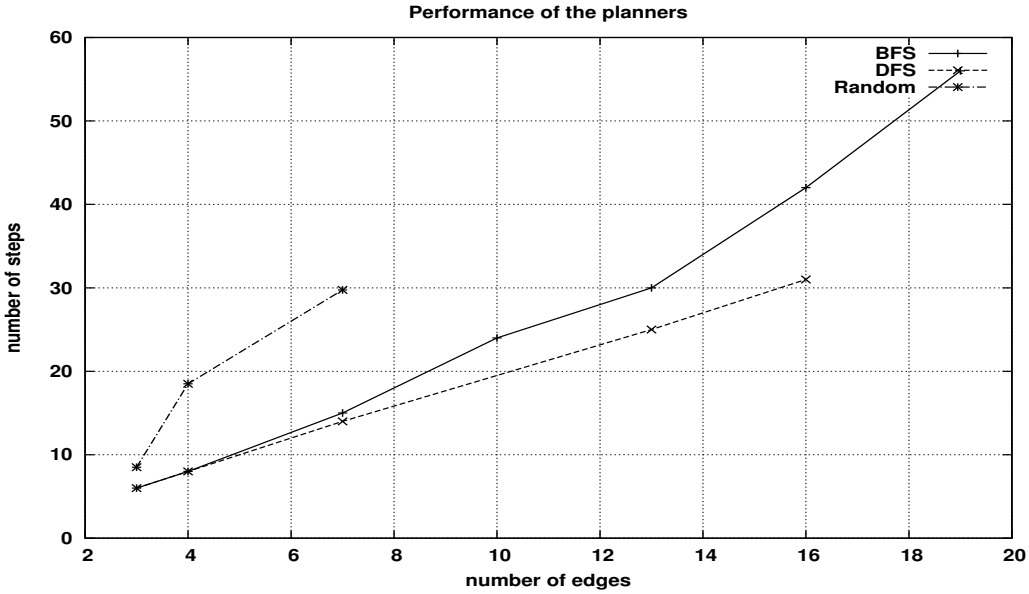


(b) no planarity constraint

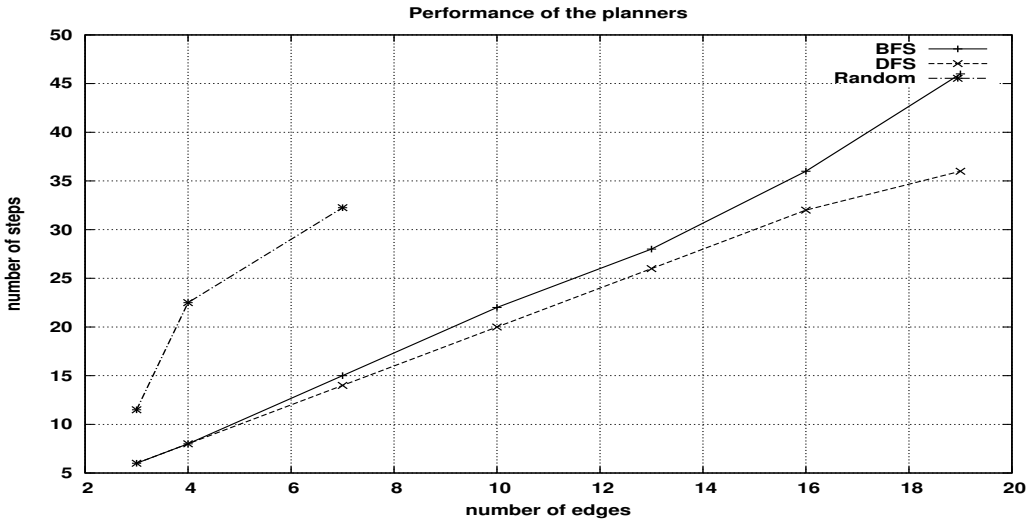


(c) all constraints

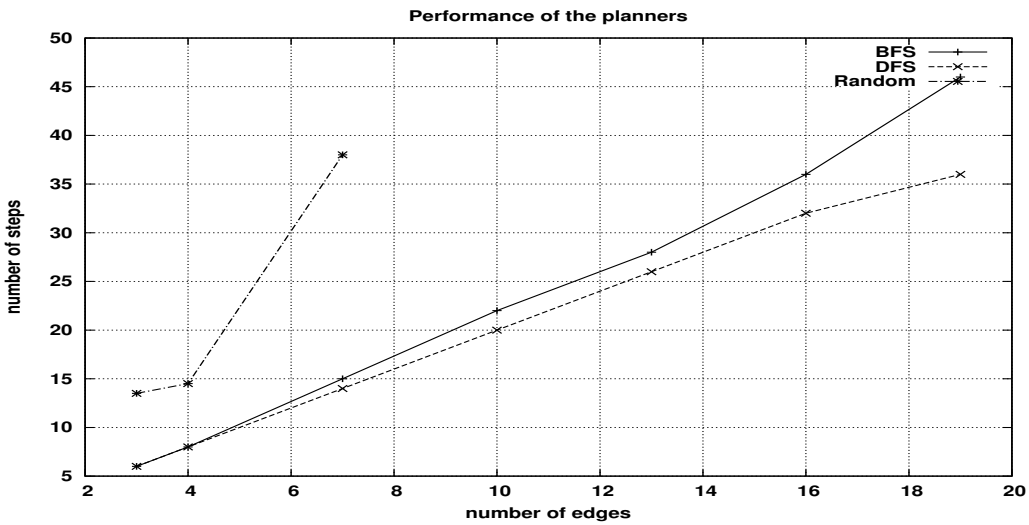
Figure 5.19: Dependency of the number of hypotheses on the number of edges for open-loop environments



(a) no direction constraints



(b) no planarity constraint



(c) all constraints

Figure 5.20: Dependency of the number of steps on the number of edges for open-loop environments

As a result of experiments in open-loop environments we can conclude again that the **DFS** planner had the **best results** in the number of hypotheses and the steps, but there was not so big difference between this planner and the BFS planner. It was caused by no existence of closed-loop in any environment. The BFS planner was the second best again giving not much worse results than the DFS planner. Thus this kind of planner can be also used for open-loop environments with bigger number of edges. As almost unusable is the Random planner, which complexity grows markedly with higher number of the edges.

When considering tested constraints, also for open-loop environments the direction constraints influenced the number of hypotheses the most but only with a small influence to the number of steps required to get valid map. The planarity constraint had only small affect to the number of hypotheses and steps. As expected, using planner with all constraints had the best results in amount of hypotheses and steps.

Chapter 6

Conclusion

The robotics is a wide multi-disciplinary branch of science. This work addressed only the problem of mapping the part of mapping an unknown environment. Two fundamental methods of building a map the environment have been described in Chapter (3). Then multi-hypothesis topological map method has been introduced and discussed in Chapter (3), Section (3.2). The implementation was designed and based on these theoretical principles in Chapter (4). Then follows Chapter (5) with provided experiments. Since experiments with real robots could not be done because of variance of the real arena, we have provided more experiments on various environments with closed and open loops using three types of planners tested in the simulator. Their performance and complexity have been measured and summarized in particular graphs.

According to overall measured results, we can conclude that the **multi-hypothesis topological mapping method** is **working** and **suitable for mapping an unknown environment** because of its robustness, simplicity and low memory and time consumptions. We have confirmed that for generating valid reliable maps of environments all constraints must be applied. Then we have shown that omission of direction constraints or planarity constraint is possible but with higher complexity and not always correct results. The most pronounced feature of **multi-hypothesis topological mapping method** that when using all constraints it **always generates valid map independently of the used planner**. Moreover, we have found that the most effective planner for all environments is **Depth-first search** planner which had the **best results** in all tests.

Bibliography

[1] P. Agent. The advantages of absolute encoders for motion control. *Sensors*, pages 19–24, 1991.

[2] J. Borenstein, H. R. Everett, L. Feng, and D. Wehe. Mobile robot positioning – sensors and techniques, 1996.

[3] J. M. Boyer and W. J. Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8:2004, 2004.

[4] H. R. Everett. *Sensors for mobile robots: theory and application.* A. K. Peters, Ltd., Natick, MA, USA, 1995.

[5] B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[6] C. Ken. ROS Introduction. *http://www.ros.org/wiki/ROS/Introduction.*

[7] G. Ligozat. Reasoning about cardinal directions. *Journal of Visual Languages and Computing*, 9:23–44, 1998.

[8] G. Metta, P. Fitzpatrick, and L. Natale. Yarp: Yet another robot platform. *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics*, 3(1), 2006.

[9] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching, and Annals of the Computation Laboratory of Harvard University*, pages 285–292. Harvard University Press, 1959.

[10] R. Moratz. Representing relative direction as a binary relation of oriented points. In *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy*, pages 407–411, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

[11] R. R. Murphy. *Introduction to AI Robotics.* MIT Press, Cambridge, MA, USA, 1st edition, 2000.

[12] L. Nick. ROS Concepts. *http://www.ros.org/wiki/ROS/Concepts.*

[13] S. Thrun. Learning maps for indoor mobile robot navigation, 1997.

[14] J. O. Wallgrün. Qualitative spatial reasoning for topological map learning. *Spatial Cognition & Computation*, 10(4):207–246, 2010.

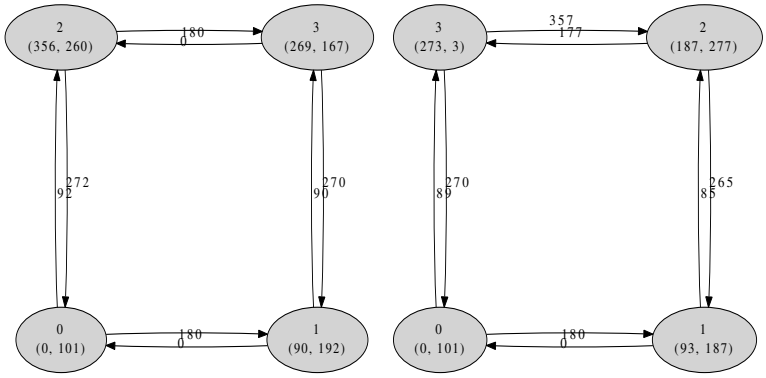
Appendices

Appendix A

Appendix

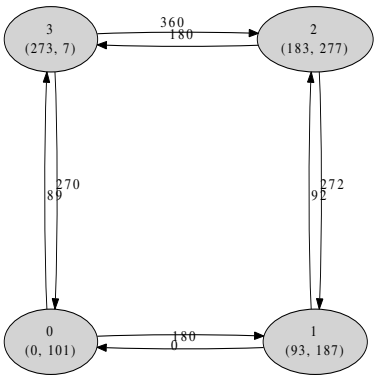
This chapter contains Figures of maps generated by planners refered in Section (5).

A.1 Closed-loop environments



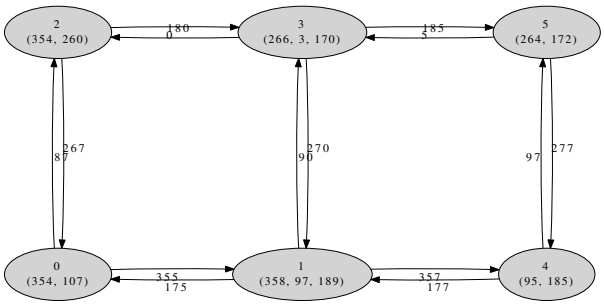
(a) BFS

(b) DFS

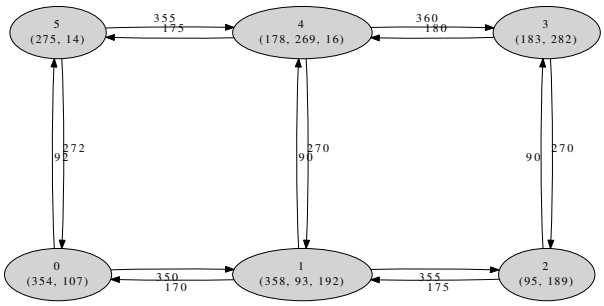


(c) Random

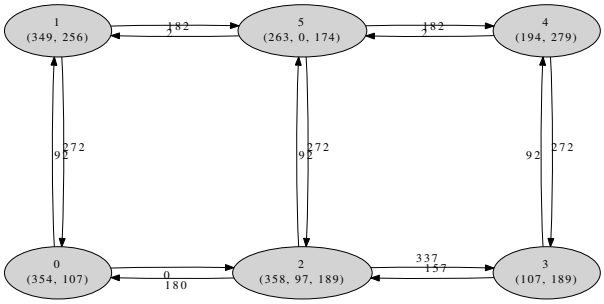
Figure A.1: An example of maps for closed-loop environment with 4 edges generated by planners using all constraints



(a) BFS

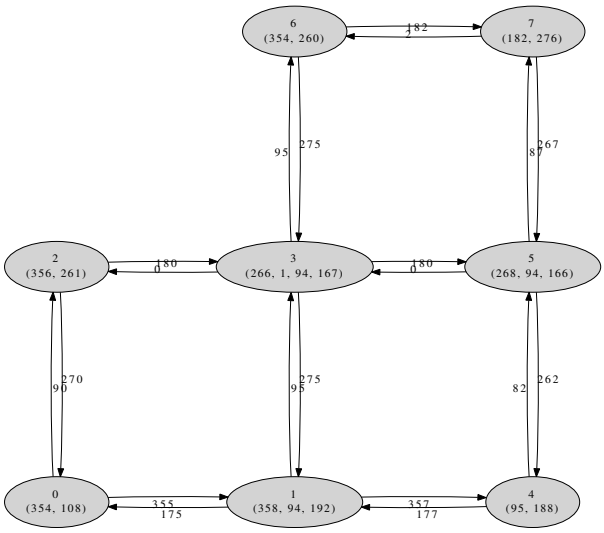


(b) DFS

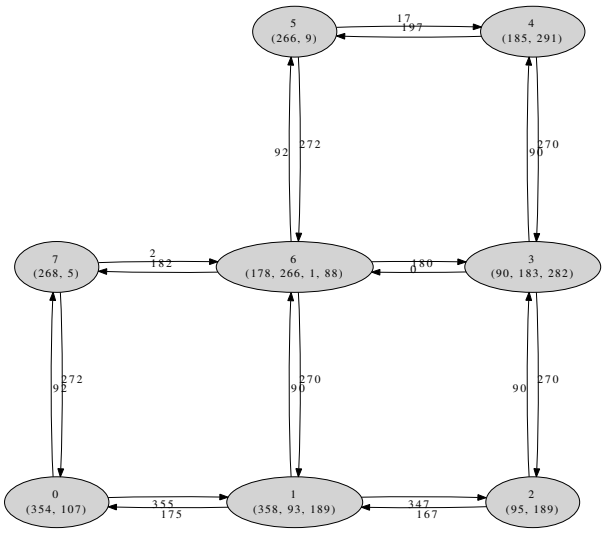


(c) Random

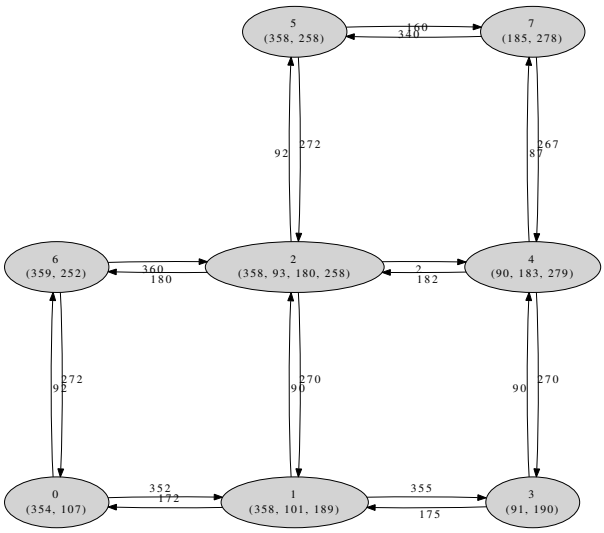
Figure A.2: An example of maps for closed-loop environment with 7 edges generated by planners using all constraints



(a) BFS

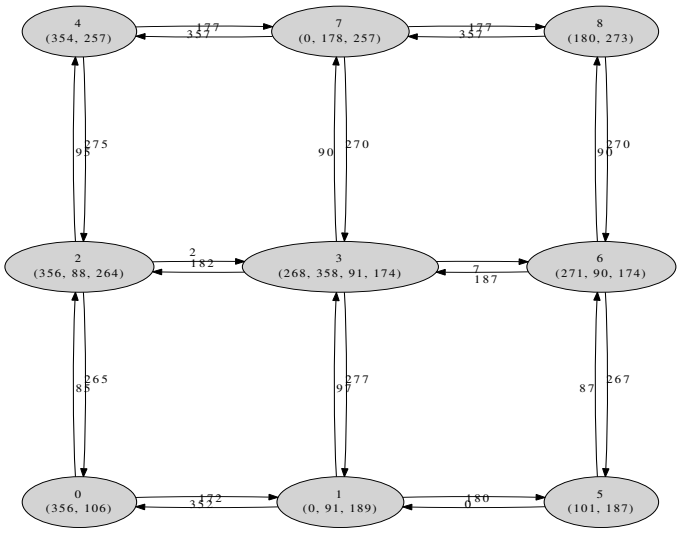


(b) DFS

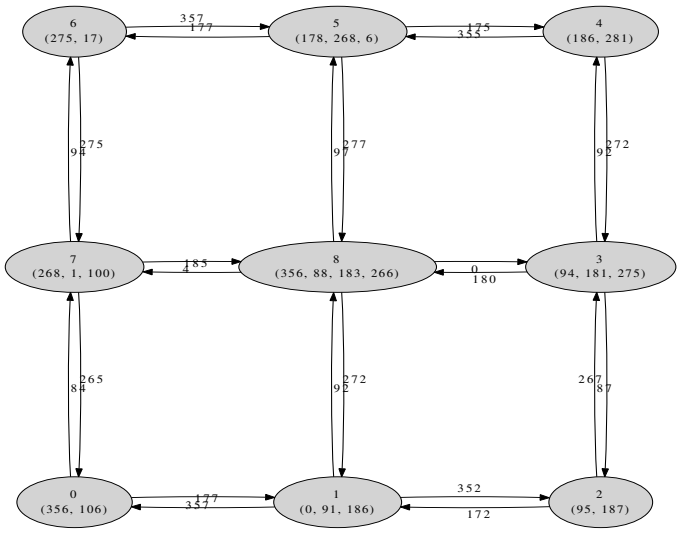


(c) Random

Figure A.3: An example of maps for closed-loop environment with 10 edges generated by planners using all constraints

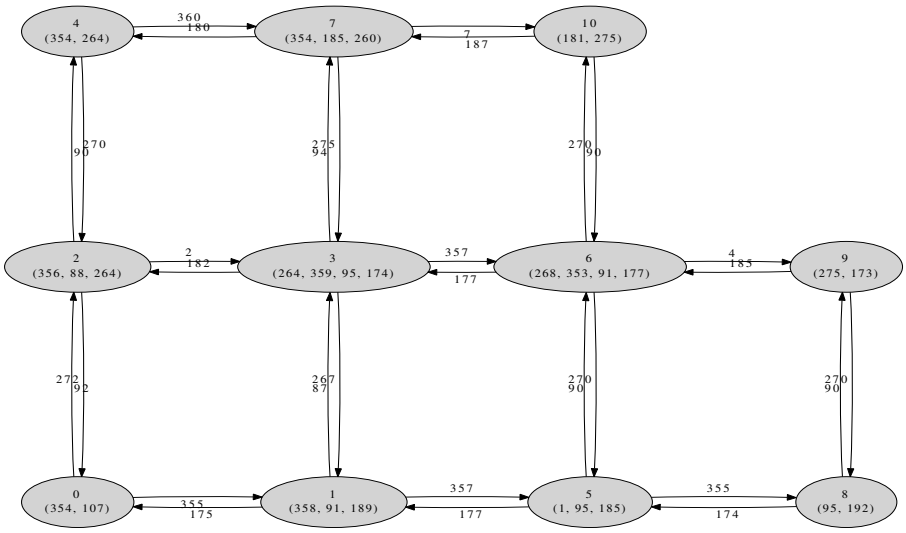


(a) BFS

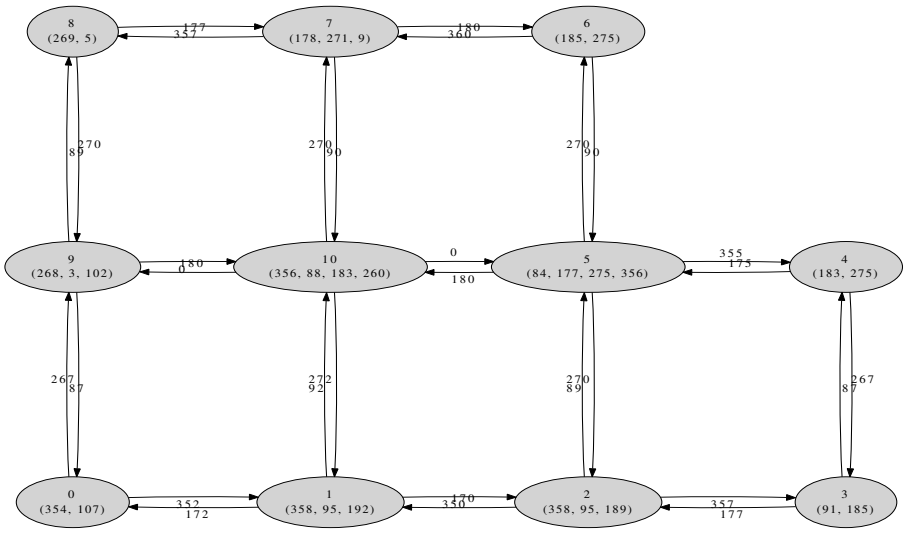


(b) DFS

Figure A.4: An example of maps for closed-loop environment with 12 edges generated by planners using all constraints

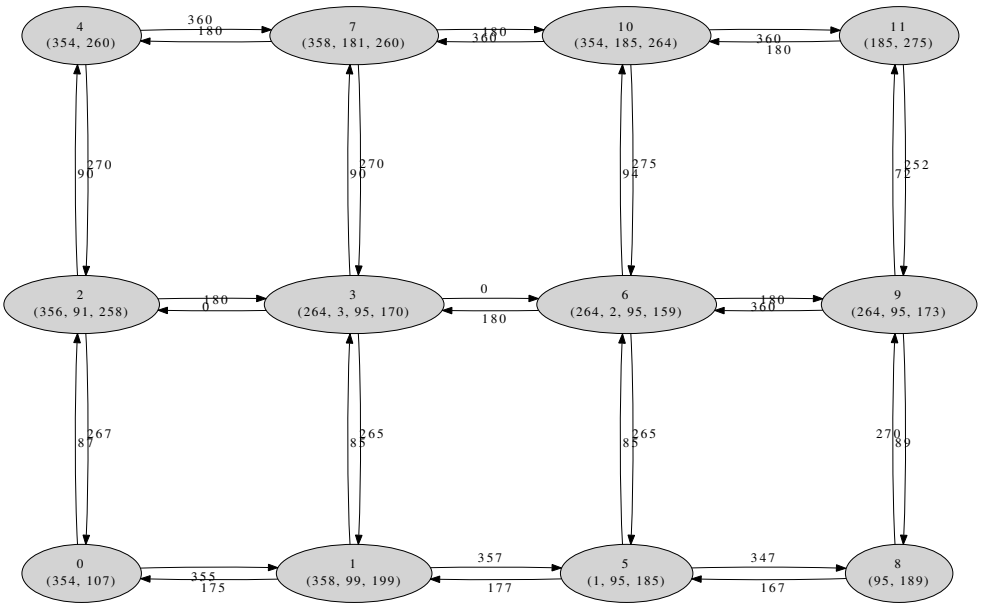


(a) BFS

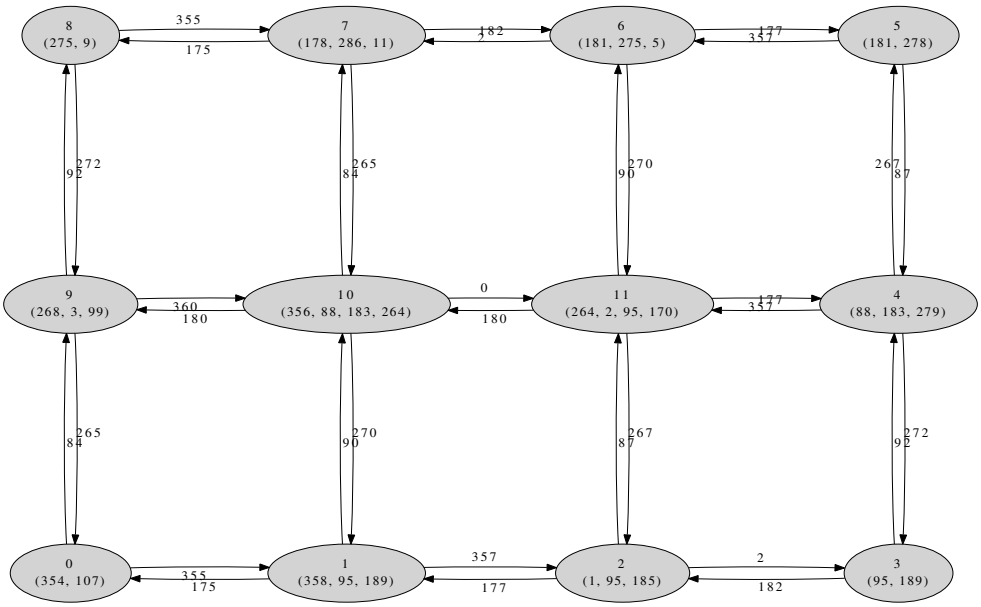


(b) DFS

Figure A.5: An example of maps for closed-loop environment with 15 edges generated by planners using all constraints

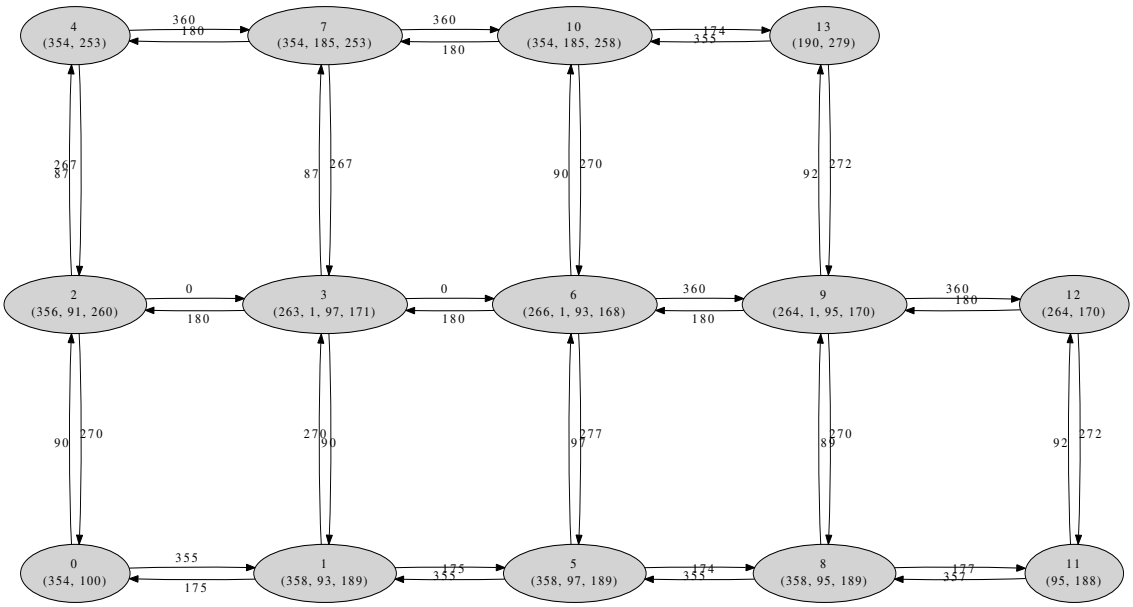


(a) BFS

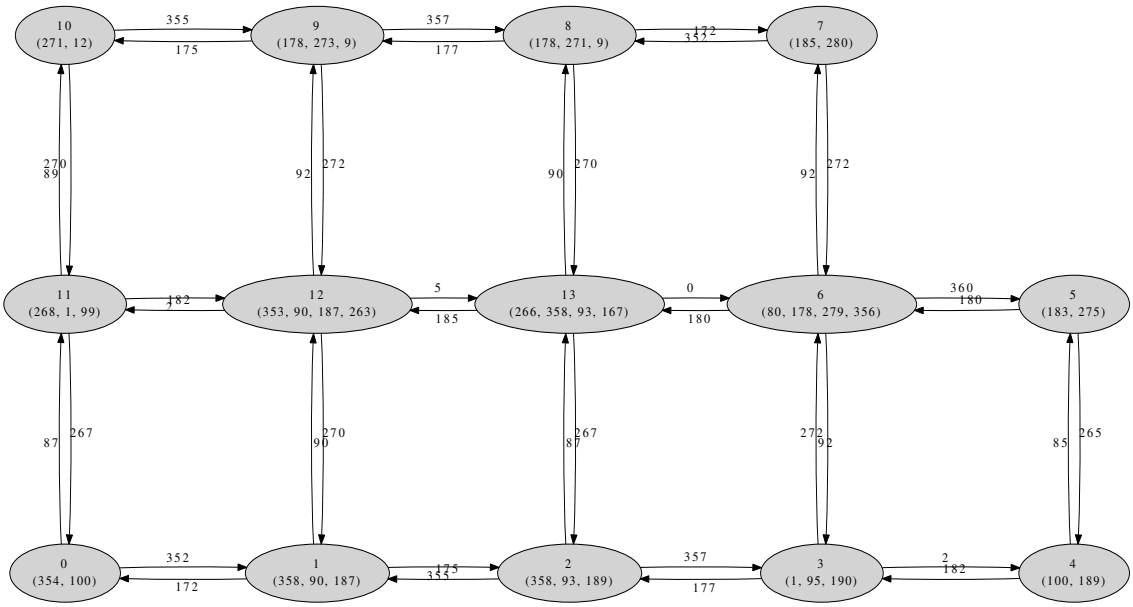


(b) DFS

Figure A.6: An example of maps for closed-loop environment with 17 edges generated by planners using all constraints



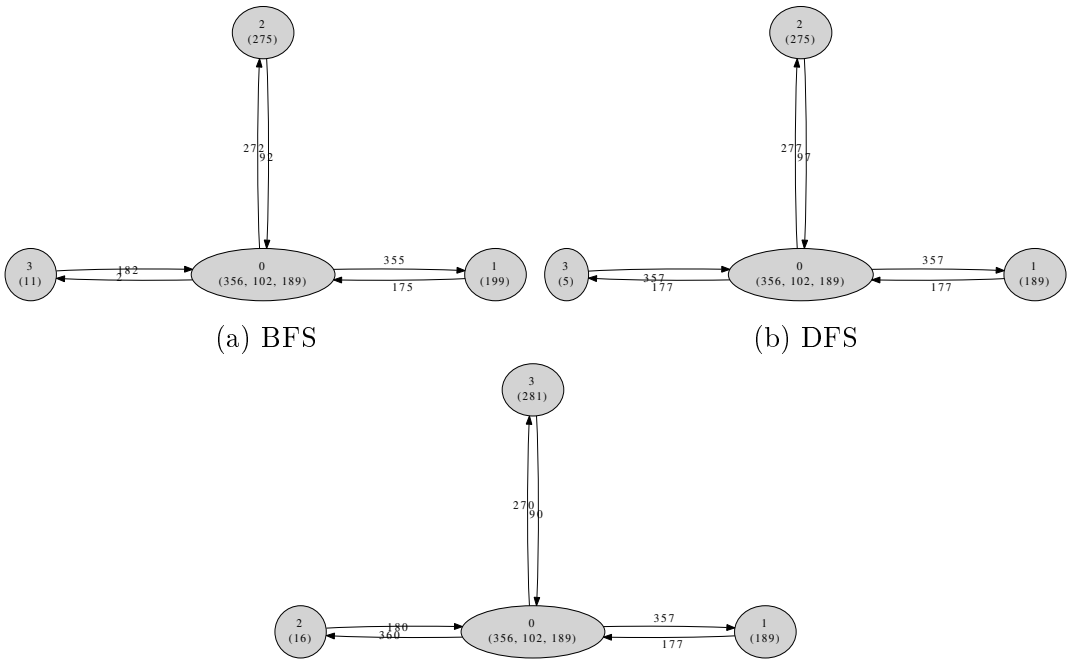
(a) BFS



(b) DFS

Figure A.7: An example of maps for closed-loop environment with 20 edges generated by planners using all constraints

A.2 Open-loop environments

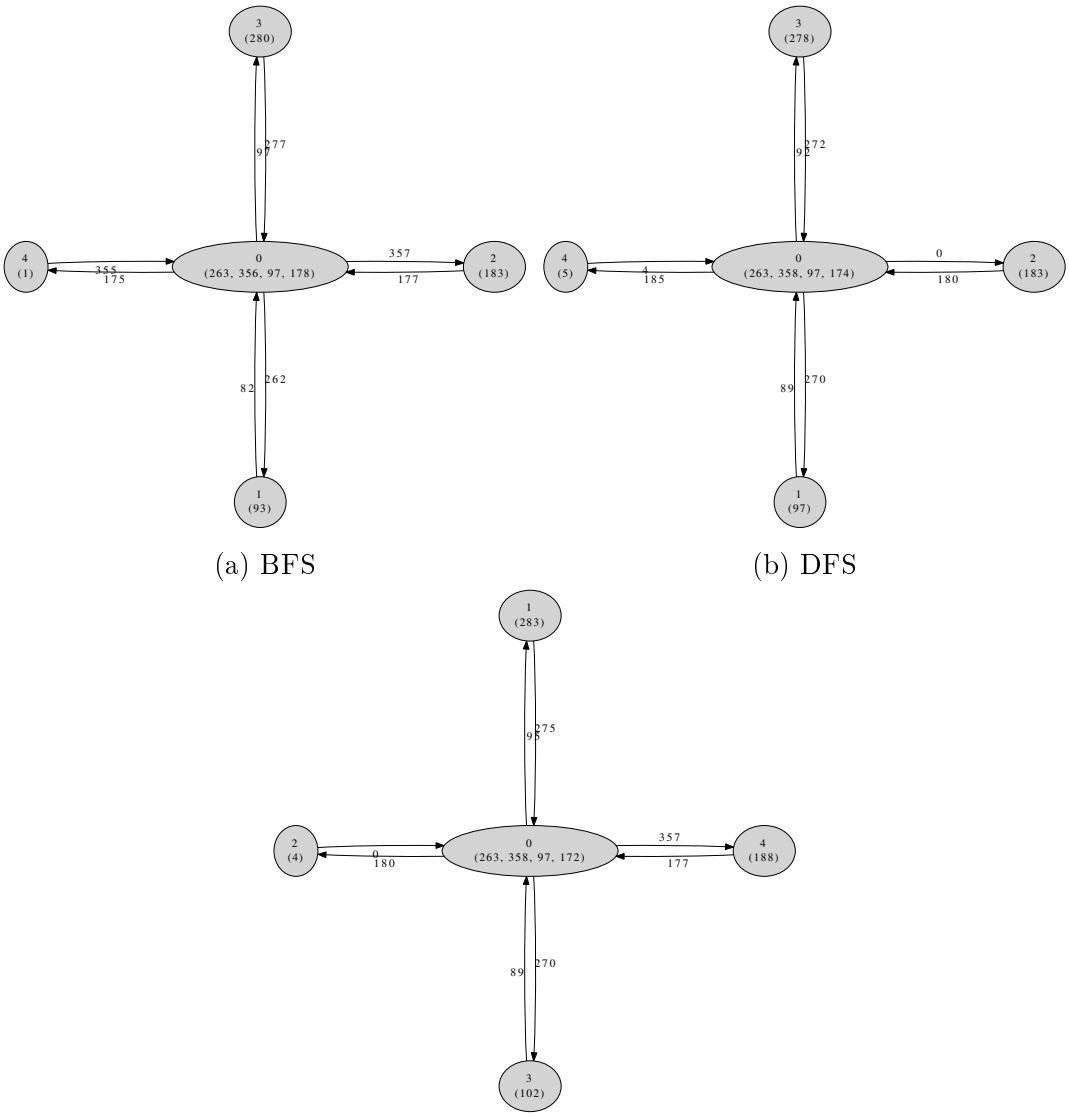


(a) BFS

(b) DFS

(c) Random

Figure A.8: An example of maps for open-loop environment with 3 edges generated by planners using all constraints

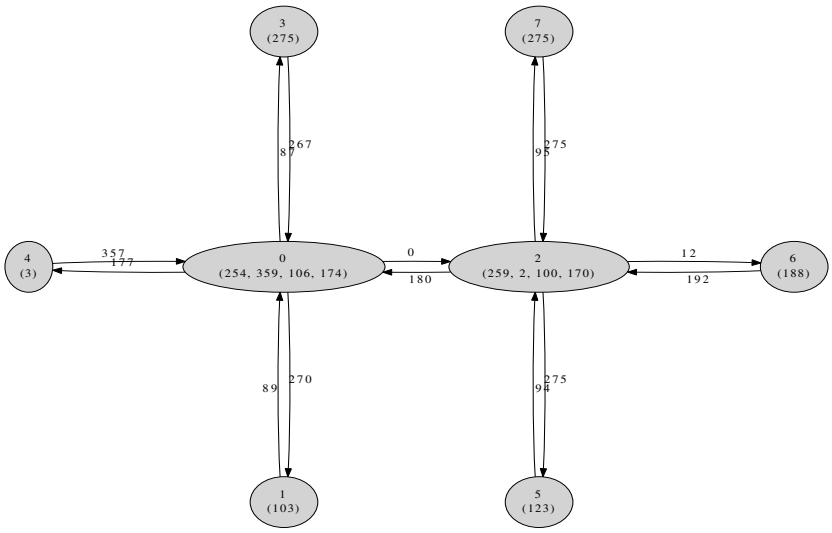


(a) BFS

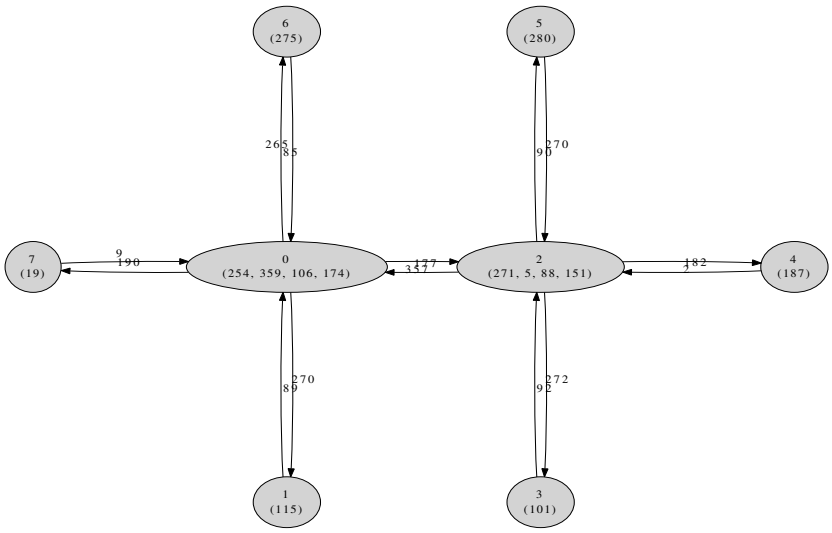
(b) DFS

(c) Random

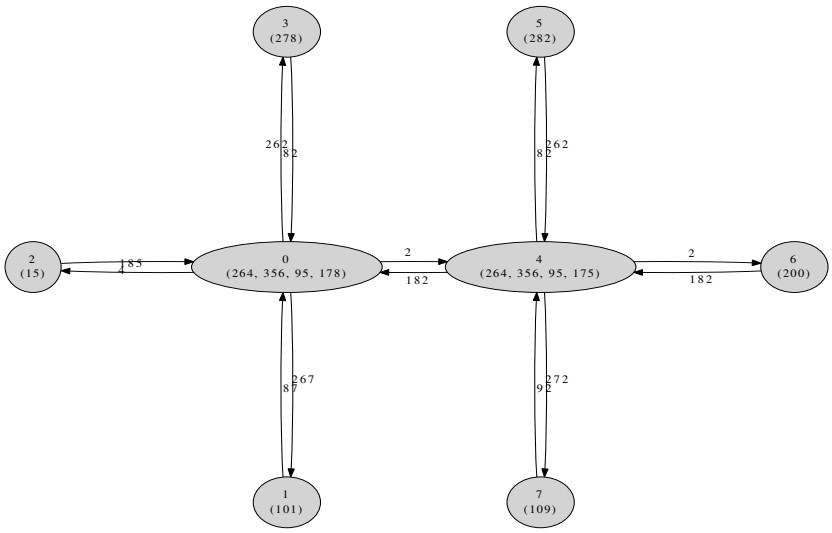
Figure A.9: An example of maps for open-loop environment with 4 edges generated by planners using all constraints



(a) BFS

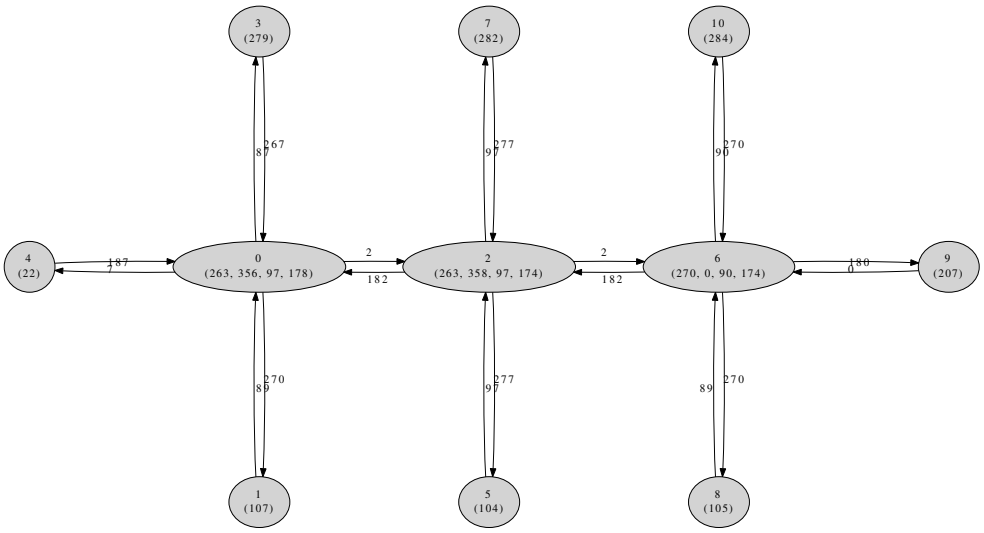


(b) DFS

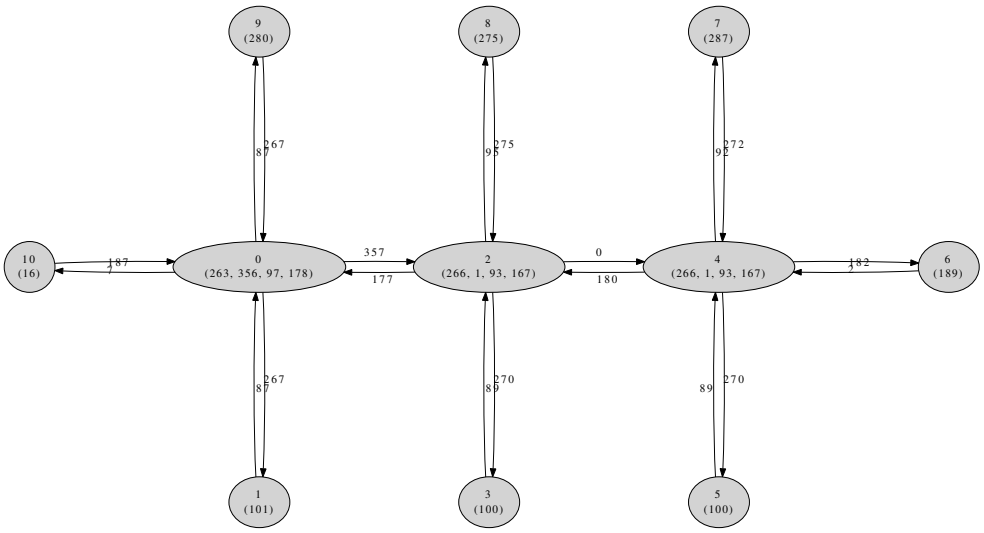


(c) Random

Figure A.10: An example of maps for open-loop environment with 7 edges generated by planners using all constraints

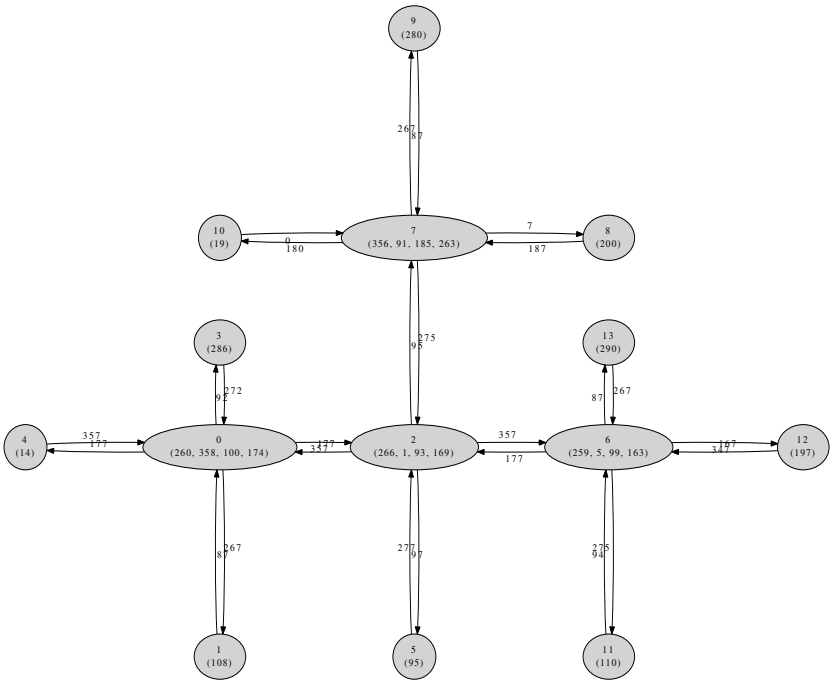


(a) BFS

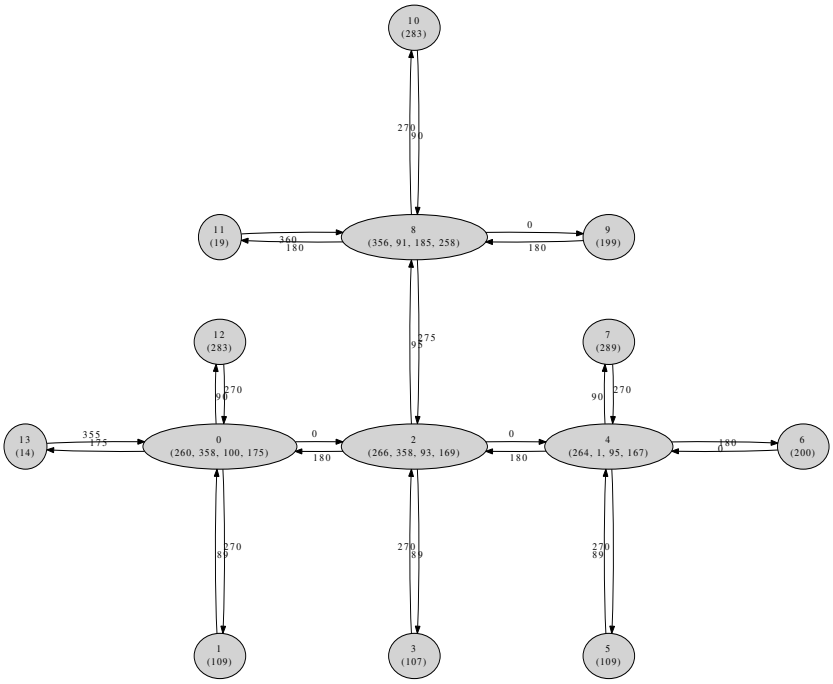


(b) DFS

Figure A.11: An example of maps for open-loop environment with 10 edges generated by planners using all constraints

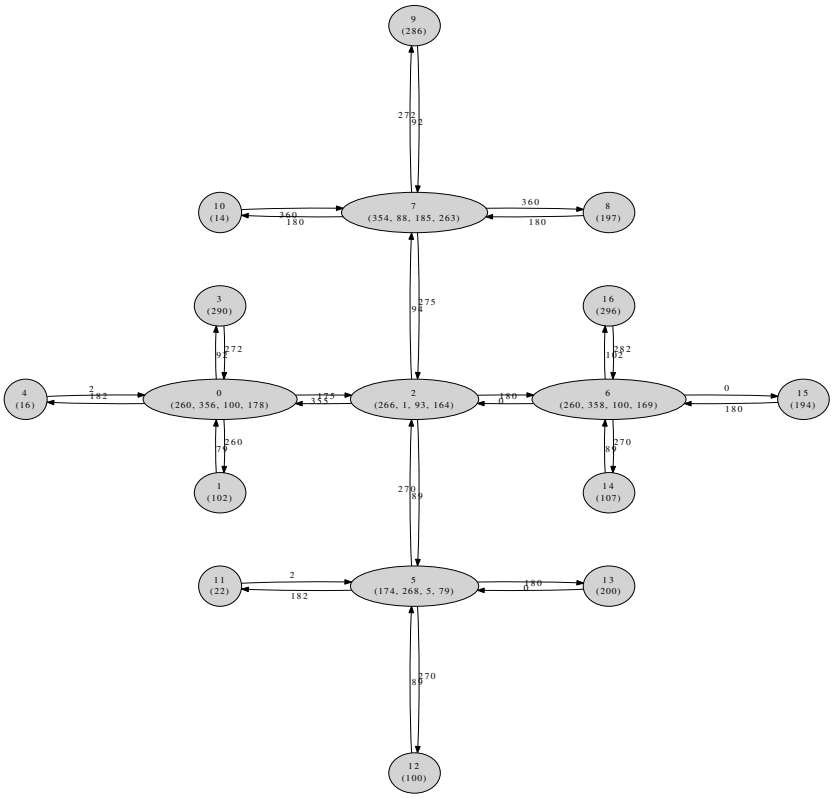


(a) BFS

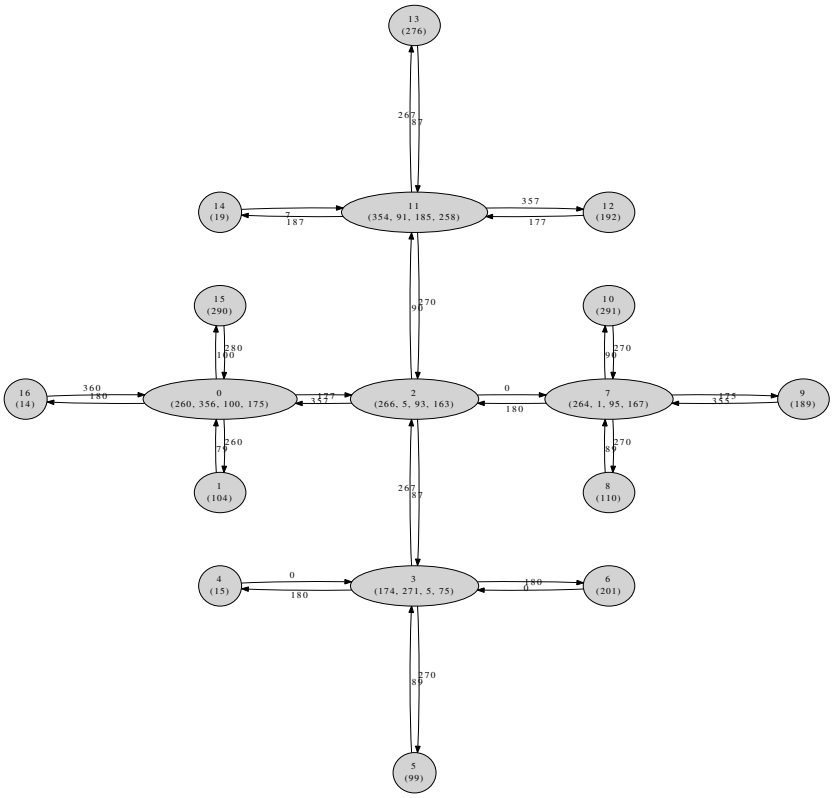


(b) DFS

Figure A.12: An example of maps for open-loop environment with 13 edges generated by planners using all constraints

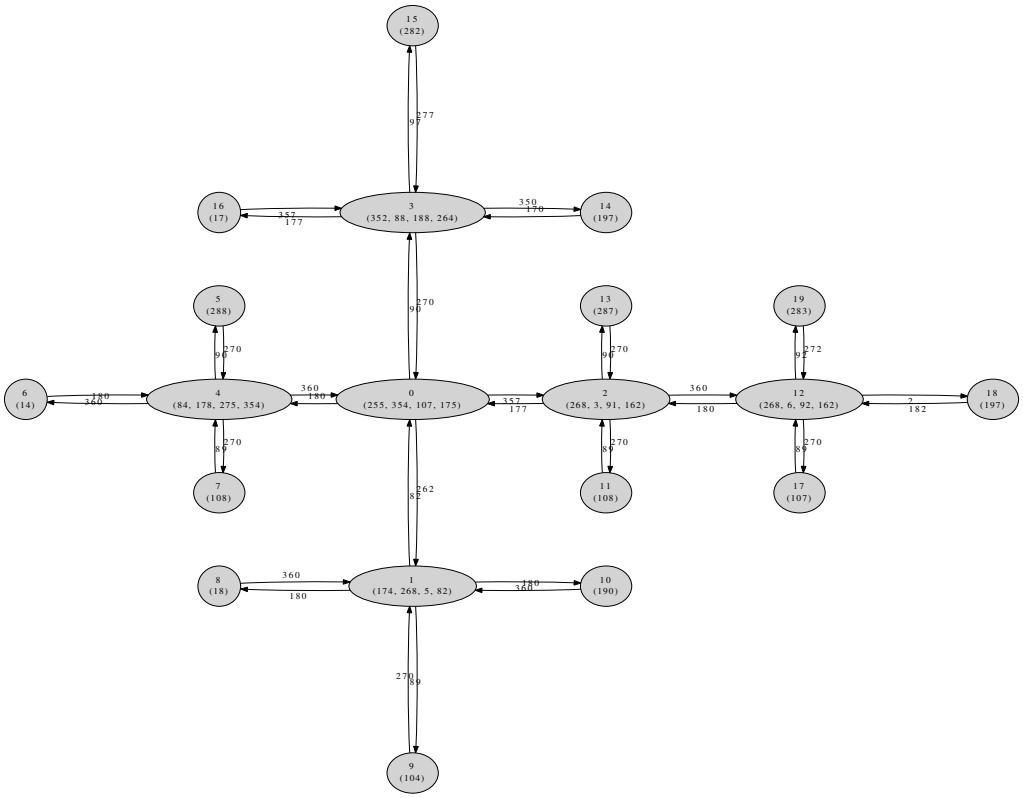


(a) BFS

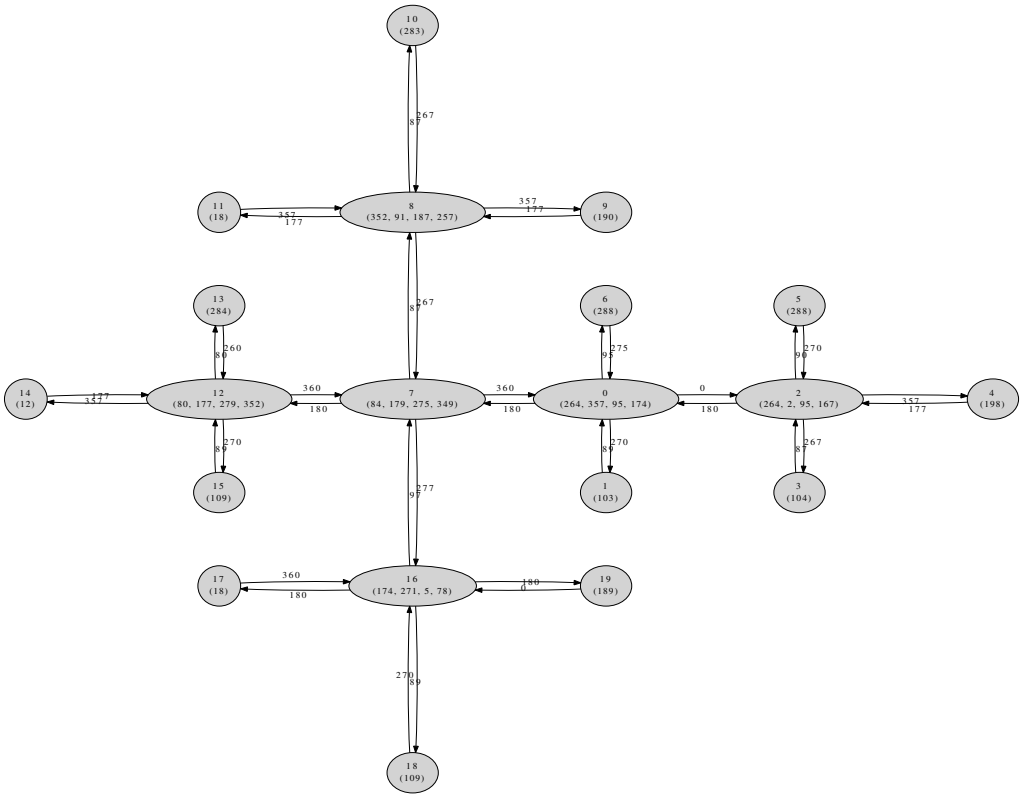


(b) DFS

Figure A.13: An example of maps for open-loop environment with 16 edges generated by planners using all constraints



(a) BFS



(b) DFS

Figure A.14: An example of maps for open-loop environment with 19 edges generated by planners using all constraints