Chapter 9
Mapping

We have seen that a robot can use its capability to detect obstacles to localize itself, based on information about the position of the obstacles or other information on the environment. This information is normally provided by a map. It is relatively easy to construct a map of industrial environments such as factories, since the machines are anchored in fixed locations. Maps are less relevant for a robotic vacuum cleaner, because the manufacturer cannot prepare maps of the apartments of every customer. Furthermore, the robots would be too difficult to use if customers had to construct maps of their apartments and change them whenever a piece of furniture is moved. It goes without saying that it is impossible to construct in advance maps of inaccessible locations like the ocean floor.

The solution is to have the robot build its own map of the environment. To build a map requires localization so that the robot knows where it is, but localization needs a map, which needs …. To overcome this chicken-and-egg problem, robots use *simultaneous localization and mapping (SLAM)* algorithms. To perform SLAM robots use information known to be valid even in unexplored parts of the environment, refining the information during the exploration.

This is what humans did to create geographic maps. Observations of the sun and stars were used for localization and maps were created as exploration proceeded. Initially, tools for localization were poor: it is relatively easy to measure latitude using a sextant to observe the height of the sun at noon, but accurate measurements of longitude were impossible until accurate clocks called chronometers were developed in the late eighteenth century. As localization improved so did maps, including not only land and seacoasts, but also terrain features like lakes, forests and mountains, as well as artificial structures like buildings and roads.

Sections 9.1 and 9.2 introduce methods of representing maps in a computer. Section 9.3 describes how a robot can create a map using the frontier algorithm. Section 9.4 explains how partial knowledge of the environment helps in constructing

M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_9

a map. A SLAM algorithm is the subject of the final three sections. The algorithm is first presented in Sect. 9.5 using a relatively simple example. Activities for SLAM are collected in Sects. 9.6 and 9.7 explains the formal algorithm.

9.1 Discrete and Continuous Maps

We are used to graphical maps that are printed on paper, or, more commonly these days, displayed on computers and smartphones. A robot, however, needs a non-visual representation of a map that it can store in its memory. There are two techniques for storing maps: discrete maps (also called *grid maps*) and continuous maps.

Figure 9.1a shows an 8×8 grid map with a triangular object. The location of the object is stored as a list of the coordinates of each grid cell covered by the object. The object in the figure consists of the cells at:

$$(5, 3),\ (5, 4),\ (5, 5)\ ,\ (4, 5)\ ,\ (5, 6),\ (4, 6),\ (3, 6)\ .$$

Figure 9.1b shows a *continuous map* of the same object. Instead of storing the positions of the object, the coordinates of boundary positions are stored:

$$A = (6, 3),\ B = (3, 7),\ C = (6, 7)\ .$$

Discrete maps are not very accurate: it is hard to recognize the object in Fig. 9.1a as a triangle. To improve accuracy, a finer grid must be used: 16×16 or even 256×256. Of course, as the number of grid points increases, so must the size of the memory in the robot. In addition, a more powerful computer must be used to process the grid

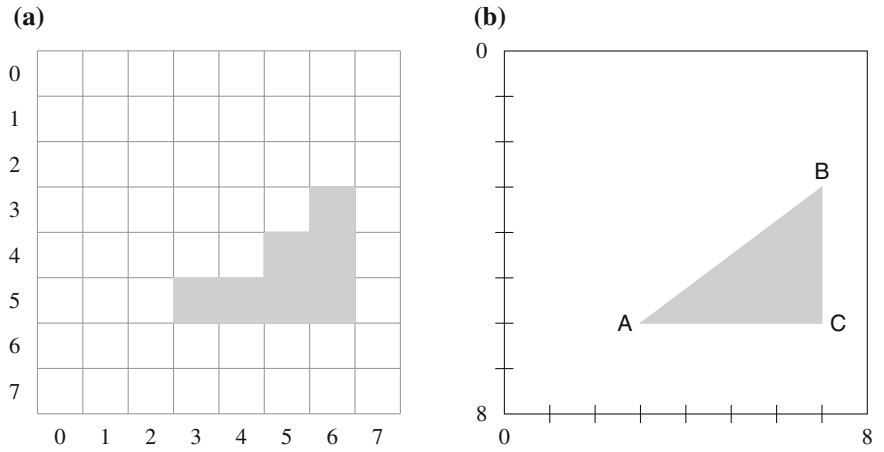

Fig. 9.1 **a** A discrete map of the occupied cells of an object, **b** A continuous map of the same object

cells. Mobile robots have constraints on weight, cost, battery capacity, and so on, so very fine grids may not be practical.

If the objects in the environment are few and have a simple shape, a continuous map is more efficient, in addition to being more accurate. In Fig. 9.1b, three pairs of numbers represent the triangle with far greater accuracy than the seven pairs of the discrete map. Furthermore, it is easy to compute if a point is in the object or not using analytic geometry. However, if there are many objects or if they have very complex shapes, continuous maps are no longer efficient either in memory or in the amount of computation needed. The object in Fig. 9.1b is bounded by straight lines, but if the boundary were described by high-order curves, the computation would become difficult. Consider a map with 32 objects of size one, none of which touch each other. The discrete map would have 32 coordinates, while the continue map would need to store the coordinates of the four corners of each object.

In mobile robotics, discrete maps are commonly used to represent maps of environments, as we did in Chap. 8.

9.2 The Content of the Cells of a Grid Map

A geographic map uses conventional notations to describe the environment. Colors are used: green for forests, blue for lakes, red for highways. Symbols are used: dots of various sizes to denote villages and towns, and lines for roads, where the thickness and color of a line is used to indicate the quality of the road. Robots use grid maps where each cell stores a number and we need to decide what a number encodes.

The simplest encoding is to allocate one bit to each cell. A value of 1 indicates that an object exists in that cell and a value of 0 indicates that the cell is empty. In Fig. 9.1a, gray represents the value 1 and white represents the value 0.

However, sensors are not accurate and it may be difficult to be certain if a cell is occupied by an object or not. Therefore, it makes sense to assign a probability to each cell indicating how certain we are that the object is in that cell. Figure 9.2 is a copy of Fig. 9.1a with the probabilities listed for each cell. Cells without a number are assumed to have a probability of 0.

It can be seen that cells with a probability of at least 0.7 are the ones considered in Fig. 9.1a to be cells occupied by the object. Of course, we are free to choose some other threshold, for example 0.5, in which case more cells are considered to be occupied. In this example, we know that the object is triangular, so we can see that a threshold of 0.5 makes the object bigger than it actually is, while the higher threshold 0.7 gives a better approximation.

0								
1								
2						.4	.6	.3
3			.1	.2	.4	.6	.9	.5
4			.3	.6	.5	.7	.8	.6
5			.4	.8	1.0	1.0	.9	.5
6			.2	.3	.2	.3	.3	.1
7								
	0	1	2	3	4	5	6	7

Fig. 9.2 A probabilistic grid map

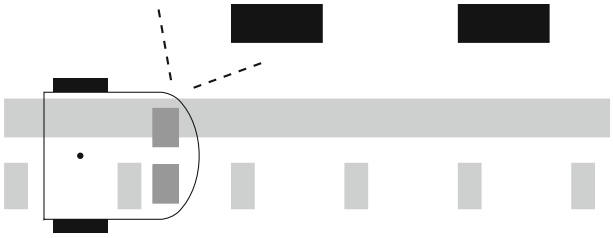

Fig. 9.3 The *black rectangles* are the obstacles to be measured. The *gray line* guides the robot and the *gray marks* are used for localization

Activity 9.1: Probabilistic map of obstacles

- Place your robot on a line in front of a set of obstacles that the robot can detect with a lateral sensor (Fig. 9.3). If you implemented localization as described in Activity 8.4, you can use this information to establish where the robot is. Otherwise, draw regular lines on the ground that can be read by the robot for localization. Build a probabilistic map of the obstacles.
- How do the probabilities change when obstacles are added or removed?

9.3 Creating a Map by Exploration: The Frontier Algorithm

Consider a robotic vacuum cleaner newly let loose in your apartment. Obviously, it does not come pre-programmed with a map of your apartment. Instead, it must explore the environment to gather information that will be used to construct its own map. There are several ways of exploring the environment, the simplest of which is random exploration. The exploration will be much more efficient if the robot has a partial map that it can use to guide its exploration.

9.3.1 Grid Maps with Occupancy Probabilities

The map in Fig. 9.4 is a grid map where each cell is labeled with its *obstacle probability*, which is the probability that there is an obstacle in the cell. The obstacle can be a wall, a table or anything that does not allow the robot to pass through this cell. The question marks represent cells that have not yet been explored. In the absence of any knowledge about the contents of a cell, we can assume that the probability that there is an obstacle there is 0.5, since it could just as easily be occupied or not. A question mark is used instead of the value 0.5 to clarify the unexplored status of the cells.

The center of the map is free from obstacles and the occupancy probabilities of these cells, called *open cells*, are low (0.1 or 0.2). There are three known obstacles, at the top right, the top left and the bottom center. The obstacles are characterized by high occupancy probabilities (0.9 or 1.0) and are denoted by gray cells. A *frontier cell* is an open cell that is adjacent (left, right, up, down) to one or more unknown

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	1	?	?	?	?	?	0.9	1	0.9	?	?
?	?	?	?	?	1	0.1	0.1	?	?	?	1	0.2	1	?	?
?	?	?	?	?	1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	1	?	?
?	?	?	?	?	0.9	0.1	0.1	0.1	0.1	0.1	0.1	0.2	1	?	?
?	?	?	?	?	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	?	?	?
?	?	?	?	?	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	?	?	?
?	?	?	?	?	?	?	0.2	0.1	0.1	0.2	0.2	0.1	?	?	?
?	?	?	?	?	?	?	1	1	0.9	1	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Fig. 9.4 Grid map of an environment with occupancy probabilities

cells. The set of frontier cells is called the *frontier*. The red lines of small squares in Fig. 9.4 represent the boundary between the frontier and the unknown cells in the environment. The unknown cells adjacent to the frontier are the interesting ones that should be explored in order to expand the current map.

9.3.2 The Frontier Algorithm

The *frontier algorithm* is used to expand the map by exploring the frontier. The robot moves to the closest frontier cell, senses if there are obstacles in unknown adjacent cells and updates the map accordingly.

The grid map in Fig. 9.5 is the same as the map in Fig. 9.4 with the addition of the robot which occupies a cell colored blue. The frontier cell closest to the robot is the cell two steps above its initial location. The arrow shows that the robot has moved to that cell. The robot uses its local sensors to determine if there are obstacles in adjacent unknown cells. (The sensors can detect obstacles in all eight adjacent cells, including the ones on the diagonal.) Suppose that the cell to its upper left certainly contains an obstacle (probability 1.0), while the cells directly above and to the right almost certainly do not contain an obstacle (probability 0.1). Figure 9.6 shows the map updated with this new information and the new position of the frontier.

Figure 9.7 shows the result of the next iteration of the algorithm. The robot has moved up one cell to the closest frontier cell, detected obstacles in the two adjacent unknown cells and updated the map. The upper right obstacle is completely known and there is no frontier cell in the vicinity of the current position of the robot.

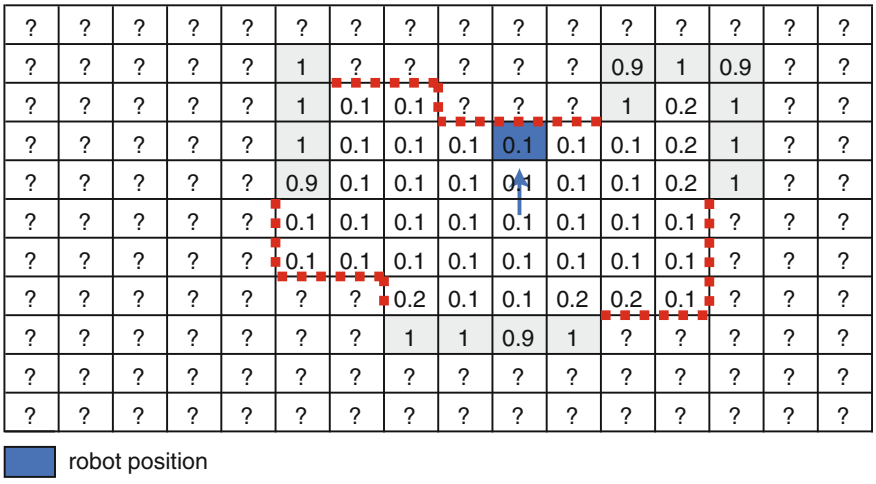

Fig. 9.5 The robot moves to the frontier

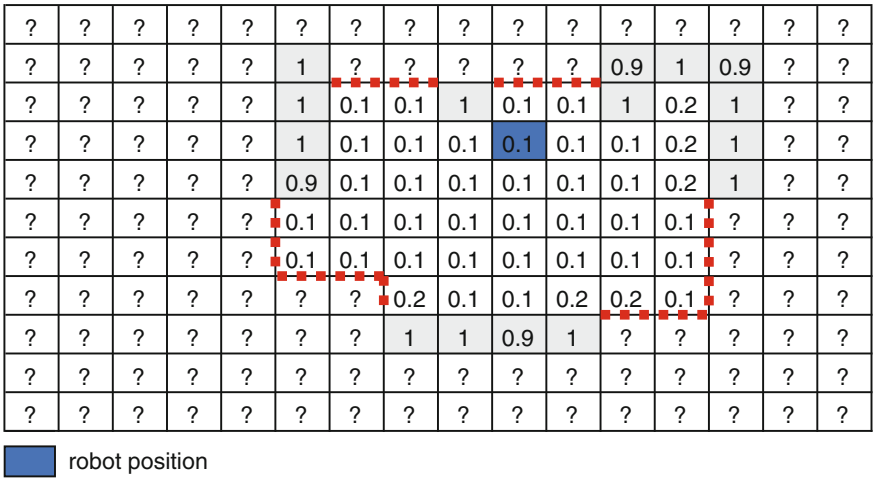

Fig. 9.6 The robot updates unknown cells adjacent to the frontier

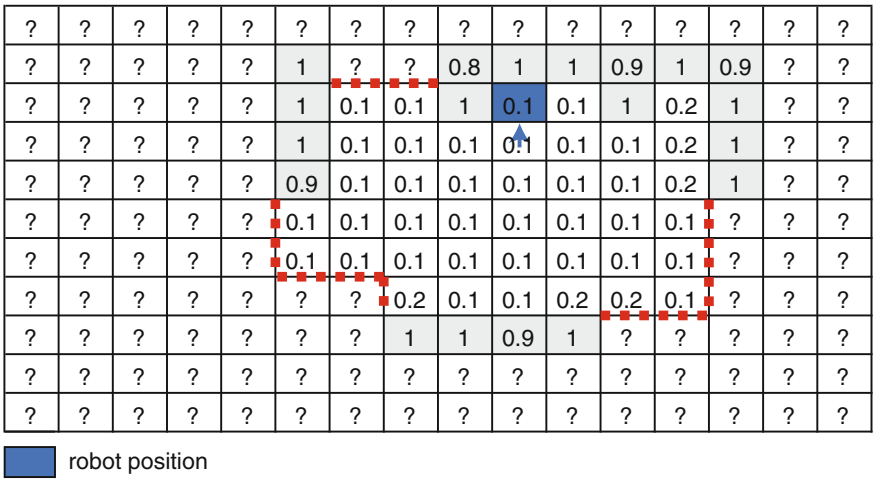

Fig. 9.7 Second iteration of the frontier algorithm

Figure 9.8 shows the next iteration of the algorithm. The robot is blocked by the upper right obstacle and has to avoid it as it moves to the nearest frontier cell.

Figure 9.9 shows the completed map constructed by the robot after it has explored the entire frontier as shown by the path with the blue arrows.

Algorithm 9.1 formalizes the frontier algorithm. For simplicity, the algorithm recomputes the frontier at each step. A more sophisticated algorithm would examine the cells in a neighborhood of the robot's position and add or remove the cells whose status as frontier cells has changed.

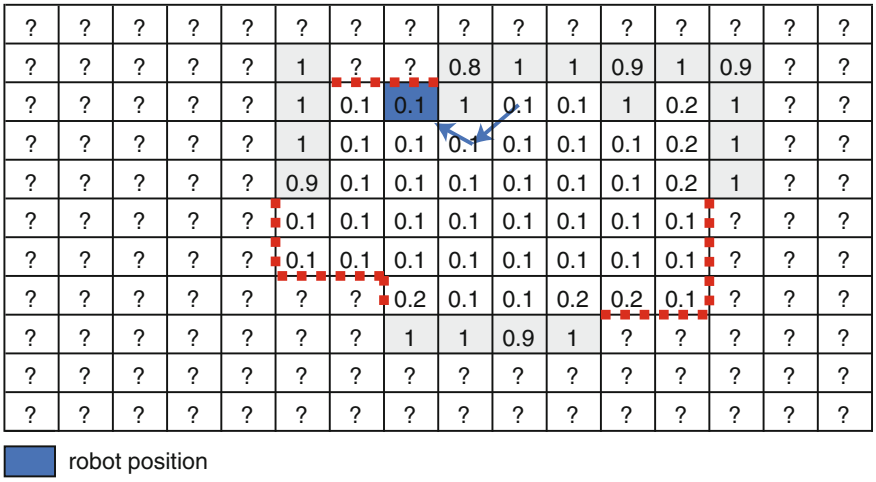

Fig. 9.8 The robot avoids an obstacle while moving to the next frontier

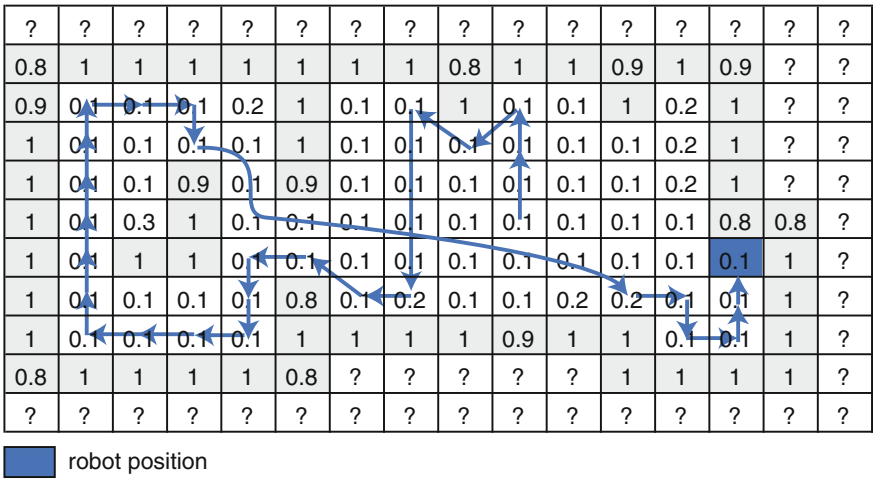

Fig. 9.9 The map constructed by the frontier algorithm and the path explored by the robot

The example to which we applied the frontier algorithm is a relatively simple environment consisting of two rooms connected by a door (at the sixth column from the left in Fig. 9.9), but otherwise closed to the outside environment. However, the frontier algorithm works in more complex environments.

The frontier algorithm can be run in parallel by multiple robots. Each robot will explore that portion of the frontier that is closest to its position. The robots share their partial maps so that consistency of the maps is maintained. Since each robot explores a different area of the environment, the construction of the map will be much more efficient.

Algorithm 9.1: Frontier algorithm

```
        float array grid                    // Grid map
        cell list frontier                  // List of frontier cells
        cell robot                          // Cell with robot
        cell closest                        // Closest cell to robot
        cell c                              // Index over cells
        float low                           // Low occupancy probability
 1: loop
 2:     frontier ← empty
 3:     for all known cells c in the grid
 4:         if grid(c) < low and
 5:             exists unknown neighbor of c
 6:                 append c to frontier
 7:     exit if frontier empty

 8:     closest ← cell in frontier nearest robot
 9:     robot ← closest
10:     for all unknown neighbors c of closest
11:             sense if c is occupied
12:             mark grid(c) with occupancy probability
```

9.3.3 Priority in the Frontier Algorithm

The frontier algorithm can use criteria other than distance to choose which frontier to explore. Consider the exploration of the grid map shown in Fig. 9.10. The robot is at cell (3, 3) marked with the blue circle. There are six known obstacle cells and five known open cells, of which the three cells (1, 3), (2, 2), (3, 2), marked with red squares, are the frontier cells. (Here the diagonal neighbors are not considered as adjacent.)

In Algorithm 9.1, the robot uses distance to a frontier cell as the criterion for deciding where to move. In Fig. 9.10 the cell to the robot's left at (3, 2) is the closest cell since it is only one step away, while the other two frontier cells are two steps away. We can consider a different criterion by taking into account the number of unknown cells adjacent to a frontier cell. Starting with a frontier cell with more

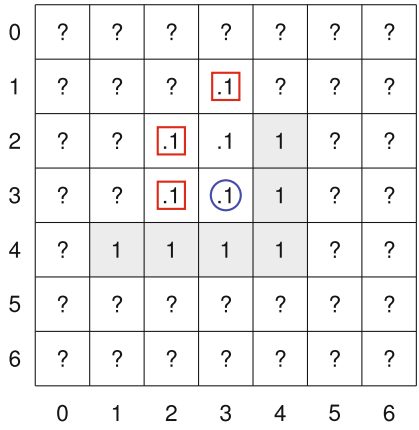

Fig. 9.10 Exploration of a labyrinth

unknown cells might make the algorithm more efficient. We define the priority of a frontier cell as:

$$p_{cell} = \frac{a_{cell}}{d_{cell}},$$

where a_{cell} is the number of adjacent unknown cells and d_{cell} is the distance from the robot. The priorities of the three frontier cells are:

$$p_{(3,2)} = 1/1 = 1, \quad p_{(2,2)} = 2/2 = 1, \quad p_{(1,3)} = 3/2 = 1.5\,.$$

The priority of cell (1, 3) is the highest and the exploration starts there.

Activity 9.2: Frontier algorithm

- Implement the frontier algorithm. You will need to include an algorithm to move accurately from one cell to another and an algorithm to move around obstacles.
- Run the program on the grid map in Fig. 9.9. Do you get the same path? If not, why not?
- Run the program on the grid map in Fig. 9.10.
- Modify Algorithm 9.1 to use the priority described in Sect. 9.3.3. Is the path different from the one generated by the original program?
- Try to implement the frontier algorithm on your robot. What is the most difficult aspect of the implementation?

9.4 Mapping Using Knowledge of the Environment

Now that we know how to explore an environment, let us consider how to build a map during the exploration. In Chap. 8 we saw that a robot can localize itself with the help of external landmarks and their representation in a map. Without such external landmarks, the robot can only rely on odometry or inertial measurement, which are subject to errors that increase with time (Fig. 5.6). How is it possible to make a map when localization is subject to large errors?

Even with bad odometry, the robot can construct a better map if it has some information on the structure of the environment. Suppose that the robot tries to construct the plan of a room by following its walls. Differences in the real speeds of the left and right wheels will lead the robot to conclude that the walls are not straight (Fig. 9.11a), but if the robot knows *in advance* that the walls are straight and perpendicular to each other, the robot can construct the map shown in Fig. 9.11b. When it encounters a sharp turn, it understands that the turn is a 90° corner where two walls meet, so its mapping of the angles will be correct. There will also be an error when measuring the lengths of the walls and this can lead to the gap shown in the figure between the first and last walls. The figure shows a small gap which would not be important, but if the robot is mapping a large area, the problem of *closing a loop* in a map is hard to solve because the robot has only a local view of the environment.

Consider a robotic lawnmower given the task of mowing a lawn by moving back and forth; it has to close the loop by returning to its charging station (Fig. 9.12). It is not possible to implement this behavior using odometry alone, since small errors in velocity and heading lead to large errors in the position of the robot. It is highly unlikely that through odometry alone the robot will mow the entire surface of the lawn and return to its charging station. Landmarks such as signaling cables in the ground need to be used to close the loop.

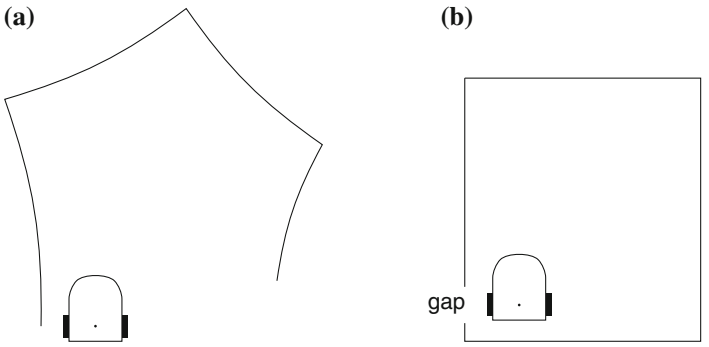

Fig. 9.11 **a** Perceived motion of a robot based on odometry, **b** Odometry together with knowledge of the geometry of the walls

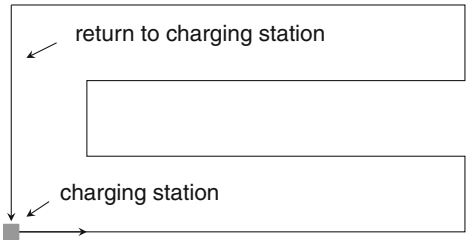

Fig. 9.12 A robotic lawnmower mowing an area and returning to its charging station

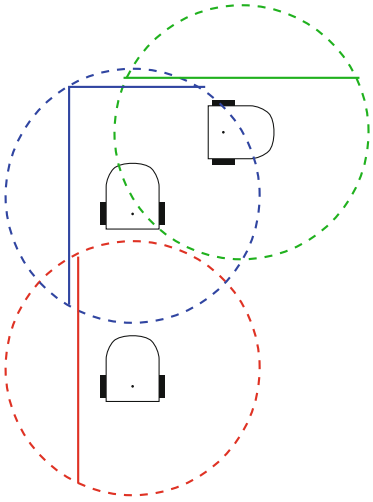

Fig. 9.13 Long range sensor measurements can detect overlap

Map construction can be significantly improved by using sensor data that can give information on regular features in the environment, in particular, at a long range. The regular features can be lines on the ground, a global orientation, or the detection of features that *overlap* with other measurements. Suppose that we have a distance sensor that can measure distances over a large area (Fig. 9.13). The measurement over a large area enables the robot to identify features such as walls and corners from a measurement taken at a single location. Large area measurements facilitate identifying overlaps between the local maps that are constructed at each location as the robot moves through the environment. By comparing local maps, the localization can be corrected and map accurately updated. This is the topic of the SLAM algorithm described in the next section.

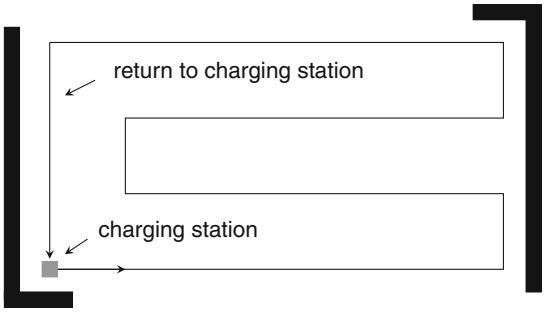

Fig. 9.14 A robotic lawnmower with landmarks

Activity 9.3: Robotic lawnmower

- Write a program that causes a robotic lawnmower to move along a path using odometry alone (Fig. 9.12). Run the program several times. Does the robot return to the charging station? If not, how large are the errors? Are the errors consistent or do they change from one run to the next?
- Place landmarks of black tape around the lawn (Fig. 9.14). Program your robot to recognize the landmarks and correct its localization. How large do the landmarks have to be so that the robot detects them reliably?

9.5 A Numerical Example for a SLAM Algorithm

The SLAM algorithm is quite complicated so we first compute a numerical example and later give the formal presentation.

Figure 9.15a shows a robot in a room heading towards the top of the diagram. The robot is near a corner of the room and there is a projection from the wall to the robot's left, perhaps a supporting pillar of the building. Figure 9.15b is the corresponding map. The large dot shows the position of the robot and the associated arrow shows its heading. The thick dotted line represents the real wall. The white cells represent locations that are known to be free, the gray cells represent obstacles, and the cells with question marks are still unexplored. A cell is considered to be part of the obstacle if a majority of the area of the cell is behind the wall. For example, the two horizontal segments of the projection from the wall are near the boundary of the cells they pass through, but these cells are considered part of the obstacle because almost all their area is behind the wall.

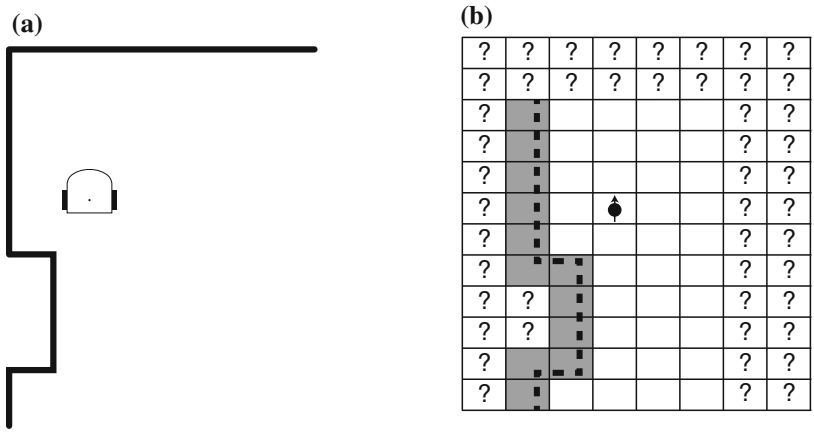

Fig. 9.15 **a** A robot near the wall of a room, **b** The corresponding map

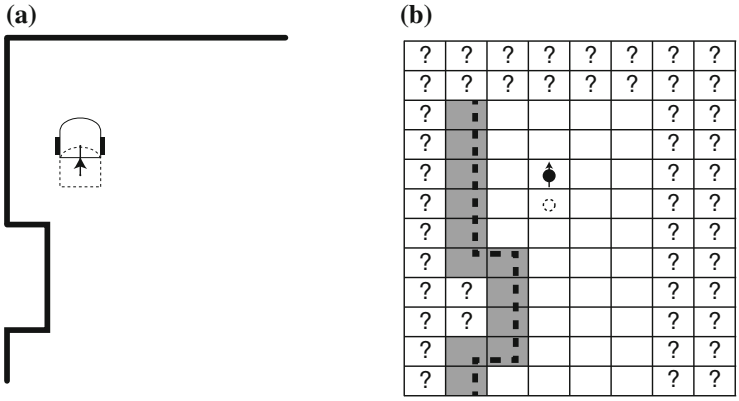

Fig. 9.16 **a** The intended motion of the robot, **b** The map for the intended motion

For the purpose of presenting the details of the SLAM algorithm, the map is highly simplified. First, the cells are much too large, roughly the same size as the robot itself. In practice, the cells would be much smaller than the robot. Second, we specify that each (explored) cell is either free (white) or it contains an obstacle (gray); real SLAM algorithms use a probabilistic representation (Sect. 9.2).

Suppose that the robot in Fig. 9.15a intends to move forwards to the new position shown in Fig. 9.16a. Figure 9.16b shows the map corresponding to the position after the intended move, where the robot has moved the height of one cell up from its initial position. Unfortunately, the right wheel moves over an area of low friction and although the robot ends up in the correct position, its heading is too far to the right. The actual position of the robot is shown in Fig. 9.17a and b is the corresponding map.

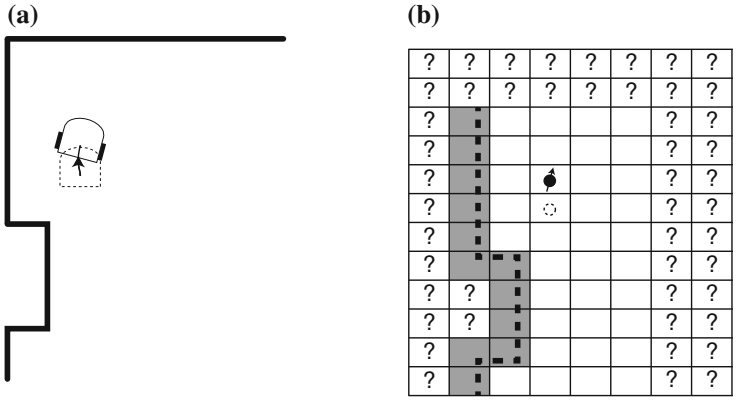

Fig. 9.17 **a** The actual motion of the robot, **b** The map for the actual motion

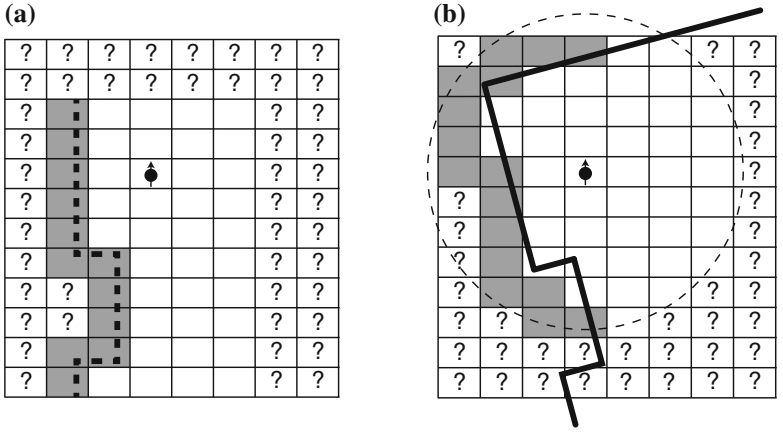

Fig. 9.18 **a** The intended perception of the robot, **b** The actual perception of the robot

Figure 9.18a (which is the same as Fig. 9.16b) shows the *intended* perception of the robot because the robot has moved one cell upwards relative to the map in Fig. 9.15b. From this position it can detect the obstacle to its left and investigate the unknown cells in front of it.

However, because of the error in odometry, the *actual* perception of the robot is different. Figure 9.18b shows the actual position of the wall as seen by the robot overlaid on top of the cells, where cells are colored gray if the majority of their area is known to be behind the wall. (Examine several cells to verify that this is true.) We assume that the robot can sense walls at a distance of up to five times the size of a cell as shown by the dashed circle and we also assume that the robot knows that any wall is one cell thick.

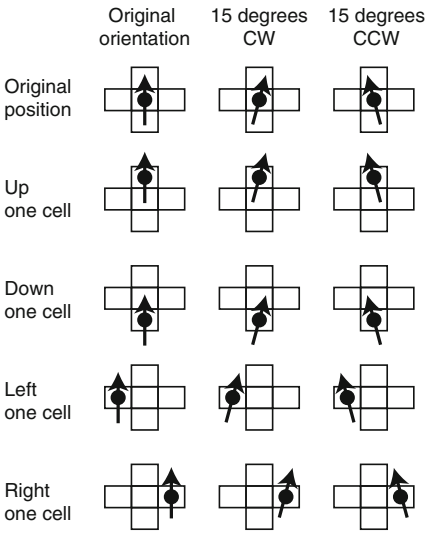

Fig. 9.19 Possible poses of the robot

There is a clear mismatch between the current map and the sensor data which should correspond to the known part of the map. Obviously, the robot is not where it is expected to be based on odometry. How can this mismatch be corrected? We assume that odometry does give a reasonable estimation of the pose (position and heading) of the robot. For each relatively small possible error in the pose, we compute what the perception of the current map would be and compare it with the actual perception computed from the sensor data. The pose that gives the best match is chosen as the actual pose of the robot and the current map updated accordingly.

In the example, assume that the robot is either in the expected cell or in one of its four neighbors (left, right, up, down) and that the heading of the robot is either correct or turned slightly to the right (15° clockwise (CW)) or slightly to the left (15° counterclockwise (CCW)). The $5 \times 3 = 15$ possible poses are shown in Figs. 9.19 and 9.20 shows the perception of the map computed from the current map for each pose. (To save space, only a 8×5 fragment of the 12×8 map is displayed.)

The next step is to choose the map that gives the best fit with the sensor measurements. First transform the 8×5 maps into 8×5 matrices, assigning -1 to empty cells, $+1$ to obstacle cells and 0 to other cells. The left matrix in Fig. 9.21 is associated with the current map and the center matrix in the figure is associated with the perception map corresponding to the pose where the robot is in the correct cell but the heading is 15° CW (the middle element of the top row of Fig. 9.20).

To compare the maps, multiply elements of corresponding cells. Let $m(i, j)$ be the (i, j)'th cell of the current map and $p(i, j)$ be the (i, j)'th cell of the perception map obtained from sensor values. $S(i, j)$, the *similarity* of (i, j)'th cell, is:

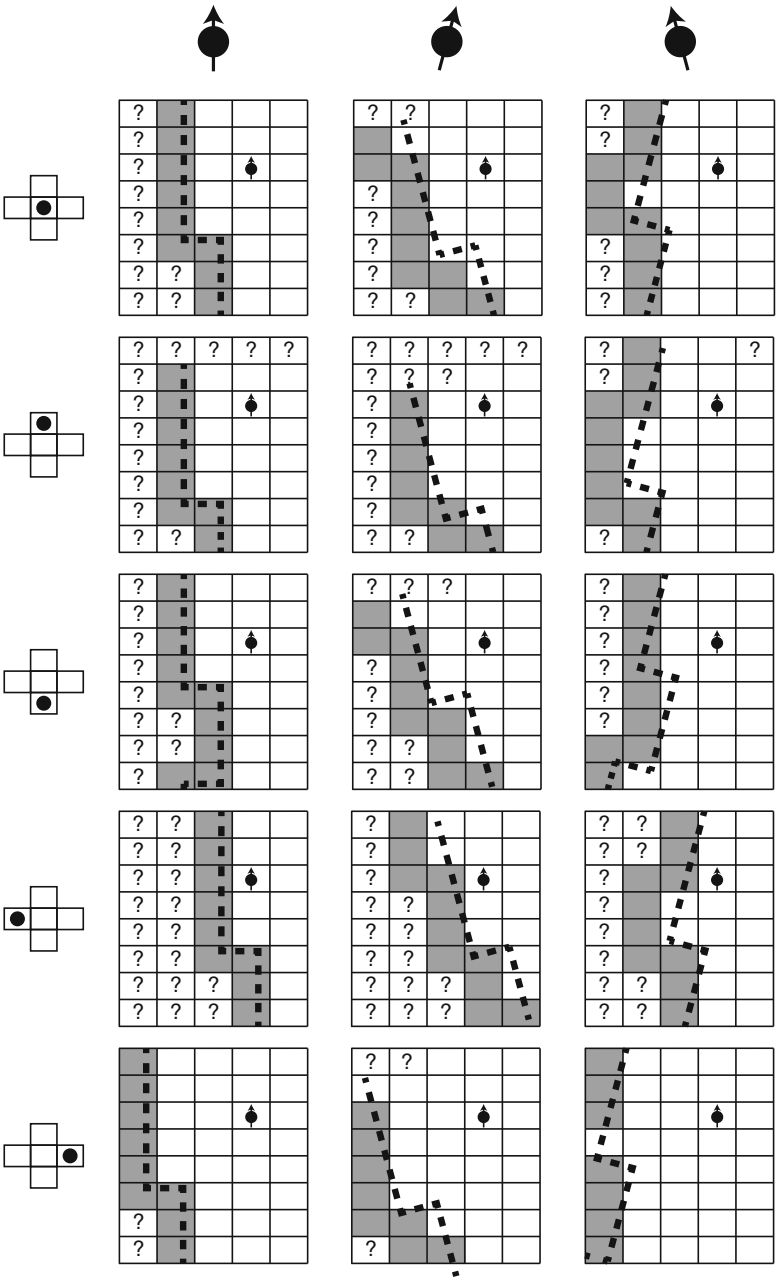

Fig. 9.20 Estimations of perception of the robot for different poses

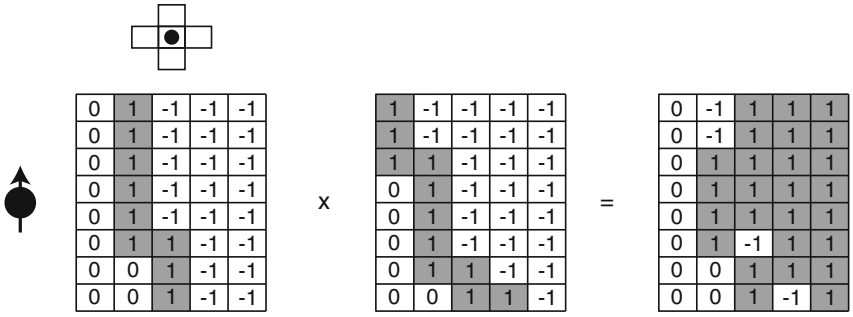

Fig. 9.21 Computation of the matching between two maps

Table 9.1 Similarity $\mathcal{S}$ of the sensor-based map with the current map

	Intended orientation	15° CW	15° CCW
Intended position	22	**32**	20
Up one cell	23	25	16
Down one cell	19	28	21
Left one cell	6	7	18
Right one cell	22	18	18

$$S(i, j) = m(i, j)\, p(i, j)\,,$$

which can also be expressed as:

$$\begin{aligned} S(i, j) &= 1 \quad \text{if } m(i, j) \neq 0,\ p(i, j) \neq 0,\ m(i, j) = p(i, j) \\ S(i, j) &= -1 \quad \text{if } m(i, j) \neq 0,\ p(i, j) \neq 0,\ m(i, j) \neq p(i, j) \\ S(i, j) &= 0 \quad \text{if } m(i, j) = 0 \text{ or } p(i, j) = 0\,. \end{aligned}$$

The right matrix in Fig. 9.21 shows the result of this computation for the two matrices to its left. There are a lot of 1's meaning that the matrices are similar and thus we can conclude that the perception maps are similar. For a quantitative result, compute the sum of the similarities to obtain a single value for any pair m, p:

$$\mathcal{S} = \sum_{i=1}^{8} \sum_{j=1}^{5} S(i, j)\,.$$

Table 9.1 gives the values of the similarity $\mathcal{S}$ for all the perception maps in Fig. 9.20 compared with the current map. As expected, the highest similarity is obtained for the map corresponding to the pose with the correct position and with the heading rotated by 15° CW.

Once we have this result, we correct the pose of the robot and use data from the perception map to update the current map stored in the robot's memory (Fig. 9.22).

9.6 Activities for Demonstrating the SLAM Algorithm

The following two activities demonstrate aspects of the SLAM algorithm. Activity 9.4 follows the algorithm and is intended for implementation in software. Activity 9.5 demonstrates a key element of the algorithm that can be implemented on an educational robot.

The activities are based on the configuration shown in Fig. 9.23. The robot is located at the origin of the coordinate system with pose $((x, y), \theta) = ((0, 0), 0°)$.[1] Given the uncertainty of the odometry, the robot might actually be located at any of the coordinates $(-1, 0)$, $(1, 0)$, $(0, -1)$, $(0, 1)$ and its orientation might be any of $-15°$, 0, $-15°$ (as shown by the dashed arrows), giving 15 possible poses. The three gray dots at coordinates $(2, 2)$, $(2, 0)$, $(2, -2)$ represent known obstacles on the current map. (To save space the dots are displayed at coordinates $(2, 1)$, $(2, 0)$, $(2, -1)$.) The obstacles can be sensed by multiple horizontal proximity sensors, but for the purposes of the activities we specify that there are three sensors.

In the SLAM algorithm the *perception* of an obstacle is the value returned by a distance sensor. To avoid having to define model for the sensors, the activities will define a perception as the *distance* and *heading* from the sensor to the obstacle.

Activity 9.4: Localize the robot from the computed perceptions

- For each of the 15 poses compute the set of perceptions of each obstacle. For example, if the robot is at the pose $((0.0, 1.0), -15.0°)$, the set of perceptions of the three obstacles is:

$$[(2.2, 41.6°),\ (2.2, -11.6°),\ (3.6, -41.3°)]\,.$$

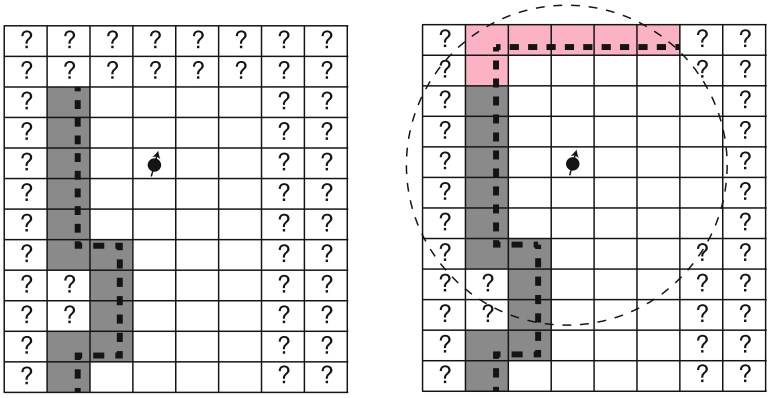

Fig. 9.22 Map before and after the update using data from the perception map

[1] It is convenient to take the heading of the robot as 0°.

- Given a set of measured perceptions, compute their similarities to the perceptions of the 15 poses of the robot. Choose the pose with the best similarity as the actual pose of the robot. For example, for the set of measured perceptions:

$$[(2.0, 32.0°),\ (2.6, -20.0°),\ (3.0, -30.0°)]\,,$$

  and the similarity computed as the sum of the absolute differences of the elements of the perceptions, the pose with the best similarity is $((0.0, 1.0), -15.0°)$.
- Experiment with different similarity functions.
- We have computed that the robot's pose is approximately $((0.0, 1.0), -15.0°)$. Suppose that a new obstacle is placed at coordinate $(3, 0)$. Compute the perception (d, θ) of the object from this pose and then compute the coordinate (x, y) of the new obstacle. The new obstacle can be added to the map with this coordinate.
- Is the computed coordinate significantly different from the coordinate $(3, 0)$ that would be obtained if the robot were at its intended pose $((0, 0), 0°)$?

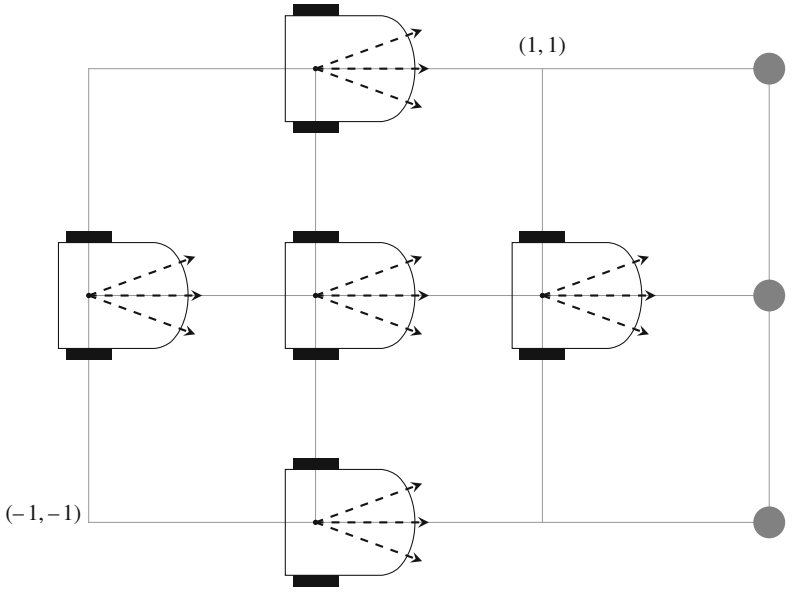

Fig. 9.23 Configuration for the SLAM algorithm

Activity 9.5: Localize the robot from the measured perceptions

- Place three objects as shown in Fig. 9.23.
- Write a program that stores the set of values returned by the three horizontal proximity sensors. Place your robot successively at each of the 15 poses and record the sets of values. You now have a database of perceptions: a set of three sensor readings for each pose.
- Place the robot at one of the poses and store the set of values returned by the sensors. Compute the similarity of this set to each of the sets in the database. Display the pose associated with the best similarity.
- Experiment with placing the robot at various poses and several times at each pose. How precise is the determination of the pose?
- Experiment with different similarity functions.

9.7 The Formalization of the SLAM Algorithm

Algorithm 9.2 is a SLAM algorithm that finds the position whose perception map is closest to the perception map obtained from the sensor data. The robot is localized to this position and the map updated to what is perceived at this position.

The algorithm is divided into three phases. In the first phase (lines 2–4), the robot moves a short distance and its new position is computed by odometry. The perception map at this location is obtained by analyzing the sensor data.

Assuming that the odometry error is relatively small, we can define a set of test positions where the robot might be. In the second phase (lines 5–8), the expected map at each of these positions is computed and compared with the current map. The best match is saved.

In the third phase (lines 9–11), the position with the best match becomes the new position and the current map is updated accordingly.

In practice, the algorithm is somewhat more complicated because it has to take into account that the perception map obtained from the sensors is limited by the range of the sensors. The overlap will be partial both because the sensor range does not cover the entire current map and because the sensors can detect obstacles and free areas outside the current map. Therefore, the size of the perceived map **p** will be much smaller than the expected map **e** and the function compare(**p**, **e**) will only compare the areas that overlap. Furthermore, when updating the current map, areas not previously in the map will be added. In Fig. 9.22 there are cells in the current map that are outside the five-cell radius of the sensor and will not be updated. The light red cells were unknown in the current map as indicated by the question marks, but in the perception map they are now known to be part of the obstacle. This information is used to update the current map to obtain a new current map.

Algorithm 9.2: SLAM

```
                matrix m ← partial map          // Current map
                matrix p                        // Perception map
                matrix e                        // Expected map
                coordinate c ← initial position // Current position
                coordinate n                    // New position
                coordinate array T              // Set of test positions
                coordinate t                    // Test position
                coordinate b ← none             // Best position
 1: loop
 2:     move a short distance
 3:     n ← odometry(c)                         // New position based on odometry
 4:     p ← analyze sensor data

 5:     for every t in T                        // T is the positions around n
 6:         e ← expected(m, t)                  // Expected map at test position
 7:         if compare(p,e) better than b
 8:             b ← t                           // Best test position so far

 9:     n ← b                                   // Replace new position by best position
10:     m ← update(m,p,n)                       // Update map based on new position
11:     c ← n                                   // Current position is new position
```

9.8 Summary

Accurate robotic motion in an uncertain environment requires that the robot have a map of the environment. The map must be maintained in the robot's computer; it can be either a grid map of cells or a graph representation of a continuous map. In an uncertain environment, a map will typically not be available to the robot before it begins its tasks. The frontier algorithm is used by a robot to construct a map as it explores its surroundings. More accurate maps can be constructed if the robot has some knowledge of its environment, for example, that the environment is the inside of a building consisting of rectangular rooms and corridors. Simultaneous localization and mapping (SLAM) algorithms use an iterative process to construct a map while also correcting for errors in localization.

9.9 Further Reading

Two textbooks on path and motion planning are [4, 5]. See also [6, Chap. 6].

The frontier algorithm was proposed by Yamauchi [8] who showed that a robot could use the algorithm to successfully explore an office with obstacles.

Algorithms for SLAM use probability, in particular Bayes rule. Probabilistic methods in robotics are the subject of the textbook [7].

A two-part tutorial on SLAM by Durrant-Whyte and Bailey can be found in [1, 2]. A tutorial on graph-based SLAM is [3].

Sebastian Thrun's online course *Artificial Intelligence for Robotics* is helpful: https://classroom.udacity.com/courses/cs373.

References

1. Bailey, T., Durrant-Whyte, H.: Simultaneous localization and mapping: part ii. IEEE Robot. Autom. Mag. **13**(3), 108–117 (2006)
2. Durrant-Whyte, H., Bailey, T.: Simultaneous localization and mapping: part i. IEEE Robot. Autom. Mag. **13**(2), 99–110 (2006)
3. Grisetti, G., Kümmerle, R., Stachniss, C., Burgard, W.: A tutorial on graph-based slam. IEEE Intell. Transp. Syst. Mag. **2**(4), 31–43 (2010)
4. Latombe, J.C.: Robot Motion Planning. Springer, Berlin (1991)
5. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
6. Siegwart, R., Nourbakhsh, I.R., Scaramuzza, D.: Introduction to Autonomous Mobile Robots, 2nd edn. MIT Press, Cambridge (2011)
7. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. MIT Press, Cambridge (2005)
8. Yamauchi, B.: A frontier-based approach for autonomous exploration. In: IEEE International Symposium on Computational Intelligence in Robotics and Automation, pp. 146–151 (1997)