bathrooms. It would perhaps have been quicker in this case to turn left, rather than right, at the first obstacle but that strategy might give a worse outcome somewhere else. Many variants of the *bug* algorithm have been developed, but while they improve the performance for one type of environment they can degrade performance in others. Fundamentally the robot is limited by not using a map. It cannot see the big picture and therefore takes paths that are locally, rather than globally, optimal.

5.2 Map-Based Planning

The key to achieving the *best* path between points A and B, as we know from everyday life, is to use a map. Typically best means the shortest distance but it may also include some penalty term or cost related to traversability which is how easy the terrain is to drive over – it might be quicker to travel further but faster over better roads. A more sophisticated planner might also consider the size of the robot, the kinematics and dynamics of the vehicle and avoid paths that involve turns that are tighter than the vehicle can execute. Recalling our earlier definition of a robot as a

goal oriented machine that can sense, plan and act,

this section concentrates on planning.

There are many ways to represent a map and the position of the vehicle within the map. Graphs, as discussed in Appendix I, can be used to represent places and paths between them. Graphs can be efficiently searched to find a path that minimizes some measure or cost, most commonly the distance traveled. A simpler and very computer-friendly representation is the occupancy grid which is widely used in robotics.

An occupancy grid treats the world as a grid of cells and each cell is marked as occupied or unoccupied. We use zero to indicate an unoccupied cell or free space where the robot can drive. A value of one indicates an occupied or nondriveable cell. The size of the cell depends on the application. The memory required to hold the occupancy grid increases with the spatial area represented and inversely with the cell size. However for modern computers this representation is very feasible. For example a cell size 1×1 m requires▸ just 125 kbyte km^{-2}.

Considering a single bit to represent each cell. The occupancy grid could be compressed or could be kept on a disk with only the local region in memory.

In the remainder of this section we use code examples to illustrate several different planners and all are based on the occupancy grid representation. To create uniformity the planners are all implemented as classes derived from the `Navigation` superclass which is briefly described on page 133. The *bug2* class we used previously was also an instance of this class so the remaining examples follow a familiar pattern.

Once again we state some assumptions. Firstly, the robot operates in a grid world and occupies one grid cell. Secondly, the robot does not have any nonholonomic constraints and can move to any neighboring grid cell. Thirdly, it is able to determine its position on the plane. Fourthly, the robot is able to use the map to compute the path it will take.

In all examples we will use the house map introduced in the last section and find paths from bedroom 3 to the kitchen. These parameters can be varied, and the occupancy grid changed using the tools described above.

5.2.1 Distance Transform

Consider a matrix of zeros with just a single nonzero element representing the goal. The distance transform of this matrix is another matrix, of the same size, but the value of each element is its distance▸ from the original nonzero pixel. For robot path planning we use the default Euclidean distance. The distance transform is actually an image processing technique and will be discussed further in Chap. 12.

The distance between two points (x_1, y_1) and (x_2, y_2) where $\Delta_x = x_2 - x_1$ and $\Delta_y = y_2 - y_1$ can be Euclidean $\sqrt{\Delta_x^2 + \Delta_y^2}$ or CityBlock (also known as Manhattan) distance $|\Delta_x| + |\Delta_y|$.

Making a map. An occupancy grid is a matrix that corresponds to a region of 2-dimensional space. Elements containing zeros are free space where the robot can move, and those with ones are obstacles where the robot cannot move. We can use many approaches to create a map. For example we could create a matrix filled with zeros (representing all free space)

```
>> map = zeros(100, 100);
```

and use MATLAB operations such as

```
>> map(40:50,20:80) = 1;
```

or the MATLAB builtin matrix editor to create obstacles but this is quite cumbersome. Instead we can use the Toolbox map editor makemap to create more complex maps using an interactive editor

```
>> map = makemap(100)
```

that allows you to add rectangles, circles and polygons to an occupancy grid. In this example the grid is 100×100. See online help for details.

The occupancy grid in Fig. 5.4 was derived from a scanned image but online buildings plans and street maps could also be used.

Note that the occupancy grid is a matrix whose coordinates are conventionally expressed as (row, column) and the row is the vertical dimension of a matrix. We use the Cartesian convention of a horizontal x-coordinate first, followed by the y-coordinate therefore the matrix is always indexed as `y,x` in the code.

To use the distance transform for robot navigation we create a DXform object, which is derived from the Navigation class

```
>> dx = DXform(house);
```

and then create a plan to reach a specific goal

```
>> dx.plan(place.kitchen)
```

which can be visualized

```
>> dx.plot()
```

as shown in Fig. 5.5. We see the obstacle regions in red overlaid on the distance map whose grey level at any point indicates the distance from that point to the goal, in grid cells, taking into account travel *around* obstacles.

The hard work has been done and to find the shortest path from *any* point to the goal we simply consult or query the plan.◀ For example a path from the bedroom to the goal is

For the bug2 algorithm there was no planning step so the query in that case was the simulated robot querying its proximity sensors.

```
>> dx.query(place.br3, 'animate');
```

which displays an animation of the robot moving toward the goal. The path is indicated by a series of green dots as shown in Fig. 5.5.◀

By convention the plan is based on the goal location and we query for a start location, but we could base the plan on the start position and then query for a goal.

The plan is the distance map. Wherever the robot starts, it moves to the neighboring cell that has the smallest distance to the goal. The process is repeated until the robot reaches a cell with a distance value of zero which is the goal.

If the path method is called with an output argument the path

```
>> p = dx.query(place.br3);
```

is returned as a matrix, one row per point, which we can visualize overlaid on the occupancy grid and distance map

```
>> dx.plot(p)
```

The path comprises

```
>> numrows(p)
ans =
   336
```

points which is considerably shorter than the path found by *bug2*.

This navigation algorithm has exploited its global view of the world and has, through exhaustive computation, found the shortest possible path. In contrast, *bug2* without

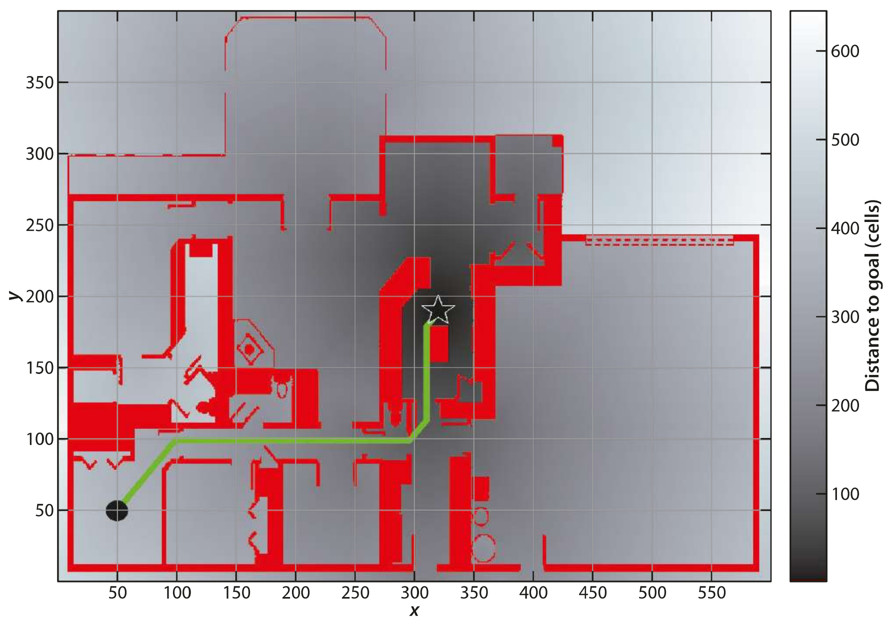


Fig. 5.5.
The distance transform path. Obstacles are indicated by *red cells*. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the *scale* on the right-hand side

the global view has just bumped its way through the world. The penalty for achieving the optimal path is computational cost. This particular implementation of the distance transform is iterative. Each iteration has a cost of $O(N^2)$ and the number of iterations is at least $O(N)$, where N is the dimension of the map.

We can visualize the iterations of the distance transform by

```
>> dx.plan(place.kitchen, 'animate');
```

which shows the distance values propagating as a wavefront outward from the goal. The wavefront moves outward, spills through doorways into adjacent rooms and outside the house.▸ Although the plan is expensive to create, once it has been created it can be used to plan a path from *any* initial point to that goal.

More efficient algorithms exist such as fast marching methods and Dijkstra's method, but the iterative wavefront method used here is easy to code and to visualize.

We have converted a fairly complex planning problem into one that can now be handled by a Braitenberg-class robot that makes local decisions based on the distance to the goal. Effectively the robot is rolling *downhill* on the distance function which we can plot as a 3D surface

```
>> dx.plot3d(p)
```

shown in Fig. 5.6 with the robot's path and room locations overlaid.

For large occupancy grids this approach to planning will become impractical. The roadmap methods that we discuss later in this chapter provide an effective means to find paths in large maps at greatly reduced computational cost.

The scale associated with this occupancy grid is 4.5 cm per cell and we have assumed the robot occupies a single grid cell – this is a very small robot. The planner could therefore find paths that a larger real robot would be unable to fit through. A common solution to this problem is to *inflate* the occupancy grid – making the obstacles bigger is equivalent to leaving the obstacles unchanged and making the robot bigger. For example, if we inflate the obstacles by 5 cells

```
>> dx = DXform(house, 'inflate', 5);
>> dx.plan(place.kitchen);
>> p = dx.query(place.br3);
>> dx.plot(p)
```

the path shown in Fig. 5.7b now takes the wider corridors to reach its goal. To illustrate how this works we can overlay this new path on the inflated occupancy grid

```
>> dx.plot(p, 'inflated');
```

and this is shown in Fig. 5.7a. The inflation parameter of 5 has grown the obstacles by 5 grid cells in all directions, a bit like applying a very thick layer of paint.◀ This is equivalent to growing the robot by 5 grid cells in all directions – the robot grows from a single grid cell to a disk with a diameter of 11 cells which is equivalent to around 50 cm.

This is morphological dilation which is discussed in Sect. 12.6.

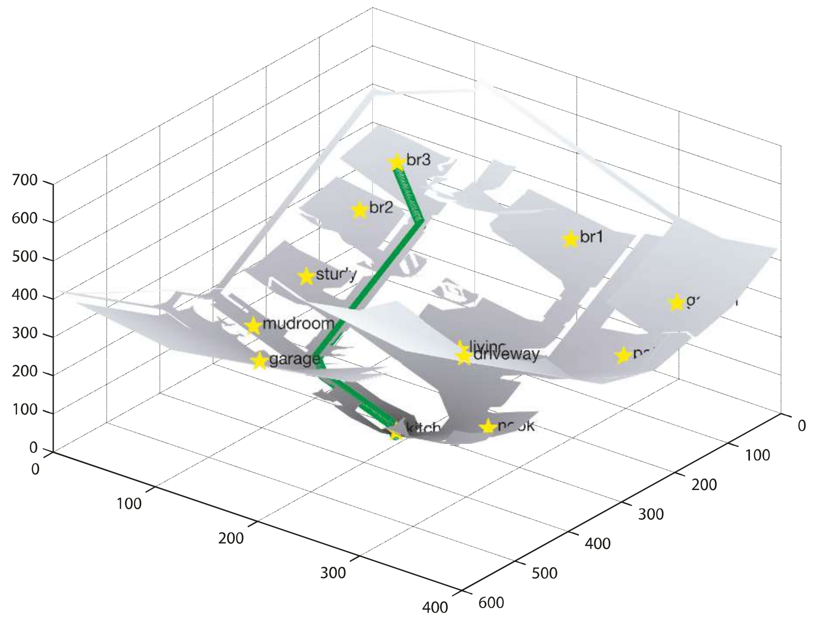
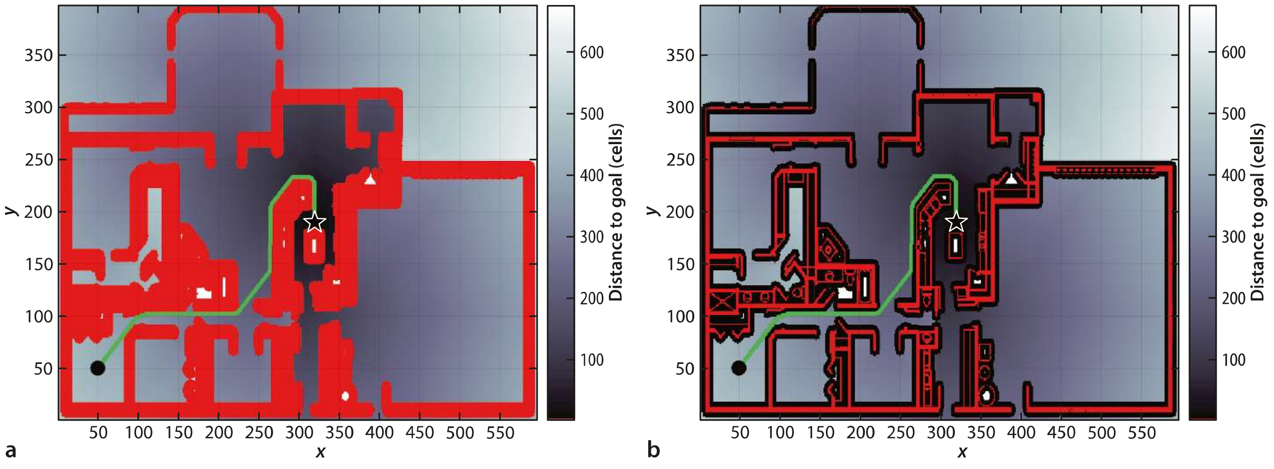


Fig. 5.6. The distance transform as a 3D function, where height is distance from the goal. Navigation is simply a downhill run. Note the discontinuity in the distance transform where the split wavefronts met

Fig. 5.7. Distance transform path with obstacles inflated by 5 cells. **a** Path shown with inflated obstacles; **b** path computed for inflated obstacles overlaid on original obstacle map, black regions are where no distance was computed ▼



Navigation superclass. The examples in this chapter are all based on classes derived from the `Navigation` class which is designed for 2D grid-based navigation. Each example consists of essentially the following pattern. Firstly we create an instance of an object derived from the `Navigation` class by calling the class constructor.

```
nav = MyNavClass(map)
```

which is passed the occupancy grid. Then a plan is computed

```
nav.plan()
nav.plan(goal)
```

and depending on the planner the goal may or may not be required. A path from an initial position to the goal is computed by

```
p = nav.query(start, goal)
p = nav.query(start)
```

again depending on whether or not the planner requires a goal. The optional return value `p` is the path, a sequence of points from `start` to `goal`, one row per point, and each row comprises the *x*- and *y*-coordinate. If `start` or `goal` is given as [] the user is prompted to interactively click the point. The 'animate' option causes an animation of the robot's motion to be displayed.

The map and planning information can be visualized by

```
nav.plot()
```

or have a path overlaid

```
nav.plot(p)
```

5.2.2 D*

A popular algorithm for robot path planning is D* which finds the best path through a graph, which it first computes, that corresponds to the input occupancy grid. D* has a number of features that are useful for real-world applications. Firstly, it generalizes the occupancy grid to a cost map which represents the cost $c \in \mathbb{R}$, $c > 0$ of traversing each cell in the horizontal or vertical direction. The cost of traversing the cell diagonally is $c\sqrt{2}$. For cells corresponding to obstacles $c = \infty$ (`Inf` in MATLAB).

D* is an extension of the A* algorithm for finding minimum cost paths through a graph, see Appendix I.

Secondly, D* supports incremental replanning. This is important if, while we are moving, we discover that the world is different to our map. If we discover that a route has a higher than expected cost or is completely blocked we can incrementally replan to find a better path. The incremental replanning has a lower computational cost than completely replanning as would be required using the distance transform method just discussed.

D* finds the path which minimizes the total cost of travel. If we are interested in the shortest time to reach the goal then cost is the time to drive across the cell and is inversely related to traversability. If we are interested in minimizing damage to the vehicle or maximizing passenger comfort then cost might be related to the roughness of the terrain within the cell. The costs assigned to cells will also depend on the characteristics of the vehicle: a large 4-wheel drive vehicle may have a finite cost to cross a rough area whereas for a small car that cost might be infinite.

To implement the D* planner using the Toolbox we use a similar pattern and first create a D* navigation object

```
>> ds = Dstar(house);
```

The D* planner converts the passed occupancy grid `map` into a cost map which we can retrieve

```
>> c = ds.costmap();
```

where the elements of `c` will be 1 or ∞ representing free and occupied cells respectively.

A plan for moving to the goal is generated by

```
>> ds.plan(place.kitchen);
```

which creates a very dense directed graph (see Appendix I). Every cell is a graph vertex and has a cost, a distance to the goal, and a link to the neighboring cell that is closest to the goal. Each cell also has a state $t \in$ {NEW, OPEN, CLOSED}. Initially every cell is in the NEW state, the cost of the goal cell is zero and its state is OPEN. We can consider the set of all cells in the OPEN state as a wavefront propagating outward from the goal. The cost of

The distance transform also evolves as a wavefront outward from the goal. However D* represents the frontier efficiently as a list of cells whereas the distance transform computes the frontier on a per-cell basis at every iteration – the frontier is implicitly where a cell with infinite cost (the initial value of all cells) is adjacent to a cell with finite cost.

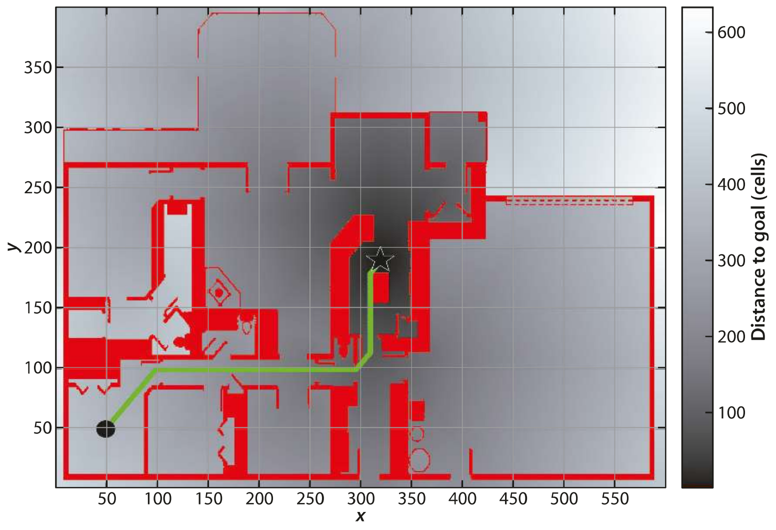


Fig. 5.8.
The D* planner path. Obstacles are indicated by *red cells* and all driveable cells have a cost of 1. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the *scale* on the right-hand side

reaching cells that are neighbors of an OPEN cell is computed and these cells in turn are set to OPEN and the original cell is removed from the open list and becomes CLOSED. In MATLAB this initial planning phase is quite slow◀ and takes over a minute and

D* is more efficient than the distance transform but it executes more slowly because it is implemented entirely in MATLAB code whereas the distance transform is a MEX-file written in C.

```
>> ds.niter
ans =
      245184
```

iterations of the planning loop.

The path from an arbitrary starting point to the goal

```
>> ds.query(place.br3);
```

is shown in Fig. 5.8. The robot has again taken a short and efficient path around the obstacles that is almost identical to that generated by the distance transform.

The real power of D* comes from being able to efficiently change the cost map during the mission. This is actually quite a common requirement in robotics since real sensors have a finite range and a robot discovers more of world as it proceeds. We inform D* about changes using the `modify_cost` method, for example to raise the cost of entering the kitchen via the bottom doorway

```
>> ds.modify_cost( [300,325; 115,125], 5 );
```

we have raised the cost to 5 for a small rectangular region across the doorway. This region is indicated by the yellow dashed rectangle in Fig. 5.9. The other driveable cells have a default cost of 1. The plan is updated by invoking the planning algorithm again

```
>> ds.plan();
```

and this time the number of iterations is only

```
>> ds.niter
ans =
      169580
```

which is 70% of that required to create the original plan.◀ The new path for the robot

The cost increases with the number of cells modified and the effect those changes have on the distance map. It is possible that incremental replanning takes more time than planning from scratch.

```
>> ds.query(place.br3);
```

is shown in Fig. 5.9. The cost change is relatively small but we notice that the increased cost of driving within this region is indicated by a subtle brightening of those cells – in a cost sense these cells are now further from the goal. Compared to Fig. 5.8 the robot has taken a different route to the kitchen and avoided the bottom door. D* allows updates to the map to be made at any time while the robot is moving. After replanning the robot simply moves to the adjacent cell with the lowest cost which ensures continuity of motion even if the plan has changed.

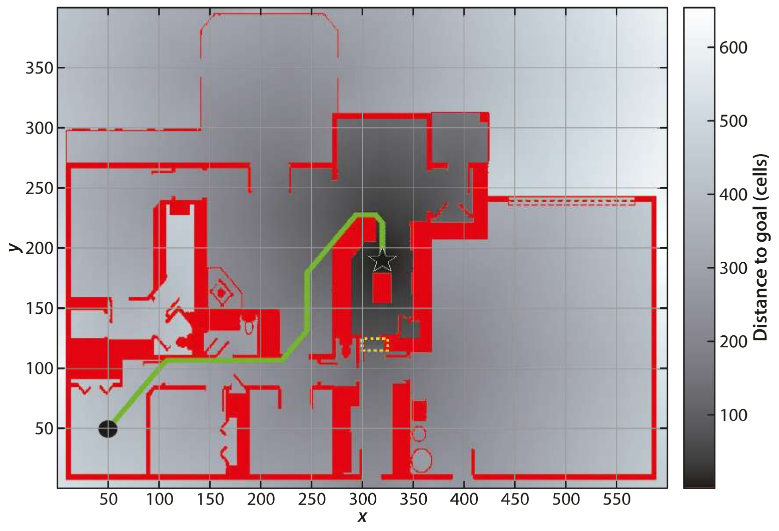
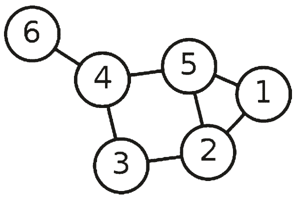


Fig. 5.9. Path from D* planner with modified map. The higher-cost region is indicated by the *yellow dashed rectangle* and has changed the path compared to Fig. 5.7

A graph is an abstract representation of a set of objects connected by links typically denoted $G(V, E)$ and depicted diagrammatically as shown to the right. The objects, V, are called vertices or nodes, and the links, E, that connect some pairs of vertices are called edges or arcs. Edges can be directed (arrows) or undirected as in this case. Edges can have an associated weight or cost associated with moving from one of its vertices to the other. A sequence of edges from one vertex to another is a path. Graphs can be used to represent transport or communications networks and even social relationships, and the branch of mathematics is graph theory. Minimum cost path between two nodes in the graph can be computed using well known algorithms such as Dijstrka's method or A*.

The navigation classes use a simple MATLAB graph class called `PGraph`, see Appendix I.



5.2.3 Introduction to Roadmap Methods

In robotic path planning the analysis of the map is referred to as the *planning phase*. The *query phase* uses the result of the planning phase to find a path from A to B. The two previous planning algorithms, distance transform and D*, require a significant amount of computation for the planning phase, but the query phase is very cheap. However the plan depends on the goal. If the goal changes the expensive planning phase must be re-executed. Even though D* allows the path to be recomputed as the costmap changes it does not support a changing goal.

The disparity in planning and query costs has led to the development of roadmap methods where the query can include both the start and goal positions. The planning phase provides analysis that supports changing starting points and changing goals. A good analogy is making a journey by train. We first find a local path to the nearest train station, travel through the train network, get off at the station closest to our goal, and then take a local path to the goal. The train network is invariant and planning a path through the train network is straightforward. Planning paths to and from the entry and exit stations respectively is also straightforward since they are, ideally, short paths. The robot navigation problem then becomes one of building a network of obstacle free paths through the environment which serve the function of the train network. In the literature such a network is referred to as a roadmap. The roadmap need only be computed once and can then be used like the train network to get us from any start location to any goal location.

We will illustrate the principles by creating a roadmap from the occupancy grid's free space using some image processing techniques. The essential steps in creating the roadmap are shown in Fig. 5.10. The first step is to find the free space in the map which is simply the complement of the occupied space

```
>> free = 1 - house
```

and is a matrix with nonzero elements where the robot is free to move. The boundary is also an obstacle so we mark the outermost cells as being not free

```
>> free(1,:) = 0; free(end,:) = 0;
>> free(:,1) = 0; free(:,end) = 0;
```

and this map is shown in Fig. 5.10a where free space is depicted as white.

The topological skeleton of the free space is computed by a morphological image processing algorithm known as thinning▸ applied to the free space of Fig. 5.10a

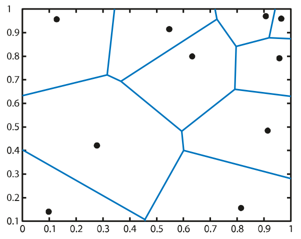
Also known as skeletonization. We will cover this topic in Sect. 12.6.3.

```
>> skeleton = ithin(free);
```

and the result is shown in Fig. 5.10b. We see that the obstacles have grown and the free space, the white cells, have become a thin network of connected white cells which are equidistant from the boundaries of the original obstacles.

Figure 5.10c shows the free space network overlaid on the original map. We have created a network of paths that span the space and which can be used for obstacle-free travel around the map.▸ These paths are the edges of a generalized Voronoi

The junctions in the roadmap are indicated by black dots. The junctions, or triple points, are identified using the morphological image processing function `triplepoint`.



The Voronoi tessellation of a set of planar points, known as sites, is a set of Voronoi cells as shown to the left. Each cell corresponds to a site and consists of all points that are closer to its site than to any other site. The edges of the cells are the points that are equidistant to the two nearest sites. A generalized Voronoi diagram comprises cells defined by measuring distances to objects rather than points. In MATLAB we can generate a Voronoi diagram by

```
>> sites = rand(10,2)
>> voronoi(sites(:,1), sites(:,2))
```

Georgy Voronoi (1868–1908) was a Russian mathematician, born in what is now Ukraine. He studied at Saint Petersburg University and was a student of Andrey Markov. One of his students Boris Delaunay defined the eponymous triangulation which has dual properties with the Voronoi diagram.

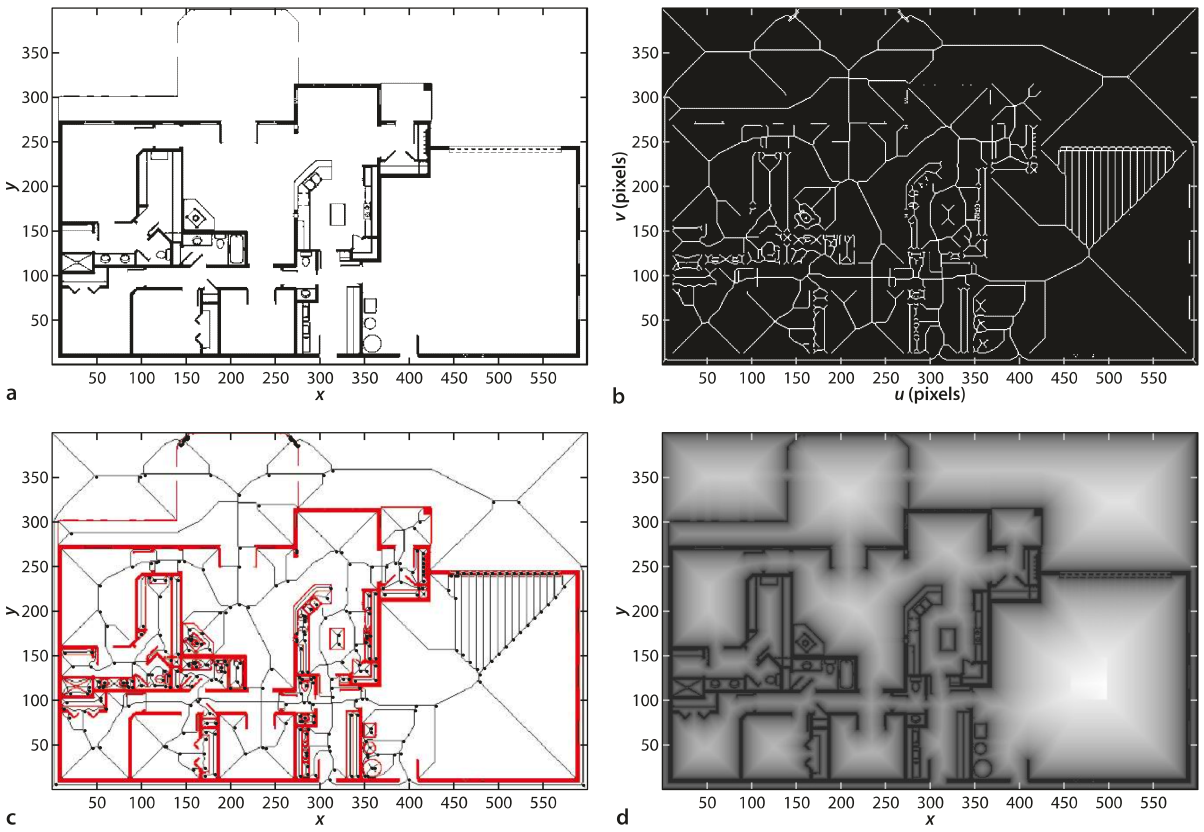
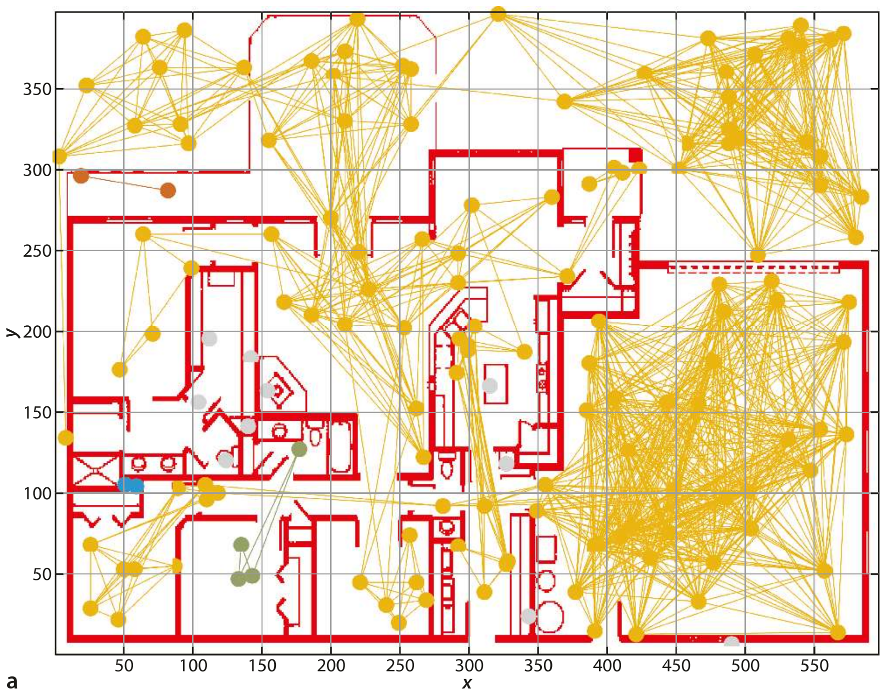


Fig. 5.10. Steps in the creation of a Voronoi roadmap. **a** Free space is indicated by *white cells*; **b** the skeleton of the free space is a network of adjacent cells no more than one cell thick; **c** the skeleton with the obstacles overlaid in red and road-map junction points indicated by *black dots*; **d** the distance transform of the obstacles, pixel values correspond to distance to the nearest obstacle

diagram. We could obtain a similar result by computing the distance transform of the obstacles, Fig. 5.10a, and this is shown in Fig. 5.10d. The value of each pixel is the distance to the nearest obstacle and the ridge lines correspond to the skeleton of Fig. 5.10b. Thinning or skeletonization, like the distance transform, is a computationally expensive iterative algorithm but it illustrates well the principles of finding paths through free space. In the next section we will examine a cheaper alternative.

5.2.4 Probabilistic Roadmap Method (PRM)

The high computational cost of the distance transform and skeletonization methods makes them infeasible for large maps and has led to the development of probabilistic methods. These methods sparsely sample the world map and the most well known of these methods is the probabilistic roadmap or PRM method.



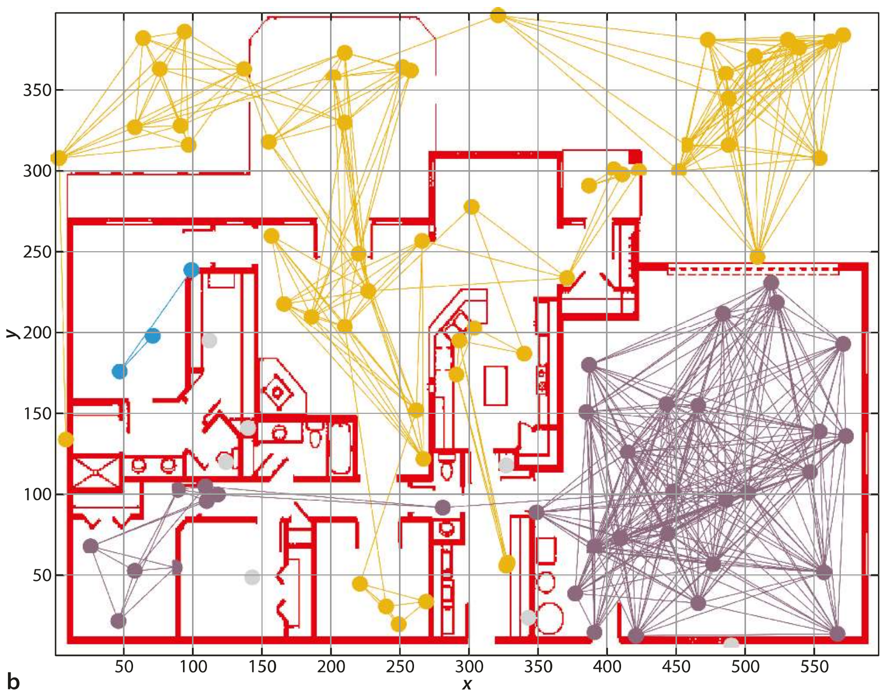


Fig. 5.11. Probablistic roadmap (PRM) planner and the random graphs produced in the planning phase. **a** Well connected network with 150 nodes; **b** poorly connected network with 100 nodes

To use the Toolbox PRM planner for our problem we first create a `PRM` object

```
>> prm = PRM(house)
```

and then create the plan

```
>> prm.plan('npoints', 150)
```

To replicate the following result be sure to initialize the random number generator first using `randinit`. See page 139.

with 150 roadmap nodes. Note that we do not pass the goal as an argument since the plan is independent of the goal. Creating the path is a two phase process: planning, and query. The planning phase finds *N* random points, 150 in this case, that lie in free space. Each point is connected to its nearest neighbors by a straight line path that does not cross any obstacles, so as to create a network, or graph, with a minimal number of disjoint components and no cycles. The advantage of PRM is that relatively few points need to be tested to ascertain that the points and the paths between them are obstacle free. The resulting network is stored within the PRM object and a summary can be displayed

```
>> prm
prm =
PRM navigation class:
  occupancy grid: 397x596
  graph size: 150
  dist thresh: 178.8
  2 dimensions
  150 vertices
  1223 edges
  14 components
```

which indicates the number of edges and connected components in the graph. The graph can be visualized

```
>> prm.plot()
```

as shown in Fig. 5.11a. The dots represent the randomly selected points and the lines are obstacle-free paths between the points. Only paths less than 178.8 cells long are selected which is the distance threshold parameter of the PRM class. Each edge of the graph has an associated cost which is the distance between its two nodes. The color of the node indicates which component it belongs to and each component is assigned a unique color. In this case there are 14 components but the bulk of nodes belong to a single large component.

This is derived automatically from the size of the occupancy grid.

The query phase finds a path from the start point to the goal. This is simply a matter of moving to the closest node in the roadmap (the start node), following a minimum cost A* route through the roadmap, getting off at the node closest to the goal and then traveling to the goal. For our standard problem this is

```
>> prm.query(place.br3, place.kitchen)
>> prm.plot()
```

and the path followed is shown in Fig. 5.12. The path that has been found is quite efficient although there are two areas where the path doubles back on itself. Note that we provide the start and the goal position to the query phase. An advantage of this planner is that once the roadmap is created by the planning phase we can change the goal and starting points very cheaply, only the query phase needs to be repeated. The path taken is

```
>> p = prm.query(place.br3, place.kitchen);
>> about p
p [double] : 9x2 (144 bytes)
```

which is a list of the node coordinates that the robot passes through – via points. These could be passed to a trajectory following controller as discussed in Sect. 4.1.1.3.

There are some important tradeoffs in achieving this computational efficiency. Firstly, the underlying random sampling of the free space means that a different roadmap is created every time the planner is run, resulting in different paths and path lengths. Secondly, the planner can fail by creating a network consisting of disjoint components. The roadmap in Fig. 5.11b, with only 100 nodes has several large disconnected components and the nodes in the kitchen and bedrooms belong to different components. If the start and goal nodes are not connected by the roadmap, that is, they are close to different components the query method will report an error. The only solution is to rerun the planner and/or increase the number of nodes. Thirdly, long narrow gaps between obstacles such as corridors are unlikely to be exploited since the probability of randomly choosing points that lie along such spaces is very low.

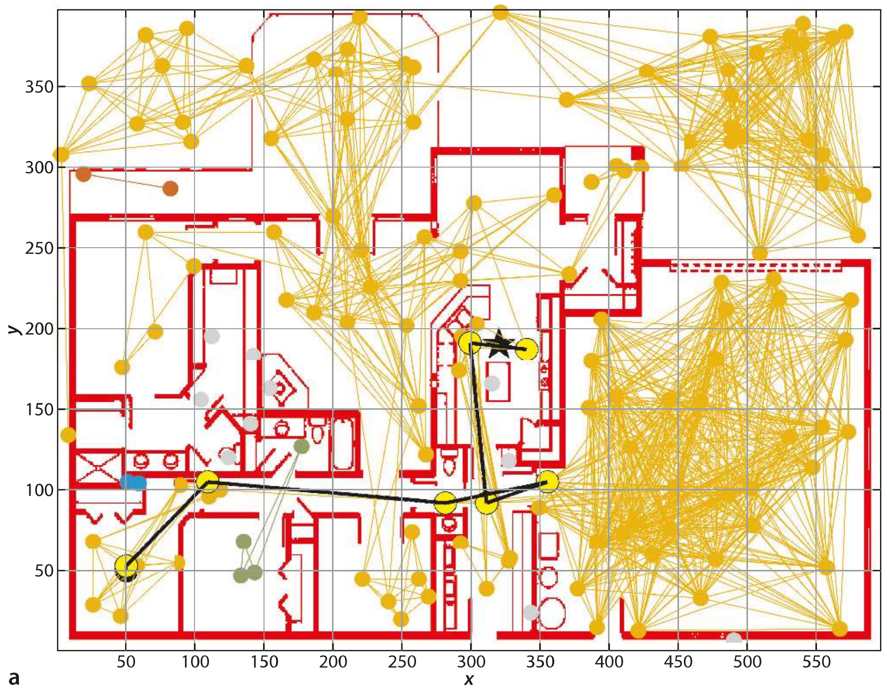
Random numbers. The MATLAB random number generator (used for rand and randn) generates a very long sequence of numbers that are an excellent approximation to a random sequence. The generator maintains an internal state which is effectively the position within the sequence. After startup MATLAB always generates the following random number sequence

```
>> rand
ans =
    0.8147
>> rand
ans =
    0.9058
>> rand
ans =
    0.1270
```

Many algorithms discussed in this book make use of random numbers and this means that the results can never be repeated. Before all such examples in this book is an invisible call to randinit which resets the random number generator to a known state

```
>> randinit
>> rand
ans =
    0.8147
>> rand
ans =
    0.9058
```

and we see that the random sequence has been restarted.



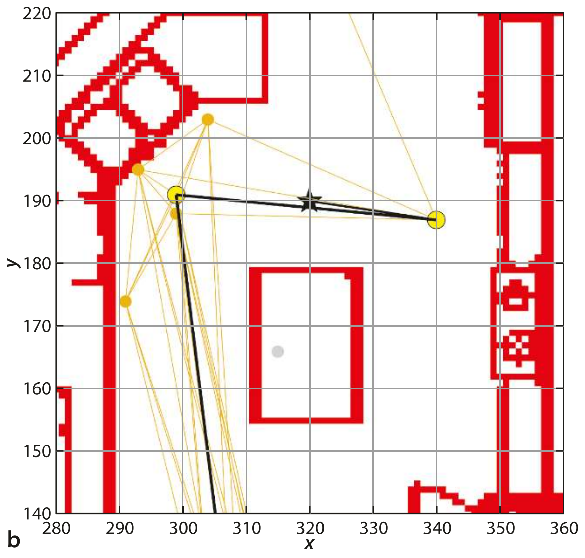


Fig. 5.12.
Probablistic roadmap (PRM) planner **a** showing the path taken by the robot via nodes of the roadmap which are highlighted in yellow; **b** closeup view of goal region where the short path from the final roadmap node to the goal can be seen

5.2.5 Lattice Planner

The planners discussed so far have generated paths independent of the motion that the vehicle can actually achieve, and we learned in Chap. 4 that wheeled vehicles have significant motion constraints. One common approach is to use the output of the planners we have discussed and move a point along the paths at constant velocity and then follow that point, using techniques such as the trajectory following controller described in Sect. 4.1.1.3.

An alternative is to *design* a path from the outset that we know the vehicle can follow. The next two planners that we introduce take into account the motion model of the vehicle, and relax the assumption we have so far made that the robot is capable of omnidirectional motion.