Section 9: Path Planning

Path planning is a fundamental problem in robotics. Generally speaking, the objective of path planning is to determine a path $\boldsymbol{\gamma}$ for the robot to follow that brings it from an initial state $\mathbf{x}_0$ to a final state $\mathbf{x}_1$ while avoiding obstacles. The path planning problem is related, but distinct, to the control problems we studied in the previous section. Path planning often neglects or simplifies the complex vehicle dynamics and instead considers larger scale motions in environments with obstacles.

Consider the scenarios in Fig. 1a and 1b. Both paths begin and end at the same locations. The first path considers the vehicle's heading angle while the second path treats the vehicle as a particle and neglects its heading. Both of these approaches are valid ways to formulate the path planning problem and the correct choice depends on the task at hand (i.e., whether the initial and final heading is important). If we consider the particle model of the robot (Fig. 1b) then the robot's configuration space is

$$\mathcal{C} = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$$

where the minimum and maximum values indicate the dimensions of the bounding rectangle in which the robot is constrained to operate. In Fig. 1 obstacles are indicated by the $\mathcal{C}_{\text{obs}}$ region in grey which is a subset of the robot's configuration space

$$\mathcal{C}_{\text{obs}} \subseteq \mathcal{C} \ .$$

Similarly, the free space in which the robot can move is indicated by the $\mathcal{C}_{\text{free}}$ region in white which is also a subset of the robot's configuration space

$$\mathcal{C}_{\text{free}} \subseteq \mathcal{C} \ .$$

Together, the union of the free space with the obstacle space forms the entire configuration space:

$$\mathcal{C}_{\text{obs}} \cup \mathcal{C}_{\text{free}} = \mathcal{C} \ .$$

Every point in the configuration space is either in the free space or in the obstacle space, but not both. Thus the intersection of $\mathcal{C}_{\text{obs}}$ and $\mathcal{C}_{\text{free}}$ is an empty set

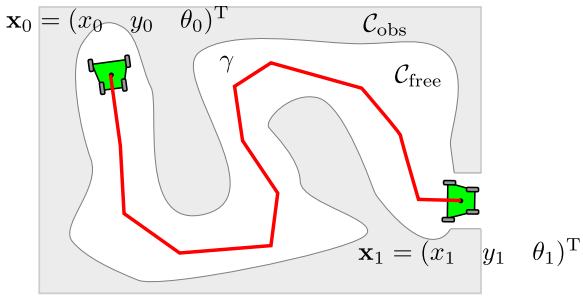
$$\mathcal{C}_{\text{obs}} \cap \mathcal{C}_{\text{free}} = \emptyset \ .$$

With these definitions we can formally define a path $\boldsymbol{\gamma}$ as a function that maps a time-like parameter τ (that ranges from 0 at the beginning of the path to 1 at the end of the path) to points in the configuration space

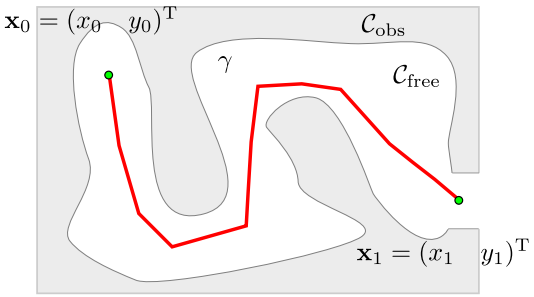
$$\boldsymbol{\gamma}(\tau) : [0, 1] \to \mathcal{C}_{\text{free}} \ .$$

The path begins at $\mathbf{x}_0$ and ends at $\mathbf{x}_1$

$$\boldsymbol{\gamma}(0) = \mathbf{x}_0 \in \mathcal{C}_{\text{free}}$$
$$\boldsymbol{\gamma}(1) = \mathbf{x}_1 \in \mathcal{C}_{\text{free}}$$



(a) The initial and goal states include the car's heading.



(b) The initial and goal states do not include the car's heading.

Figure 1: A sketch of the motion planning problem.

Goals of Path Planning

The problem of finding a path that transfers the robot from state $\mathbf{x}_0$ to state $\mathbf{x}_1$ is one formulation of the path planning problem. However, many other formulations exist that aim to achieve different objectives:

- **Determine if a feasible path exists.** In some cases it is not clear if a path from $\mathbf{x}_0$ to $\mathbf{x}_1$ even exists or is feasible for the vehicle to execute. This situation may arise, for example, in the case of a ground vehicle trying to escape a maze, or a underwater vehicle fighting a strong currents in a river.
- **Determine which path is optimal.** In situations when there are many possible paths we typically want to identify one that is the "best" in some sense. This is achieved by defining a cost function $J(\boldsymbol{\gamma})$ that returns a scalar cost (e.g., the length of the path, or the energy expended along the path) and we seek to find the optimal path $\boldsymbol{\gamma}^*$ that minimizes the cost J.
- **Determine a path that satisfies complex goals.** Robots may also encounter problems that require visiting multiple locations and performing several tasks, as well as avoiding both static and moving obstacles. In this situation we may be primarily concerned with satisfying the constraints of the problem, and may or may not also consider path optimization.

Planning Algorithm Considerations

Path planning has been a widely studied topic in the robotics literature. There are many algorithms that exist to plan paths for mobile robots. The following is a list of characteristics that may differentiate path planning algorithms and help in selecting one for a given task:

- **Motion Model.** Path planning algorithms make assumptions on how the robot moves (e.g., the dimension of the configuration space, equations of motion). This can range from a simple models (e.g., the robot can move in any direction at will) to more complex models that account for the robot's dynamics (e.g., the robot's motion must satisfy a system of differential equations). The degree to which the robot's motion must be accurately modelled is highly dependent on the application.
- **Environment.** Path planning algorithms make assumptions regarding the structure of the environment. Some algorithms assume the environment is perfectly known, while others assume it is only partially known and discovered by the robot as it moves around. While some planning algorithms incorporate static and moving obstacles, other algorithms neglect obstacles altogether.
- **Completeness.** When a planning algorithm is said to be *complete* it means that it will always find a solution if one exists or it will determine if one does not exist. This is an important property,

but even planning algorithms that may be incomplete can also be useful (e.g., a certain application may only require an algorithm to solve a path planning problem quickly and if the application is not critical, then occasional failures may be acceptable).

- **Generality.** Some path planning algorithm can accommodate many different vehicle motion models and environments while others can only solves a very specific problem.
- **Optimality.** Different planning algorithms may be used to optimize different criteria (e.g., time vs. energy) and further they may differ with respect to how well they perform this optimization.
- **Computational Complexity.** Ultimately, path planning algorithms are designed to be implemented on a mobile robot with limited computational resources. Not only must the robot's computer be capable of finding a solution, it must do so fast enough so that the solution is useful.

Bug Algorithm Path Planning

Bug algorithms are inspired by the way in which insects navigate around obstacles. They make the following assumptions:

- the robot operates in a 2D world and has a sensor that informs it of its own (x, y) position and that of the goal (x_g, y_g)
- the robot does not know the shape and location of the obstacles
- the robot has a sensor that enables it to detect obstacles and follow their boundaries
- the robot always moves in the same direction (e.g., clockwise) when it initially encounters an obstacle

We will discuss several variations of the Bug algorithms but the main idea behind each one is as follows:

1. Move in a straight line towards the goal (x_g, y_g)
2. If the i-th obstacle is encountered (label this point the hit point H_i)
3. Follow the boundary of the obstacle until the leave point L_i is reached
4. Return to Step 1 until the goal is reached

The algorithms differ mainly in in the way they determine the leave point after encountering each obstacle.

Bug 0. After reaching a hit point, the robot follows the boundary of the obstacle until it is able to make further progress towards the goal (this becomes the leave point). This procedure is illustrated in Fig 2. However, Bug 0 is a *incomplete* algorithm because it can fail to return a solution even when one exists. Consider the example scenario in Fig. 3. A left-turning Bug 0 algorithm will fail to find the path to the goal since the leave point L_2 coincides with the leave point L_1 it will remain trapped in a infinite loop.

Bug 1. An improvement over Bug 0 requires the robot to have some memory of where it has already been. In the Bug 1 algorithm, the robot circumnavigates each obstacle it encounters. By remembering the boundary of the obstacle it can determine the point along the obstacle boundary at which it was closest to the goal and label that as a leave point (Fig 4). This approach overcomes the problems faced by Bug 0 and Bug 1 is a complete algorithm.

Bug 2. However, an obvious shortcoming of Bug 1 is that it is time-consuming to circumnavigate each obstacle. The Bug 2 algorithm uses a reference line drawn from the initial state to the goal as a

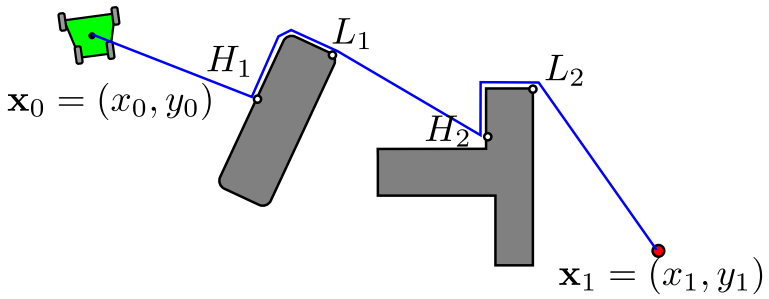


Figure 2: Sketch of the Bug 0 algorithm

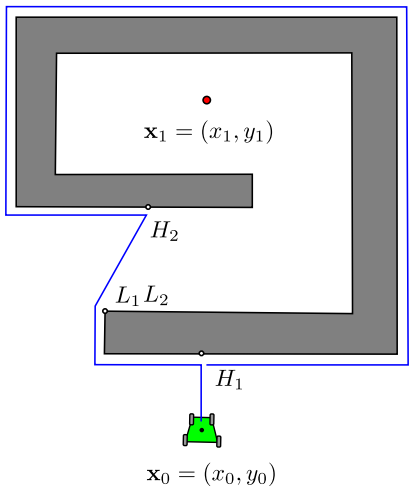


Figure 3: A scenario in which a left-turning Bug 0 algorithm fails

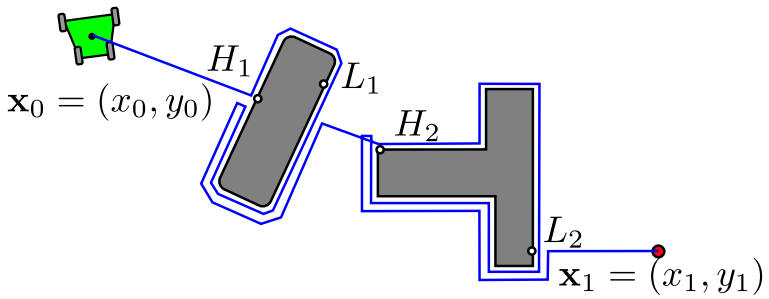


Figure 4: Sketch of the Bug 1 algorithm

guide. By always turning left and forcing all hit points and leave points to lie on the reference line (Fig 5), the robot is able to more efficiently to reach the goal. Bug 2 is also a complete planning algorithm (it will always return a solution if one exists).

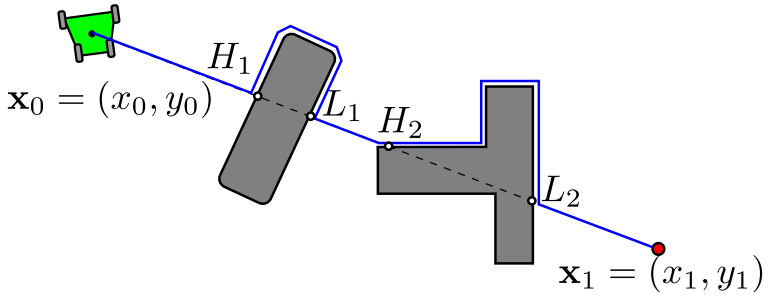
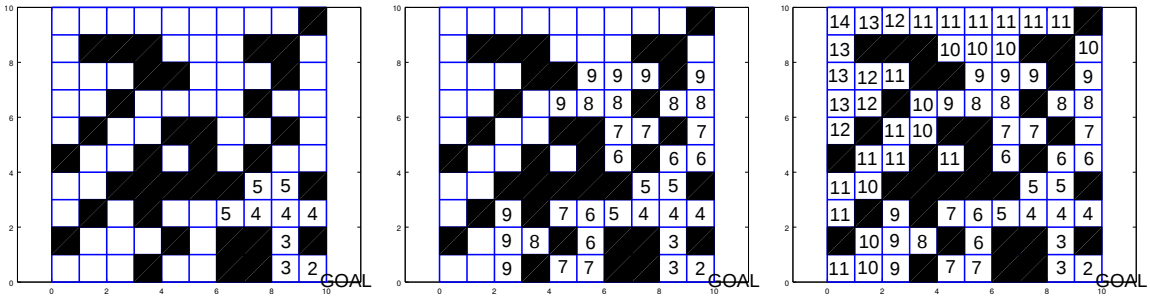


Figure 5: Sketch of the Bug 2 algorithm

Wavefront Planning

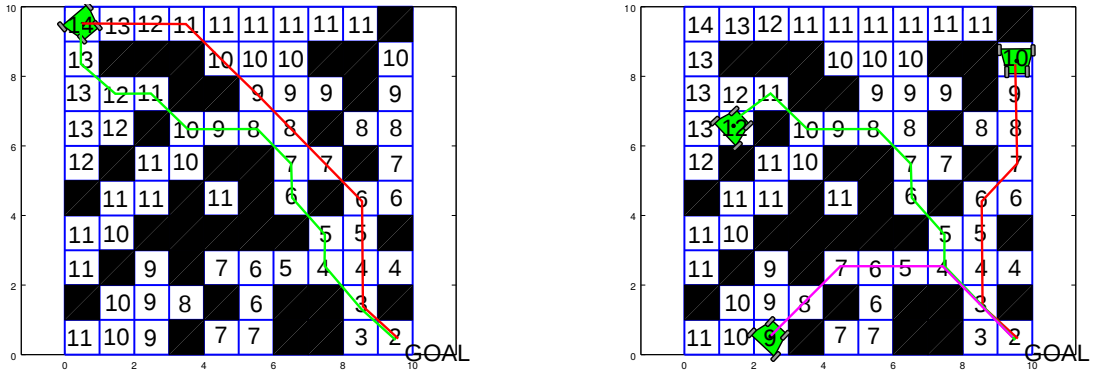
The wavefront planning approach imposes a grid over the configuration space and assumes that the robot can move in any direction (up, down, left, right, and also in the diagonal directions – up-right, up-left, down-right, down-left). The robot incurrs one unit of cost with each "move". This approach aims to assign a cost to each cell that represents the number of moves that must be made from that cell to reach the goal. The procedure works as follows: Initially, a one is assigned to each cell occupied by an obstacle and a zero to each free cell. The cell corresponding to the goal is then re-assigned a value of two. Next, all of the zero-valued cells next to the goal cell are assigned a value of three. Then, all of the zero-valued cells next to the three-valued cells are assigned a value of four. The next set of adjacent cells are labelled 5 (Fig. 6a). At every iteration the adject cells are incremented by one. At any given iteration, the highest numbered cells represent a wavefront of a fixed distance from the goal (e.g., all of the cells labelled 9 in Fig. 6b are 9 moves away from the goal). This process repeats until all the zero-valued cells in the grid are updated (Fig. 6c).



(a) Wavefront expanded to 5 (b) Wavefront expanded to 9 (c) Wavefront expanded to entire grid

Figure 6: The process of wavefront expansion for a 10 x 10 grid. The goal cell is in the lower right corner. Black cells indicate obstacles (with cell value 1).

Then by selecting a start point in the grid, the robot can arrive at the shortest path to the goal by iteratively moving to a cell with lower cost. As shown Fig. 7a, there can be more than one way to reach the goal from a given start point. Further, once the wavefront has been expanded to include the entire map, the path from any start point can be easily computed (Fig. 7b). (In fact, if one is seeking only the path from a specific start point, the wave front expansion can be terminated once this start point is reached.) This approach is straightforward to program and can be used to solve larger problems (Fig. 8).



(a) Sketch of the Bug 2 algorithm (b) Sketch of the Bug 2 algorithm

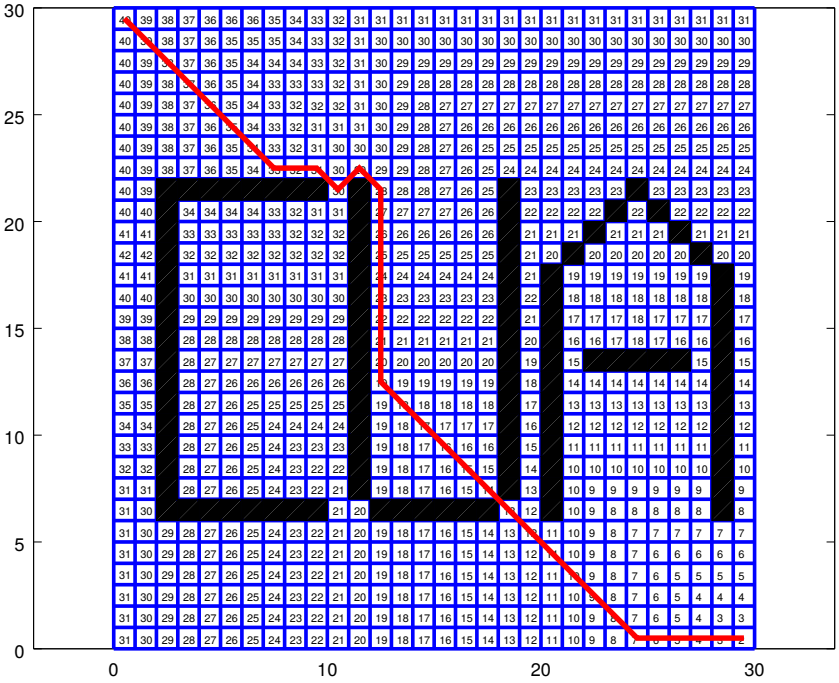


Figure 8: Sketch of the wavefront algorithm for a large environment

Nearest-neighbor Search: Travelling Salesman Problem

The *travelling salesman problem* is often framed as follows: determine the shortest path that visits a list of cities (given by (x, y) points) while only visiting each city once. (Usually an extra condition is included that requires the salesman to return to the city he began in.) The path the salesman takes consists of straight line segments joining each city. This problem amounts to one of combinatorial optimization: what is the sequence of cities to visit? The travelling salesman problem has a broad range of applications (e.g., connecting a series of circuit components with the least amount of wire, finding the fastest route to pick up a carpool of passengers). Here we will study the *nearest neighbor* algorithm that generates a (suboptimal) solution to this problem.

The nearest-neighbor algorithm begins by choosing a point at random from the list of points being considered. The chosen point is added to a *sequence list* (i.e., a vector that contains the sequence of points that will define the path). The chosen point also becomes the *current point*. A *available points list* is maintained that gives all of the points that are not yet part of the path. The point in the available list that is closest to the current point is: added to the sequence list, removed from the available list, and then becomes the new current point. This process repeats until all points are added to the sequence list.

Many path planning algorithms (and algorithms in general) are often described using *pseudocode*. Pseudocode is a general way to communicate the procedure of an algorithm without using syntax that is tied to a specific language. Consider the pseudocode of the nearest-neighbor algorithm shown in Algorithm 1.

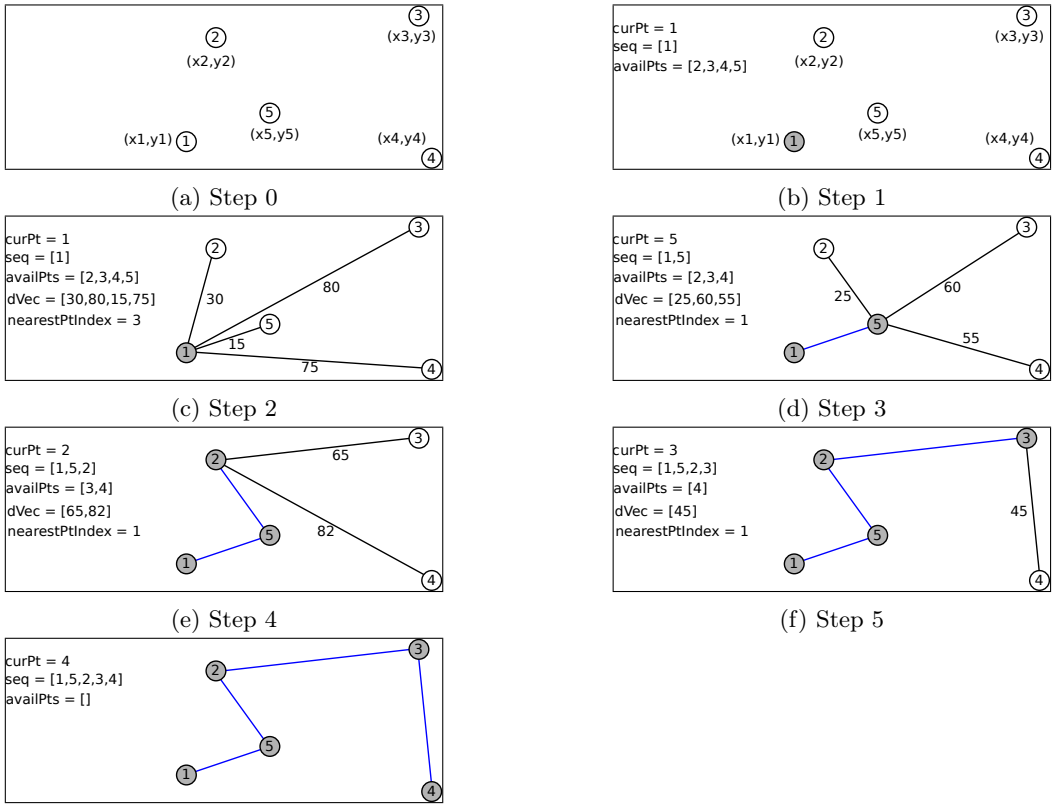
We will explain this pseudocode with an example. Consider the set of points shown in Fig. 9a. Each point is labelled by a number inscribed in a circular marker. We assume that the vectors $x = [x1, x2, x3, x4, x5]$ and $y = [y1, y2, y3, y4, y5]$ describing the positions of the points are provided as inputs to the algorithm. The output of the algorithm is the *seq* vector. This input-output relationship is described on Line 1. Given the vectors x and y we can determine that there $N = 5$ elements (Line 2). (The $\leftarrow$ symbol is an assignment operator that indicates the variable on the left is assigned the expression or quantity on the right.)

Algorithm 1 Nearest Neighbor Algorithm for a Set of N Planar Points

```
1: procedure [SEQ] = NearestNeighborPath(x,y)
2:     N ← total number of points
3:     curPt ← 1                          ▷ most recently visited point (begin at first point)
4:     availPts ← [2 : 1 : N]             ▷ vector of points not yet visited
5:     seq(1) ← curPt                     ▷ vector storing the nearest-neighbor sequence
6:     i ← 2                              ▷ store the number of points in seq plus one
7:     while (length(availPts) > 0) do    ▷ while there are points left to visit
8:         dVec ← empty                   ▷ changes size each iter. so initialize empty
9:         dVec ← computeDistance(curPt, availPts, x, y)   ▷ distances from curPt to the availPts
10:        nearestPtIndex ← minIndex(dVec)   ▷ index of the availPt that is nearest the curPt
11:        seq(i) ← availPts(nearestPtIndex)   ▷ add the nearest neighbor to the sequence
12:        curPt ← seq(i)                 ▷ update the curPt
13:        availPts(i) ← empty            ▷ remove the nearestPt entry from the availPts list
14:        i ← i + 1                      ▷ update
15:    end while
16: end procedure
```

The first step of the algorithm is to determine the current point (which we call $curPt$). If the list of points is provided randomly we can simply choose the first point in the list so that $curPt = 1$ (Line 3). The available points list is a vector containing the remaining points $availPts = [2, 3, 4, 5]$, see Fig. 9b (Line 4). The $curPt$ is the first element in the sequence vector seq (Line 5). Line 6 initializes a counter i that stores the number of points is seq plus one. The while loop (Line 7) executes the commands on Lines 8-14 in a loop until the terminating condition $length(availPts) > 0$ is met (i.e., the iterations continue until the $availPts$ vector is empty).

The while loop begins by initializing $dVec$ to an empty vector (Line 8). $dVec$ will hold the distances to the $availPts$ but since the $availPts$ list is changing size with each iteration we want to make sure $dVec$ is appropriately initialized. (Note, in MATLAB an empty vector can be initialized as `dVec = []`.) Line 9 denotes an abstract operation of assigning to $dVec$ the distance from the $curPt$ to each of the $availPts$. (Although this is represented as one line in the pseudocode it may take multiple lines to implement this using the syntax of a particular language.) Following our example, $dVec = [30, 80, 15, 75]$ as shown in Fig. 9c. The next step (Line 10) is to assign to the variable $nearestPtIndex$ the integer that corresponds to the smallest element of $dVec$ (in MATLAB, this index can be obtained as the second output of the `min` command). The $nearestPtIndex$ indicates which element of the $availPts$ vector will be added to the seq (Line 11). This point also becomes the new $curPt$ (Line 12). Finally, the point is removed from the $availPts$ list (Line 13). (In MATLAB, an element at position $index$ of a vector a can be removed from the vector as follows: `a(index) = []`. The end of the first iteration is shown in Fig. 9d. On Line 14 the counter is incremented and the procedure returns to the while condition. The loop runs several more times as shown in Fig. 9e-Fig. 9g. An example result for a larger set of x and y points is shown in Fig. 10.



(a) Step 0 (b) Step 1

(c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5

(g) Step 6

Figure 9: Example steps of a nearest-neighbor search algorithm to visit a series of $N = 5$ points

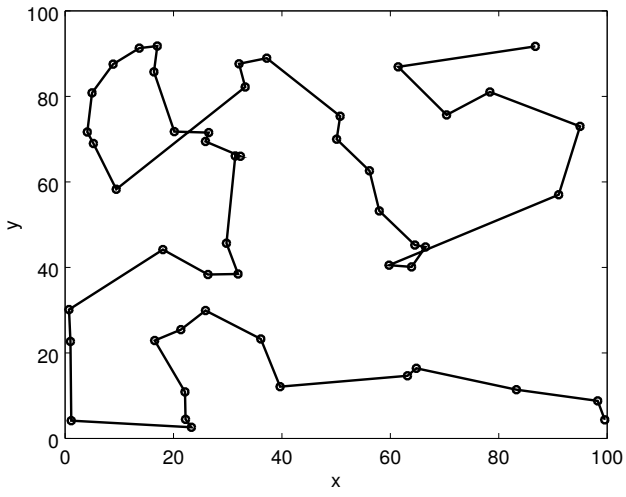


Figure 10: Nearest-neighbor solution for $N = 50$ points.