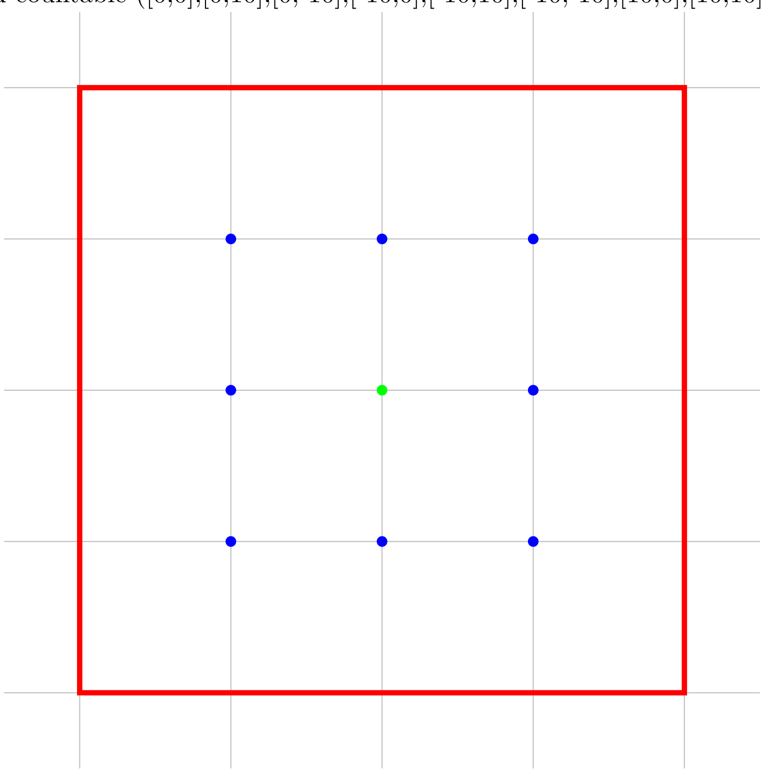
Anthill Puzzle - Solution

May 29, 2023

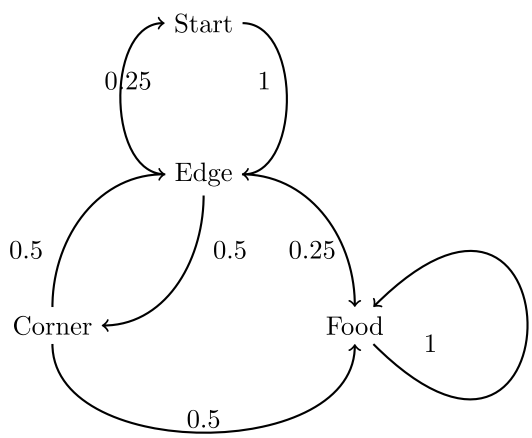
1 Question 1

The following figure provides a visual representation of the situation. The green point represents the origin (anthill), whereas the red square indicates the location of the food all around the anthill. The blue dots represent all the possible locations of the ant while searching for food. Since the text specifies that the ant moves in a random direction for 10 cm each time, those locations are discrete and countable ([0,0],[0,10],[0,-10],[-10,0],[-10,10],[-10,-10],[10,0],[10,10],[10,-10]).



We can leverage the discrete nature of the problem to adopt a Markov Chain approach to obtain the solution. The states of our Markov Chain are the possible positions of the ant along with the *Food state* which is an absorbing state. However, it is convenient to notice that the symmetry of the problem allow us to make some simplifications. For example, we notice that the states [10,10] and ([10,-10]) are identical for our problem. In both of these states, which we can think of as *Corner states* we have a $\frac{1}{2}$ probability of reaching the goal and a $\frac{1}{2}$ probability of reaching an *Edge state*.

This reasoning suggests that we can model the problem as a Markov Chain with 4 states: (*Start, Edge, Corner, Food*). As outlined before, *Food* is the absorbing final state. The state *Corner* represents the positions [(-10,-10),(-10,10),(10,10),(10,-10)], whereas the state *Edge* represents the positions [(0,-10),(0,10),(-10,0),(10,0)]. In the following figure, the Markov Chain representation with the transition probabilities.



Let i be a state of the Markov Chain, thus $i \in$ [*Start, Edge, Corner, Food*]. Let's define X_i as the expected number of turns to reach the state *Food* from state i. We have the following system of equations:

$X_{\text{Food}} = 0$

$X_{\text{Corner}} = \frac{1}{2}(1 + X_{\text{Edge}}) + \frac{1}{2}(1 + X_{\text{Food}})$

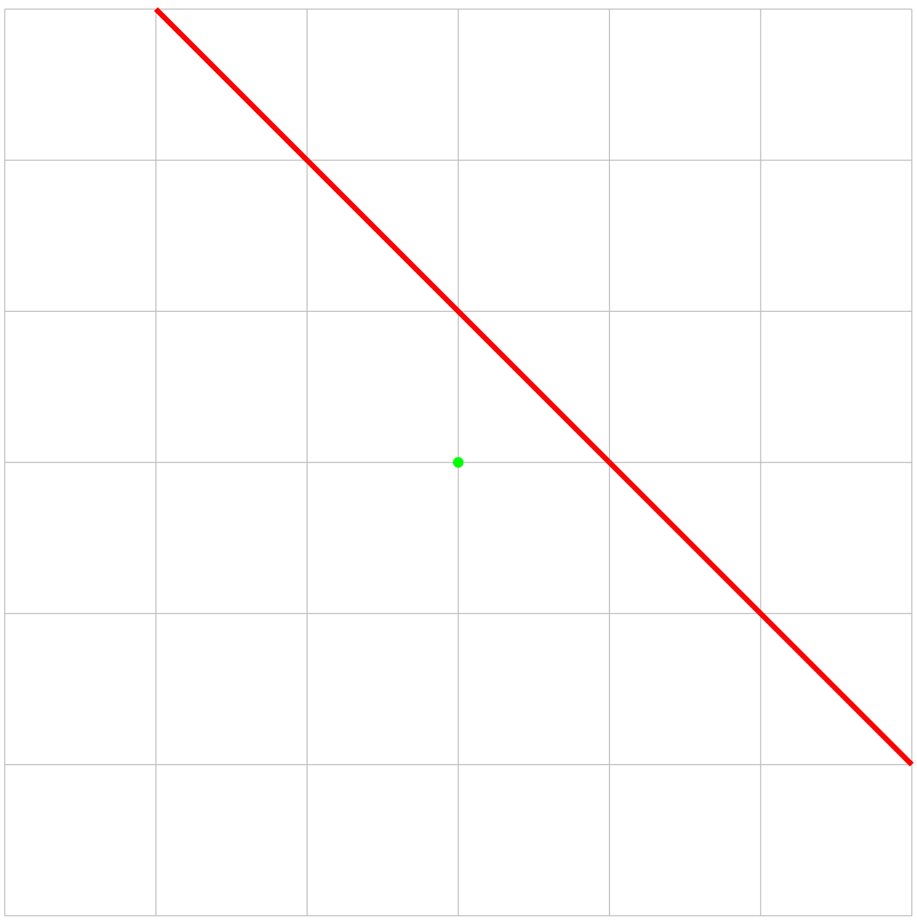
$X_{\text{Edge}} = \frac{1}{4}(1 + X_{\text{Start}}) + \frac{1}{2}(1 + X_{\text{Corner}}) + \frac{1}{4}(1 + X_{\text{Food}})$

$X_{\text{Start}} = (1 + X_{\text{Edge}})$

It's a linear system with 4 equations and 4 unknowns. Solving it for X_{Start}, we get $X_{\text{Start}} = \mathbf{4.5}$, which is the answer to the question.

2 Question 2

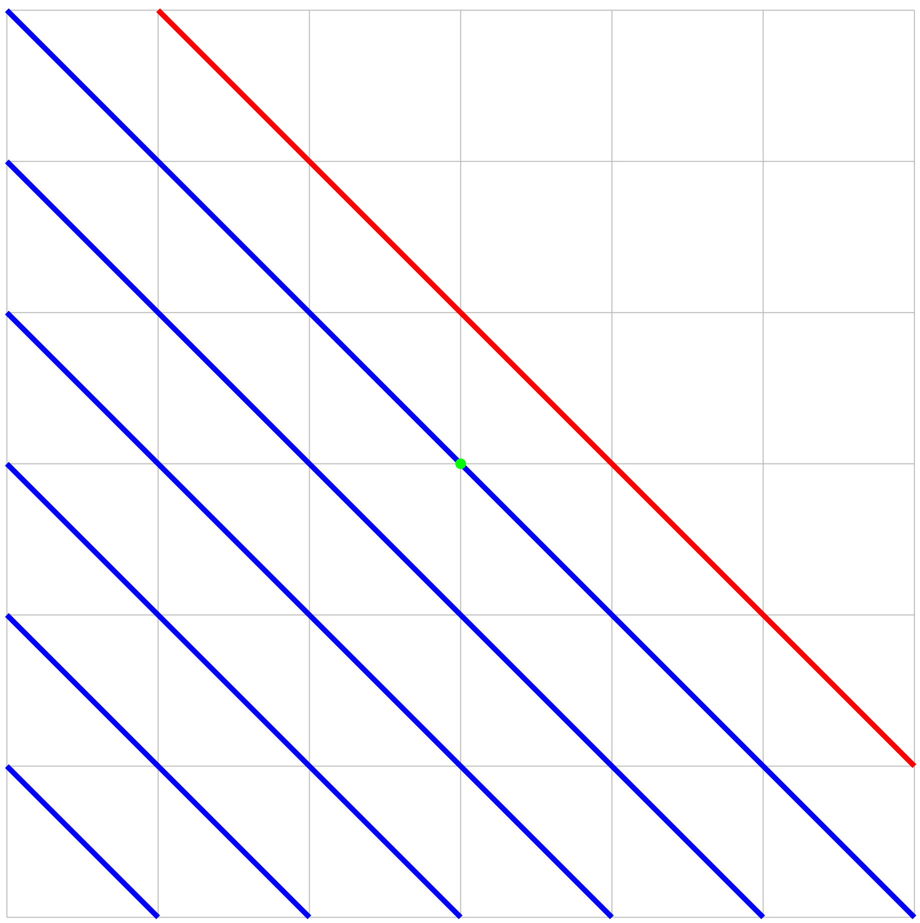
For the second question, the food is located on a diagonal line passing through the points ([10,0],([0,10]). In the following figure, we represent the situation with the starting point (anthill) once again marked in green.



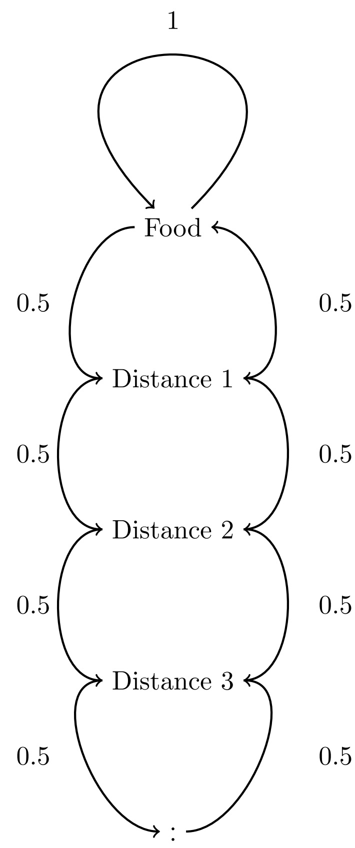
This time it looks like we have infinite possible positions to keep into consideration. However, we can make the following observation. At every step, the ant either moves 1 step closer or further to the *food line* and these two events both happen with probability $\frac{1}{2}$. As we show in the following figure, the position of the ant can be fully described by which parallel line (to the *food line*) it occupies

at every given step. We mark each blue line with an integer representing its distance from the red *food line*.
So the question becomes:

Consider a marker on the line of integers starting at 1. Each second, the marker moves either one step to the left or one step to the right with equal probability. What is the average amount of time needed for the marker to reach the point 0?



Again, this reduced problem can be modelled as a Markov Chain:



Once again, we are dealing with a Random Walk problem on a Markov Chain with an infinite number of states. Let i denote a state of the Markov Chain, thus $i \in$ [*Food, Distance 1, Distance 2, ...*]. Furthermore, let X_i denote the average number of steps taken to get to the state *Food* from state i.

Naturally, $X_{\text{food = 0}}$.
For each i = *Distance k*, with $k \in N$, by the Markov property of Random Walks, we have:

$$X_{\text{Distance}_{\text{i+1}}} = 2X_{\text{Distance}_{\text{i}}}$$

We are interested in computing $\text{X}_{\text{Distance}_1}$

Let us generalize the problem for any probability p to move closer to the food. In our case, $p = 0.5$. For a generic p, we have that:

$X_{\text{Distance}_1} = p + (X_{\text{Distance}_2} + 1)(1 - p)$

$X_{\text{Distance}_1} = p + (2X_{\text{Distance}_1} + 1)(1 - p)$

$X_{\text{Distance}_1} = \frac{1}{2p-1}$

As p approaches $\frac{1}{2}$, $\mathrm{X}_{\text{Distance}_1}$ tends to ∞ , which is the answer to the question.

3 Question 3

We can write a Python program, which uses the *Shapely* library to draw our boundary. Furthermore, the method *contains* allows to efficiently check if a point is inside or outside the boundary. This holds true for every kind of shapes (circle, ellipses, polygons). In our case, we need to draw the required ellipse. We can write a code that run multiple simulations for every ant starting at the point(0,0). At each step, we randomly move the ant and check if is still within the boundary. Once the ant is out of the boundary, we can return the number of steps it took. After multiple simulations, we can estimate a confidence interval for the average number of steps. It turns out that the closest integer in this interval is **14**, which is the answer to the question. Here we provide the snippet of the Python code used for the simulation.

```
import random
import numpy as np
from shapely import Point, Polygon
from shapely.affinity import scale
from scipy.stats import norm
import matplotlib.pyplot as plt


"""
This class defines the ellipse which form the boundary"
"""
class Shape_Ellipse():

    def __init__(self, center, semi_major, semi_minor):

        self.center = Point(center[0], center[1])
        self.ellipse = scale(self.center.buffer(1),
        ↪  xfact=semi_major, yfact=semi_minor)

    """
```

```python
    A line's endpoints are part of its boundary and are therefore not contained.
    """
    def does_contain(self,point):

        return self.ellipse.contains(Point(point[0],point[1]))


    def plot(self):

        fig, ax = plt.subplots()
        ax.set_aspect('equal')
        ax.add_patch(plt.Polygon(list(self.ellipse.exterior.coords), alpha=0.5))
        ax.autoscale_view()
        plt.show()


"""
This class allows to run simulations for the problem
"""
class Simulation_Ellipse():

    def __init__(self, center, semi_major, semi_minor):

        self.position = [0,0]
        self.ellipse = Shape_Ellipse(center, semi_major, semi_minor)

    """
    Randomly update ant position
    """
    def get_next_position(self):

        dir = random.randint(0,3)

        curr_x = self.position[0]
        curr_y = self.position[1]

        if dir == 0:
            self.position = [curr_x + 10, curr_y]

        elif dir == 1:
            self.position = [curr_x - 10, curr_y]

```

```
        elif dir == 2:
            self.position = [curr_x, curr_y + 10]

        elif dir == 3:
            self.position = [curr_x, curr_y - 10]


    """
    Run a single simulation for a single ant and returns the number of steps taken.
    """
    def run_simulation(self):

        count = 0
        self.position = [0,0]

        while self.ellipse.does_contain(self.position):

            count += 1
            self.get_next_position()

        return count


    """
    Compute a confidence interval for the mean of the population
    """
    def get_confidence_interval(self, confidence, n):

        "Store observations"
        observations = []
        for i in range(n):
            observations.append(self.run_simulation())

        """
        Compute sample mean and sample_variance
        """
        sample_mean = sum(observations) / n

        sample_variance = 0
        for i in range(n):
            sample_variance += (observations[i] - sample_mean) ** 2
        sample_variance = sample_variance / (n - 1)

```

```
        z_alpha_halved = norm.ppf((1-confidence) / 2 + confidence, loc=0, scale=1)

        print("The ", confidence*100, "% confidence interval for the mean of the population is: [",
                sample_mean - z_alpha_halved * (np.sqrt(sample_variance) /  np.sqrt(n)) ,
                sample_mean + z_alpha_halved * (np.sqrt(sample_variance) /  np.sqrt(n)),
                "]")

    def plot(self):

        self.ellipse.plot()


"""
Define here the parameters of the Ellipse
"""
CENTER = [2.5,2.5]
semi_major = 30
semi_minor = 40
sim = Simulation_Ellipse(CENTER,semi_major,semi_minor)


"""
Define here the statistical parameters
"""
CONFIDENCE = 0.95
SAMPLES = 10000

sim.get_confidence_interval(CONFIDENCE, SAMPLES)

```