title random bee walker | author samyak ahuja | Github samyakducs

Random Walker : Bee

Files

Programming was done predominantly in javascript and to use a graphics library p5js (for interactive experience) additional web files were generated. **app.js** file includes all the relevant programming.

To launch the program just open the **index.html** file in a browser, preferably google chrome because of certain JavaScript Engine support issues.

The simulation is also hosted on github and with internet access can be accessed **here**.

Design Decisions

To start with this problem is just a variant of the random walker problem where the probability of moving in a certian direction is 1/6 and the direction themselves are 6 which are separated by 60-degree angles in a cartesian plane.

So problems that needed solving were

- Creating a Mathematical Equation for moving to the next position in the path.
- Calculating the distance between the starting position and the finishing position of the path.
- Deciding on the way the problem needs to be solved, There are two main methods:
  - Solving by the simulation method where multiple runs of the algorithm are done and then the expected values are calculated based on the data collected.
  - Solving by a purely mathematical approach where the problem will be solved by using mathematical equations only.

1. Moving to the next position in the path.

First a random value from 0 to 5 is generated because in a hexagonal grid there are 6 directions for a Walker to move. Now we need to move to the next hexagon based on this random variable.

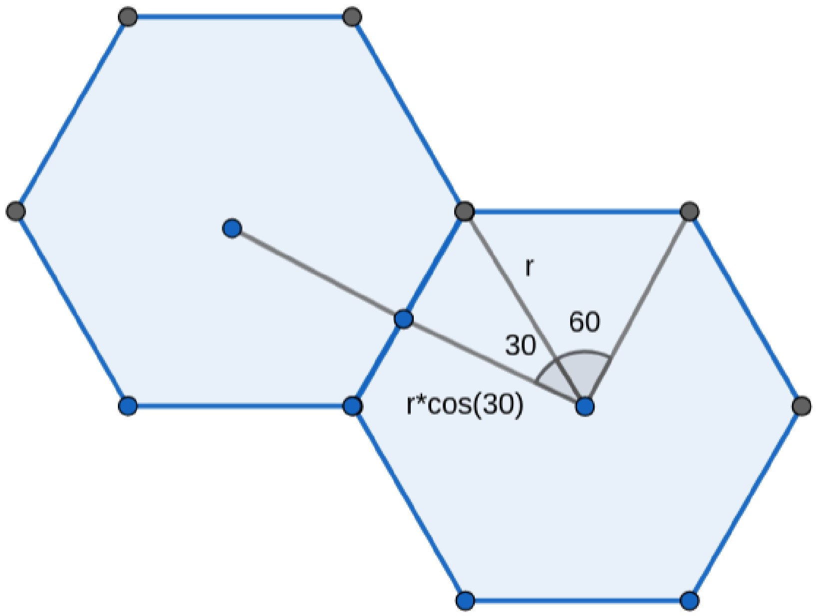
Mapping from this random variable to a change in the x and y location of the Walker can be done in two ways:

- Directly coding the changes in x and y using if-else statements.
- Using cos and sin functions to move the Walker in a particular direction.

I choose the second option just because it is more elegant and shorter to write.

```
let theta = (random_var * 60) + 30
this.x = this.x + 2 * this.radius * Math.cos(radians(theta))
this.y = this.y - 2 * this.radius * Math.sin(radians(theta))
```

The angle theta is computed by mulipying the random variable(0-5) with 60 since vertices are 60-degree apart. Finally 30 is added because we want to bisect the edge of the Hexagon in order to get to the center of the next hexagon.



Here the x location is changed by 2 times the radius of a hexagon (which is assumed to be equal to the distance between the center and any vertex of the hexagon) in the direction of cos(theta). Similar thing can be said for y location but one thing to be noted here is that the y location is decremented instead of incrementing. This is because of the way computers interpret coordinate axes, where it is inverted as compared to normal coordinate axes we see everyday.

2. Calculating the Distance

Distance between the starting and the final positions in the path can be calculated by using various methods but the method I choose is that of Euclidean distance, which is the shortest distance between any 2 points in Euclidean Space.

This gives an idea of displacement between the center of the starting hexagon and the center of the finishing hexagon. The equation that I used for calculating is:

```
let dist2Hex = bee.radius * cos(radians(30))
let distance = dist(bee.getX(), bee.getY(), w / 2, h / 2) / dist2Hex
```

Here *dist2Hex* is the unit of measurement that I used which is equal to the distance between the center of a hexagon and the point at which the edge of the hexagon is bisected by the line joining the centers of 2 neighbouring hexagons.

The distance calculated originally was in terms of pixels and hence to normalize it I divided it by the unit of measurement *dist2Hex*

3. Method of Computation

I took an iteration to be defined as a block that executes, during which it goes from the starting state to the finishing state in a defined number of steps all according to the constraints defined in the problem, and at the end appends the distance between the starting and final states to a list.

This list gets populated during numerous iterations that are run. In my case I did 1000 such iterations. During each iteration I incrementally updated the mean and at the end of all iterations for a particular number of steps(like 16) I calculated the standard deviation.

The same process was repeated for some other number of steps, say 64.

Result and Analysis

Results obtained for paths of length 16 and 64 for any one generation were:

Number of steps	Expected Value	Standard Deviation
16	8.27	4.48
64	16.94	8.73

The standard deviation is too high for the solution to be called stable