Lecture 13:

Distributions, Random walks

(download slides and .py files to follow along)

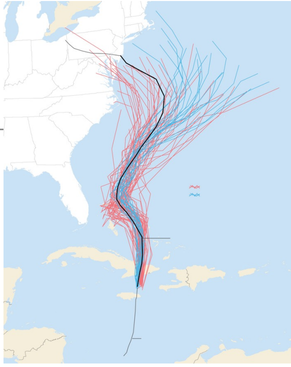
Tim Kraska

MIT Department of Electrical Engineering and Computer Science

Announcements

- Pset 3 is due tonight.
- Pset 4 is out after class.

Recap: Monte Carlo Simulation



- A method of estimating the value of an unknown quantity using the principles of inferential statistics
- Inferential statistics
  - *Population*: a set of examples
  - *Sample*: a proper **subset** of a population
  - Key fact: a *random sample* tends to exhibit the same properties as the population from which it is drawn
    - Provided size of sample is sufficiently large and random
      - The key question: When should we believe that a sample has the same properties as the population?

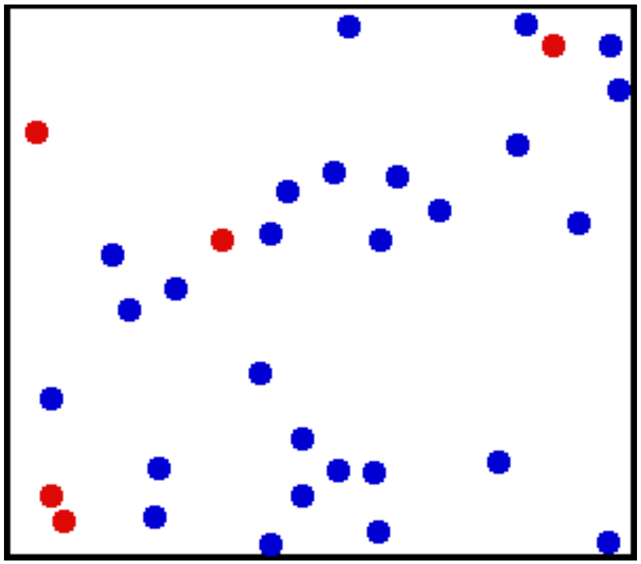
Simulations Are Used a Lot

- To model systems that are mathematically intractable
  - Otherwise may not be possible to analyze a system
- To extract useful intermediate results
- To support iterative development by successive refinement
  - Start with basic system, then incrementally add features
- To support exploring of variations by asking/answering “what if” questions
- Let’s illustrate with random walks

What are Random Walks?

Method to model a system where:

- Objects start at a location and **choose a random direction** in which to take each step
- Distribution of choice of direction (and speed) can be different
  - For different object types
  - For different environmental conditions

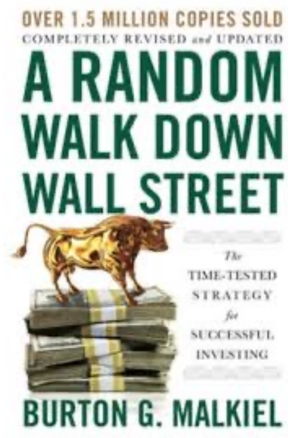


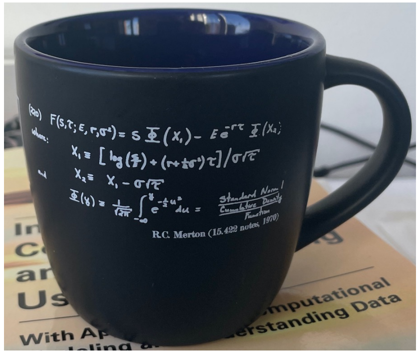
Random Walks in the World

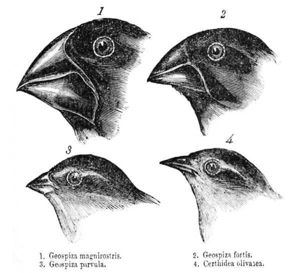
- In modeling stocks, change in price often modeled as a Gaussian random walk
  - Step size chosen from a normal distribution
  - Basic assumption of Black-Scholes-Merton option pricing model

Scholes and Merton won a Nobel Prize for this work

- In population genetics, random walk describes statistical properties of genetic drift
  - Change in frequency of specific gene variant in a population







Darwin's finches

Why Do We Cover Random Walks?

- Important for modeling behavior in many domains
  - Understanding the stock market, modeling diffusion processes, modeling cell movement, suggesting who to follow on Instagram, etc.
- Solve computational problems (Search, AI, lighting simulation)
- Good illustration of how to use simulations to understand things
- Excuse to cover some important programming topics
  - Practice classes
  - Practice plotting

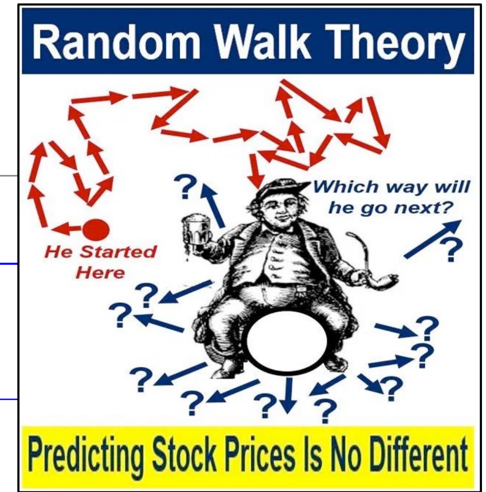
Our Random Walk

- Often called the drunkard's walk
- A random walk on a two-dimensional surface
  - Walker starts at some initial location
- Each step a fixed distance (initially)
- Step is taken in a direction chosen randomly from a set of choices

Homer's Odysseus – classic wanderer

Homer – Just Odd

Simplified Drunkward’s Walk



Homer starts at the origin, and takes a unit step in one of the cardinal directions at random with equal likelihood.

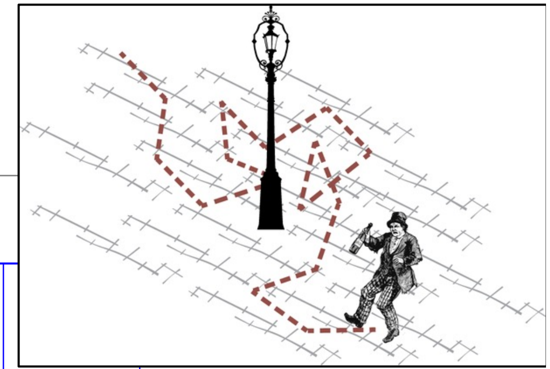
How does distance from origin at end of walk relate to number of steps?

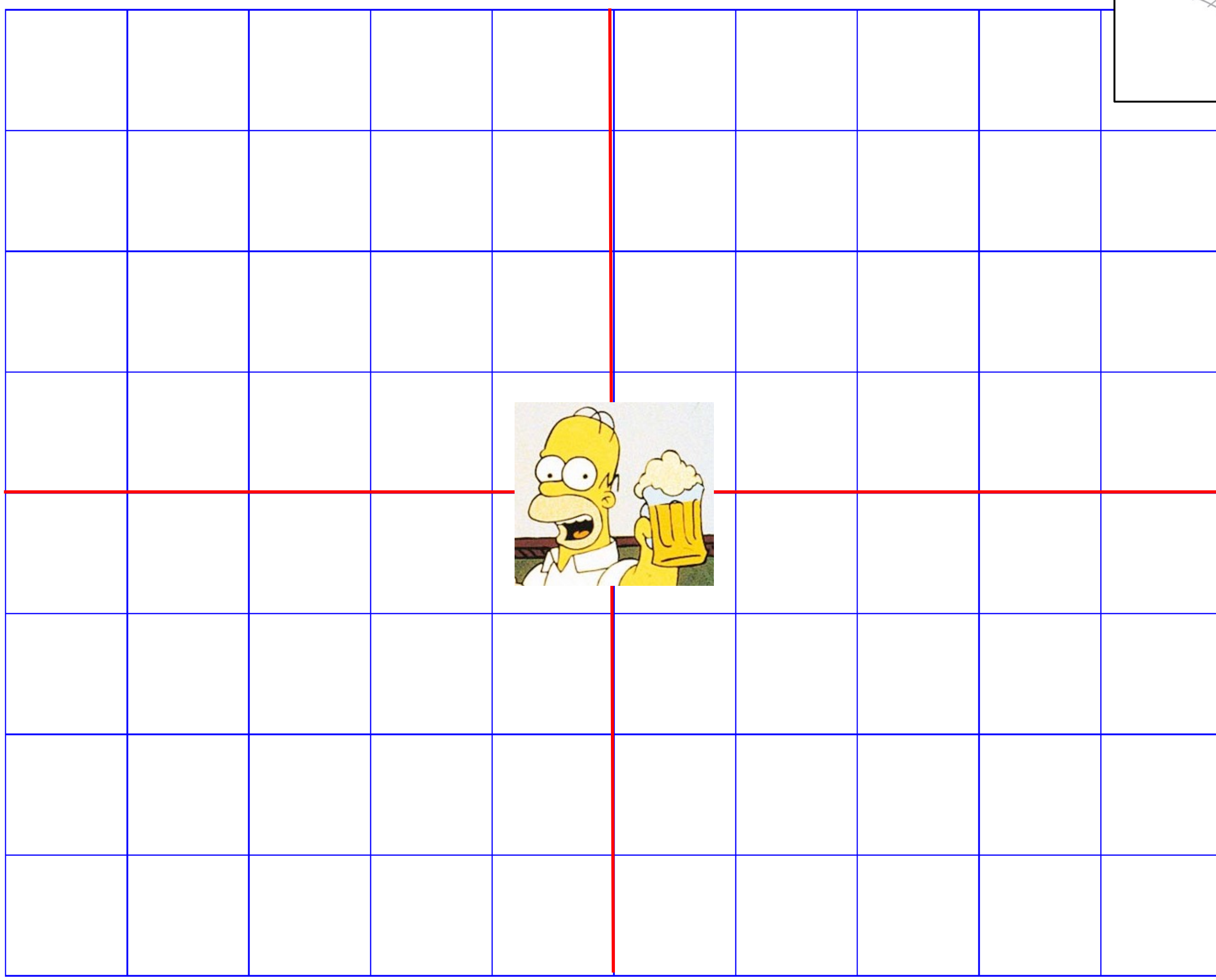
How Far will Homer Get?

How will the expected distance from the starting point change as the number of steps, d, grows?

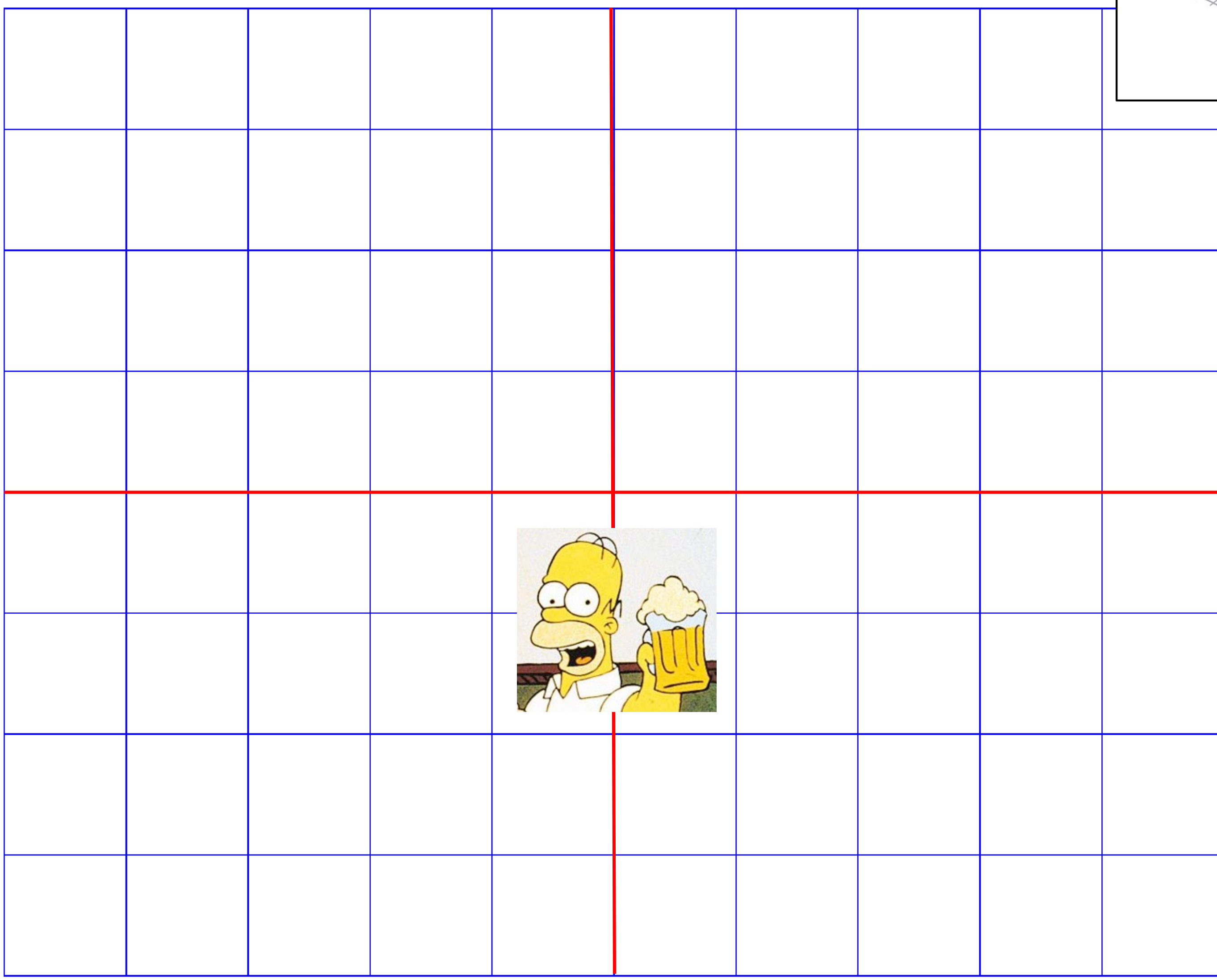
- It won’t
- O(d)
- O(d**2)
- O(log d)
- O(d**0.5)
- O(2**d)

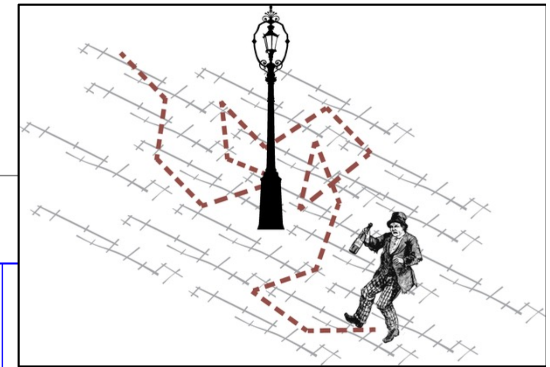
Hand Simulation



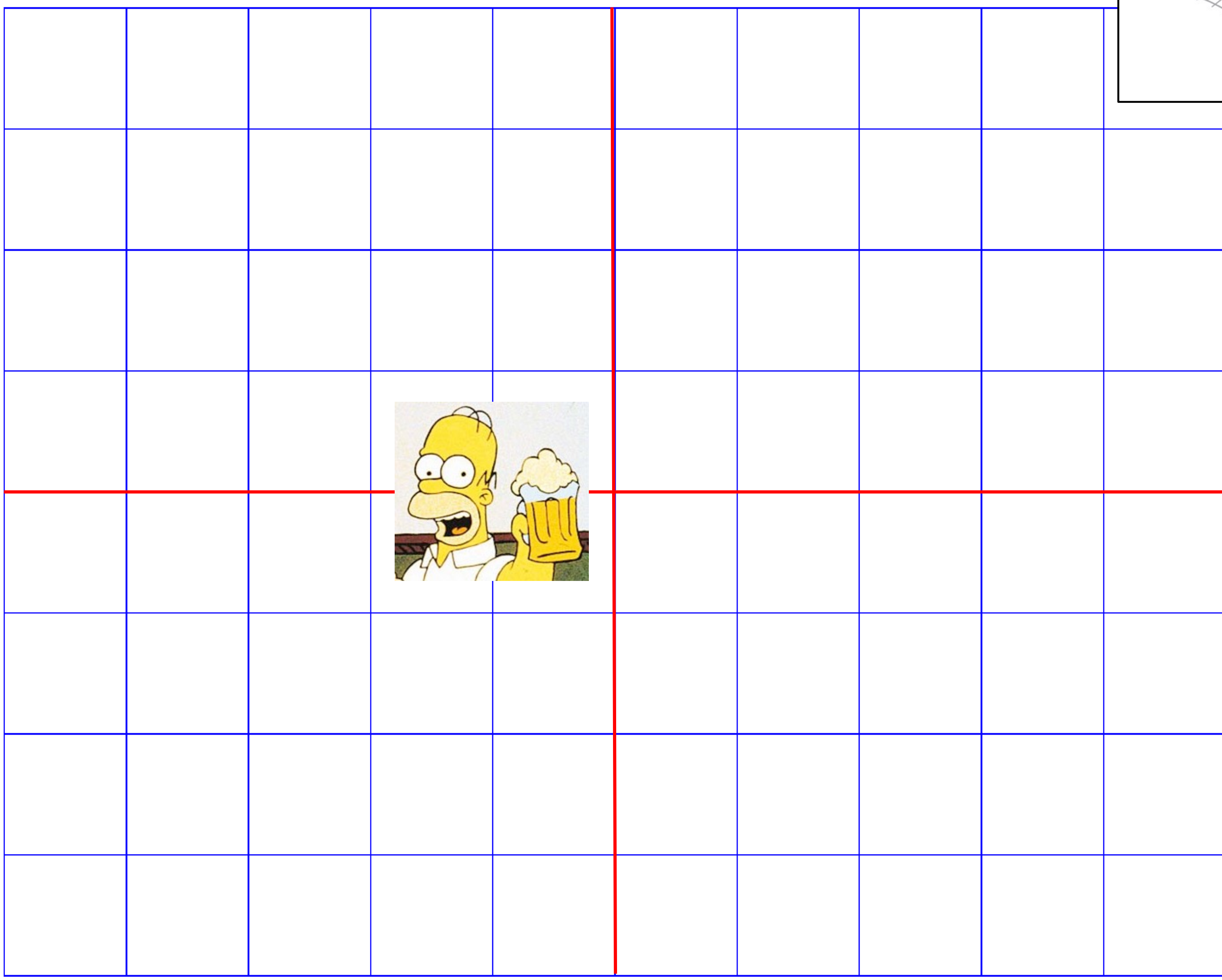


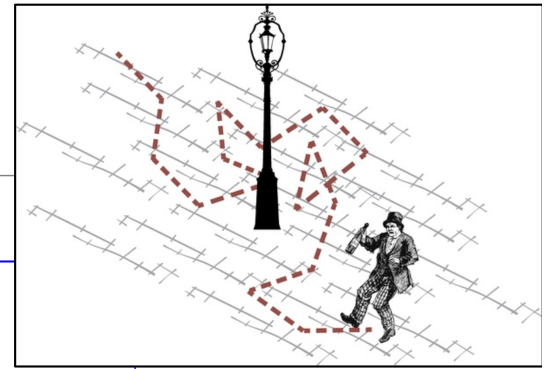
One Possible First Step



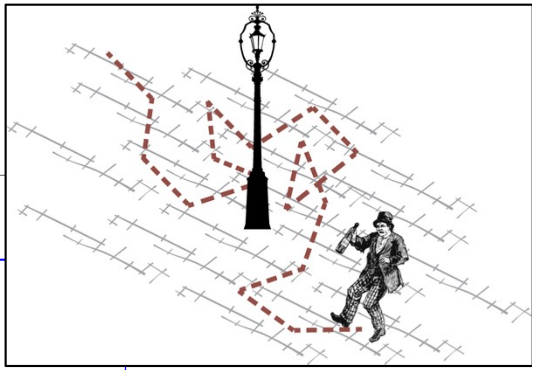


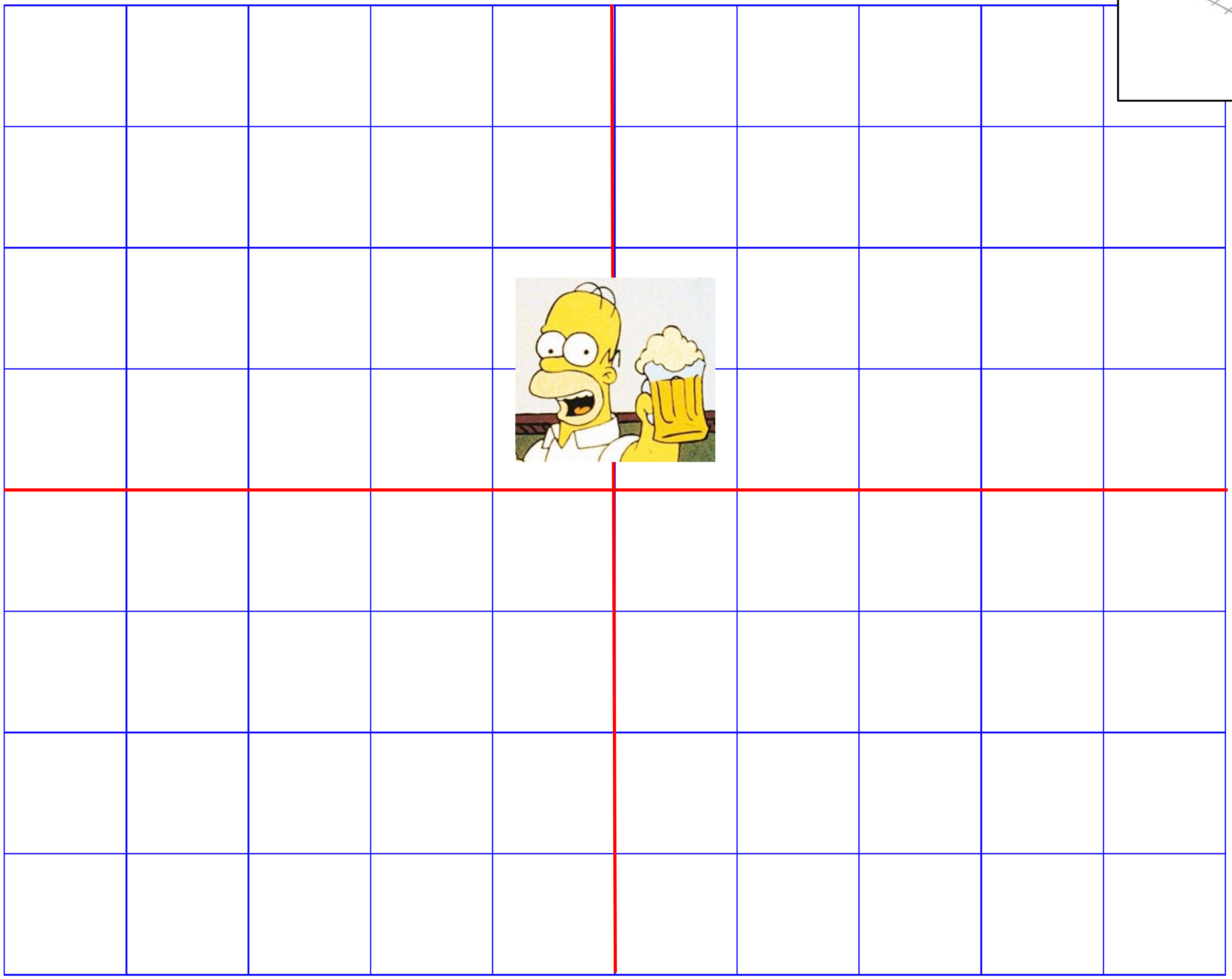
Another Possible First Step



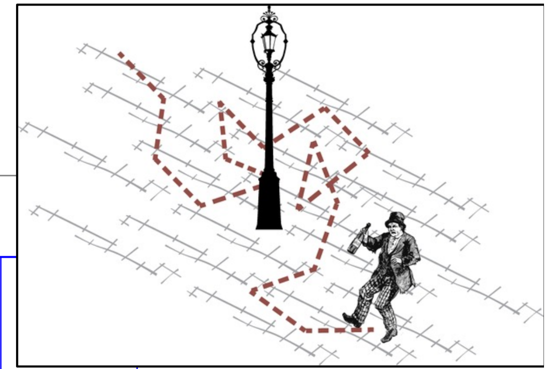


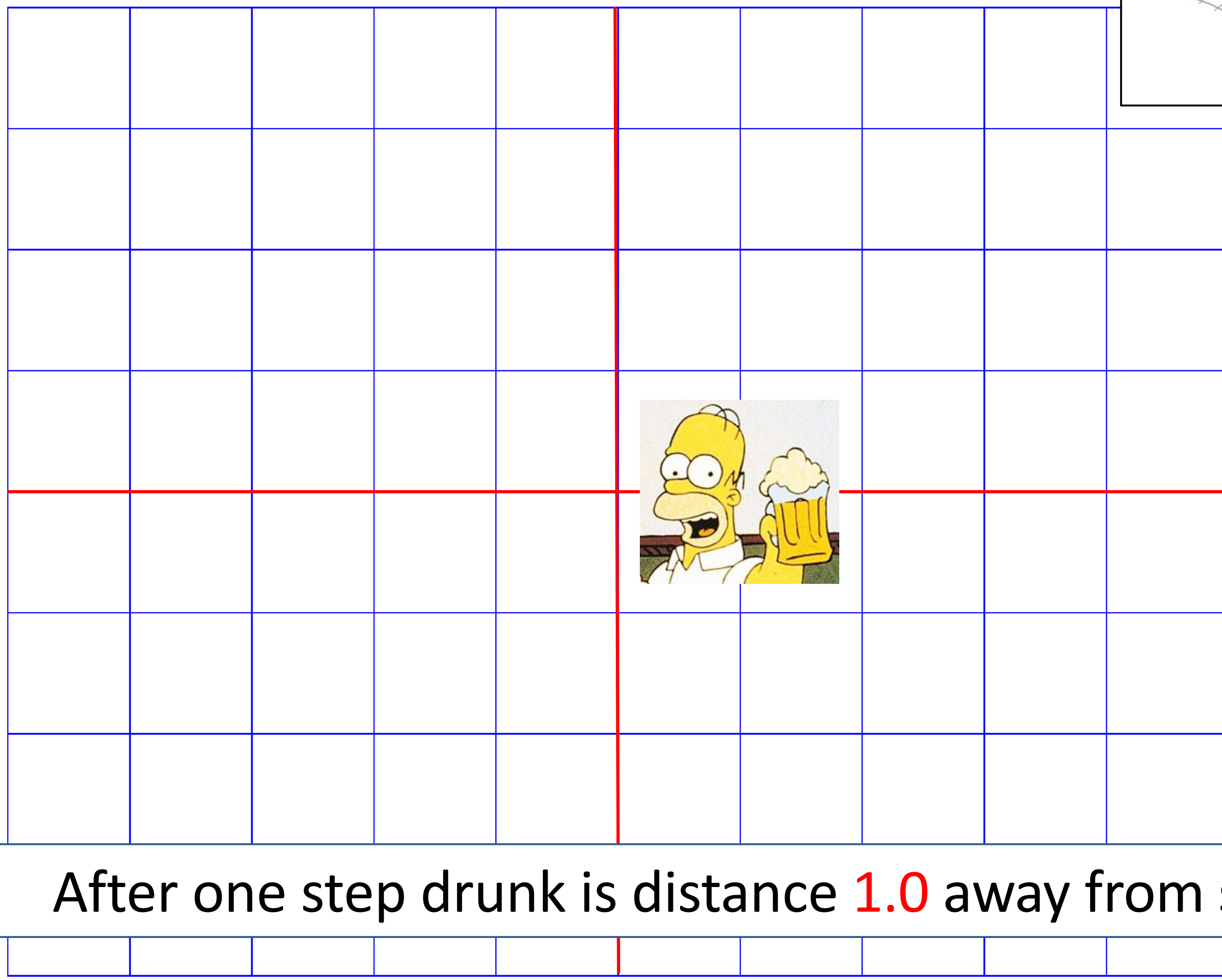
Yet Another Possible First Step





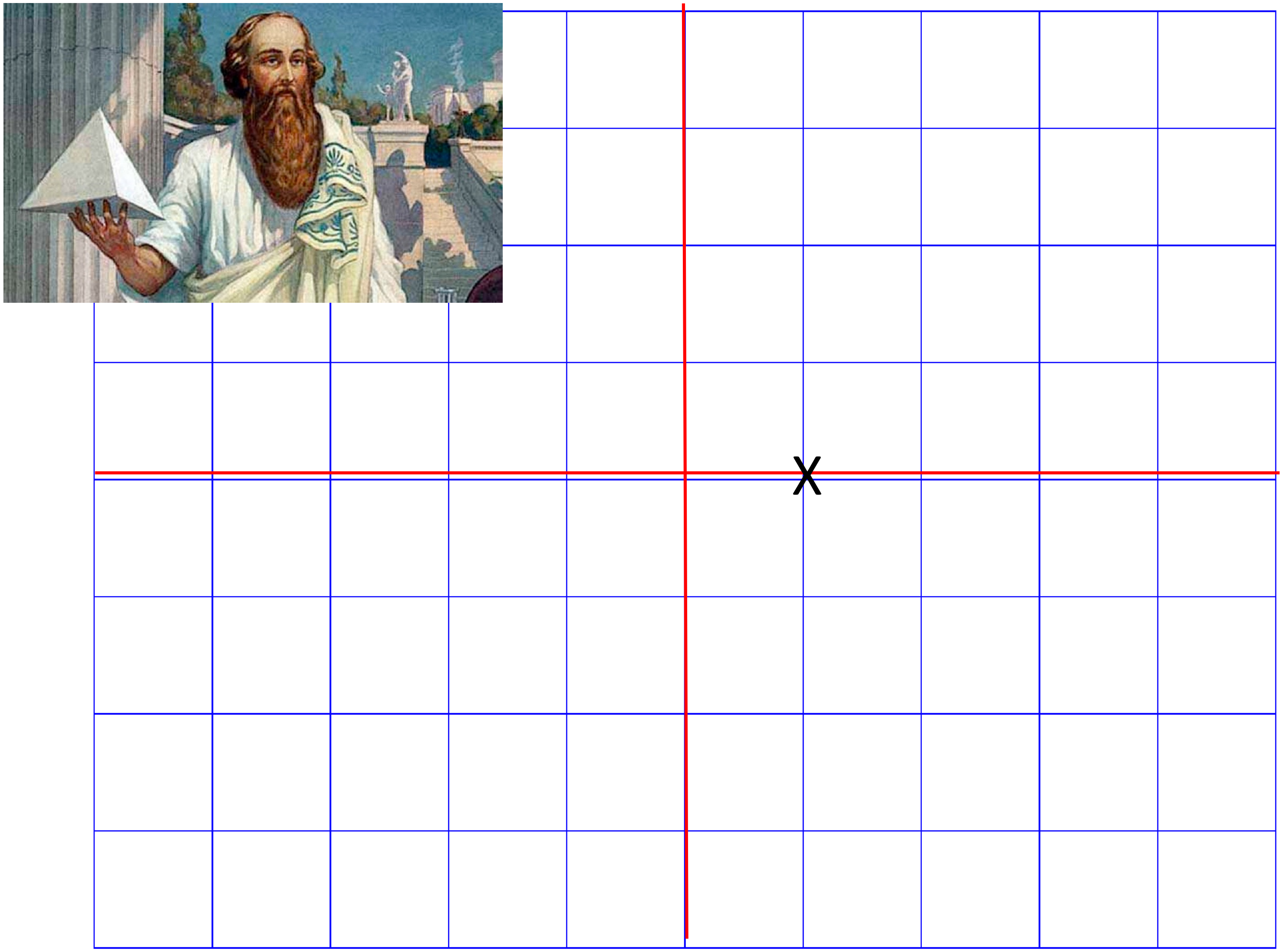
Last Possible First Step





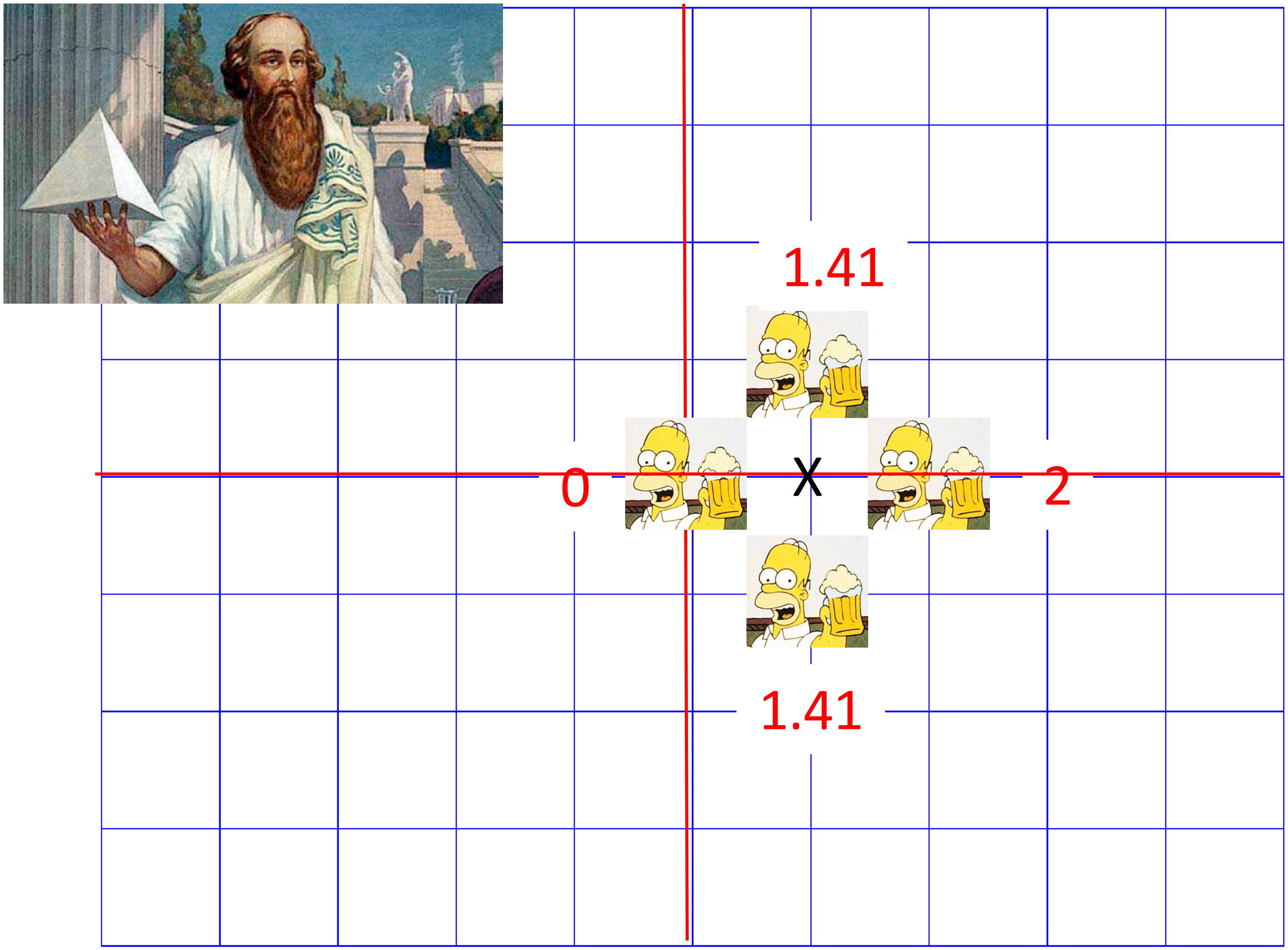
After one step drunk is distance 1.0 away from start

Possible Locations after Two Steps



Assuming 1st step to right

Possible Locations after Two Steps

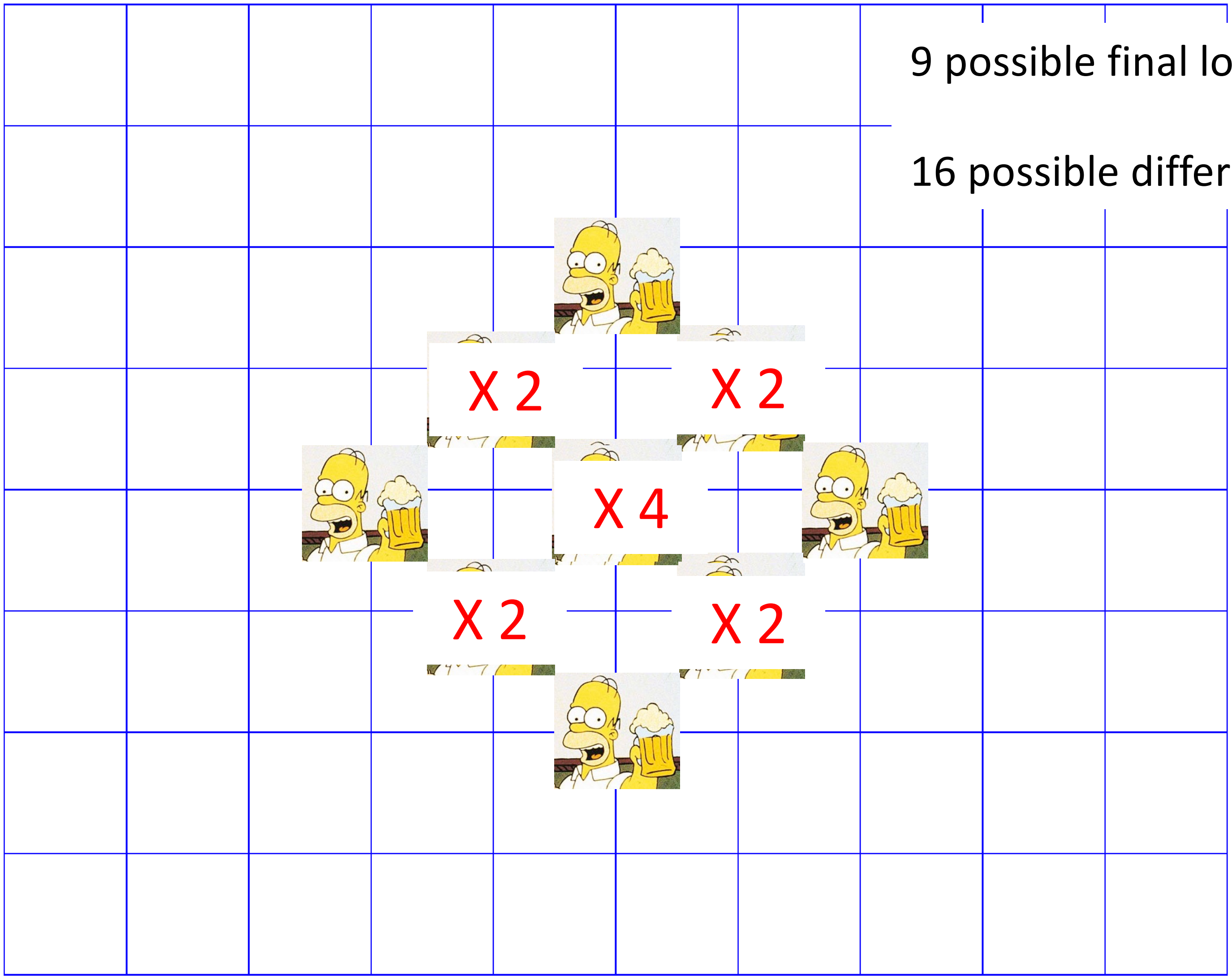


$a^2 + b^2 = c^2$

$c = (a^2 + b^2)^{0.5}$

Assuming 1st step to right

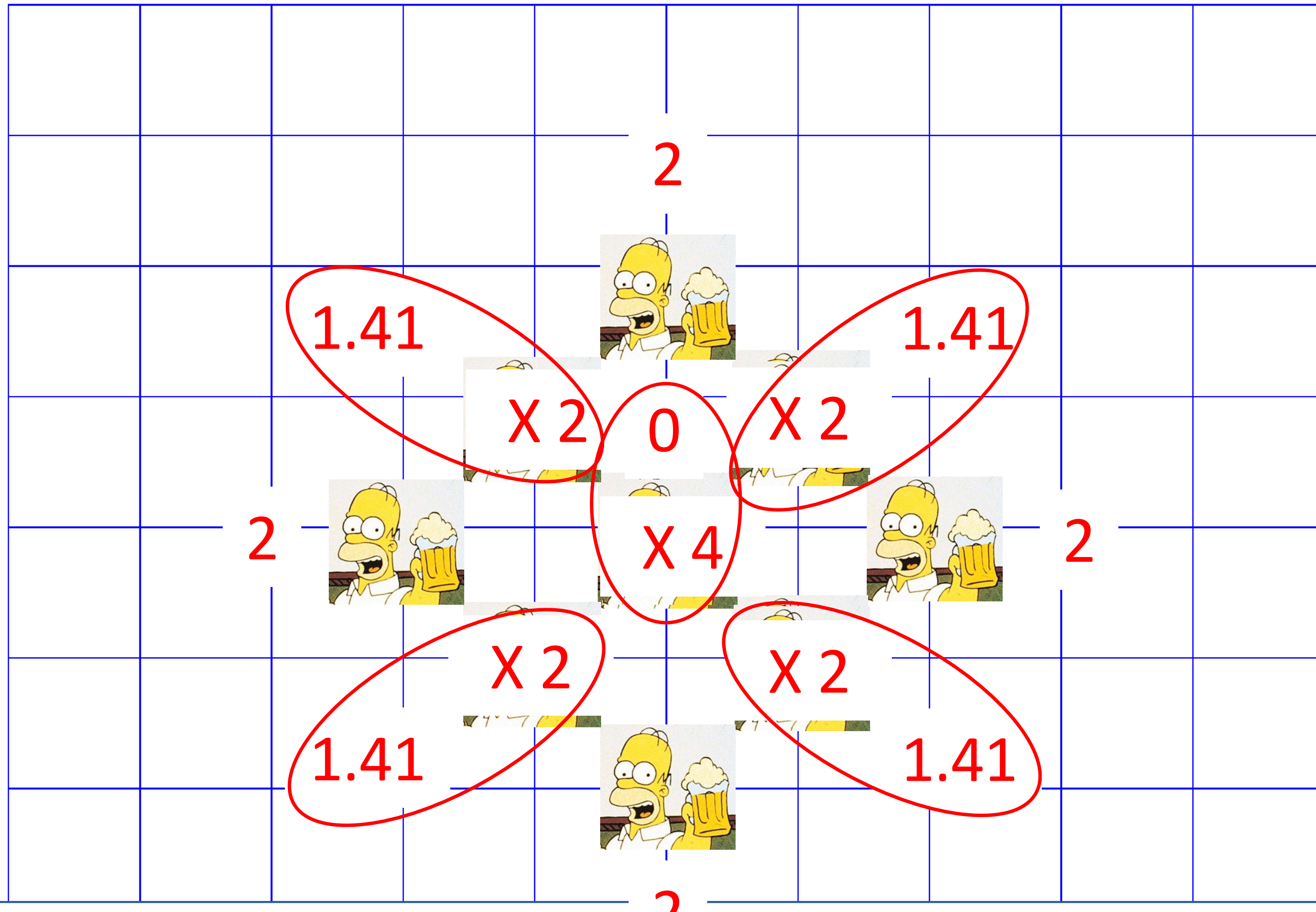
Possible Distances After Two Steps



9 possible final locations

16 possible different paths

Possible Distances After Two Steps



Average if equally likely is:

(2 + 2* 1.41 + 2 + 2* 1.41 + 2 + 2* 1.41 + 2 + 2 *1.4 + 4*0)/16

= 1.205

Expected Distance After 100,000 Steps?

- Can see that expected distance seems to increase with number of steps, and certainly maximum distance does
  - Went from **0** to **1** to **1.205** as expected distance
- But can we accurately model the actual expected distance (i.e., with a mathematical formula)?
  - Seems pretty hard
- What about computing all possible paths, and measuring actual average distance?

Feasible?

How many possible **paths** are there after 100,000 steps?

- 100,000
- 400,000
- 100,000**2
- 4**100,000
- 2**100,000

Feasible?

How many possible **paths** are there after 100,000 steps?

- 100,000
- 400,000
- 100,000**2
- 4**100,000
- 2**100,000

Note, the number of possible paths is different from the number of possible final locations – latter is roughly 100,000**2 = 10B

Enumerating all Possibilities Is Not Practical

- Need a different approach to problem
- A probabilistic simulation
  - Won’t yield precisely correct answer
  - But should yield a good approximation

Modeling Random Walks

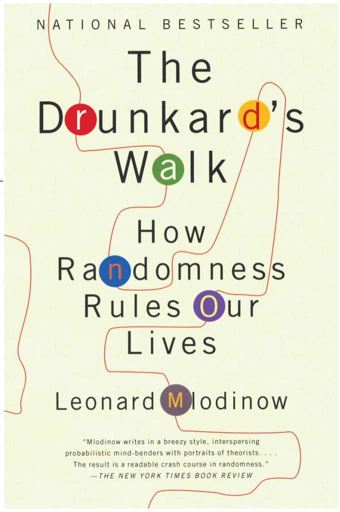
1. Perform an experiment where you take **d steps at random based on some distribution** and ask some questions about what happened, e.g., what is the distance from starting point
   - A particular event

Modeling Random Walks

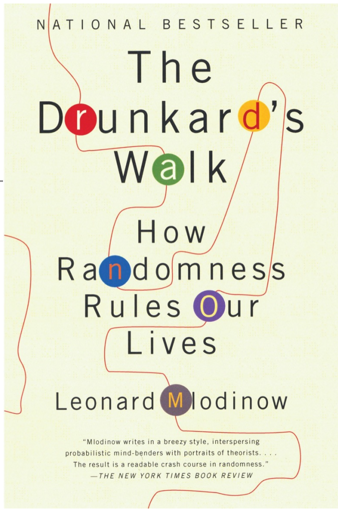
1. Perform an experiment where you take **d steps at random based on some distribution** and ask some questions about what happened, e.g., what is the distance from starting point
   - A particular event
2. With many **experiment trials/repetitions**, what is the average distance away from the start location (or other properties of the set of end or intermediate points)?
   - Sampling space of possible events
3. Can't sample all events, so want to draw statistical conclusions about events—topic of a later lecture

Structure of Simulation

- Simulate one walk of *d* steps
  - Record distance from start at end of walk
  - Record final location at end of walk



Structure of Simulation

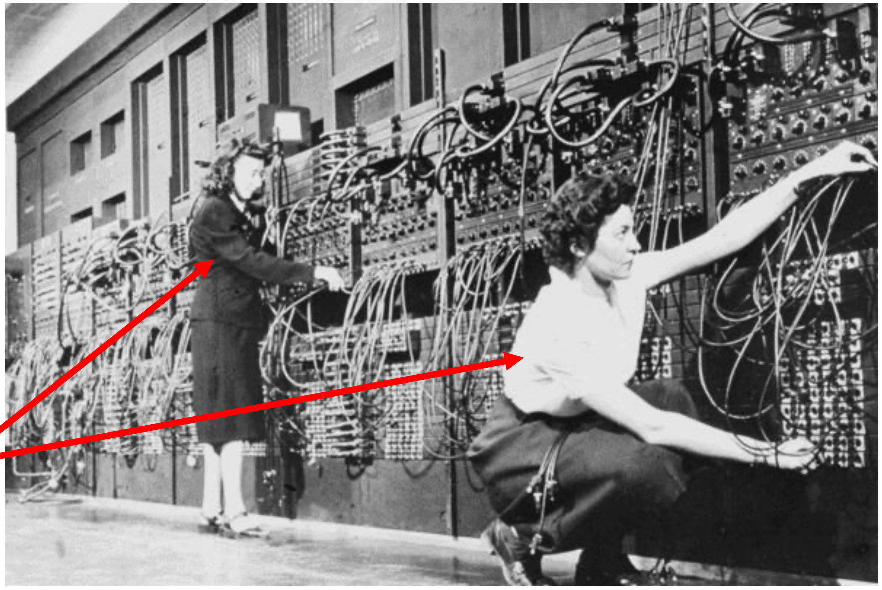


- Simulate one walk of *d* steps
  - Record distance from start at end of walk
  - Record final location at end of walk
- Simulate *n* such walks, each of *d* steps
  - Record all distances and final locations
- Report average distance from origin over set of *n* walks
  - How does this change as we increase *d*?
- Will come back to final locations over set of walks later

Monte Carlo Simulations

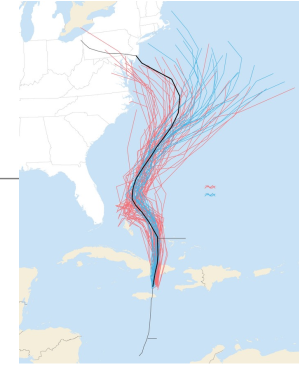
- Stanislaw Ulam, recovering from an illness, was playing a lot of solitaire (1946)

- Tried to figure out probability of winning, and failed
- Thought about playing lots of hands and counting number of wins (i.e., sampling)
  - ~10,000 hands needed
- Asked Von Neumann if he could build a program to simulate many hands on ENIAC
  - Creation of Monte Carlo simulation method



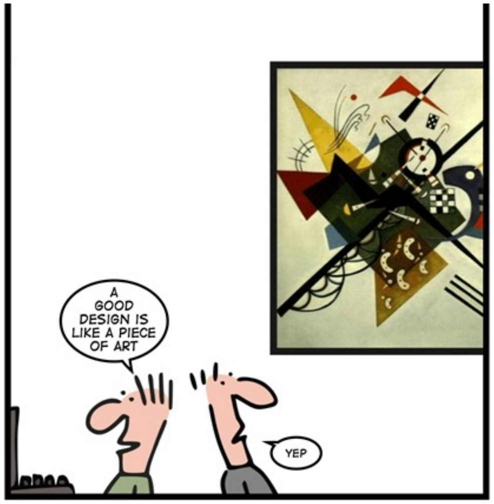
Programming, circa 1946

Monte Carlo Simulation



- A method of estimating the value of an unknown quantity using the principles of inferential statistics
- Inferential statistics
  - *Population*: a set of examples
  - *Sample*: a proper **subset** of a population
  - Key fact: a *random sample* tends to exhibit the same properties as the population from which it is drawn
    - Provided size of sample is sufficiently large and random
- From where does name come?
  - Ulam and von Neuman used the method to study particle interactions in nuclear weapons research. Because the research was secret, they needed a code name. Ulam named it after the casino in Monaco.

Random Walk - Overall plan



- First, some useful abstractions
- Location—a place
  - An object, like a drunk, has a location
  - Location can change as object moves
- Field—a collection of drunks with locations
  - Implicit collection of locations
  - Explicitly represent only locations occupied by drunks
  - Associate location directly with a drunk
  - Represent by collection of drunks, not collection of possible locations

Random Walk - Overall plan

- Drunk
  - Can move, change location
- Simulation
  - Create one or more drunks, with initial location
  - Randomly move d steps, record final location, distance from start
  - Do this n times, measure **average** final distance, collect set of final locations

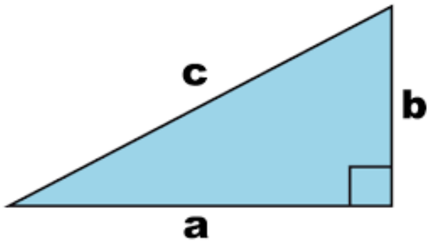
Location

```
def make_location(x, y):
    return (x, y)

def move(loc, dx, dy):
    x, y = loc
    return (x + dx, y + dy)

def get_x(loc):
    return loc[0]

def get_y(loc):
    return loc[1]

def dist(loc_a, loc_b):
    return hypot(loc_a[0] - loc_b[0], loc_a[1] - loc_b[1])

def loc_str(loc):
    return '<' + str(get_x(loc)) + ', ' + str(get_y(loc)) + '>'
```

Locations are tuples

'Yes, business is brisk – Location, Location, Location!'

Python has functions for everything math.hypot() calculates Euclidian norm



$a^2 + b^2 = c^2$

Fields

```python
def make_field():
    return {}

def add_drunk(field, drunk_id, loc):
    if drunk_id in field:
        raise ValueError("Duplicate drunk")
    field[drunk_id] = loc

def get_loc(field, drunk_id):
    if drunk_id not in field:
        raise ValueError("Drunk not in field")
    return field[drunk_id]
```

Modeling Drunks

```
def usual_step():
    return random.choice([(0, 1), (0, -1), (1, 0), (-1, 0)])
```

Basic activity of a drunk is to wander; take_step will return an x and y tuple, denoting **change** in location

Modeling Drunks

```python
def usual_step():
    return random.choice([(0, 1), (0, -1), (1, 0), (-1, 0)])

def masochist_step():
    return random.choice([(0.0, 1.1), (0.0, -0.9), (1.0, 0.0), (-1.0, 0.0)])
```

Takes a random step in cardinal directions, but if northward, moves a bit further than if southward

Modeling Drunks

```
def usual_step():
    return random.choice([(0, 1), (0, -1), (1, 0), (-1, 0)])

def masochist_step():
    return random.choice([(0.0, 1.1), (0.0, -0.9), (1.0, 0.0), (-1.0, 0.0)])

def liberal_step():
    return random.choice([(0.0, 1.0), (0.0, -1.0), (0.9, 0.0), (-1.1, 0.0)])

def conservative_step():
    return random.choice([(0.0, 1.0), (0.0, -1.0), (1.1, 0.0), (-0.9, 0.0)])

def liberal_masochist_step():
    # Flip between liberal and masochist tendencies
    if random.choice([True, False]):
        return liberal_step()
    else:
        return masochist_step()

def corner_step():
    return random.choice([(0.71, 0.71), (0.71, -0.71), (-0.71, 0.71), (-0.71, -0.71)])

def continuous_step():
    return (random.uniform(-1,1), random.uniform(-1,1))
```

Simulating a Single Walk

Takes a function as input

```
def move_drunk(field, drunk_id, step_fn):
    if drunk_id not in field:
        raise ValueError("Drunk not in field")
    dx, dy = step_fn()
    field[drunk_id] = move(field[drunk_id], dx, dy)
```

Executes the function

```
def walk(field, drunk_id, step_fn, num_steps):
    """Moves drunk_id num_steps times; returns distance from start to end."""
    start = get_loc(field, drunk_id)
    for _ in range(num_steps):
        move_drunk(field, drunk_id, step_fn)
    return dist(start, get_loc(field, drunk_id))
```

FUNCTIONS AS PARAMETERS

DETOUR

Motivation

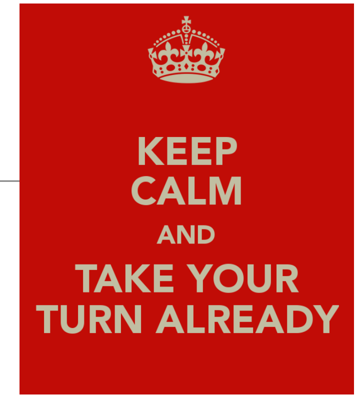
- We want to create a function such that:
  - Given an input list
  - It outputs a new list with only the elements of the input that pass a given test (a.k.a. predicate)
  - We want the function to be general for any test

Solution:
test function as parameter

```
def filtered_list
```

General filtering function

```
def filtered_list(orig_list, test):
    out_list = []
    for e in orig_list:
        if test(e):
            out_list.append(e)
    return out_list

def test_odd(x):
    return x%2 == 1

l = [1, 2, 3, 4, 5]

print(filtered_list(l, test_odd))
```



A list of functions

```
def add_all(n, fns):
    ans = 0
    for f in fns:
        ans += f(n)
    return ans

def f(x): return x**3
def g(x): return x*2
def h(x): return -2*x
list_of_functions = [f, g, h]

print (add_all(2, list_of_functions))
```

What is printed?

A) error

B) 0

C) 8

D) -12

E) something else

In general: HIGHER-ORDER PROCEDURES

- Everything in Python is an object of some particular type (e.g., ints, floats, strings, tuples, lists, Booleans, even None!)
- Objects
  - have a type
  - can appear in RHS of assignment statement (i.e., we can bind a name to an object)
  - can be used as an argument to a function
  - can be returned as a value from a function
- While this may seem intuitive for data structures, functions are also **first-class objects**
  - In other words, functions can be treated just like numbers, or strings, or tuples, or lists, with respect to these criteria

Higher-order procedure: can take a function as argument, or return a function as value, or both

Lambda

DETOUR

Lambda creates a function without naming it

- Convenient sugar (shorter way to write things) when one-line functions are needed as parameters

```
lambda param1, parame2… : output_expression
```

- Equivalent to defining

```
def func (param1, param2,…):
      return output_expression
```

And passing func

Lambda creates a function without naming it

- Convenient sugar (shorter way to write things) when one-line functions are needed as parameters
- For example:

```
lambda x: x%2 == 1
```

```
def test_odd(x):
    return x%2 == 1

l = [1, 2, 3, 4, 5]

print(filtered_list(l, test_odd))

#the code below is equivalent but creates a function with lambda without naming it
print(filtered_list(l, lambda x: x%2 == 1))
```

Returning Functions

```
def make_specialized_filtering_function(test):
    return lambda orig_list : filtered_list(orig_list, test)

f = make_specialized_filtering_function()
```

Careful whether you mean the function object (without parentheses and parameters)
or you want to call the function with given parameters

Comprehension

DETOUR

Some very common patterns

- Applying a function to all elements of a list
- Filtering a list according to a predicate (Boolean function)

```
def filtered_list(orig_list, test):
    out_list = []
    for e in orig_list:
        if test(e):
            out_list.append(e)
    return out_list
```

- Python offers a sugar to do both easily

LIST COMPREHENSIONS

```
[expr for elem in iterable if mytest]
```

Default to always returning Tru
if no specific test specified

Comprehension examples

```
l = [n for n in range(200) if n%2==1]
l2 = [n**2 for n in l]
l2 = [n**2 for n in l if  n%2==1]
```

LIST COMPREHENSIONS

```
[expr for elem in iterable if mytest]
```

- evaluating this is same as calling function below (where `fn` is a function that computes *expr*)

```
def list_comp(old_list, fn, test=lambda x: True):
    new_list = []
    for e in old_list:
        if test(e):
            new_list.append(fn(e))
    return new_list

list_comp(iterable,
          lambda elem: expr,
          lambda elem: mytest)
```

HIGHER ORDER PROCEDURES

- **Functions are first-class objects**
  - they have a type
  - they can be assigned as a value bound to a name
  - they can be **used as an argument** to another function
  - they can be **returned as a value** from another function
- This enables the creation of concise, easily read code
- **Environment model** explains evolution of code, even when function accepts a function as argument or return a function as value

Back to Random Walk

Simulating a Single Walk

Takes a function as input

```
def move_drunk(field, drunk_id, step_fn):
    if drunk_id not in field:
        raise ValueError("Drunk not in field")
    dx, dy = step_fn()
    field[drunk_id] = move(field[drunk_id], dx, dy)
```

Executes the function

```
def walk(field, drunk_id, step_fn, num_steps):
    """Moves drunk_id num_steps times; returns distance from start to end."""
    start = get_loc(field, drunk_id)
    for _ in range(num_steps):
        move_drunk(field, drunk_id, step_fn)
    return dist(start, get_loc(field, drunk_id))
```

Animated visualization of single walk

- Using extra code
  - No need to understand it yet in detail.

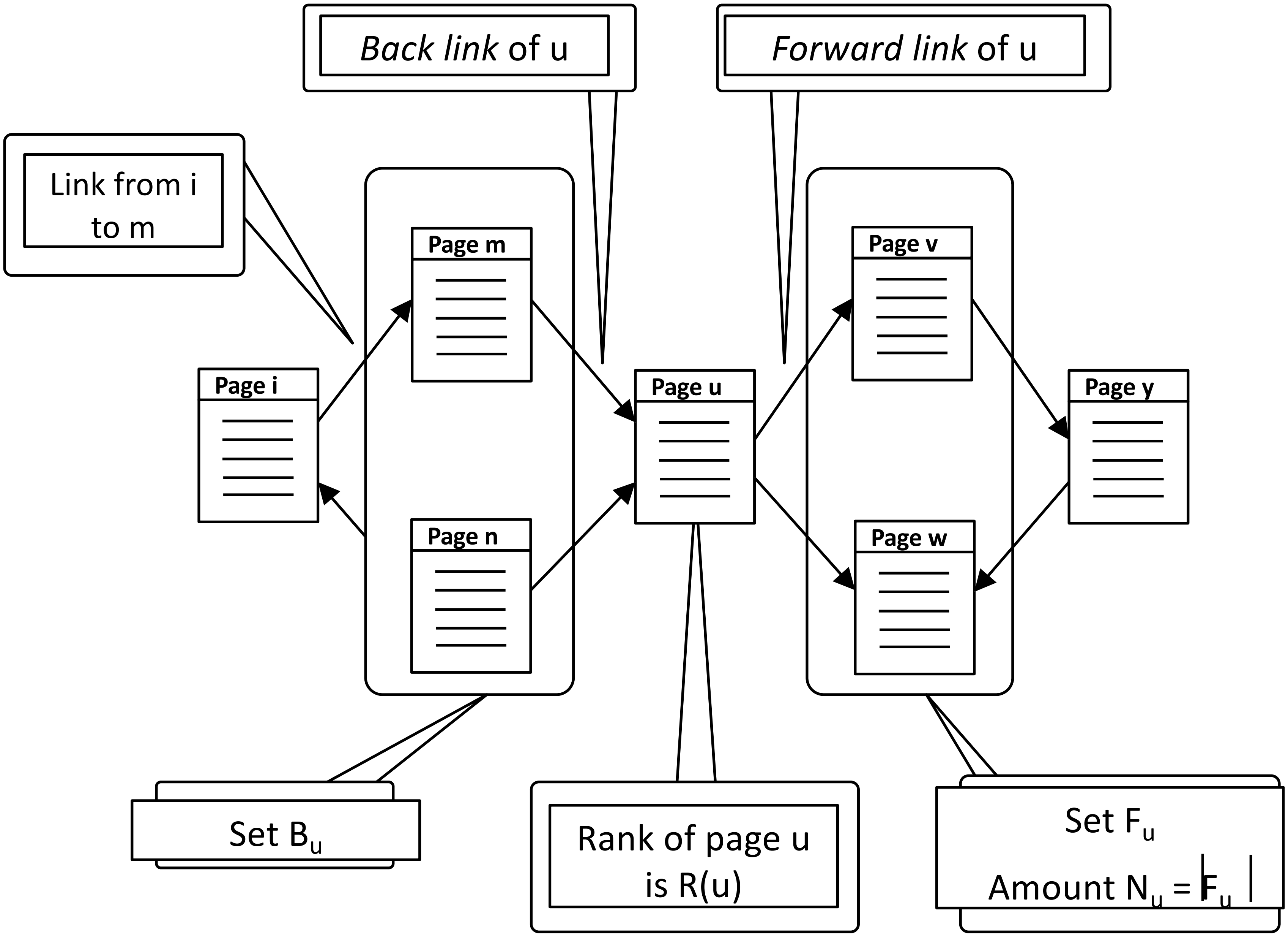
```
def walk_with_viz(field, drunk_id, step_fn, num_steps, pause=0.0001, axis_max=30, clr='black'):
    """Same as walk(), but draws the path live."""
    Lx, Ly = [], []
    start = get_loc(field, drunk_id)

    plt.xlim(-axis_max, axis_max)
    plt.ylim(-axis_max, axis_max)
    plt.plot(0, 0, 'o', color=clr, markersize=3)

    Lx.append(get_x(start))
    Ly.append(get_y(start))

    for _ in range(num_steps):
        move_drunk(field, drunk_id, step_fn)
        loc = get_loc(field, drunk_id)
        Lx.append(get_x(loc))
        Ly.append(get_y(loc))
        plt.plot(Lx, Ly, linewidth=1, color=clr)
        plt.draw()
        plt.show(block=False)
        plt.pause(pause)

    return dist(start, get_loc(field, drunk_id))
```

Simulating Multiple Walks

```python
def sim_walks(num_steps, num_trials, step_fn):
    """Runs num_trials walks of num_steps; returns list of final distances."""
    origin = make_location(0, 0)
    distances = []
    for _ in range(num_trials):
        f = make_field()
        add_drunk(f, "Homer", origin)
        distances.append(round(walk(f, "Homer", step_fn, num_steps), 1))
    return distances

def drunk_test(walk_lengths, num_trials, step_fn, step_name="step_fn"):
    """For each length, run sim_walks and print summary stats."""
    for num_steps in walk_lengths:
        distances = sim_walks(num_steps, num_trials, step_fn)
        print(step_name, "random walk of", num_steps, "steps")
        print(" Mean =", round(sum(distances) / len(distances), 4))
        print(" Max =", max(distances), "Min =", min(distances))
```

Example Random Walk Applications

The Web as Directed Graph



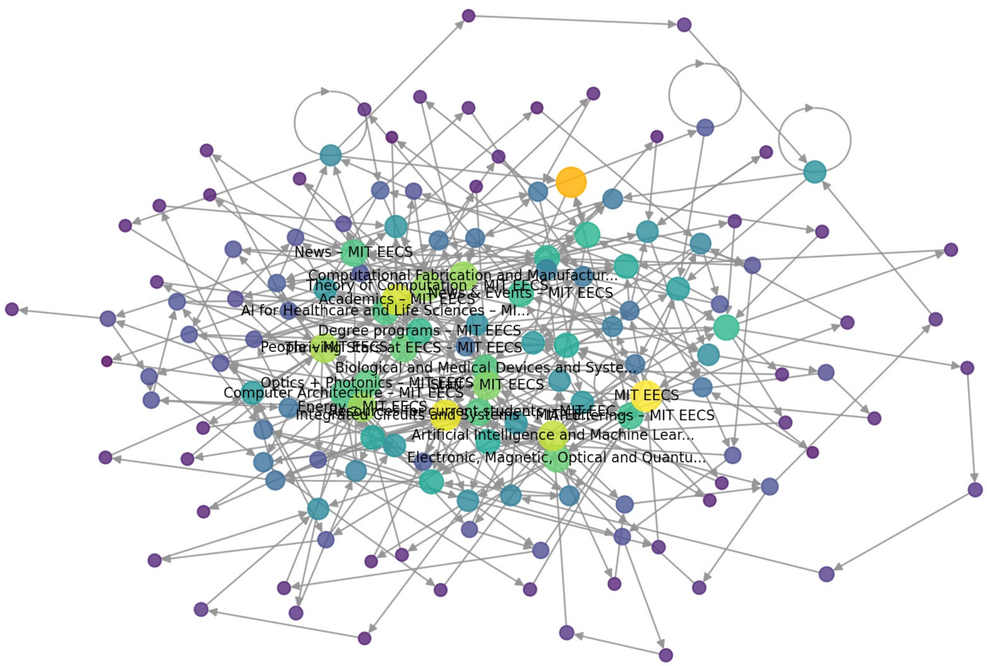
Demo 2: Graphs + Random Walk

- Page rank (Google) idea:
  - Each web page “votes” for the “importance” of pages it links to
  - The weight of a vote depends on the importance of the page itself

- So… “
to know the importance you must know the importance

Graphs + Random Walk = two trillion dollars?

- Idea 2 : random walk on graph of web pages
  - Slowly “accumulates” importance
  - More “important” pages visited more.
  - Related to friendship paradox



Demo 2: Path finding and random walk

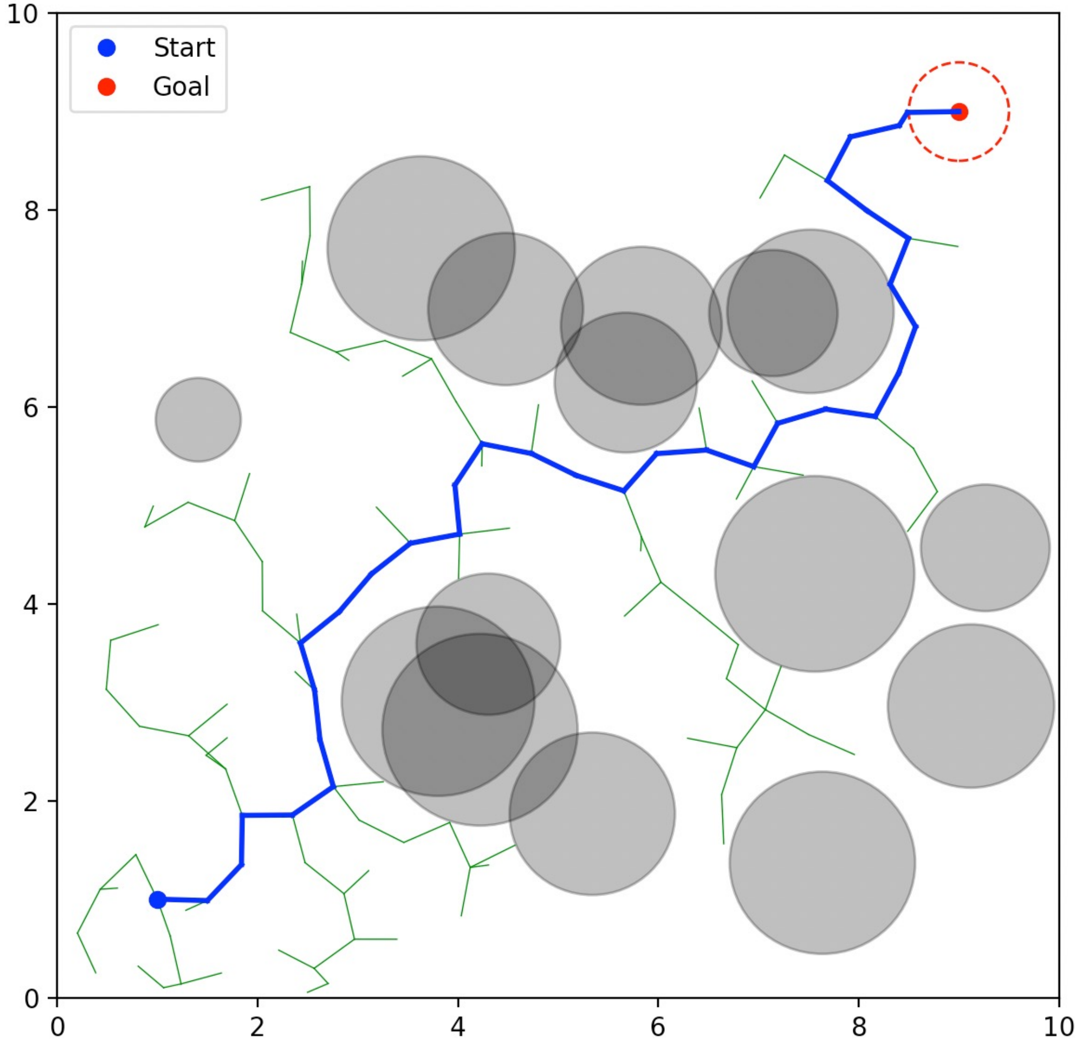
- For the case where we don't have a graph
- Continuous space + obstacles
- Combine random walk in continuous space + creation of a graph

Path finding and random walk

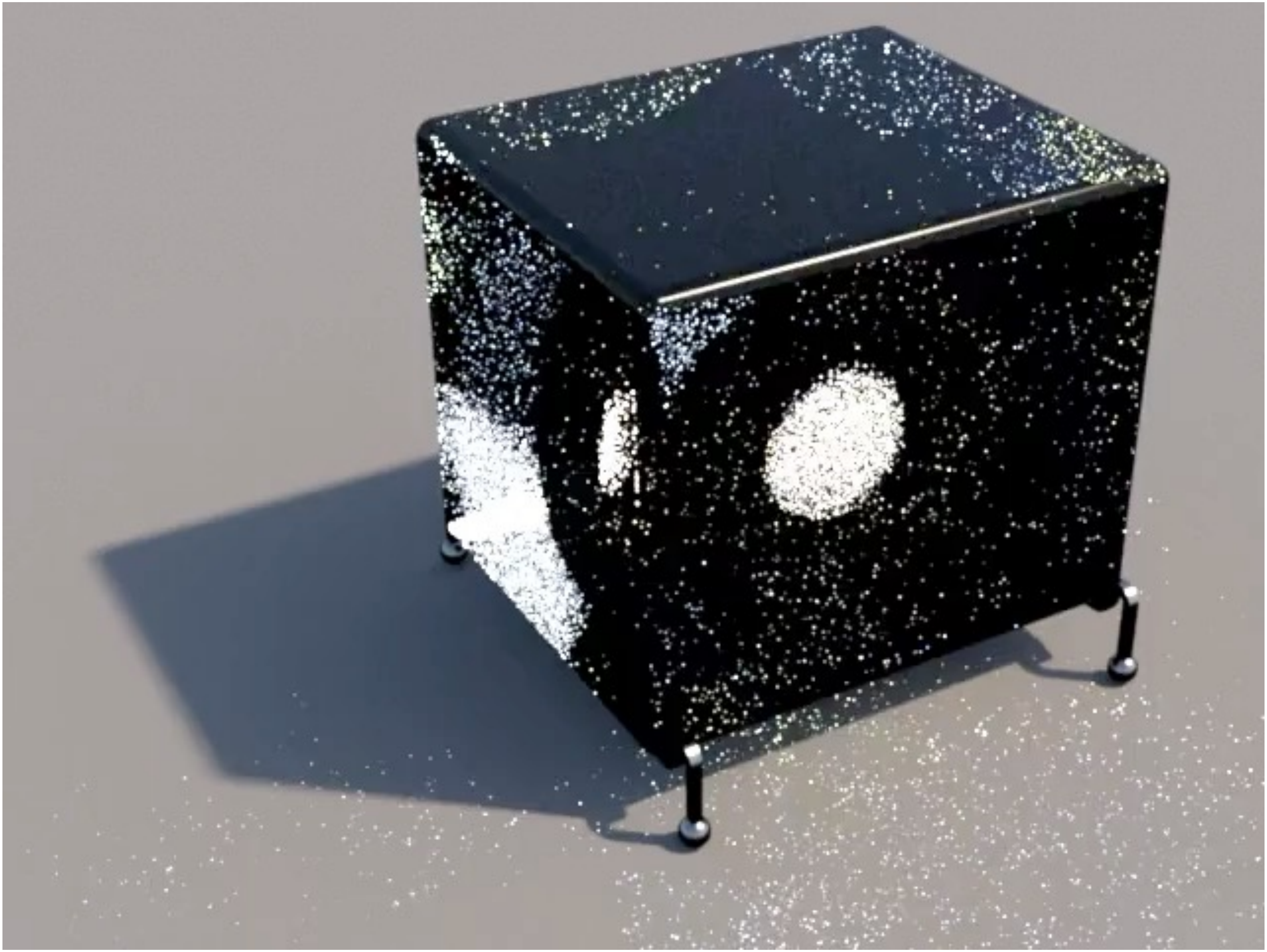
- RRT : Rapidly-exploring Random Trees
- At each step :
  - Pick random point in space
  - Find closes point in existing graph, ignoring obstacles. Grow graph in that direction.

Path finding and random walk

- RRT



Demo 3: Raytracing



Demo 3: Raytracing