Math 3610 Project 2: Big Brother

Team 1: Matthew Ekey (mpe26),
Sid Reddy (sgr45), Minhua Yan (my393)

Summary

In this manuscript, we explore a policy for drone surveillance of Gotham City. Our goal is to design an efficient patrol strategy for drones that regularly observes all geographic locations in the city, while reducing the cost of the program. We formulate the optimization problem as *minimizing the number of drones* such that *observation frequency constraints are satisfied*. A broad strategy and its variants are analyzed for different use cases, including drone unreliability, variable-priority regions in the city, and preventing program insiders from finding paths through the city that avoid surveillance.

1 Introduction

We propose a surveillance plan for Gotham to detect and deter jaywalking, using MAV with the following features:

- 5 hours of continuous flight before needing to refuel
- Programmable behavior

The plan is designed to ensure that all geographic locations are watched at least once every k minutes, where k depends on the priority of the region. We also propose extensions of the plan to satisfy the following requirements:

- Surveillance coverage does not severely degrade when drones go down for repairs, which happens randomly
- High-traffic areas are observed more frequently, and low-traffic areas are watched less frequently
- Surveillance plan designers don't have an advantage over average citizens in planning paths through the city that avoid drone observation

Our aim is to satisfy the above requirements with as few drones as possible. In this manuscript, we describe four implementations that each seek to minimize the number of drones needed to fulfill certain constraints. We will compare the plans, and analyze their advantages and disadvantages.

2 Assumptions

- Gotham is a rectangular grid, with an even width and even length
- Drones can recharge or repair at any location in the city. This might be realistic if the drones are solar-powered, and are repaired by employees who travel to the location where a drone breaks down and repair the drones on the spot.
- Drones never crash into each other
- Jaywalking only happens at intersections

3 Parameters and justifications

- s = drone speed (meters / second) : 10 m/s
- d = distance between two adjacent grid points (meters) : 250 m
- t_b = maximum time between recharging (seconds) : 18000 s
- t_r = time needed to recharge (seconds) : 500 s
- f_m = minimum observation frequency for a medium-priority street (1 / second) : $\frac{1}{900}$ / s
- f_l = minimum observation frequency for a low-priority street (1 / second) : $\frac{1}{1200}$ / s
- f_h = minimum observation frequency for a high-priority street (1 / second) : $\frac{1}{300}$ / s

s **drone speed**: According to [1], the speed of a drone sold at 849 dollars in 2015 is 35-40 km/h (9.7-11.1 m/s). Therefore, it's reasonable to assume that the type of drones NYC government can purchase in mass in 2084 has a similar speed.

d **distance between two adjacent grid points**: We simplified Gotham map as a rectangular with uniformly sized blocks based on Gotham maps. In our simplification, there are $15 \cdot 100$ blocks in Gotham. Given the length (about 22 km) and width (about 4 km) of Gotham, we take 250 meters as the distance between two adjacent grid points.

t_b **maximum time between recharging**: For efficiency, we recommend to NYC government that the drones get either gas refuel or battery change when they run out of energy, in which case 500 s would be a good approximation for the time needed.

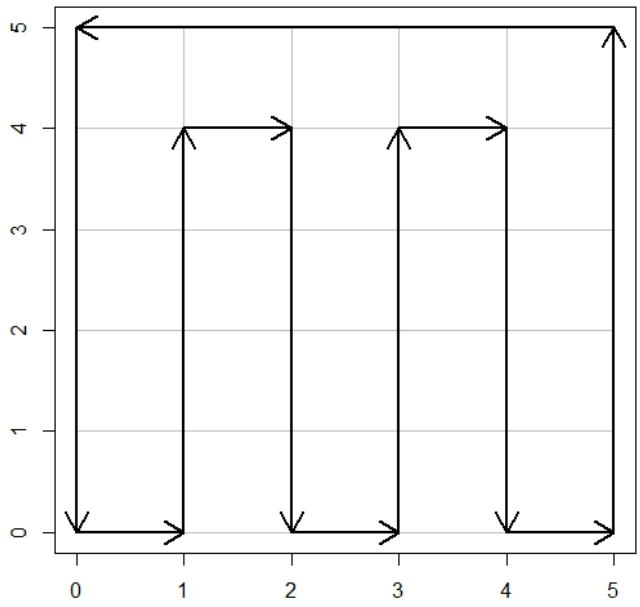

Figure 1: An example of the sweeping patrol pattern that drones execute over their assigned rectangular region

4 Plan 1

4.1 The plan

Each drone is assigned to a rectangular region that forms a subset of the grid, and multiple drones may be assigned to the same region. A drone's patrol pattern will be to follow a counter-clockwise, zigzag-like curve (see Figure 1) that allows it to continuously loop through the cells in the region while spending an equal amount of time on each cell in the long run. Assume that drones initially spawn such that the patrol path for their assigned region is covered uniformly, and that each drone is initially fully charged.

Let's assign n drones to the rectangular region covering the entire city. The total distance a drone must travel to loop through all cells exactly once is dMN, and it travels at a maximum speed of s. A drone must recharge $\frac{dMN}{st_b}$ times along the way, and each recharge takes t_r seconds. Thus, it takes a total of $\frac{dMN}{s} + \frac{dMN}{st_b} \cdot t_r = \frac{dMN(t_b+t_r)}{st_b}$ seconds for a drone to loop through the entire

city exactly once. Since n drones are distributed uniformly across the patrol path through the city, the observation frequency for any spot in the city is $\frac{n}{\left(\frac{dMN(t_b+t_r)}{st_b}\right)}$ observations per second.

Our goal is to minimize n such that $\frac{n}{\left(\frac{dMN(t_b+t_r)}{st_b}\right)} \geq f_m$, which yields the solution

$$n^* = \lceil \frac{f_m dMN(t_b + t_r)}{st_b} \rceil = 46$$

4.2 Simulation

We simulated our drone surveillance strategy and tracked the time between observations for discretized intervals along the patrol path for the entire city. The pseudocode for the algorithm is shown below.

```
step_length = drone_speed * step_duration
path_length = region_area * cell_dist / step_length
times_since_last_observation = zeros(num_steps, path_length)

drone_positions = 0:(path_length/num_drones):path_length

for t=1:num_steps
    drone_positions = (drone_positions + 1) % path_length
    times_since_last_observation[t, :] =
        times_since_last_observation[t-1, :] + step_size
    times_since_last_observation[t, drone_positions] = 0

assert (times_since_last_observation < 1 /
    minimum_observation_frequency).all()
```

Running this algorithm for `num_drones` $= n^* = 46$ results in the assert statement succeeding, while running it for `num_drones` $= n^* = 45$ or less results in the assert statement failing. Thus, for our chosen parameter values, the city needs at least 46 drones on patrol in order to prevent any location from going unobserved for longer than 15 minutes.

5 Plan 2

5.1 Charge Penalty for Turns

Assume that when a drone makes a 90-degree turn, it incurs a charge penalty that subtracts t_n seconds from the maximum time till next recharging. Given the patrol pattern described in **Plan 1** (see Figure 1), a drone that patrols

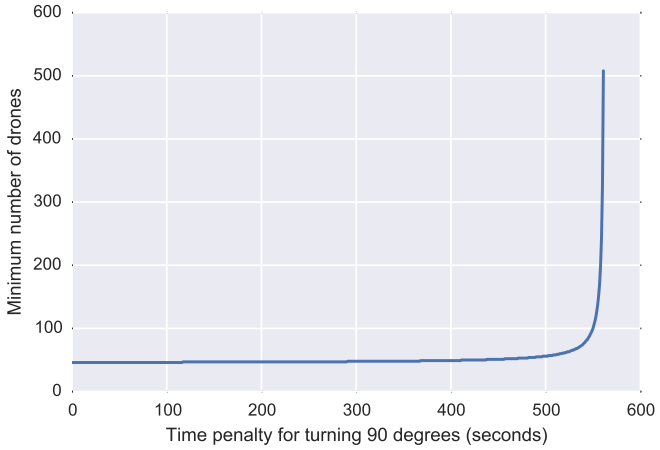

Figure 2: For reasonably small time penalty t_n, there is an insignificant increase in the minimum number of drones needed to patrol the city. From the curve, it's clear that for small t_n, the prediction for the minimum number of drones is robust to noise in the estimate of t_n. For large t_n, the prediction is extremely sensitive to noise in t_n.

a PxQ rectangular region must make $2 \cdot \min(P, Q)$ turns during one complete loop. Thus, it takes a total of $\frac{dMN(t_b - 2t_n \min(M,N) + t_r)}{s(t_b - 2t_n \min(M,N))}$ seconds to complete one loop of the entire city, which leads to a modified optimization problem with the solution

$$n^* = \lceil \frac{f_m dMN(t_b - 2t_n \min(M, N) + t_r)}{s(t_b - 2t_n \min(M, N))} \rceil$$

Note that we must select parameters such that $t_b - 2t_n \min(M, N) > 0$, otherwise our sweeping patrol pattern (see Figure 1) is not feasible.

For our parameter values, we get the curve in Figure 2.

5.2 Repairs

Assume that at any given time, a drone is currently down for repairs with constant probability p_r. We modify our patrol strategy to use a randomized initialization of drones along the patrol path (instead of the deterministic, uniform initialization described in **Part 1**), and give probabilistic guarantees on satisfying observation frequency constraints (i.e., give upper bounds on the probability that at any given time, a location has been unobserved for longer than k minutes). Instead of deterministically placing drones throughout the patrol

path for a PxQ region such that the drones are evenly spaced, we sample a drone's position uniformly at random from the interval $[0, dPQ]$. The key idea here is that randomization makes it easier to analyze the effects of drones randomly going down for repairs. Note that in the long run, if drones randomly go down for repairs and immediately start patrolling once they're repaired, then a uniformly-spaced drone stream will eventually turn into a random distribution of drones.

We modify the simulation in **Part 1** to account for randomized drone positioning and random repairs. The pseudocode is shown below.

```
step_length = drone_speed * step_duration
path_length = region_area * cell_dist / step_length
times_since_last_observation = zeros(num_steps, path_length)

drone_positions = random.choice(1:path_length, size=num_drones
    , replace=True)

for t=1:num_steps
    active = random(num_drones) > repair_probability
    drone_positions = (drone_positions + active) % path_length
    times_since_last_observation[t, :] =
        times_since_last_observation[t-1, :] + step_size
    times_since_last_observation[t, drone_positions[active]] =
         0

assert (times_since_last_observation < 1 /
    minimum_observation_frequency).all()
```

Running this algorithm with our choice of parameter values produces the output shown in Figure 3.

6 Plan 3

The key idea here is that we can efficiently handle regions of varying frequency constraints by assigning drones to different regions (instead of assigning all drones to cover the entire city, which we did in **Plan 1** and **Plan 2**).

For the city of Gotham, we use a grid with dimensions $M = 16$ and $N = 100$ (see Figure 4). Special priority regions are as follows.

- R_p (Gotham Central Park) spans from $(6, 58)$ to $(9, 79)$, and has area $A_p = 4 \cdot 22 = 88$
- R_u (Gotham University) spans from $(6, 20)$ to $(9, 23)$, and has area $A_u = 4 \cdot 4 = 16$

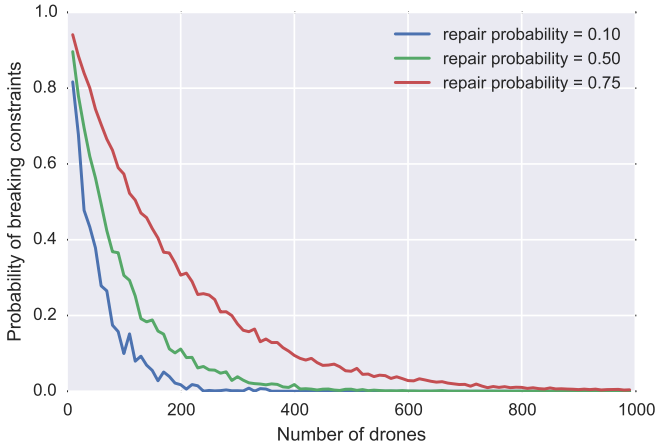

Figure 3: The probability p_r of a drone being in repair at any given time affects the degree to which increasing the number of drones reduces the probability of not satisfying the observation frequency constraints. For relatively large upper bound δ on the probability of breaking the constraints, the prediction of the minimum number of drones is robust to perturbations of δ. For small upper bound δ on the probability of breaking constraints, the prediction of the minimum number of drones is more sensitive to perturbations in δ.

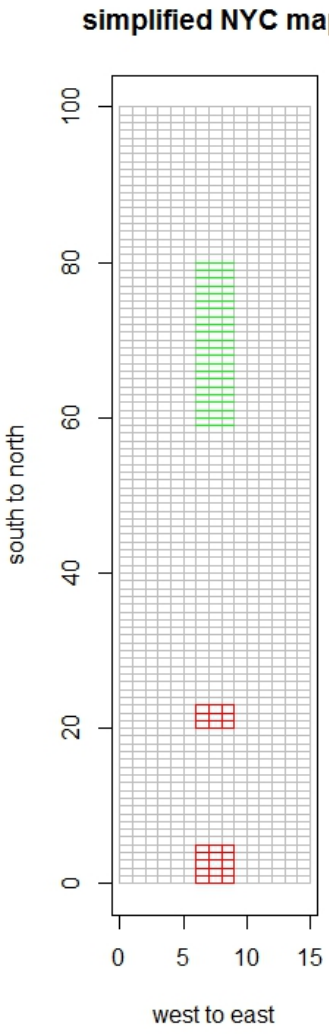

Figure 4: A downsampled version of our simplified grid map of Gotham. Low traffic region (Gotham Central Park) is highlighted in green, High traffic regions (Gotham University and Financial District) are highlighted in red.

- R_f (Financial District) spans from $(6, 0)$ to $(9, 5)$, and has area $A_f = 4 \cdot 6 = 24$

Our medium-priority regions cover any area not covered by special priority regions.

- R_1 spans from $(0, 0)$ to $(5, 99)$, and has $A_1 = 6 \cdot 100 = 600$ intersections
- R_2 spans from $(10, 0)$ to $(15, 99)$, and has $A_2 = 6 \cdot 100 = 600$ intersections
- R_3 spans from $(6, 6)$ to $(9, 19)$, and has $A_3 = 4 \cdot 14 = 56$ intersections
- R_4 spans from $(6, 24)$ to $(9, 57)$, and has $A_4 = 4 \cdot 34 = 136$ intersections
- R_5 spans from $(6, 80)$ to $(9, 99)$, and has $A_5 = 4 \cdot 20 = 80$ intersections

Now, we can decompose the optimization problem into subproblems for each region, then aggregate all minimum drone numbers to get the minimum number of drones needed to satisfy the observation frequency constraints for all regions.

Using the result from **Plan 1**, the minimum number of drones for a region with area A and priority p is $n(A, p) = \lceil \frac{f_p d A (t_b + t_r)}{s t_b} \rceil$. Thus, the minimum number of drones for all regions is

$$n(A_p, l) + n(A_u, h) + n(A_f, h) + \sum_{i=1}^{5} n(A_i, m) = 52$$

6.1 Simulation

We run the same simulation used in **Plan 1** for each region, and find that $n(A_\cdot, \cdot)$ is indeed the minimum number of drones needed to satisfy the observation frequency constraints for a given region, and thus 52 drones are necessary to satisfy the constraints for all regions. See **Plan 3** in **Appendix A** for details.

7 Plan 4

Our strategy in **Plan 1** involves placing drones at evenly spaced intervals on a patrol path going through the entire city, so if we manage to enter the city without immediately getting caught by a drone, we can simply follow what we know to be the patrol path, and as long as we move at the same speed s as the drones, we will completely avoid detection.

One approach to making it harder to avoid detection is to decide if a given drone should travel *clockwise* or *counter-clockwise* uniformly at random (i.e., by flipping a coin). The direction that a drone travels does not matter if we use the randomized drone initialization strategy described in **Repairs**, since adding or

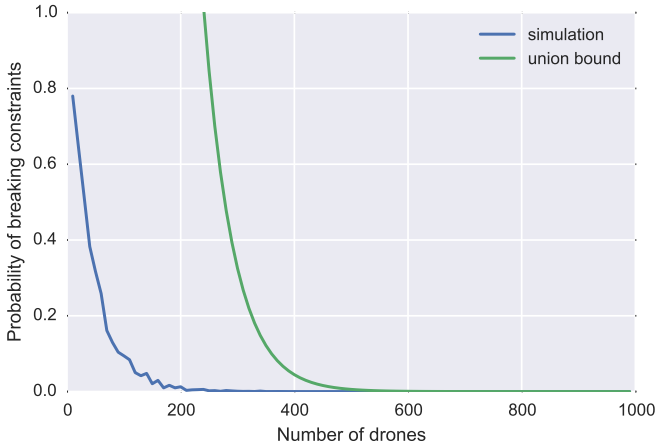

Figure 5: The simulations provide a much stronger guarantee than the bound we derived analytically. For relatively large upper bound δ on the probability of breaking the constraints, the prediction of the minimum number of drones is robust to perturbations of δ. For small upper bound δ on the probability of breaking constraints, the prediction of the minimum number of drones is more sensitive to perturbations in δ.

subtracting a constant from a uniform random variable yields a uniform random variable.

Ignoring repairs, we can bound the probability that the observation frequency constraints are broken given the number of drones n. The constraints are broken if a gap larger than $\frac{s}{f_p}$ meters exists between any two consecutive drones along the patrol path of length dMN meters. Let X_i represent the gap between drone i and drone $(i+1) \pmod n$. Since the drone positions are uniform random variables on the interval $[0, dMN]$, we have the following.

$$\mathbb{P}\left[X_i > \frac{s}{f_p}\right] \leq \left(\frac{dMN - \frac{s}{f_p}}{dMN}\right)^n = \left(1 - \frac{s}{f_p dMN}\right)^n$$

Applying the union bound,

$$\mathbb{P}\left[\bigcup_{i=1}^{n} X_i > \frac{s}{f_p}\right] \leq \sum_{i=1}^{n} \mathbb{P}\left[X_i > \frac{s}{f_p}\right] = n\left(1 - \frac{s}{f_p dMN}\right)^n$$

To verify this bound and perhaps provide a stronger guarantee, we modified the simulation in **Repairs** to ignore repair status and to decide on a drone's patrol orientation by flipping a coin. The pseudocode is shown below.

```
step_length = drone_speed * step_duration
path_length = region_area * cell_dist / step_length
times_since_last_observation = zeros(num_steps, path_length)

drone_positions = random.choice(1:path_length, size=num_drones
    , replace=True)

drone_orientations = 2 * (random(num_drones) < 0.5) - 1

for t=1:num_steps
    drone_positions = (drone_positions + drone_orientations) %
         path_length
    times_since_last_observation[t, :] =
        times_since_last_observation[t-1, :] + step_size
    times_since_last_observation[t, drone_positions] = 0

assert (times_since_last_observation < 1 /
    minimum_observation_frequency).all()
```

8 Strengths and Weaknesses

The strengths of our model are as follows.

- The patrol strategy of following a closed loop makes it convenient to give analytical expressions for the minimum number of drones needed to guarantee that observation frequency constraints are satisfied
- The actual number of drones recommended by our system is relatively small (46 in **Part 1**, 52 in **Part 3**, and < 300 for reasonable parameter choices in **Part 2** and **Part 4**) compared to the upper bound of 1600 (i.e., placing a drone at each grid cell and having them stay in one place).

The weaknesses of our model are as follows.

- The zigzag-like curve in Figure 1 may not be optimal, in the sense that it may not minimize the number of turns necessary to patrol a PxQ region in a closed loop
- Deploying drones in a closed loop through a rectangular region was done for the sake of convenient analysis. It's possible that other strategies (e.g., concentric squares, spirals, random walk) reduce the number of drones needed to satisfy the observation frequency constraints, and that simulations would be sufficient to demonstrate the trade-off between frequency constraint guarantees and the number of drones.

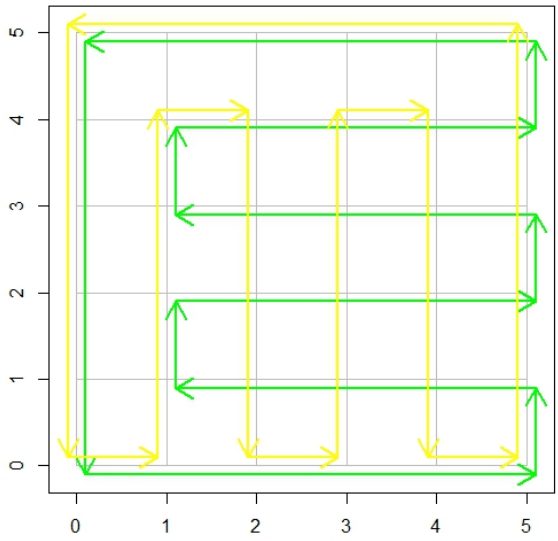

Figure 6: Two drones taking different zigzag routes supplement each other's coverage and make sure that all the segments are surveilled. We can add two other drones if we want to make sure all the segments are covered with equal strength.

9 Future Work

Jaywalking in the middle of the streets: We assumed that jaywalking only happens in intersections for simplification. However, in real life, jaywalking can happen in the middle of streets and this type of jaywalking is actually more dangerous. To cover all the street sections, we can program different drones to follow different zigzag routes such that they combined together will offer a complete coverage over all the streets (figure 3).

Recharge and repair right at the point: We also assumed that a drone can get recharged right at the point where it runs out of energy. This simplified our programming since we didn't need to worry about getting drones to a recharge station before gas/battery runs out and the only trouble the need of recharge brought was drones out of the stream for 500s. However, it's unlikely that there are recharge stations all over the city. For future work, we can reprogram the plans under the condition of recharge station distribution in Gotham. We can

also try to identify the optimal plan for building recharge stations.

10 Conclusions

Given the parameters values: drone speed 10m/s, distance between two adjacent intersections 250m, defaults lasting time 18000s, recharge time 500s, time penalty for a 90-degree turn 10s, we need at least 46 drones to make sure that each intersection in Gotham is watched at least once in 15 minutes. Taking time penalty for sharp turns into consideration doesn't significantly change the minimum number of drones needed as long as the penalty is not greater than 500s. However, malfunction has a big influence on the quality of surveillance and would cause the number of drones needed to meet the requirement increase greatly. With the probability that one random drone is broken at a certain time being 0.1, we need 200 drones to make sure the probability of failing to watch each intersection once in 15 minutes is virtually 0, while a malfunction probability of 0.5 requires 400 drones and a malfunction probability of 0.75 requires more than 800, to achieve the same. Dividing Gotham into subsections according to traffic density (Central Park with low density only needs to be watched once in 20 minutes, while NYU and the financial district needs to be watched once every 5 minutes), we found that a minimum of 52 drones are needed. To reduce the planners' advantage in jaywalking, we switched from a deterministic plan to one where we put drones randomly uniformly on the route and also have their directions randomly determined. In this way, the planners won't be able to predict when or from which direction the next drone will come.

11 References

[1] `http://www.helipal.com/multi-rotors_speed-racing-drone-rtf.html`

12 Individual Contributions

Sid: The design of the overall patrol strategy, as well as the analysis and simulations of Plans 1-4 (sections **Plan 1**, **Plan 2**, **Plan 3**, **Plan 4**, **Appendix A**). Wrote the **Summary** and **Strengths and Weaknesses**. Made small edits to the rest.

Minhua: Wrote the **Introduction**, **Assumptions**, **Parameters and justifications**, **Future Work**, and **Conclusions**, and made occasional edits to the rest. Did online research and identified the parameters, made figures for the simplified map and routes. Also participated in the discussion of plans.

Matthew: Helped with initial algorithm development. Did online research for parameter justifications and helped outline model assumptions.

13 Appendix A: Source Code

The source code below was exported from an IPython notebook that we have uploaded to Github here: `https://github.com/rddy/math3610-p2/blob/master/bigbrother.ipynb`

```
# coding: utf-8
```

```
# In[1]:
```

```
from __future__ import division
import math
```

```
from matplotlib import pyplot as plt
import seaborn as sns
import numpy as np
```

```
get_ipython().magic(u'matplotlib inline')
```

```
# In[2]:
```

```
import matplotlib as mpl
mpl.rc('savefig', dpi=300)
```

```
# In[3]:
```

```
city_width = 16 # blocks
city_length = 100 # blocks
city_area = city_width * city_length # blocks^2
drone_speed = 10 # meters / second
cell_dist = 250 # meters
time_between_recharging = 18000 # seconds
charging_time = 500 # seconds
medium_priority_freq = 1 / 900 # 1 / second
high_priority_freq = 1 / 300 # 1 / second
low_priority_freq = 1 / 1200 # 1 / second
repair_probability = 0.001
time_penalty_for_turn = 500 # seconds
```

```
# ##Part 1
```

```
# In[4]:

n_min = lambda freq, region_area: math.ceil(freq * cell_dist *
    region_area * (time_between_recharging + charging_time) /
    (drone_speed * time_between_recharging))
print n_min(medium_priority_freq, city_area)


# In[5]:

step_size = 1 # seconds
num_steps = 60 * 60 * 6 # 6 hours


# In[6]:

def simulate_surveillance(num_drones, region_area,
    repair_probability=0, uniform_init=True, rand_dirs=False):
    step_length = drone_speed * step_size
    path_length = int(math.ceil(region_area * cell_dist /
        step_length))
    times_since_last_observation = np.zeros((num_steps,
        path_length))

    if uniform_init:
        drone_positions = np.arange(0, path_length,
            path_length / num_drones, dtype=int)
    else:
        drone_positions = np.random.choice(range(path_length),
            size=num_drones, replace=True)

    if rand_dirs:
        drone_dirs = 2 * (np.random.random(num_drones) < 0.5)
            - 1
    else:
        drone_dirs = np.ones(num_drones)

    for t in xrange(num_steps):
        not_in_repair = np.random.random(num_drones) >=
            repair_probability
        drone_positions = (drone_positions + not_in_repair *
            drone_dirs) % path_length
        times_since_last_observation[t, :] =
            times_since_last_observation[t-1, :] + step_size
        times_since_last_observation[t, drone_positions[
            not_in_repair].astype(int)] = 0

    return times_since_last_observation
```

```
# In[7]:

times_since_last_observation = simulate_surveillance(n_min(
    medium_priority_freq, city_area), city_area)
assert (times_since_last_observation < 1 /
    medium_priority_freq).all()
print "Tests passed"


# In[8]:

step_size = 10 # seconds


# ##Part 2

# Charge time penalty for making turns

# In[9]:

assert time_between_recharging - 2 * time_penalty_for_turn *
    min(city_width, city_length) > 0


# In[10]:

n_min_with_charge_penalty = lambda freq, region_width,
    region_length, time_penalty_for_turn: math.ceil(freq *
    cell_dist * region_width * region_length * (
    time_between_recharging - 2 * time_penalty_for_turn * min(
    region_width, region_length) + charging_time) / (
    drone_speed * (time_between_recharging - 2 *
    time_penalty_for_turn * min(region_width, region_length)))
    )
print n_min_with_charge_penalty(medium_priority_freq,
    city_width, city_length, time_penalty_for_turn)


# In[11]:

xs = range(time_between_recharging // (2 * min(city_width,
    city_length)))
ys = [n_min_with_charge_penalty(medium_priority_freq,
    city_width, city_length, x) for x in xs]


# In[12]:

plt.xlabel('Time penalty for turning 90 degrees (seconds)')
plt.ylabel('Minimum number of drones')
```

```
plt.plot(xs, ys)
plt.savefig('part2a.pdf', dpi=300)
plt.show()


# Consider repairs

# In[13]:

def prob_freq_constraint_broken(num_drones, repair_probability
    ):
    times_since_last_observation = simulate_surveillance(
        num_drones, city_area, repair_probability=
        repair_probability, uniform_init=False)
    return np.mean(times_since_last_observation > 1 /
        medium_priority_freq)


# In[14]:

xs = np.arange(10, 1000, 10)
repair_probabilities = [0.1, 0.5, 0.75]
ys = [[prob_freq_constraint_broken(x, rp) for x in xs] for rp
    in repair_probabilities]


# In[15]:

plt.xlabel('Number of drones')
plt.ylabel('Probability of breaking constraints')
for rp, ys_of_rp in zip(repair_probabilities, ys):
    plt.plot(xs, ys_of_rp, label='repair probability = %0.2f'
        % rp)
plt.legend(loc='best')
plt.savefig('part2b.pdf', dpi=300)
plt.show()


## Part 3

# In[16]:

low_priority_areas = [88] # blocks^2
medium_priority_areas = [56, 80, 136, 600, 600] # blocks^2
high_priority_areas = [16, 24] # blocks^2


# In[17]:

areas = low_priority_areas + medium_priority_areas +
```

```
    high_priority_areas
priority_freqs = ([low_priority_freq] * len(low_priority_areas
    )) + ([medium_priority_freq] * len(medium_priority_areas))
     + ([high_priority_freq] * len(high_priority_areas))


# In[18]:

print sum(n_min(freq, area) for area, freq in zip(areas,
    priority_freqs))


# In[19]:

assert all((simulate_surveillance(n_min(freq, area), area) < 1
     / freq).all() for area, freq in zip(areas, priority_freqs
    ))
print "Tests passed"


# ##Part 4

# In[20]:

def prob_freq_constraint_broken(num_drones):
    times_since_last_observation = simulate_surveillance(
        num_drones, city_area, uniform_init=False, rand_dirs=
        True)
    return np.mean(times_since_last_observation > 1 /
        medium_priority_freq)


# In[21]:

xs = np.arange(10, 1000, 10)
ys = [prob_freq_constraint_broken(x) for x in xs]


# In[22]:

plt.xlabel('Number of drones')
plt.ylabel('Probability of breaking constraints')
plt.plot(xs, ys, label='simulation')
plt.plot(xs, xs*(1 - drone_speed / (medium_priority_freq *
    cell_dist * city_area))**xs, label='union bound')
plt.ylim([0, 1])
plt.legend(loc='best')
plt.savefig('part4.pdf', dpi=300)
plt.show()
```

 # In [ ]: