Optisim

Stochastic Optimization of a Nomadic Trucker

Jason Andri Gíslason

University of Iceland
January 11, 2022

Contents

1 Introduction

Generally people, companies and governments want to transport goods between places. There is an entire industry devoted to facilitating transport of goods between locations. So for instance if you are the manager of a trucking company, you might have a fleet of drivers over some operating area, and will transport goods across that operating area. But you have a big problem in front of you as there are an abundance of jobs that each driver can take. The problem complexity arises from the fact that you do not know what jobs will be available in the destination city where the driver will be when they will have completed the prior job. This problem is something that has been solved in the trucking industry.

The purpose of this project was not to do something new or original but rather to solve some stochastic optimization problem using methods from the book *Reinforcement Learning and Stochastic Optimization* by Warren B. Powell. We will need to create the underlying data from scratch and then construct a mathematical model of the system and then use methods from stochastic optimization to solve it.

2 Definitions

We'll start off by defining the notation that will be used throughout this report. Let the set of cities be $\mathcal{I}$ with $i, j \in \mathcal{I}$. The set of regions, $\mathcal{R}$ partitions $\mathcal{I}$ and we let $\varphi : \mathcal{I} \to \mathcal{R}$ be a mapping from cities to the region where the city is located. Let $\text{dist}(i, j)$ be the distance between the cities i and j. Time is indexed by $t \in \mathcal{T} = \hat{T} \cup T$, where $\hat{T} = \{t_0 - \hat{t} + 1, \ldots, t_0 - 1\}$ is past time and $T = \{t_0, t_0 + 1, \ldots, t_0 + \tau\}$ is present and future time, $\hat{t}$ is how many days in the past we have data and τ is the time horizon for the optimization.

3 Parameter Estimation with Bayesian Statistics

In this chapter we are going to generate the data that will be used as a stand in for the real world data that is unavailable to the public. Then we will use bayesian statistics to estimate the distribution of these prices which we will then use in later chapters to optimize the problem. The work here was done in conjunction with the final project of a Bayesian Statistics Course that I took while working on this project.

3.1 The Data

To start off with we will generate the data and take a quick look at it to make sure it looks good.

3.1.1 Generation

To generate the data we will first generate demand between the directed pairs of cities (i, j). For each city sample

$$\begin{aligned} U_{i_{\text{in}}} &\sim \text{Unf}(0.1, 0.9) \\ U_{i_{\text{out}}} &\sim \text{Unf}(0.1, 0.9) \end{aligned}$$

Now do the same for each region

$$\begin{aligned} U_{r_{\text{in}}} &\sim \text{Unf}(0.25, 0.75) \\ U_{r_{\text{out}}} &\sim \text{Unf}(0.25, 0.75) \end{aligned}$$

Now let

$$D_{ij} = \frac{D_{\text{avg}} U_{i_{\text{out}}} U_{j_{\text{in}}} U_{\varphi(i)_{\text{out}}} U_{\varphi(j)_{\text{in}}}}{\mathbb{E}(U_{j_{\text{in}}}) \mathbb{E}(U_{i_{\text{out}}}) \mathbb{E}(U_{\varphi(i)_{\text{out}}}) \mathbb{E}(U_{\varphi(j)_{\text{in}}})}$$

where D_{avg} is some average demand. The denominator is there to balance the expectation of the uniform random variables. This is done to keep the magnitude of D_{avg} similar to D_{ij}, here we let $D_{\text{avg}} = 40$. The demand will be time invariant for each simulation.

Now for each pair of cities ij and time t (how far back we want historical data) we sample

$$n_{ijt} \sim \text{Poi}(D_{ij})$$

and let it be the amount of jobs from i to j at time t. Now to sample to prices of loads we sample for each ijt, n_{ijt} times from

$$1000 \cdot \text{Gamma}(D_{ij}, D_{ij}^{-1})$$

and now we have our prices for the loads at ijt.

3.1.2 A look at the data

For this problem we will let $\mathcal{I} = \{1, 2, \ldots, mn\}$ and $\mathcal{R} = \{1, 2, \ldots, m\}$ where n is the number of cities in each region. We partition the regions such that $\varphi(i) = \lfloor i/n \rfloor$. Doing this allows us to scale the data as we want, adding more regions or cities as we please. If we would look at a 'world' like the United States we would have around 10 regions and 20-100 cities in a region. Then we can choose how many days into the past we want to go for the historical data, we would want something like 100-300 days. Showing that kind of data is impossible due to its high dimentionality. But here we will show the data that we will be working on for the project, just to save time for computation.

We will let $n = 4$ and $m = 3$. We want a few dozen days to be able to estimate n effectively so we take $\hat{t} = 20$.

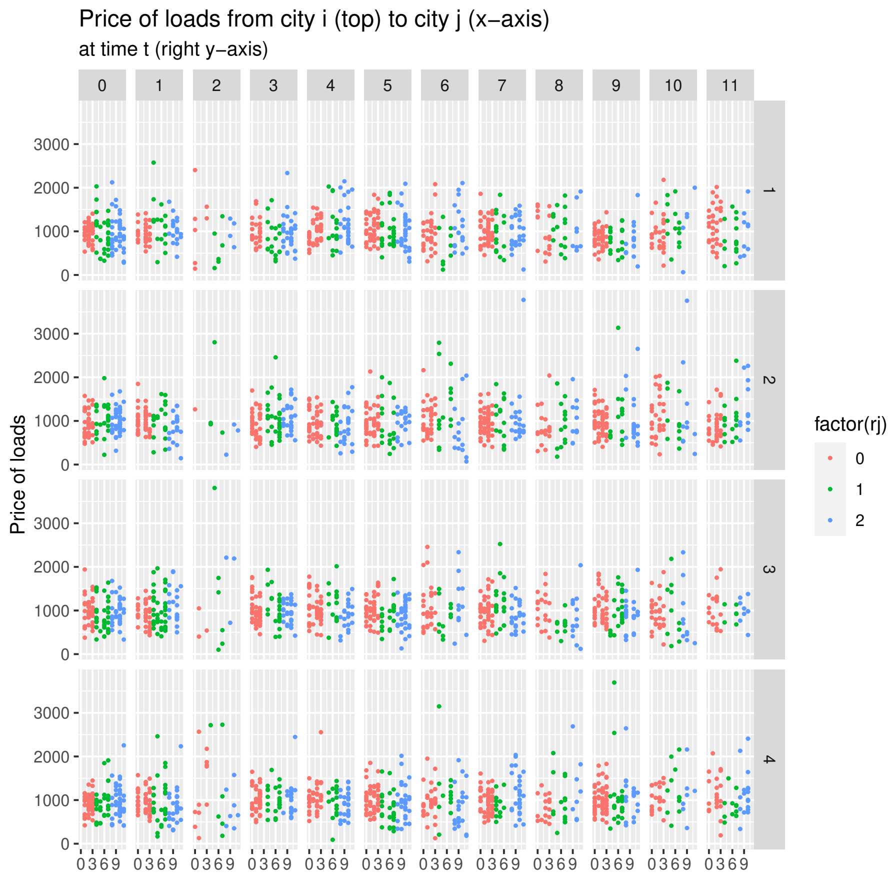

Figure 1: Plot of price of loads, we only plot the first 4 timesteps to avoid cluttering

3.2 The Model

Here we will create the statistical model that we will use to estimate the distribution of prices.

3.2.1 Defining the model

Now we will create our model. We do not have access to the demand parameter. The only data we have access to are the prices of loads

$$y_{ijtk} = \text{the } k\text{-th price from city } i \text{ to city } j \text{ at time } t$$

from the y's we can count the n_{ijt}'s. The model we want to propose is twofold. We need estimations of the prices and we would also like estimations of the n's to be able to estimate how many jobs are available.

So we assume that y follows a gamma distribution $p(y_{ijtk}|\alpha_{ij}, \beta_{ij}) = \text{Gamma}(\alpha_{ij}, \beta_{ij})$. The gamma distribution has an improper prior distribution

$$p(\alpha_{ij}, \beta_{ij}|p, q, r, s) \propto \frac{p^{\alpha_{ij}-1} e^{\beta_{ij} q} \beta_{ij}^{\alpha_{ij} s}}{\Gamma(\alpha_{ij})^r}$$

To make the posterior distribution a little cleaner we will not show the ijt index of α, β, n and y. Then the posterior distribution of y is

$$p(\alpha, \beta|y) \propto \frac{\beta^{\alpha(s+n)} p^{\alpha-1} \prod_{k=1}^{n} y_k^{\alpha-1} \exp\{-\beta(q + \sum_{k=1}^{n} y_k)\}}{\Gamma(\alpha)^{r+n}}$$

We will pick the parameters as $(r, s, q, p) = (0.2, -0.01, 0.2, 1.5)$. As for estimating n we assume that n follows a poission distribution. That is $n_{ijt} \sim \mathrm{Poi}(\lambda_{ij})$, and we let the prior be a gamma distribution $p(\lambda_{ij}) = \mathrm{Gamma}(\psi, \omega)$, then the posterior of n_{ijt} will become

$$p(\lambda_{ij}|\psi, \omega) = \mathrm{Gamma}\left(\psi + \sum_{t \in \hat{T}} n_{ijt}, \omega + |\hat{T}|\right)$$

we will use $(\psi, \omega) = (9, 1)$

3.2.2 List of variables and parameters

y_{ijtk}	The k-th price from city i to city j at time t
n_{ijt}	The number of observations of y_{ij} at time t
α_{ij}	Parameter of y_{ijtk}
βij	Parameter of y_{ijtk}
λ_{ij}	Parameter of n_{ijt}

3.3 Inference

3.3.1 The substitution

The project is divided in two. First we need estimates of λ_{ij}. Now since the posterior distribution of the λ_{ij} is a known distribution it is trivial to sample from it or to get posterior intervals and means.

The complicated part is in estimating the prices. Since the posterior distribution of p_{ijtk} is known up to a constant we will need to use the metropolis algorithm to sample from the distribution. There is a slight problem in which our sampling distribution will need to be asymmetric since the gamma distribution is asymmetric. Therefore will use a substitution

$$\alpha_{ij}(a_{ij}) = e^{a_{ij}}$$
$$\beta_{ij}(b_{ij}) = e^{b_{ij}}$$

Then the posterior distriburion in terms of a and b becomes

$$\begin{aligned} p(a, b|y) &= p(\alpha(a), \beta(b)|y)\det(J(\alpha(a), \beta(b)) \\ &= p(e^a, e^b|y)e^{a+b} \end{aligned}$$

Using this substitution we can choose a symmetric proposal distribution in the Metropolis Hastings algorithm, namely a normal distribution with variance $\sigma_a^2 = \sigma_b^2 = 0.075^2$. It is a coincidence that the variances are equal. They were chosen such that the acceptance rate of the algorithm would be around 44% for most city pairs. Some city pairs had distributions where the acceptance rate would be around higher or lower, but this value looked to make most of the pairs have an acceptance rate of around 44%

3.3.2 Metropolis Algorithm

Now we use this algorithm with 5 chains and a burn in of 666 samples and then a final length of 4000 for all pairs of $(i, j), i \neq j$. Then we have alot of samples for a and b but we want samples of α and β so we simply perform element wise exponentiation of the elements from the substitution equation to convert to α and β. We initialize $\alpha_{ij}^0 = 6.8$ and $\beta_{ij}^0 = 1/6800$. So $a_{ij}^0 = \ln\left(\alpha_{ij}^0\right)$ and $b_{ij}^0 = \ln\left(\beta_{ij}^0\right)$.

Algorithm 1 Metropolis Algorithm for city pair $(i, j), i \neq j$ at step t

Require: $t > 0$

$a^* \sim N(a^{t-1}, \sigma_a^2)$
$r \leftarrow \ln\big(p(a^*, b^{t-1}|y)\big) - \ln\big(p(a^{t-1}, b^{t-1}|y)\big)$
$U \sim \text{Unf}(0, 1)$
if $\min(r, 0) \geq \ln(U)$ **then**
  $a^t \leftarrow a^*$
else
  $a^t \leftarrow a^{t-1}$
end if
$b^* \sim N(b^{t-1}, \sigma_b^2)$
$r \leftarrow \ln(p(a^t, b^*|y)) - \ln\big(p(a^t, b^{t-1}|y)\big)$
$U \sim \text{Unf}(0, 1)$
if $\min(r, 0) \geq \ln(U)$ **then**
  $b^t \leftarrow b^*$
else
  $b^t \leftarrow b^{t-1}$
end if

3.4 Results

3.4.1 Immediate Results

After generating 20000 samples of α's and β's for each one of the 121 different city pairs there is alot of data to go about. We could calculate a posterior interval for each of the α_{ij}'s and β_{ij}'s, but since we are interested in the price of loads which we assume come from a gamma distribution with parameters α_{ij} and β_{ij} will only look at the means of the posteriors for the α's and β's.

Since we effectively have 121 models that were computed. Displaying them all in a useful way is difficult. Therefore we will only show a handful of city pairs. First we have a comparison of the density of the data versus the density of the model.

Now we can visually inspect the model to see if it fits the data. From the four city pairs that are shown in 2 we see that two are very good fits and the other two are okay. Now those two bad fits are the worst fits from all of the pairs, at least according to the p-values.

(i, j)	95% interval for α_{ij}	95% interval for β_{ij}
$(1, 2)$	$(13.26, 17.73)$	$(1.35, 1.82)10^{-2}$
$(9, 5)$	$(2.03, 4.36)$	$(1.87, 4.29)10^{-3}$
$(10, 4)$	$(2.63, 5.26)$	$(2.58, 5.47)10^{-3}$
$(3, 7)$	$(8.06, 11.7)$	$(8.12, 11.2)10^{-3}$

We also might want to check if there is a relation between α_{ij}, β_{ij} and λ_{ij}.

There is a clear linear relationship between α and λ and β and λ.

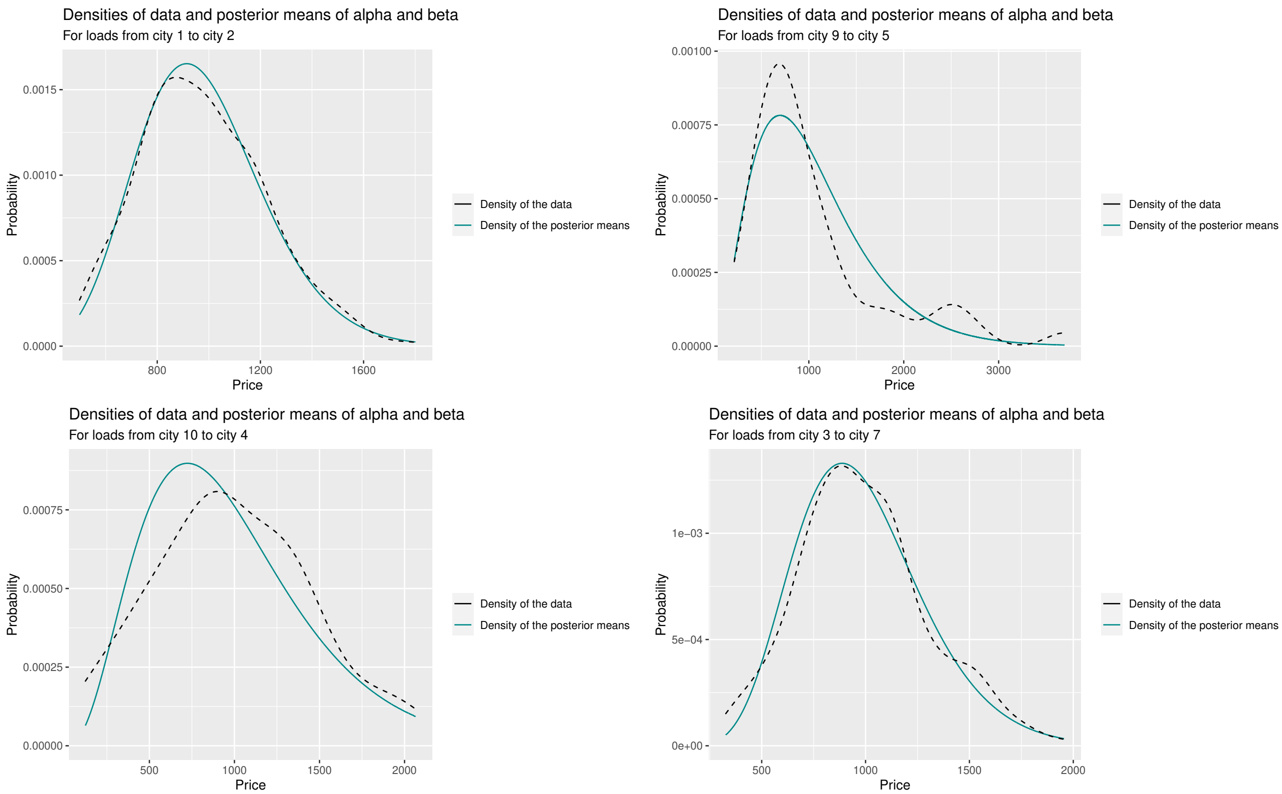

Figure 2: Densities for four pairs of cities

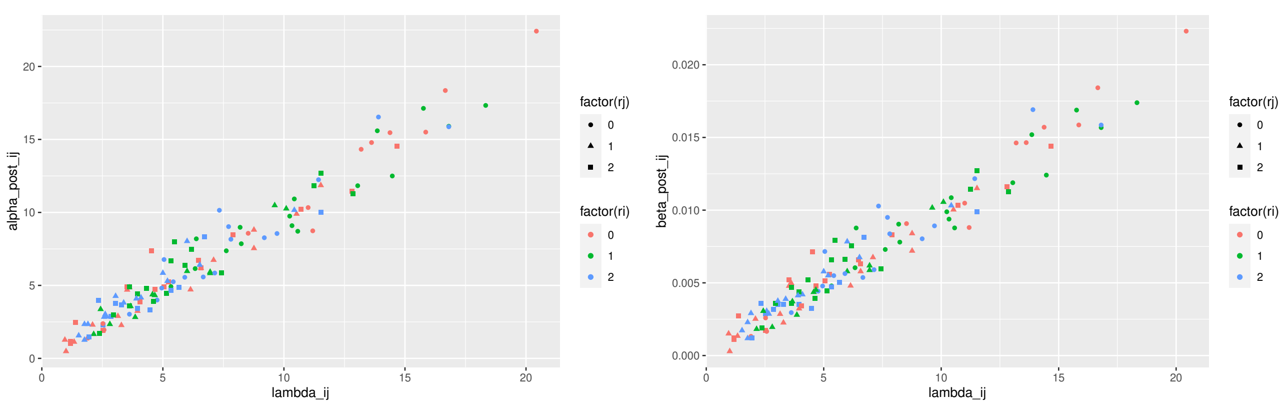

Figure 3: Scatterplots of α_{ij} and β_{ij} as a function of λ_{ij}

3.4.2 Model Validation

Now is the model accurate. We don't really know for certain. Therefore we will use bayesian p-values to get an estimate of roughly how good the model is. From figure 2 we know that the model is a good approximation for the real world. So we will use the following discrepancy measure

$$T(y_{ij}, \alpha_{ij}, \beta_{ij}) = \sum_{k=1}^{n_{ij}} \left(y_{ijk} - \frac{\alpha_{ij}}{\beta_{ij}} \right)^2$$

Then using a random subset of the samples of α_{ij} and β_{ij} from the metropolis sample we can sample y_{ij}^{rep} and compute p-values for all city pairs. Most p-values are on the interval $[0.2, 0.6]$ which is fine, but not amazing.

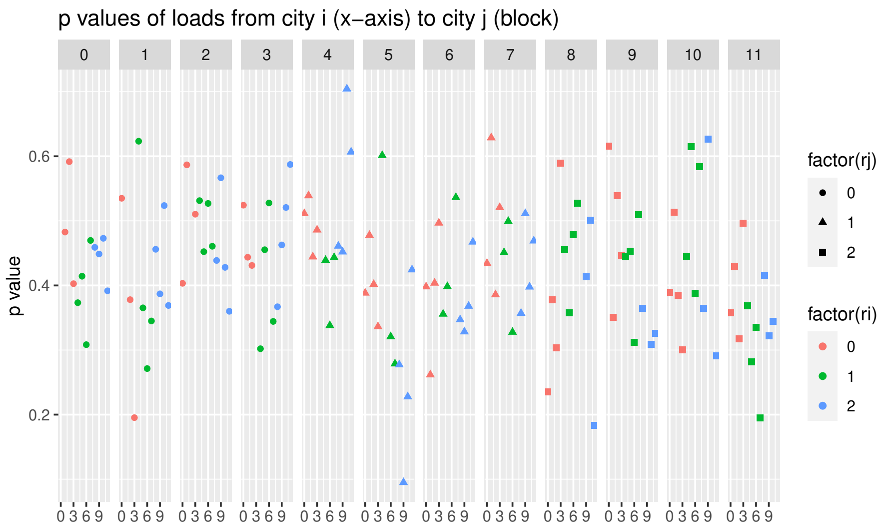

Figure 4: Plot of p-values for each city pair

There are two outliers, the densities of those pairs are plotted in figure 2.

3.5 Conclusion

3.5.1 Discussion

The model is not perfect, but it does not have to be. We only wanted a rough approximation of the prices of loads between city pairs. However there are some obvious flaws with the model.

The linearity between α and λ is expected since they are equal in the data generation. However β should not be linearly related to λ, it should be inversely proportional to λ, so it appears that the parameter sampling is dominated by α. It is not trivial to expain this, since these parameters fit the data quite well, much better than if one of the parameters is wildly wrong. This, of course, we don't really know if we are assuming that we do not know the underlying structure of the data. But from the Meta perspective it tells us there is something wrong, but that it is mostly correct.

The problem this model is trying to answer is what is the estimated price of a load from city i to city j. From this model we could calculate a 95% interval for the price. But that is not really a 95% intverval since we are uncertain that the α's and β's are exactly correct. We have intervals for them. Incorporating two sided intervals into the interval for creates many more intervals that can be chosen for the price interval. So for the purposes of simplicty we will stick to using the posterior means for α and β. This creates more error but we are fine with it.

The single biggest problem with the model is the fact that in the metropolis hastings algorithm we are using the same variance for α_{ij} and β_{ij} irrespective of which pair ij is being sampled. Ideally we would want to tune the variance so the acceptance rate is 44% for each city pair. This is another problem in and of itself.

3.5.2 Possible improvements to the model

So from the disussion above we can list some improvements to the model. The biggest improvement would be to tune the acceptance rate for all of the city pairs to be around 44%. There is a way to implement this if we let the variance of both parameters be the same. Then we can use a simple iteration to get closer to the desired acceptance rate. It becomes more nuanced when we assume that the variance of α and β should be different. Then we would need to use some kind of gradient descent.

One aspect that was considered for the model but was cut due to complexity was adding hierarchy in the model since it is sensible that there is some regional factor to the prices. And of course in the data generation there is. But we don't necessarily know that.

The other improvement would be to incorporate the posterior interval of α and β. Maybe using the samples of α_{ij} and β_{ij} and sampling a couple of prices which yeilds a sample of prices. This could be a solution.

4 Optimization

In this chapter we will create the state variable and consider a few policies and then we will implement and test them to see which one is best.

4.1 The State Variable

Since the only things that change over time are the prices of loads and the location of the truck we have

$$S_t = (R_t, I_t)$$

where

$$R_t = \text{the location of the truck at time } t$$

and I_t is the observed prices of loads at time t that is

$$I_t = (p^{\max}_{tij})_{(i,j)\in\mathcal{I}^2}$$

and we only need to update R over time since prices are observed not calculated so

$$R_{t+1}|S^x_t = x_t$$

$$I_{t+1} \text{ is observed}$$

Now there is a little more information in the inital state variable with

$$S_0 = (R_0, I_0, A, B, \Lambda)$$

where

$$A = (\alpha_{ij})_{(i,j)\in\mathcal{I}^2}, B = (\beta_{ij})_{(i,j)\in\mathcal{I}^2} \text{ and } \Lambda = (\lambda_{ij})_{(i,j)\in\mathcal{I}^2}$$

4.2 Policies

We want to solve the following problem

$$\max_{\pi} \mathbb{E}\left\{\sum_{t\in T} C(S_t, X^\pi(S_t))|S_0\right\}$$

where $C(S_t, x_t) = p^{\max}_{tR_tx_t}$. That is we want to find the best policy π. There are an infinite amount of polices out there but we will design four polices, each of which is in itself an optimization problem. But thankfully they are all relatively easy to solve.

4.2.1 Epsilon greedy policy

The first policy we'll consider is the epsilon greedy policy. It is a probabilistic policy as it chooses actions with probabilities, namely:

$$X^{EG}(S_t|\varepsilon) = \begin{cases} \text{random } i \text{ from } \mathcal{I} \text{ with probability } \varepsilon \\ \arg\max_{x_t} C(S_t, x_t) \text{ with probabilty } 1-\varepsilon \end{cases}$$

Now let's look at the performance of this policy using many values of ε. So disappointingly adding some

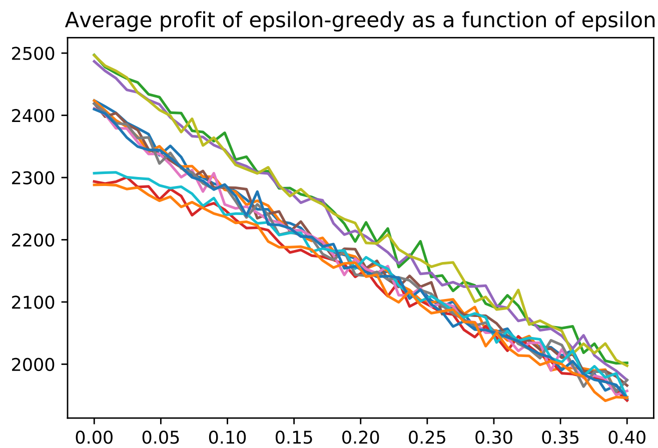

Figure 5: The colors represent different starting cities

randomness to the pure greedy policy only diminishes the performance. So for this class of policy we will furthermore use $\varepsilon = 0$.

4.2.2 Boltzman exploration policy

This policy like epsilon greedy is probabilistic and it chooses actions $i \in \mathcal{I}$ with probability

$$\mathbb{P}(X^{\text{BE}}(S_t) = x_t | S_t, \theta) = \frac{e^{\theta C(S_t, x_t)}}{\sum_{x'_t \in \mathcal{I}} e^{\theta C(S_t, x'_t)}}.$$

The performance of this policy was Interestingly the performance of the Boltzmann exploration appears to

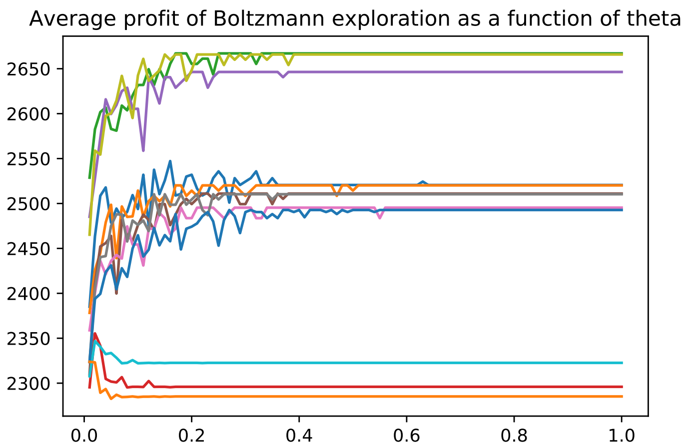

Figure 6: The colors represent different starting cities

converge to a stable value for $\theta > 0.3$.

4.2.3 Lookahead Policy

Now we will design a more standard (and more complicated) policy. Here we will use a deterministic lookahead hybrid model:

$$X^{LA-DET-HYB}(S_t) = \arg\max_{x_t \in \mathcal{I}} \left(C(S_t, x_t) + \sum_{t'=t+1}^{t+H} \overline{C}^x(\tilde{S}_{tt'}, \tilde{x}_{tt'}) \right)$$

where for $t' = t+1, \ldots, t+H$

$$\tilde{x}_{tt'} = \arg_j \max_{y^*_{tt'ij}}$$

and $y^*_{tt'ij}$ is the optimal value in the linear integer program.

Decision Variable:

$$y_{tt'ij} = \begin{cases} 1 \text{ if leg } i \to j \text{ is on the optimal path at time } t' \\ 0 \text{ otherwise} \end{cases} \quad , t' = t, \ldots, t+H, (i,j) \in \mathcal{I} \times \mathcal{I}$$

Objective Function:

$$\max\left\{C^y_{tij}y_{tt'ij}+\sum_{t'=t+1}^{t+H}\sum_{(i,j)\in\mathcal{I}\times\mathcal{I}}\overline{C}^y_{t'ij}y_{tt'ij}\right\}$$

Constraints:

$$\sum_{(i,j)\in\mathcal{I}\times\mathcal{I}} y_{tt'ij}\leq 1,\ \text{for } t'=t,\dots,t+H$$

$$\sum_{k\in\mathcal{I}} y_{tt'ki}-\sum_{j\in\mathcal{I}} y_{t(t'+1)ij}=0,\ \text{for } t'=t,\dots,t+H-1,\ \text{for } i\in\mathcal{I}$$

$$\sum_{j\in\mathcal{I}} y_{ttij}=0,\ \text{for } i\neq R_t$$

where

$$\overline{C}^y_{ij}(t,\theta)=g^\theta_{ij}=\theta\text{th percentile of Gamma}(\alpha_{ij},\beta_{ij})$$

$$\overline{C}^x(\tilde{S}_{tt'},\tilde{x}_{tt'})=g^\theta_{R_{tt'}\tilde{x}_{tt'}}$$

The performance of the lookahead was more sporadic since it is more complex to compute. There is no clear

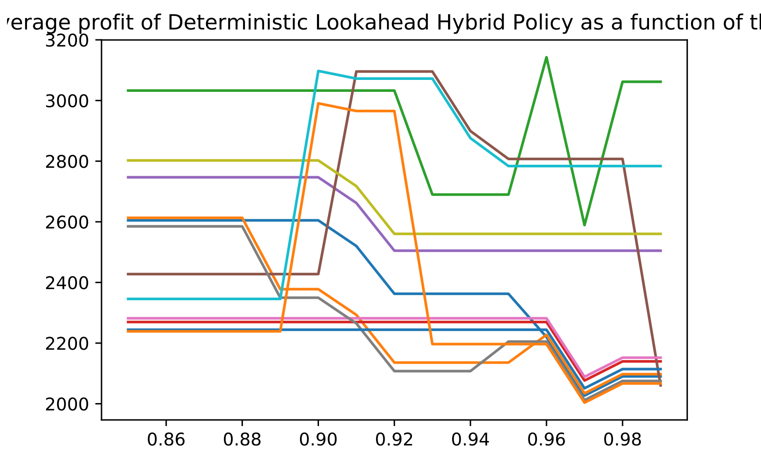

Figure 7: The colors represent different starting cities

optimal value for θ that can be read of the graph but we will pick $\theta=0.9$.

4.2.4 Value function approximation

Now we will look at a value function approximation policy. It is as follows:

$$X^{VFA}(S_t) = \arg\max_{x_t \in \mathcal{I}} (C(S_t, x_t) + \overline{V}_t^x(R_t^x))$$

where $\overline{V}_t^x(R_t^x)$ is the value of being at x_t.

The tricky part of a value function approximation is computing the value of choosing an action. However we will use a quite primitive way of calculating it for this example. We'll start by initialising the value of going to each city as 0. Then we simulate using our infered distribution of load prices of the cities and moving around and updating the value of going to a city with the average with the previous prices chosen to that city. To avoid getting caught in loops we will also pick a random city with probability 0.05 but then we will not update the value since we are not using the VFA policy.

Since this model is not parameterized we cannot plot its performance as a function of a parameter. We will however look at how this policy performs as a function of time horizon to see if the profit converges to some value to save computing time. Interesting that there is a consistent high profit average when we only move

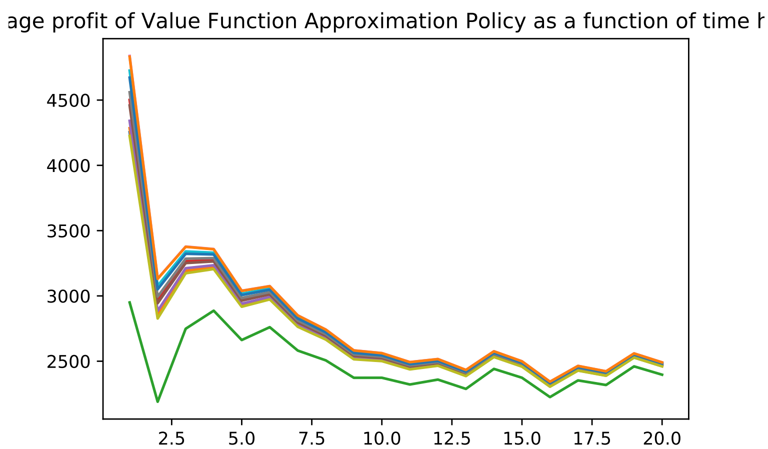

Figure 8: The colors represent different starting cities

once. It is consistent starting in all cities but one. It needs to be said that in this simulation (using the "real" data) all of the simulations started at the same time but in different cities but the prices are all independent so there should not be such a high correlation. However for short paths will have a higher variance.

5 Results

Now we are ready to compare all of the models that have been created. We will compare five models, four of the best above and also a random choice policy as a sort of contol. For each time horizon all of the policies started in the same city and at the same time for accurate comparison. However for each time horizon many starting cities and starting times were computed to even out noise in the data.

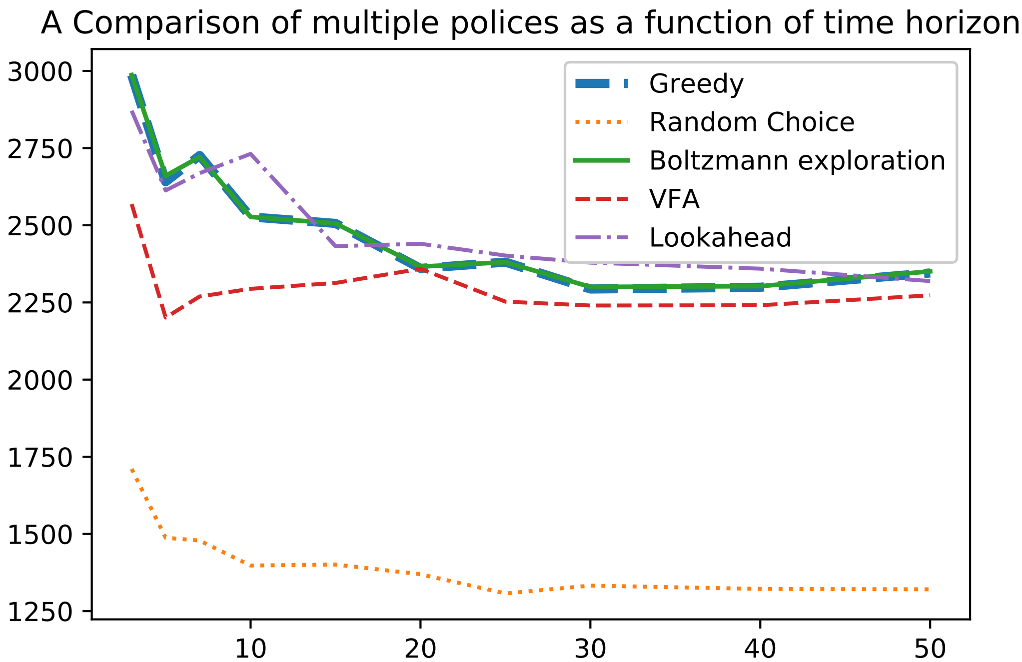

Figure 9

A Appendix Parameter Estimation Code

Graphs of the outputs were generated in R as well as some of the massaging of the data.

A.1 Data Generation

A.1.1 Demand (Python)

```
import pandas as pd
import numpy as np
n = 4
m = 3
I = list(range(n*m))
R = list(range(m))
def phi(i):
    return i // n
def pos(i):
    x = i \% n
    y = (i - x) // n
    return (x, y)
D_avg = 5
def gen_demand(I, R, csv=False):
    demand = pd.DataFrame(
        columns=["D", "i", "j", "ri", "rj"])
    r_min = 0.25
    r_max = 0.75
    i_min = 0.1
    i_max = 0.9
    U_in_r = [np.random.uniform(r_min, r_max) for r in R]
    U_out_r = [np.random.uniform(r_min, r_max) for r in R]
    U_in_i = [np.random.uniform(i_min, i_max) for i in I]
    U_out_i = [np.random.uniform(i_min, i_max) for i in I]
    dem = dict()
    for i in I:
        for j in I:
            if i != j:
                D = ((i_min+i_max)/2)**(-2)*((r_min+r_max)/2)**(-2)*
                D_avg*(U_in_i[j]*U_out_i[i] * U_out_r[phi(i)]*U_in_r[phi(j)])
                dem[(i, j)] = D
                demand = demand.append({"D": D, "i": i,
                        "j": j, "ri": phi(i), "rj": phi(j)}, ignore_index=True)
    demand.index = ['']*len(demand)
    if csv:
        demand.to_csv('data/demand.csv', index=False, header=True)
    return demand, dem
```

```
dataframe, demand = gen_demand(I, R, csv=True)
```

A.1.2 Loads (Python)

```
import pandas as pd
import numpy as np
n = 4
m = 3
I = list(range(n*m))
R = list(range(m))

def phi(i):
    return i // n

def pos(i):
    x = i \% n
    y = (i - x) // n
    return (x, y)

data = pd.read_csv('data/demand.csv')
demand = dict(
    zip(zip(data['i'].tolist(), data['j'].tolist()), data['D'].tolist()))

def gen_loads(demand, I, R, T=10, shift=0, use_dist=False):
    distance_modifier = 1.1 if use_dist else 0
    loads = pd.DataFrame(
        columns=["p", "t", "i", "j", "ri", "rj"])
    for t in range(1, T+1):
        for i in I:
            for j in I:
                if i != j:
                    n = np.random.poisson(demand[(i, j)])
                    for _ in range(n):
                        price = 1000 * np.random.gamma(
                            shape=demand[(i, j)], scale=1/demand[(i, j)])
                        loads = loads.append({"p": price, "t": t+shift, "dist": dist(i, j), "i": i,
                        "j": j, "ri": phi(i), "rj": phi(j)}, ignore_index=True)
    loads.index = ['']*len(loads)
    loads.to_csv('data/loads.csv', index=False, header=True)
    return loads

gen_loads(demand, I, R, T=20)
```

A.2 Metropolis Algorithm (Python)

```python
import numpy as np
import pandas as pd
import math
from tqdm import tqdm
import time


def log_posterior_alphabeta(a, b, y):
    r = 0.2
    s = -0.01
    q = 0.2
    p = 1.5
    n = len(y)
    return ((a-1)*(np.log(p) + sum([np.log(i) for i in y])) -
    b*(q + sum(y)) - (r+n)*np.log(math.gamma(a)) + a*(s+n)*np.log(b))


def log_posterior(a, b, y):
    return log_posterior_alphabeta(np.exp(a), np.exp(b), y) + a + b


def J_a(a, i, j):
    sigma_a = 0.075
    return np.random.normal(loc=a, scale=sigma_a)


def J_b(b, i, j):
    sigma_b = 0.075
    return np.random.normal(loc=b, scale=sigma_b)


data_loads = pd.read_csv('data/loads.csv', header=0)


def gibbs(alpha_0, beta_0, i, j, L=10000, chains=4, burn=1000):
    print("Gibbs currently sampling: ", i, "-->", j)
    a = np.empty(shape=(burn, chains))
    b = np.empty(shape=(burn, chains))
    a[0, ] = np.full(chains, np.log(alpha_0))
    b[0, ] = np.full(chains, np.log(beta_0))
    y = data_loads[(data_loads.i == i)
                   & (data_loads.j == j)]['p'].to_numpy()
    chain_acc_a = 0
    chain_acc_b = 0
    for k in range(chains):
```

```
    acceptance_rate_a = 0
    acceptance_rate_b = 0
    for t in (range(1, burn)):
        a_star = J_a(a[t-1, k], i, j)

        logr = log_posterior(
            a_star, b[t-1, k], y) - log_posterior(a[t-1, k], b[t-1, k], y)

        if min(logr, 0) >= np.log(np.random.uniform()):
            acceptance_rate_a += 1
            a[t, k] = a_star
        else:
            a[t, k] = a[t-1, k]

        b_star = J_b(b[t-1, k], i, j)

        logr = log_posterior(
            a[t, k], b_star, y) - log_posterior(a[t, k], b[t-1, k], y)

        if min(logr, 0) >= np.log(np.random.uniform()):
            acceptance_rate_b += 1
            b[t, k] = b_star
        else:
            b[t, k] = b[t-1, k]
    chain_acc_a += acceptance_rate_a/burn
    chain_acc_b += acceptance_rate_b/burn
print("Burn in acceptance rate: ",
      round(chain_acc_a/chains, 2), round(chain_acc_b/chains, 2))
a_burn = a[burn-1, ]
b_burn = b[burn-1, ]
a = np.empty(shape=(L, chains))
b = np.empty(shape=(L, chains))
a[0, ] = a_burn
b[0, ] = b_burn
chain_acc_a = 0
chain_acc_b = 0
for k in range(chains):
    acceptance_rate_a = 0
    acceptance_rate_b = 0
    for t in (range(1, L)):
        a_star = J_a(a[t-1, k], i, j)

        logr = log_posterior(
            a_star, b[t-1, k], y) - log_posterior(a[t-1, k], b[t-1, k], y)

        if min(logr, 0) >= np.log(np.random.uniform()):
```

```python
                acceptance_rate_a += 1
                a[t, k] = a_star
            else:
                a[t, k] = a[t-1, k]

            b_star = J_b(b[t-1, k], i, j)

            logr = log_posterior(
                a[t, k], b_star, y) - log_posterior(a[t, k], b[t-1, k], y)

            if min(logr, 0) >= np.log(np.random.uniform()):
                acceptance_rate_b += 1
                b[t, k] = b_star
            else:
                b[t, k] = b[t-1, k]
        chain_acc_a += acceptance_rate_a/(L-1)
        chain_acc_b += acceptance_rate_b/(L-1)
        if k == 0:
            alpha = np.exp(a[:, k])
            beta = np.exp(b[:, k])
        else:
            alpha = np.concatenate((alpha, np.exp(a[:, k])), axis=None)
            beta = np.concatenate((beta, np.exp(b[:, k])), axis=None)
    print("Final acceptance rate: ",
          round(chain_acc_a/chains, 2), round(chain_acc_b/chains, 2), "\n")
    return alpha, beta


def gen_all_gibbs():
    final_array = np.empty(shape=(4, 0))
    for i in I:
        for j in I:
            if i != j:
                alpha, beta = gibbs(6.8, 1/6800, i, j,
                                    L=4000, chains=5, burn=666)
                arr = np.column_stack((alpha, beta, np.full(
                    len(alpha), i), np.full(len(alpha), j)))
                if i == 0 and j == 1:
                    final_array = arr
                else:
                    final_array = np.vstack((final_array, arr))
    df = pd.DataFrame(
        {'alpha': final_array[:, 0], 'beta': final_array[:, 1],
        'i': final_array[:, 2], 'j': final_array[:, 3]})
    df.to_csv('data/gibbs.csv', index=False, header=True)
```

```
n = 4
m = 3
I = list(range(n*m))
t_0 = time.time()
gen_all_gibbs()
print('Time to execute: ', time.strftime(
    '\%H:\%M:\%S', time.gmtime(time.time()-t_0)))
```

A.3 Code for p-values (Python)

```
import numpy as np
import pandas as pd
import math
from tqdm import tqdm
import time
def T(y, alpha, beta):
    return np.mean(np.array([(i - alpha/beta)**2 for i in y]))
n = 4; m = 3
I = list(range(n*m))

data_loads = pd.read_csv(
    'data/loads.csv', header=0).drop(['t', 'ri', 'rj', 'dist'], axis=1)
theta_samples = pd.read_csv('data/gibbs.csv', header=0)

n_data = data_loads.groupby(by=['i', 'j']).count().to_numpy()

def n(i, j):
    return n_data[11 * i + j-1, 0]

def get_alpha_beta(i, j):
    return theta_samples[(theta_samples.i == i) & (theta_samples.j == j)].
        drop(['i', 'j'], axis=1).to_numpy()

def get_pval(i, j):
    n_ij = n(i, j)
    k = 3000
    set = get_alpha_beta(i, j)
    np.random.shuffle(set)
    y = data_loads[(data_loads.i == i)
                  & (data_loads.j == j)]['p'].to_numpy()
    I = np.zeros(k)
    for i, theta in enumerate(set[:k]):
        alpha, beta = theta
        y_rep = np.random.gamma(size=n_ij, shape=alpha, scale=1/beta)
        T_y = T(y, alpha, beta)
        T_rep = T(y_rep, alpha, beta)
        I[i] = 1 if T_rep >= T_y else 0
    return round(np.mean(I), 4)

def get_all_pval():
    p_data = pd.DataFrame(columns=['p_val', 'i', 'j'])
    for i in I:
        for j in I:
            if i != j:
                p = get_pval(i, j)
```

```
                p_data = p_data.append(
                    {'p_val': p, 'i': i, 'j': j}, ignore_index=True)
    p_data.index = ['']*len(p_data)
    p_data.to_csv('data/p_data.csv', index=False, header=True)
    return

get_all_pval()
```

B Appendix Optimizaion Code

This code was written in a single jupyter notebook which can be found on the GitHub repository for this project (OptiSim).

B.1 Setup

```
import numpy as np
import pandas as pd
from tqdm import tqdm
import time as tm
from world_gen import *
from matplotlib import pyplot as plt
from scipy.stats import gamma
from scipy.special import logsumexp

I = gen_I()
R = gen_R()

data_loads = pd.read_csv('data/loads_new.csv')
L = len(set(data_loads['t'].to_list()))
data = pd.read_csv('data/p.csv')
T = list(range(1, L+1))
p = dict(
    zip(zip(data['i'].tolist(), data['j'].tolist(), data['t'].tolist()), data['p'].tolist()))
```

B.2 Organization

```
def gen_S0(i_0,t_0):
    data = pd.read_csv('data/p_hat.csv')
    get_a = dict(zip(zip(data['i'].tolist(), data['j'].tolist()), data['alpha_post_ij'].tolist()))
    get_b = dict(zip(zip(data['i'].tolist(), data['j'].tolist()), data['beta_post_ij'].tolist()))
    lambda_est = pd.read_csv('data/lambda.csv')
    lamb = dict(zip(zip(lambda_est['i'].tolist(),
        lambda_est['j'].tolist()), lambda_est['lambda_ij'].tolist()))
    return i_0,get_prices(t_0),get_a,get_b,lamb,t_0
```

```
def get_prices(t):
    prices = dict()
    for i in I:
        for j in I:
            if i != j:
                prices[(i,j)]=round(p[(i,j,t)])
            else:
                prices[(i,j)]=0
    return prices


def C(S_t,x_t):
    R_t,I_t,t = S_t
    return I_t[R_t,x_t]


def policy_eval(policy,S_0,T = 20):
    R_0,I_0,A,B,Lambda,t_0 = S_0
    t=t_0
    S_t = (R_0,I_0,t_0)
    path = [R_0]
    profit = 0
    while t <= T+t_0:
        x_t = policy(S_t,S_0)
        profit += C(S_t,x_t)

        t+=1
        R_t = x_t
        I_t = get_prices(t)
        S_t = (R_t,I_t,t)
        path.append(x_t)
    return profit,path
```

B.3 Policies

B.3.1 Epsilon Greedy

```
def EG_policy(epsilon):
    def policy(S_t,S_0):
        if epsilon == 1:
            return np.random.choice(I)
        elif epsilon == 0:
            return np.argmax([C(S_t,x_t) for x_t in I])
        else:
            U = np.random.uniform()
            if U <= epsilon:
                return np.random.choice(I)
```

```
            else:
                return np.argmax([C(S_t,x_t) for x_t in I])
    return policy
```

B.3.2 Boltzmann

```
def BE_policy(theta):
    def policy(S_t,S_0):
        log_div = logsumexp(np.array([theta* C(S_t,x_t) for x_t in I]))
        probs = np.array([np.exp(theta* C(S_t,x_t) - log_div) for x_t in I])
        return np.random.choice(I,p = probs)
    return policy
```

B.3.3 Lookahead

```
import gurobipy as gp
from gurobipy import GRB

def C_y(t,i,j):
    return p[(i,j,t)] if i!= j else 0

def C_bar_y(t,i,j,A,B,theta):
    return gamma.ppf(a = A[(i,j)],scale = 1/B[(i,j)],q=theta) if i!=j else 0

def LA_DET_HYB_policy(theta):
    def policy(S_t,S_0):
        R_0,I_0,A,B,Lambda,t_0 = S_0
        R_t,I_t,t = S_t
        H = 10

        T_rem_H = set([t + k for k in range(H)])
        T = set([t + k for k in range(H+1)])
        model = gp.Model("Policy")

        Y = model.addVars(I, I, T, name=["y[\%s,\%s,\%s]" \% (i, j, t_prime)
            for i in I for j in I for t_prime in T], vtype=GRB.BINARY)

        model.setObjective(gp.quicksum(C_y(t,i,j) * Y[i,j,t] + gp.quicksum(C_bar_y(t_prime,i,j,A,B,theta)
            for t_prime in [t + k for k in range(1,H+1)]) for i in I for j in I),GRB.MAXIMIZE)

        model.addConstrs(gp.quicksum(Y[i, j, t_prime] for i in I for j in I) <= 1 for t_prime in T)
        model.addConstrs((gp.quicksum(Y[k,i,t_prime] for k in I) - gp.quicksum(Y[i,j,t_prime+1]
            for j in I)) == 0 for t_prime in T_rem_H for i in I)
        model.addConstrs(gp.quicksum(Y[i,j,t] for j in I) == 0 for i in set(I)-set([R_t]))

        model.optimize()
        for j in I:
```

```
            if Y[R_t,j,t].x>0:
                return j
    return policy
```

B.3.4 VFA

```
def update_value(R_t,value):
    Values = {int(y[0]): float(y[1]) for y in [x.split(",") for x
        in open('data/value_func.csv').read().split('\n') if x]}
    alpha = 0.075
    Values[R_t] = (1-alpha) * Values[R_t] + alpha * value
    with open('data/value_func.csv', 'w') as f:
        for key in Values.keys():
            f.write("\%s, \%s\n" \% (key, Values[key]))


def V(x_t):
    return Opt_Values[x_t]

def VFA_policy():
    def policy(S_t,S_0):
        R_t,I_t,t = S_t
        x = np.argmax(np.array([C(S_t,x_t) + V(x_t) for x_t in I]))
        value = C(S_t,x)
        # update_value(R_t,value)
        return x
    return policy

def gen_offline_loads(i,j):
    S_0 = gen_S0(1,1)
    R_0,I_0,A,B,Lambda,t_0 = S_0
    return np.max(np.random.gamma(shape = A[(i,j)],scale = 1/B[(i,j)], size
        = int(np.ceil(Lambda[(i,j)])))) if i!=j else 0

def gen_all_offline_loads(i):
    return np.array([gen_offline_loads(i,j) for j in I])


def offline_exploration(K):
    Values  = dict(zip(I,np.array([np.array([-1]) for _ in range(len(I))])))
    eps = 0.05
    alpha = 0.1
    city = np.random.choice(I)
    for n in range(1,K+1):
        prices = gen_all_offline_loads(city)
        U = np.random.uniform()
```

```
        if U <= eps:
            x_t = np.random.choice(I)
        else:
            x_t = np.argmax(np.array([prices[i] + np.average(Values[i]) for i in I]))
            if Values[city][0] == -1:
                Values[city][0] = prices[x_t]
            else:
                Values[city] = np.append(Values[city],prices[x_t])
        city = x_t
    with open('data/value_func.csv', 'w') as f:
        for key in Values.keys():
            f.write("\%s, \%s\n" \% (key, np.average(Values[key])))

offline_exploration(4000)

Opt_Values = {int(y[0]): float(y[1]) for y in [x.split(",") for x in
    open('data/value_func.csv').read().split('\n') if x]}
```

B.4 Policy Comparison

```
def policy_simulator():
    K = [3,5,7,10,15,20,25,30,40,50]
    # K = [1,3,5,7,10]
    EG_profits = []
    Rand_profits = []
    BE_profits = []
    VFA_profits = []
    LA_profits = []
    for time_horizon in tqdm(K):
        start_times = np.random.choice(list(range(3,95-time_horizon)),size = 3)
        cities = np.random.choice(I,size = 4)
        EG_little = []
        Rand_little = []
        BE_little = []
        VFA_little = []
        LA_little = []
        for city in cities:
            for start_time in start_times:
                S_0= gen_S0(city,start_time)
                EG_little.append(
                    np.average([policy_eval(EG_policy(0),S_0,T=time_horizon)[0]/time_horizon
                        for _ in range(7)])
                )
                Rand_little.append(
                    np.average([policy_eval(EG_policy(1),S_0,T=time_horizon)[0]/time_horizon
                        for _ in range(7)])
```

```
            )
            BE_little.append(
                np.average([policy_eval(BE_policy(0.15),S_0,T=time_horizon)[0]/time_horizon
                    for _ in range(7)])
            )
            VFA_little.append(
                np.average([policy_eval(VFA_policy(),S_0,T=time_horizon)[0]/time_horizon
                    for _ in range(1)])
            )
            LA_little.append(
                np.average([policy_eval(LA_DET_HYB_policy(0.9),S_0,T=time_horizon)[0]/time_horizon
                    for _ in range(1)])
            )
    EG_profits.append(np.average(EG_little))
    Rand_profits.append(np.average(Rand_little))
    BE_profits.append(np.average(BE_little))
    VFA_profits.append(np.average(VFA_little))
    LA_profits.append(np.average(LA_little))
plt.figure()
plt.title("A Comparison of multiple polices as a function of time horizon")
plt.plot(K,EG_profits,label = "Greedy",lw=3.5,ls='--')
plt.plot(K,Rand_profits,label = "Random Choice",ls=':')
plt.plot(K,BE_profits,label = "Boltzmann exploration",lw=1.75)
plt.plot(K,VFA_profits,label = "VFA",ls='--')
plt.plot(K,LA_profits,label = "Lookahead",ls='-.')
plt.legend()
plt.savefig('figs/policy_comparison.eps',format='eps')
plt.savefig('figs/policy_comparison.png',format='png',dpi = 200)
plt.show()
```