Operations Research II Final Project: Optimizing Restaurant Seating

Jm Huibonhoa, Amy Hung, Ajay Phatak, & Steven Silverman

December 22, 2015

Contents

1 Introduction

The goal of a restaurant—or, indeed, any business—is to maximize efficiency so that it can increase its profits. For restaurants, revenue comes from two sources: walk-in customers and reservations. The restaurant that will make the most profits is precisely the one which allocates its resources most effectively and accepts the optimal combination of walk-in customers and reservations. Management must make decisions about when and where people should be seated.

This extends to such questions like whether or not to seat parties at tables that do not exactly match their size (for example, putting a party of two at a table for four because there are no other options). Accepting a reservation request will lead to higher *guaranteed* revenue, but it may decrease the amount of revenue in the future by rendering several seats unusable (and non-revenue-generating) for a certain length of time. However, if a restaurant decides to reject the customer, then there is a chance that in the future there are no requests for the table and the restaurant will lose the opportunity to sit anyone there.

This setup also brings another complication. When deciding how to seat the walk in customers, space needs to be left for the reservations that have already been made (the possibility for no-shows have not been included in this model). In general, we will assume that the larger the group is, the larger the revenue that the group will bring; equivalently, revenue per person is constant. This means that managers would want to seat the largest possible group at each table. However, combinations of groups with fewer people than the maximum number that a table can seat also need to be considered because seating a smaller group would still bring more profits than an empty table.

2 Methods

In this paper we attempt to determine when accepting a reservation at the current time is the optimal decision by using dynamic programming. There are some methods that have already been explored, including a method by Bossert that is based upon predicted future demand. We will begin with this method and use it as a basis to find an optimal seating algorithm for taking reservations. Then we will explore a extension by Fieldman which attempts to expand the model by creating a flexibility factor that allows customers to be given another time slot different from the one that was initially requested; this will be modeled by guests randomly accepting or rejecting another time slot. We will then take this one step further and attempt to offer customers a discount on the meal if they accept another time slot in hopes that the flexibility of the model will increase. The methods in the paper will be tested using random demand streams and then compared to find the most profitable method.

Throughout the paper, we make several simplifying assumptions to make the project manageable. First, we assume that future demand is uncertain and tables cannot be combined (that is, restaurants cannot make two tables for two into one table for four, or split parties among tables). We also assume all party sizes take the same amount of time to eat, and

the time to clean and prepare tables are negligible. The restaurants that are taken into consideration have staff that make the decisions on where and when to seat people. Whenever we decide whether or not to accept a reservation or request, we compare the profits that we could obtain if we took the reservation or seated the current group versus how much the restaurant could make if they did not seat this current group. There are limitations on how many times a simulation can be tested before averaged, and the number of combinations of tables that can be checked before deciding on a optimal seating or else the run time will become unreasonable.

2.1 Bossert's Method and Two-Pass Heuristic

Bossert's algorithm divides timeslots during which the restaurant can accept reservations into T distinct time periods. In each period, at most one reservation will be requested with a set probability. We also assume that the time that each party stays at a table is also constant, and the number of tables in the restaurant does not change (tables cannot be combined or shared).

As an approximation method, we use the two pass heuristic described below.

We first seat the party sizes that match the number of seats at the table and only reduce the *turn count* by 1. Turn count is the maximum number of possible groups that the table can seat in the given time period. For example, if there are six time slots (numbered 1-6) and each party takes two to eat, seating a group at time 1 would reduce the turn count from 3 to 2, but seating a group at time 2 would reduce it to 1. Then, we seat the remaining groups at suboptimal tables, meaning the number of seats at a table is greater than the number of people in the group.

With the heuristic in place, we look at the reservation requests. Using this method, we first reject requests if there is not sufficient space (the obvious step). If there is sufficient space, we compare the benefit of accepting versus rejecting the given reservation request. We reject a customer if the expected profit of doing so is not greater than the immediate gains of accepting a reservation now. To make this decision, we first tentatively reject the request and calculate the approximate future process by feeding simulated demand streams into the heuristic. Then, we calculate the profits if we accept the request using the same demand streams (but altering the state of the restaurant by seating the party). During this approximation process, we want to accept the reservations that are an exact fit for the table sizes first, and only reduce the turn count by one, as described in the heuristic. The process is repeated multiple times and then averaged. We then pick the option that would most likely give us the higher profits (determined by the higher average).

Below is the set of equations for Bossert's dynamic programming algorithm. The variables are defined below.

- n_k: the number of requests accepted over periods 1 to k - 1
- s_k: vector of length n_k, corresponds to the party sizes of the accepted requests

- t_k: vector of length n_k , corresponds to the arrival time slots of the accepted requests
- $s_k^{'}$: size of the pending reservation request
- $t_k^{'}$: requested arrival slot of the pending reservation request

Taking a more algorithmic approach in explaining this algorithm, notice that the first equation denotes the case where the reservation request is rejected due to unavailability and we proceed to consider the next reservation request. The second equation denotes the case where we consider the expected value that taking this reservation provides. and compare it to the expected utilization from rejecting the reservation using the two-pass heuristic described above. We then take the greater of the two and proceed to consider the next reservation request.

$$J_{k+1}\,(n_k,\, s_k,\, t_k, s_k,\, t_k) = \begin{cases} \text{E}\ [\ J_{k+1}\ (n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1})\ ], & \text{if cannot accommodate request,} \\ \max\ (s_k + \text{E}[\ J_{k+1}\ (\ n_k{+}1,\, s_k \cup s_i^{'},\, t_k \cup t_k^{'},\, s_{k+1}^{'}, t_k^{'}\ )], \\ \text{E}\ [J_{k+1}\ (n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1})]) \end{cases}$$

2.2 Flexibility Model

In the flexibility model, we use a strategy similar to Bossert's method but we include a flexibility element that allows the restaurant to offer to accommodate the customer's request at a slightly different time if the current requested time is sold out or significantly suboptimal. This model attempts to seat as many groups as possible and tries to give each group a time closest to their requested time. Adding this option also requires us to consider the likelihood of the customer accepting the different time, since some customers will have very stringent timing needs. We model the chances of the customer accepting the different time as a function of the spread between the requested and alternative time, with lower probability the farther away the two times are. If the customer accepts the alternative time, we take the reservation because it will be optimal (we would not have offered it otherwise). If they do not do so, we reject the request. In this model, in order to make the optimal decision the restaurant needs to consider the potential future demand. If there is low demand in the future, then the restaurant risks losing profit from having seats not being filled if they reject the request. Therefore, they would want to try to satisfy the customer and give them the requested time. If a restaurant is busy, they can risk getting their offer rejected by the customer because they are confident that there will be other requests for the tables if the current customer does not want to wait. This method is better than the naïve algorithm because it allows the restaurant to more easily place groups at tables that will only reduce maximum number of requests that can be assigned to a table by one.

Building upon the model mentioned before, we add the following variables to the flexibility model:

- q_k: a vector of length n_k, indicates the table assigned to accepted request k
- $q_k^{'}$: the table assigned to accept request k where ($t_k^{'}$ is the alternate arrival time suggested to the client)

As seen below, the difference between the Flexibility model and Bosserts model is that we first compare the value of accepting the reservation, and in the case that we reject it consider the value provided if the customer takes the optimal arrival time that we suggest to them. The probability that the customer would accept this value is p(q_k) and a rejection value of 1 - p(q_k), where the probability decreases as the suggested time increases in distance from the original reservation time. After taking the maximum of these to values we then take the maximum of the value we just chose and the expected value we get from rejecting the reservation request. Afterwards we proceed recursively as we did previously.

$$J_{k+1}\,(n_k,\, s_k,\, t_k, s_k,\, t_k, q_k) = \begin{cases} \text{E}\,[\; J_{k+1}\;(n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1}, q_k\;], & \text{if cannot accommodate request,} \\ \max\,(\;\max\,(s_k + \text{E}[\; J_{k+1}\;(\; n_k{+}1,\, s_k \cup s_i^{'},\, t_k \cup t_k^{'},\, s_{k+1}^{'}, t_k^{'}\;)], \\ s_k + \text{E}[\; J_{k+1}\;(\; n_k{+}1,\, s_k \cup s_i^{'},\, t_k \cup t_k^{'},\, s_{k+1}^{'}, t_k^{'},\, q_k \cup q_k^{'}\;) \\ \times\; \text{p}(q_k) + (1\text{-p}(q_k)) \times \text{E}\,[J_{k+1}\;(n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1},\, q_k)]\;]), \\ \text{E}\,[J_{k+1}\;(n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1}, q_k)]) \end{cases}$$

2.3 Extended Model

Building on the flexibility model, restaurants can now offer a discount on the meal if a group agrees to take a reservation time different from the one that they had originally requested. The discount value is a constant, and it makes the customers more likely to agree to our alternative time. This may give a higher total number of customers, but with the discounts, the value of these customers will be less. However, we hypothesize that optimal seating at a discount will provide a higher total utilization and consequently lead to greater total revenue during the time period.

To find the optimal discount level, we simulate the models multiple times and average the profits with increasing discount levels. We choose the discount level right before the amount of increased profits start to decrease (making the assumption that there are no non-global maxima of revenue as a function of discount: having multiple local maxima would be very strange and contradict our model of probabilistic acceptances). In our simulations, an average discount of 15% provided the greatest increase in total value.

Consequently, we add the following variable to our Extended Model:

- d: a constant value representing the discount provided

As mentioned above, we first compare the value of accepting the reservation, and in the case that we reject it consider the value provided if the customer takes the optimal arrival time that we suggest to them. Next we take the maximum of accepting the reservation, and in the case that we reject it consider the value provided if the customer takes the optimal arrival time that we suggest to them with the probability of p(d) + p(q_k) and a rejection value of 1-p(d) - p(q_k) . In our model a 15% discount provides a 15% increase in probability that the customer accepts the alternate time, therefore the probability of accepting the alternate reservation is the probability increase provided by the discount plus the original probability.

Note that if we accept the reservation that the values of the two maximums will be equal, since they will just be the value of accepting the reservation. However, at an optimal level of discount, we note that the expectation of the event where the customer accepts the reservation with a higher probability provided by the discount is greater than the expected value of the customer rejecting the optimal alternative time slot.

Lastly, we take the value above and compare it with the expected value of rejecting the reservation and take the maximum of the two. Then we proceed recursively as in the other models:

$$J_{k+1}\,(n_k,\, s_k,\, t_k, s_k,\, t_k, q_k, d) = \begin{cases} \mathrm{E}\,[\; J_{k+1}\;(n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1}, q_k, d)\;], & \text{if cannot accommodate request,} \\ \max\,(\,\max\,(\,\max\,(s_k + \mathrm{E}[\; J_{k+1}\;(\; n_k{+}1,\, s_k \cup s_i^{'},\, t_k \cup t_k^{'},\, s_{k+1}^{'}, t_k^{'}\;)], \\ s_k + \mathrm{E}[\; J_{k+1}\;(\; n_k{+}1,\, s_k \cup s_i^{'},\, t_k \cup t_k^{'},\, s_{k+1}^{'}, t_k^{'},\, q_k \cup q_k^{'}\;,\, d) \\ \times\; \mathrm{p}(q_k) + (1\text{-}\mathrm{p}(q_k)) \times \mathrm{E}\,[J_{k+1}\;(n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1},\, q_k,\, d])) \\ ,\max\,(\; s_k + \mathrm{E}[\; J_{k+1}\;(\; n_k{+}1,\, s_k \cup s_i^{'},\, t_k \cup t_k^{'},\, s_{k+1}^{'}, t_k^{'}\;)], \\ s_k + \mathrm{E}[\; J_{k+1}\;(\; n_k{+}1,\, s_k \cup s_i^{'},\, t_k \cup t_k^{'},\, s_{k+1}^{'}, t_k^{'},\, q_k, d\;) \\ \times\; p(d) + \mathrm{p}(q_k) + (1\text{-}p(d)\text{-}\;\mathrm{p}(q_k)) \times \mathrm{E}\,[J_{k+1}\;(n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1},\, q_k,\, d)]\;]\;), \\ \mathrm{E}\,[J_{k+1}\;(n_k,\, s_k,\, t_k, s_{k+1},\, t_{k+1}, d)])) \end{cases}$$

3 Results

4 Appendix: Code

Below follows the Python code we used to simulate demand and test our algorithm. Apologies for the rather inaccessible formatting.

```
##################################################
# Objects.py                                     #
# Contains class definitions for Restaurants     #
# and Tables, plus relevant methods for each     #
##################################################

class Restaurant(object):
    def __init__(self,table_array):
        self.table_dict = self.init_tables(table_array)
        self.total_capacity = self.calc_total_capacity(table_array)

      #table information is stored as an array of tuples
      #ie: [(6,1),(2,1),(4,2)] represents 1 table with
      #capacity 6, 1 table with capacity 2 and 1 table
      #with capacity 4
      #the purpose of this function is to create a dictionary of
      #table objects where each key of the dictionary is the capacity of the table
      #ie: [(6,1),(2,1),(4,1)] turns into the dictionary
      #{6:[table_object],2:[table_object],4:[table_object,table_object]}


      # we might just want to make a constructor where we pass in Tables
      # rather than table_tuples
      # (something to think about)
    def init_tables(self,table_array):
        table_dict = []
```

```
        for table_tuple in table_array:
            arr = []
            capacity = table_tuple[0]
            num_tables = table_tuple[1]
            meal_duration = 4
            max_duration = 20
            #create a table object for each table of our given capacity
            #and append it to our array
            for i in range(num_tables):
              new_table = Table(capacity,meal_duration,max_duration)
              table_dict.append(new_table)
        return table_dict

    def calc_total_capacity(self,table_array):
        total = 0
        for table_tuple in table_array:
            capacity = table_tuple[0]
            num_tables = table_tuple[1]
            total += capacity * num_tables
        return total

class Table(object):
    #A table is occupied once there are people at the
    #table. No more people can be seated at the same
    #table even if there are seats available
    def __init__(self,capacity,meal_duration,max_duration):
        self.capacity = capacity
        self.meal_duration = meal_duration
        #invariant for the array below is that the tuples
        #are sorted in ascending order where the first tuple is
        #always (0,x) where x is some end time
        self.available = [(0, max_duration)]
        self.max_duration = max_duration


    #count must be at least 1
    def turn_count(self):
        count = 0
        for (start,end) in self.available:
            count += (end-start)/self.meal_duration
        return count

    # calculate's probability of accepting alternate time slot
    # based on how far the suggested start time is from
    # the original start time
    def accept_probability(self,start_time,discount):
        optimal_suggestion = self.available[0][1]
        #below is safe since we have already called seat on this start time
        #can extend this part to search for all possible time slots
        if discount == None:
            return  float((start_time-optimal_suggestion))/float(self.max_duration)
        else:
            original_prob = round(1 - ((start_time-optimal_suggestion)/self.max_duration),2)
            discount_incentive = round(((start_time-optimal_suggestion)/self.max_duration),2) + discount
            return original_prob + discount_incentive

    def optimal_time(self):
        return self.available[0][1]

    def seat(self,start_time):
        for index in range(len(self.available)):
            (start,end) = self.available[index]
            if start <= start_time and start_time + self.meal_duration < end:
                self.available.pop(index)
                self.available.insert(index, (start, start_time))
                #below should be index + 1 not index please confirm this
                self.available.insert(index+1, (start_time+self.meal_duration, end))
                return True
        return False
```

```
    def unseat(self,start_time): # assume this is called on a valid reservation
        for index in range(len(self.available)):
            (start,end) = self.available[index]
            if start_time > start and start_time <= end: # in the right spot
                if start_time == end:
                    self.available[index] = (start, end+self.meal_duration)
        # coalesce
        to_pop = []
        for index in range(1, len(self.available)):
            (start1,end1) = self.available[index-1]
            (start2,end2) = self.available[index]
            if end1 == start2:
                self.available[index-1] = (start1,end2)
                to_pop.append(index)
        for index in to_pop:
            self.available.pop(index)


##################################################
# Heuristic.py                                   #
# Contains functions to compute the two-pass     #
# heuristic, as well as run the naive algorithm  #
# and both versions of the extended algorithms   #
##################################################


import math
import random
from objects import *
import copy

#global parameters
value_per_seat = 10
total_period = 20
meal_duration = 4

def randomStream(length):
    arrivals = []
    for i in range(length):
        arrivals.append((random.randint(2,6),random.randint(0, total_period - meal_duration)))
    return arrivals

# two-pass heuristic to estimate seats-to-go

# below is the heuristic for the naive algorithm
def heuristic_1(restaurant, demandStream):
    mealDuration = 4 # one unit = 15 minutes
    # demandStream is a list of (party size, time)
    # first loop places all exact fit reservations (as described in paper)
    fulfilled = [] # reservations we've booked
    booked = 0 # seats we've booked
    for (size, time) in demandStream:
        for table in restaurant.table_dict:
            if table.capacity == size:
                initial_count = table.turn_count()
                seatable = table.seat(time)
                if seatable:
                    final_count = table.turn_count()
                    if (final_count < initial_count - 1):
                        table.unseat(time)
                    else:
                        fulfilled.append((size,time))
                        booked += size
                        break # got it, so look at next reservation
                # otherwise it's booked for that time, so look at other tables
    # remove reservations we already made
    demandStream[:] = [x for x in demandStream if not (x in fulfilled)]
    for (size, time) in demandStream:
        for table in restaurant.table_dict:
            if table.capacity >= size:
```

```
                seatable = table.seat(time)
                if seatable:
                    fulfilled.append((size,time))
                    #missing append to fulfilled
                    booked += size
                    break # go to next reservation
    return booked

# below is the heuristic for the second extended algorithm in the paper
def heuristic_2(restaurant, demandStream, discount):
    mealDuration = 4 # one unit = 15 minutes
    # demandStream is a list of (party size, time)
    # first loop places all exact fit reservations (as described in paper)
    fulfilled = [] # reservations we've booked
    booked = 0 # seats we've booked
    for (size, time) in demandStream:
        for table in restaurant.table_dict:
            if table.capacity == size:
                initial_count = table.turn_count()
                seatable = table.seat(time)
                if seatable:
                    accept_probability = round(random.uniform(0.1, 1.0), 2)
                    accept_probability_threshold = table.accept_probability(time,None)
                    final_count = table.turn_count()
                    if (final_count < initial_count - 1) and (accept_probability > accept_probability_threshold):
                        table.unseat(time)
                    else:
                        fulfilled.append((size,time))
                        booked += size
                        break # got it, so look at next reservation
                # otherwise it's booked for that time, so look at other tables
    # remove reservations we already made
    demandStream[:] = [x for x in demandStream if not (x in fulfilled)]
    for (size, time) in demandStream:
        for table in restaurant.table_dict:
            if table.capacity >= size:
                seatable = table.seat(time)
                if seatable:
                    accept_probability = round(random.uniform(0.1, 1.0), 2)
                    accept_probability_threshold = table.accept_probability(time,discount)
                    if (accept_probability > accept_probability_threshold):
                        table.unseat(time)
                    else:
                        fulfilled.append((size,time))
                        booked += size
                        break # got it, so look at next reservation
    return booked * value_per_seat

# below is the heuristic for the our extended algorithm
def heuristic_3(restaurant, demandStream, discount):
    mealDuration = 4 # one unit = 15 minutes
    # demandStream is a list of (party size, time)
    # first loop places all exact fit reservations (as described in paper)
    fulfilled = [] # reservations we've booked
    booked = 0 # seats we've booked
    for (size, time) in demandStream:
        for table in restaurant.table_dict:
            if table.capacity == size:
                initial_count = table.turn_count()
                seatable = table.seat(time)
                if seatable:
                    accept_probability = round(random.uniform(0.1, 1.0), 2)
                    accept_probability_threshold = table.accept_probability(time,discount)
                    final_count = table.turn_count()
                    if (final_count < initial_count - 1) and (accept_probability > accept_probability_threshold):
                        table.unseat(time)
                    else:
                        fulfilled.append((size,time))
                        booked += size
                        break # got it, so look at next reservation
```

```
    # remove reservations we already made
    demandStream[:] = [x for x in demandStream if not (x in fulfilled)]
    for (size, time) in demandStream:
        for table in restaurant.table_dict:
            if table.capacity >= size:
                seatable = table.seat(time)
                if seatable:
                    accept_probability = round(random.uniform(0.1, 1.0), 2)
                    accept_probability_threshold = table.accept_probability(time,discount)
                    if (accept_probability > accept_probability_threshold):
                        table.unseat(time)
                    else:
                        fulfilled.append((size,time))
                        booked += size
                        break # got it, so look at next reservation
    return booked * value_per_seat

#requires: num_requests_accpeted <= total_period same for all other arguments except booked. heruisitc_num <= 3
def run_naive (restaurant,num_requests_accepted,accepted,accepted_arrival,pending_request_size,
               pending_requested_arrival,booked):
    #expected utilization on reject
    reject_utilization = 0
    sim_streams = 50
    for i in range(len(pending_requested_arrival)):
        reject_utilization += heuristic_1(restaurant,randomStream(sim_streams))
    reject_utilization /= float(sim_streams)
    #this is necessary to test whether a reservation makes sense without committing to the reservation
    test_restaurant = copy.deepcopy(restaurant)
    #base case if number of requests accepted is 0 we return the number of seats booked * value_per_seat
    #along with the accepted requests and arrival times
    if num_requests_accepted == 0:
        return (accepted,accepted_arrival,booked * value_per_seat)
    else:
        time = pending_requested_arrival[0]
        size = pending_request_size[0]
        #try accepting the reservation and test it against case where we reject
        for table in test_restaurant.table_dict:
            if table.capacity == size:
                seatable = table.seat(time)
                if not seatable: table.unseat(time)
        accept_utilization = 0
        for i in range(len(pending_requested_arrival)):
            accept_utilization += heuristic_1(restaurant,randomStream(sim_streams))
        accept_utilization /= float(sim_streams)
        #update parameters to be passed into recursive call
        num_requests_accepted -= 1
        pending_requested_arrival.pop(0)
        pending_request_size.pop(0)
        if reject_utilization > accept_utilization:
            return run_naive(restaurant,num_requests_accepted,accepted,accepted_arrival,pending_request_size,
                             pending_requested_arrival,booked)
        else:
            accepted.append(size)
            accepted_arrival.append(time)
            return run_naive(test_restaurant,num_requests_accepted,accepted,accepted_arrival,pending_request_size,
                             pending_requested_arrival,booked+size)


#requires: num_requests_accpeted <= total_period same for all other arguments except booked. heruisitc_num <= 3
#discount is None if heuristic_num == 2 and a value between 0 and 1 if heuristic_num == 3
def run_extendeds (restaurant,num_requests_accepted,accepted,accepted_arrival,pending_request_size,
                   pending_requested_arrival,revenue,heuristic_num,discount=None):
    switch = {
        2: heuristic_2,
        3: heuristic_3
    }
    discounted = False
    reject_utilization = 0
    sim_streams = 50
    heuristic_func = switch.get(heuristic_num, lambda: "invalid heuristic function")
```

```
    #expected utilization on reject
    for i in range(len(pending_requested_arrival)):
        reject_utilization += heuristic_func(restaurant,randomStream(sim_streams),discount)
    reject_utilization /= float(sim_streams)
    #this is necessary to test whether a reservation makes sense without committing to the reservation
    test_restaurant = copy.deepcopy(restaurant)
    #base case if number of requests accepted is 0 we return the number of seats booked * value_per_seat
    #along with the accepted requests and arrival times
    if len(pending_requested_arrival) == 0:
        return (accepted,accepted_arrival,revenue)
    else:
        time = pending_requested_arrival[0]
        size = pending_request_size[0]
        #try accepting the reservation and test it against case where we reject
        for table in test_restaurant.table_dict:
            if table.capacity == size:
                seatable = table.seat(time)
                if not seatable:
                    #capping accept probability at 70 percent
                    accept_probability = round(random.uniform(0.1, 0.7), 2)
                    accept_probability_threshold = table.accept_probability(time,discount)
                    if accept_probability > accept_probability_threshold:
                        table.unseat(time)
                    elif heuristic_num == 3: discounted = True
        accept_utilization = 0
        for i in range(len(pending_requested_arrival)):
            accept_utilization += heuristic_func(restaurant,randomStream(sim_streams),discount)
        accept_utilization /= float(sim_streams)
        #update parameters to be passed into recursive call
        num_requests_accepted -= 1
        pending_requested_arrival.pop(0)
        pending_request_size.pop(0)
        if reject_utilization > accept_utilization:
            return run_extendeds(restaurant,num_requests_accepted,accepted,accepted_arrival,pending_request_size,
                                 pending_requested_arrival,revenue,heuristic_num,discount)
        else:
            accepted.append(size)
            accepted_arrival.append(time)
            if discounted:
                revenue = revenue + size * value_per_seat * (1-discount)
            else:
                revenue = revenue + size * value_per_seat
            return run_extendeds(test_restaurant,num_requests_accepted,accepted,accepted_arrival,pending_request_size,
                                 pending_requested_arrival,revenue,heuristic_num,discount)

############################################################
# simulate.py                                              #
# Contains a wrapper to test the algorithm on a            #
# small restaurant, designated Union Grill internally      #
# (does not reflect actual Union Grill seating alignment)  #
############################################################

import math
import random
from objects import *
from heuristic import *

#generate customers according to a poisson process
def nextCustomers(rate):
    return -math.log(1.0 - random.random()) / rate


def run_simulation():
    #testing to see if oop implementation is working properly
    table_array = [(6,3),(2,3),(4,3)]
    union_grill = Restaurant(table_array)

    naive=[]
    heuristic1=[]
    heuristic2=[]
```

```
    heuristic3=[]
    num_requests = 20
    for x in xrange(0,10):
        pending = randomStream(num_requests)
        pending_size = [size for (size,time) in pending]
        pending_time = [time for (size,time) in pending]
        naive.append(run_naive(union_grill,num_requests,
                               [],[],pending_size,pending_time,0))
        #heuristic1.append(run_extendeds(union_grill,num_requests,
        #                                 [],[],pending_size,pending_time,0))
        pending_size = [size for (size,time) in pending]
        pending_time = [time for (size,time) in pending]
        heuristic2.append(run_extendeds(restaurant=union_grill,num_requests_accepted=num_requests,
                                        accepted=[],accepted_arrival=[],
                                        pending_request_size=pending_size,
                                        pending_requested_arrival=pending_time,revenue=0,heuristic_num=2))
        pending_size = [size for (size,time) in pending]
        pending_time = [time for (size,time) in pending]
        heuristic3.append(run_extendeds(union_grill,num_requests,
                                        [],[],pending_size,pending_time,0,3,discount=.15))
        print x
    print "naive:", 1.0*sum([c for (a,b,c) in naive])/len(naive)
    print "extended:", 1.0*sum([c for (a,b,c) in heuristic2])/len(heuristic2)
    print "discounted:", 1.0*sum([c for (a,b,c) in heuristic3])/len(heuristic3)
    #test

if __name__ == '__main__':
    run_simulation()
```