Carnegie Mellon University

21-393 Operations Research II

Dr. Alan Frieze

December 19, 2017

Optimizing Freshmen Dorm Assignment

Cindy Fouraker

Akanksha Kartik

Devin Noh

Harika Vedati

Acknowledgements

We would like to thank CMU Housing Service for providing us the insight into the current methodology and the actual student data of 2017 (without any identity information). We would also like to thank Professor Alan Frieze for providing us guidance throughout the research and writing process.

Introduction:

Every year, the incoming freshman at Carnegie Mellon University list their housing preferences and the Housing Department tries their best to meet the preferences of students by assigning incoming freshman to a dorm and room type of their choice. However, the increase in the number of incoming freshman each year, the limited number of freshman housing options on campus coupled with the fact that all freshmen must reside on campus makes it a lot more difficult to satisfy all student preferences. It is critical for Carnegie Mellon University to maximize student's satisfaction with their housing and this is the problem that Housing Department experiences each year.

Current Method:

The truth is even CMU, known for its most prestigious computer science program in the world does not have a perfect program which can process dormitory assignment in a single run. Housing service currently has its own program called 'AutoAllocate' that assigns students to their dorms based on factors defined by the school. To summarize, the method is what is widely known as a 'greedy method' in which each entity is examined in order of their deposit submission date and is assigned to their first available preference. Due to shortcoming of the program, housing service runs about two hundred students at a time into the program and manually modify the output to prevent erroneous assignment. Since those students who have identified roommates beforehand are guaranteed roommates, their selection of preference for roommate quality is practically ignored. Thus, the program currently runs slightly differently for those students without their roommates; their preference for potential roommate is considered at the same time while assigning them to their rooms of preference. For freshmen class of 2018,

there are about 480 students (30%) who have opted for random roommates, thus our method will focus largely on improving the satisfaction of these students.

Proposed Method:

In order to improve the current method we will give weight to the compatibility of the two randomly assigned students. We will begin by using the hungarian algorithm to assign people to a dorm and room type. We will use an exponential function to determine the cost of assigning each person to a specific dorm room. For example if a student does not get their first preference and gets assigned to their second preference the cost of that operation will be 10 and the next preference would be 30 etc. After all the students are assigned to dorm building and room type, we will greedily assign roommates. We will pair the first two people who requested a double and the first three people who requested a triple and so on. From here we manually swap roommates until we reach the optimal pairings. We will specifically consider three categorical variables

1. Smoking/Nonsmoking
2. Cleanliness
3. Morning or Night

We will assign a 1 if person is smoker and 0 if nonsmoker, 1 if messy 0 if clean, 1 if morning 0 if night. We will be using python to swap the roommates. For each student we will assign a structure of this type (smoking, cleanliness, time of day) with each part of the structure taking on either a 1 or 0 as defined above. We will compare to see if the structures for the two students match. If they are a perfect match we will assign a 0, if there is one field that is different we will assign a 1, if there are two fields that are different we will assign a 2, if all three fields are

different we will assign a 3. The goal is to minimize the total sum of all these roommate comparisons. We will pick the final matching that has the lowest overall cost.

Assumptions:

In order to feasibly complete this project, we made the following assumptions:

1. The selected sample of 145 students is representative of the whole 1921 students.
2. There are enough rooms to house all incoming Freshman.
3. Dorm types are preferred over room types.
4. Rooms are not assigned as applications come in, but rather as a whole once all applications have been submitted.
5. A nonsmoker cannot room with someone that's a smoker and males cannot room with females and vice versa.
6. Only three conditions are deemed important by the students in finding a good roommate (smoking/non-smoking, cleanliness, and morning or night person).

Implementation (Dorm/room type assignment):

Methodology

We first created a 145 by 145 cost matrix where each row consists of a student id and each column consists of the different dorms with room types (eg: Mudge Prime Double). The number of columns for a particular room type is equivalent to the number of people occupying that type of room in the dorm. For instance, a Mudge Prime Double would have two columns entries as there are two students occupying a double.

For each student ID, a cost was assigned to each dorm and room type for that student id depending on the student's preferences. The method in which the costs were assigned is described in more detail below. Once the costs were assigned for each student, we created the overall cost matrix for our Hungarian algorithm.

The Hungarian algorithm was then run on this entire cost matrix to determine an assignment of students to a dorm and room type. We implemented the Hungarian algorithm in Python using the Munkres package. Given a row and a column number corresponding to the student id and the dorm/room type, the algorithm that we implemented outputs a list of row column tuples which is the assignment of the student to the dorm and room type. The algorithm also outputs the individual costs for each such student assignment and the cumulative cost for all the student assignments put together. The cost structure is now described in more detail in the next section.

Cost Assignment:

The 145 by 145 cost matrix was done manually. Each student's dorm preference was taken into consideration to create a guideline for how the costs were assigned. Figure 1 below shows the values assigned to the corresponding row and column of the matrix.

Figure 1 : Cost assignment for students' 1st - 5th choice and the corresponding non-selected room type for that same dorm.

Condition	Cost				
	1st Preference	2nd Preference	3rd Preference	4th Preference	5th Preference
	1	4	9	16	25
Prime <-> Standard	12	24	36	48	60
Not Single <-> Not Single	24	48	72	96	120
Not Single <-> Single	36	72	108	144	180

For a better understanding of figure 1, we can look at an example. Suppose a student's choices were as following:

Ex. 1:

1st Preference	2nd Preference	3rd Preference	4th Preference	5th Preference
Standard Double - Donner	Standard Double - Hamerschlag (Males Only)	Stever Double - Stever	Standard Double - Mudge	Standard Double - Scobell (Males Only)

We would assign the student's first preference, Donner Standard Double, with a cost of 1. Seeing that he has no other preferences in Donner, we would assign the rest of the donner rooms looking at the values under the first preference column:

- Donner Standard Single -> 1st Preference -> Not Single <-> Single -> Cost = 36
- Donner Reduced Rate Triple -> 1st Preference -> Not Single <-> Not Single -> Cost = 24

Similarly, his second preference, Standard Double - Hamerschlag, will have a cost of 4. Seeing that none of his other preferences have Hamerschlag, the only other type of room in Hamerschlag, Standard Single, will have a cost of 72.

- Hamerschlag Standard Single -> 2nd Preference -> Not Single <-> Single -> Cost = 72

The third through fifth preferences and their corresponding room type cost assignments were done through the same procedure.

If a student has the same dorm type for multiple preferences, then the cost value assigned to other room types in that dorm followed the rules for the more/most preferred. For example, if a student has Mudge Standard Double as their second preference and Mudge Prime Double as their third preference, then the other room types' values will be determined looking at the second preference column.

If the student we're looking at is male, as appears to be the case in the example above, rooms from female only dorms (ex. McGill) were assigned a cost of 1000 to ensure that they were not placed into them. Similarly, to ensure that females were not placed into male only dorms, male only dorms (ex. Hamerschlag, Scobell) were assigned a cost of 1000.

Figure 2 : Costs for dorms not mentioned in the student's five preferences.

Condition	Cost
Same type of room	200
Prime <-> Standard	220
Not Single <-> Not Single	240
Single <-> Not Single	260

Looking back the example above, dorms such as Morewood E Tower and Boss were not mentioned. The cost value for these dorms will be assigned comparing the student's first preference room type with the listed conditions. In this case, the room type under the student's first preference is a "Standard Double." The room types in Boss are Prime Single and Prime Double.

- Boss Prime Single -> Not single <-> Single -> Cost = 260
- Boss Prime Double -> Standard <-> Prime -> Cost = 220

The other non-preferred room types were assigned following the same process.

Comparison (Hungarian vs. Greedy):

After forming a 145 by 145 square cost matrix, the only task left was to run the Hungarian algorithm with it. The python algorithm we acquired gives out the accumulated cost for each entry in the matrix after printing out each assignment in a single line. Thus, we could instantly find out the total cost of our assignment, which turned out to be 4133. However, we had to manually make a change to our assignment to ensure that a single gender consists a

single room. This was an inevitable process which could not be prevented given our resource. Fortunately, we only had to make three swaps and raise the cost as little as possible. After both program run and the manual process, the total cost of our method resulted in 4,688 (4133 + 555(swapping cost)).

Now that we have a quantitative result of our method, it is time to see how effective the method is compared with the current method employed by the Housing Service. Since we have already formed a cost matrix and we know the actual assignment for our samples, we could easily add up the actual total cost, which turned out be 10158. Using the Hungarian algorithm and the manual process resulted in 53.8% reduction in the total cost. Since the cost is an arbitrary term designed to represent a satisfaction level of the student groups, we cannot interpret the meaning in a more qualitative sense. However, it is obvious how effective our method is in terms of quantitative results. We can conclude that our method yields the satisfaction level of the entire freshman class that is twice as large as that yielded from the current method.

The result surprised us with the astonishing fact that finding an optimal solution is not always applicable in the real-world issues. The Housing Service employs the ‘greedy’ method to facilitate the deposit submission from the students. Thus, the method accomplishes both tasks of satisfying the need of students and that of the school. Since our method only aims to maximize the satisfaction of the students, we are able to produce effective result. We have successfully shown that there is definitely a way to raise the satisfaction level of the students if the Housing Service intends to. However, we also need to be careful not to conclude our method as the most ‘optimal’ solution since ours misses other important factors that can be

integral to the administration of the school. Also, when interpreting our result, one must not forget that we are only dealing with students who have not choses roommates beforehand. For those with matching roommates, we concede that the current method of the Housing Service is in fact optimal.

Now that we have shown the effectiveness of our dorm room/type assignment, we wanted to proceed and make further improvement to the assignment process as a whole by considering the roommates assignment based on the questions students answered. While this second task cannot result in a quantitative manner that can be compared with the current method – as we were not given resource indicating the actual roommate assignment by the Housing Service – the task will provide a possible additional improvement to our dorm/type assignment. The following text will elaborate on our second task.

Implementation (Swapping Algorithm for Roommate Assignment):

We have now assigned each of the students to a dorm and a room type using the Hungarian algorithm. Now we have a groups of students and we need to assign roommates within these groups. For example we had 18 males assigned to a Hammerschlag Standard Double room and now within these 18 people we need to make pairs of two by checking their compatibility. After using the Hungarian algorithm there were a handful of cases where we had an uneven number of males and females assigned to rooms. For example for the 2 available Henderson Prime Doubles we had 4 students assigned, 3 of whom were female and 1 of whom was male. This would not work. We ran the Hungarian algorithm on a subset of 144 students and we ran into 5 such problems. This is a very low number and were easily able to resolve these problems by either switching that student with another student who was assigned to the

same dorm or by switching a student within two dorms(by still maintaining the lowest possible cost). After doing the manual switches we continue with trying to find roommate pairings.

We use the following algorithm:

We are looking at three categories to determine compatibility smoking, cleanliness and whether or not they are a morning a night person. For each student we maintain a data type containing three values. The first value is either a 0 or a 1, a 0 indicating a nonsmoker and a 1 indicating a smoker. This category carries the most weight because we cannot assign a smoker and a nonsmoker together. The second value is for the morning and night category. When the students fill out the preference form there is a scale from 1 to 5 where each student's selects where they fall (1 indicates morning , 5 indicates night). In order to easily compare students and make it easier to find a match we assigned people who selected a 1 or a 2 as morning people, and 3,4,5 as night people. Morning people were assigned a 0 and night people were assigned a 1 in our data type. The final category indicates cleanliness. Once again in the preference form there is a scale from 1 to 5 (1 indicates very clean ,5 indicates *messy*). We assigned people who selected 1 or 2 as a clean person and 3 to 5 as messy Clean was assigned a 0 and *messy* was assigned a 1.

For each dorm type we split the students into males and females. Then we split those students into smokers and nonsmokers because there cannot be any overlap between the two groups. Once this is done we define a compare function that gives us a value either 0,1,2 indicating the number of similar categories. We already know that the smoking/non-smoking category will be the same so we are only comparing the other 2. If both categories are the same for both students a value of 0 will be assigned, if either category is different a value of 1 will be

assigned and if both categories are different a value of 2 will be assigned. We want to find roommates within the groups in order to minimize the total some of the values for each pair. This is a short summary of how we go about finding the roommate.

We create a list of all the students. We look at the first student and we search through all the other students looking for a match with a value of 0. If it cannot be found we look for a match of value 1, and if that cannot be found we finally look for a match of value 2. We maintain a set that contains all of the students who we've already used and we put these two students in that set. We can continue through the list and make sure we haven't already used the student in a previous pairing and apply the same procedure. In the end we will have a list of all the tuples with each tuple representing a pairing. For triples and quads we use the same algorithm but instead of looking for just one matching we look for two matchings for a triple or three matchings for a quad.

We apply this algorithm for every group of students (each group of students, male or female assigned to a specific dorm and room type) and find the pairings for each group. This algorithm finds the best pairings after the students have been assigned to a specific room type. The code for this algorithm and the results of our swap has been attached in the appendix. Each student is represented by their entry ID. After running the algorithm we were able to match every student with at least a match of 0 or a 1.

Improvements:

In order to make it feasible to complete the project in the time period given, we made quite a few assumptions. These assumptions, however, could be taken into account in the implementation of both the Hungarian Method and the Swapping Algorithm.

First of all, we selected 145 students out of the total 1921 total students in the Class of 2021+. The program could be used by CMU Housing and Dining Services for the entire class and possibly see an even larger difference in cost/happiness level. The 145 students that we selected were also all students that opted for a random roommate. This could be improved on by setting the previously formed roommates as a single unit and assigning dorms/rooms accordingly.

Moving on to the actual implementation of the program, the 145 by 145 cost matrix was created manually by us. A program could be written to decrease the amount of time needed for this step and also, ensure accuracy as there will be less room for error. Similarly, a program could be written for fixing errors after the Hungarian method has been run. Instead of switching males and females in dorms and rooms manually, a program could be created so that the optimal fix can be done with the click of a button.

Looking back at our assumptions, we only deemed three conditions important in the selection of roommates. A better roommate pairing could probably be formed by looking at all the questions answered by students in their housing application. In our project, we set students who put down 1 or 2 in their housing application as morning people and those who put down 3, 4, or 5 as night people. For improvement, we could set each of these responses as different answers and have a more “optimal” roommate assignment.

Conclusion:

We have successfully shown that a combination of a Hungarian method and our swapping algorithm effectively raises the satisfaction level of the students with more optimal assignment. With that being said, we could have very well ended our project at this point and conclude that the infamous problem of dormitory assignment has been solved. However, we now face a dilemma of whether our method is truly better than the current method by the Housing Service. When it comes to only the satisfaction level, then we can decisively say ours is better. However, as it is the case for most of the real world problems, there is almost always more than a single factor that we want to optimize. Because the incentive to submit deposit early is important to the school in terms of administrative aspect, the school has decided to employ current method over others. Likewise, many optimization problem requires a lot more consideration than a single factor to be truly applicable. On that note, we have found not only a satisfying result for our project but also a valuable lesson for future problems we will face.

Appendix A ***Code for Hungarian Algorithm:***

```
(*Matrix is too large to be shown*)

m = Munkres()
indexes = m.compute(matrix)
print_matrix(matrix, msg='Lowest cost through this matrix:')
total = 0
for row, column in indexes:
        value = matrix[row][column]
        total += value
        print '(%d, %d) -> %d' % (row, column, value)
        print 'total cost: %d' % total
```

Appendix B ***Code for Roommate Swapping:***

```
import math

def compare (x,y):
   totalDiff = math.fabs(x[1]- y[1]) + math.fabs(x[2]- y[2])
   return totalDiff

def sortSmokers(z):
   smokers = []
   nonsmokers = []
   for i in range(len(z)):
      if(z[i][0] == 0):
         nonsmokers.append(z[i])
      else:
         smokers.append(z[i])
   return (smokers, nonsmokers)

def findMatch(listOfStudents,usedIndices, currentStudent, maxDif):
   match = False
   matchIndex = -1
   i = 0
   while (i < len(listOfStudents) and not match):
      if ((i not in usedIndices) and i != currentStudent):
         if compare(listOfStudents[i], listOfStudents[currentStudent]) == maxDif :
            match = True
            matchIndex = i
      i = i+1
   return (match,matchIndex)

def findBest(z):
   usedIndices = set()
   pairs = []
```

```
    splitBySmoking = sortSmokers(z)
    smokers = splitBySmoking[0]
    nonsmokers = splitBySmoking[1]
    i = 0
    while (i < len(z)):
        if i not in usedIndices:
            if findMatch(z, usedIndices, i, 0)[0] == True:
                m = findMatch(z, usedIndices, i, 0)
                usedIndices.add(i)
                matchI = m[1]
                usedIndices.add(matchI)
                pairs.append((z[i][3], z[matchI][3]))
            else:
                if findMatch(z, usedIndices, i, 1)[0] == True:
                    m = findMatch(z, usedIndices, i, 1)
                    usedIndices.add(i)
                    matchI = m[1]
                    usedIndices.add(matchI)
                    pairs.append((z[i][3], z[matchI][3]))
                else:
                    m = findMatch(z, usedIndices, i, 2)
                    usedIndices.add(i)
                    matchI = m[1]
                    usedIndices.add(matchI)
                    pairs.append((z[i][3], z[matchI][3]))
        i = i +1
    return pairs

def findTriple(z, i, usedIndices):
    triple = []
    triple.append(i)
    if findMatch(z, usedIndices, i, 0)[0] == True:
                m = findMatch(z, usedIndices, i, 0)
                usedIndices.add(i)
                matchI = m[1]
                triple.append(matchI)
    else:
                if findMatch(z, usedIndices, i, 1)[0] == True:
                    m = findMatch(z, usedIndices, i, 1)
                    usedIndices.add(i)
                    matchI = m[1]
                    triple.append(matchI)
                else:
                    m = findMatch(z, usedIndices, i, 2)
                    usedIndices.add(i)
                    matchI = m[1]
                    triple.append(m[1])
    usedIndices.add(m[1])
```

```
    if findMatch(z, usedIndices, i, 0)[0] == True:
            m = findMatch(z, usedIndices, i, 0)
            usedIndices.add(i)
            matchI = m[1]
            triple.append(matchI)
    else:
            if findMatch(z, usedIndices, i, 1)[0] == True:
                m = findMatch(z, usedIndices, i, 1)
                usedIndices.add(i)
                matchI = m[1]
                triple.append(matchI)
            else:
                m = findMatch(z, usedIndices, i, 2)
                usedIndices.add(i)
                matchI = m[1]
                triple.append(m[1])
    return triple

def findBestTriple(z):
    usedIndices = set()
    triples = []
    splitBySmoking = sortSmokers(z)
    smokers = splitBySmoking[0]
    nonsmokers = splitBySmoking[1]
    i = 0
    while (i < len(z)):
        if i not in usedIndices:
            x = findTriple(z, i, usedIndices)
            triples.append((z[x[0]][3],z[x[1]][3],z[x[2]][3] ))
            usedIndices.add (x[0])
            usedIndices.add(x[1])
            usedIndices.add(x[2])
        i = i +1
    return triples

def findQuad(z, i, usedIndices):
    quad = []
    quad.append(i)
    if findMatch(z, usedIndices, i, 0)[0] == True:
            m = findMatch(z, usedIndices, i, 0)
            usedIndices.add(i)
            matchI = m[1]
            quad.append(matchI)
    else:
            if findMatch(z, usedIndices, i, 1)[0] == True:
                m = findMatch(z, usedIndices, i, 1)
                usedIndices.add(i)
                matchI = m[1]
```

```
                quad.append(matchI)
            else:
                m = findMatch(z, usedIndices, i, 2)
                usedIndices.add(i)
                matchI = m[1]
                quad.append(m[1])
    usedIndices.add(m[1])
    if findMatch(z, usedIndices, i, 0)[0] == True:
            m = findMatch(z, usedIndices, i, 0)
            usedIndices.add(i)
            matchI = m[1]
            quad.append(matchI)
    else:
            if findMatch(z, usedIndices, i, 1)[0] == True:
                m = findMatch(z, usedIndices, i, 1)
                usedIndices.add(i)
                matchI = m[1]
                quad.append(matchI)
            else:
                m = findMatch(z, usedIndices, i, 2)
                usedIndices.add(i)
                matchI = m[1]
                quad.append(m[1])
    usedIndices.add(m[1])
    if findMatch(z, usedIndices, i, 0)[0] == True:
            m = findMatch(z, usedIndices, i, 0)
            usedIndices.add(i)
            matchI = m[1]
            quad.append(matchI)
    else:
            if findMatch(z, usedIndices, i, 1)[0] == True:
                m = findMatch(z, usedIndices, i, 1)
                usedIndices.add(i)
                matchI = m[1]
                quad.append(matchI)
            else:
                m = findMatch(z, usedIndices, i, 2)
                usedIndices.add(i)
                matchI = m[1]
                quad.append(m[1])
    return quad

def findBestQuad(z):
    usedIndices = set()
    quads = []
    splitBySmoking = sortSmokers(z)
    smokers = splitBySmoking[0]
    nonsmokers = splitBySmoking[1]
```

```
i = 0
while (i < len(z)):
    if i not in usedIndices:
        x = findQuad(z, i, usedIndices)
        quads.append((z[x[0]][3],z[x[1]][3],z[x[2]][3],z[x[3]][3]))
        usedIndices.add (x[0])
        usedIndices.add(x[1])
        usedIndices.add(x[2])
        usedIndices.add(x[3])

    i = i +1
return quads
```

Appendix C Dorm Assignment Results

femaleBossPrimeDoubles = [(0,1,0,54811),(0,0,0,55186),(0,1,1,55525),(0,1,1,55691)]
femaleDonnerStandardDouble = [(0,1,0,54534),(0,1,1,54676),(0,0,0,54998),(0,1,0,55388)]
schlagStandardDouble =
[(0,1,0,53411),(0,1,1,54123),(0,1,0,54174),(0,1,1,54442),(0,1,1,54739),(0,1,0,55113),(0,1,1,552
60),(0,1,1,55344),(0,1,1,55366),(0,0,1,55385),(0,1,0,55738),(0,0,0,55784),(0,1,0,55954),(0,1,1,5
6066),(0,0,1,57828),(0,1,1,57914),(0,1,1,58043),(0,1,0,58243)]
mcgillPrimeDouble = [(0,0,0,54678),(0,1,0,55750),(0,1,0,55798),(0,1,0,58069)]
malemorewoodetowerStandardDouble =
[(0,1,1,49571),(0,1,0,55121),(0,1,1,55516),(0,1,1,56009)]
femalemorewoodetowerStandardDouble =
[(0,0,0,54066),(0,1,1,54897),(0,1,1,55511),(0,1,1,55751),(0,1,1,55835),(0,1,1,55900)]
femaleMorewoodPrimeDouble = [(0,1,0,54626),(0,0,0,55906),(0,1,0,57911),(0,1,0,57993)]
maleMudgePrimeDouble = [(0,1,0,54136),(0,1,1,55245),(0,1,0,57834),(0,1,1,57848)]
scobellStandardDouble =
[(0,1,1,53962),(0,1,1,55194),(0,1,1,55416),(0,1,1,55539),(0,1,1,55842),(0,1,1,58110)]
maleSteverStandardDouble =
[(0,1,1,53983),(0,0,0,54169),(0,1,0,54175),(0,1,0,54711),(0,1,0,54721),(0,1,1,54845)]
femaleSteverStandardDouble =
[(0,1,1,54018),(0,1,0,54892),(0,1,0,55695),(0,1,1,55804),(0,0,1,55934),(0,1,0,55936),(0,1,1,560
67),(0,1,0,57846),(0,1,1,57923),(0,1,1,58192)]
femaleDonnerReducedRateTrip =
[(0,0,0,54703),(0,0,0,54749),(0,0,1,55360),(0,1,0,57780),(0,1,0,58029),(0,1,1,58096)]
femaleMudgeReducedRateTrip =
[(0,0,1,54013),(0,1,1,54648),(0,1,1,55024),(0,1,1,56006),(0,0,0,57090),(0,1,1,58093)]
maleMudgeReducedRateTrip =
[(0,1,0,54195),(0,1,0,55054),(0,1,1,55225),(0,1,1,55667),(0,1,0,58218),(0,1,1,55466)]
maleResApartmentTrip =
[(0,1,0,54445),(0,1,0,54580),(0,1,0,54689),(0,1,0,55255),(0,1,1,57867),(0,1,1,54567)]
femaleMudgePrimeQuad =
[(0,1,0,53506),(0,1,1,54121),(0,1,1,54129),(0,1,0,54496),(0,1,1,54701),(0,1,1,55271),(0,1,1,554
06),(0,1,1,58097)]

#Example Print Statement:
print(findBest(femaleSteverStandardDouble))

Appendix C Final Roommate Swapping Results:

F Boss Prime Double
(54811, 55186), (55525, 55691)
M Boss Prime Double
(57779,57833)
M Boss Prime Single
(55512)
F Boss Prime Single
(55522)
F Donner Reduced Rate Triple
(54703, 54749, 55360), (57780, 58029, 58096)
M Donner Reduced Rate Triple
(55292,55514,55990)
F Donner Standard Double
(54534, 55388), (54676, 54998)
M Donner Standard Single
(54523)
M Hammerschlag Standard Double

(53411, 54174), (54123, 54442), (54739, 55260), (55113, 55738), (55344, 55366), (55385, 57828), (55784, 55954), (56066, 57914), (58043, 58243)
M Hammerschlag Standard Single
(55410),(58023)
M Henderson Prime Double
(54793,55599)
F Henderson Prime Double
(55473,58078)
F McGill Prime Double
(54678, 55750), (55798, 58069)
M Morewood ETower Standard Double
(49571, 55516), (55121, 56009)
F Morewood ETower Standard Double
(54066, 54897), (55511, 55751), (55835, 55900)
F Morewood ETower Standard Single
(54608),(55892)
M Morewood Prime Double
(47440, 54804)
F Morewood Prime Double
(54626, 57911), (55906, 57993)
F Morewood Prime Triple

(55008,55135,55601)

M **Mudge Prime Double**

(54136, 57834), (55245, 57848)

F **Mudge Prime Double**

(54586,55182)

F **Mudge Prime Quad**

(53506, 54496, 54121, 54129), (54701, 55271, 55406, 58097)

M **Mudge Prime Single**

(54866),(55124),(55212),(55472)

F **Mudge Reduced Rate Triple**

(56006, 54648, 55024), (54013, 58093, 57090)

M **Mudge Reduced Rate Triple**

(54195, 55054, 58218), (55225, 55667, 55466)

F **Mudge Standard Double**

(55696,55727)

M **Res Apartment Triple**

(54445, 54580, 54689), (55255, 57867, 54567)

F **Res Efficency Apartment Double**

(58105,58124)

M **Res Prime Efficiency Triple**

(44319,55168,57866)

M **Scobell Standard Double**

(53962, 55194), (55416, 55539), (55842, 58110)

M **Scobell Standard Single**

(54900),(55138), (55669)

F **Shirley Reduced Rate Triple**

(54614,55021,55741)

M **Shirley Reduced Rate Triple**

(55234,55272,55557)

M **Stever Standard Double**

(53983, 54845), (54169, 54175), (54711, 54721)

F **Stever Standard Double**

(54018, 55804), (54892, 55695), (55934, 56067), (55936, 57846), (57923, 58192)

Citations:

Munkre's Package Documentation :

http://software.clapper.org/munkres/

https://pypi.python.org/pypi/munkres/