Baseball Travelling Problem

David Michalowicz, EunJi Shim, Jialun Xu

08

**Fall**

**Abstract**

The purpose of this project is to optimize the total distance Major League Baseball (MLB) teams have to travel each regular season before the playoffs. We minimize the total distance by first simplifying the problem and solving for 4 teams in the same division. Then, we form an integer programming problem for all teams. Since integer programming problem is too complex, we create a facile solution that looks at improving the old schedule.

**Introduction**

MLB is composed of 15 teams in American League (AL) and 15 teams in National League (NL). Out of 30 MLB teams, 29 teams are located in the United States and 1 team in Canada. Each MLB team plays 5 to 7 games per week with a total of 162 games in a regular season. In order to reduce traveling, teams do not play each team the same number of times. Instead, they play more games against teams in their division than teams far from their home. However, it is still important to find an optimal schedule that will further minimize the total distance teams have to travel since having to travel shorter distances can give players more time to rest and prepare for their upcoming games. In order to do this, we first simplify the problem by making assumptions that do not exist in a real regular season.

**Assumptions**

For this research, we make assumptions to make our research problem more solvable. Note that it these assumptions are a little bit unrealistic and that they are not satisfied in a real regular season.

1. Number of home games equals the number of away games
2. Each team plays the same number of games
3. Every team plays 1 game per day, 7 games per week
4. Travel time and travel costs are never an issue.

**Approaches**

We take three different approaches to solve our research problem. The first approach is called 4 Team Problem. This method is very simple as we take only four teams and use a recursive algorithm to deduce the optimal schedule for these 4 teams. Next approach is the most complex one: integer programming. Instead of looking at only 4 teams, integer programming attempts to calculate the optimal schedule for all 30 MLB teams. Lastly, we use swapping method to find an improved schedule from our old schedule. We look at each of these method in more detail in the following sections.

**First Approach: 4 Team Problem**

In order to get a grasp on how to solve such a massive problem as scheduling 30 teams’ games over 162 days, we started by working on a small and manageable problem. The idea was to take only 4 teams and using a recursive algorithm deduce the optimal schedule which would minimize the total distance traveled by all teams. However, even on such a small scale the breadth of possible schedules is too great for a person to calculate on paper, so we wrote a program in python to solve the problem for us.

In this problem we made the following extra simplifying assumption: Each team plays each other team twice; once home and once away. This gives a total of 6 games played by each team (i.e. the schedule lasts 6 days).

In the code (which can be found below) the location of each team is represented in a list of lists, where the first sublist represents team 1, the second sublist represents team 2, and so on. Each sublist is a list of booleans containing a “True” at the index that team is currently at. For example, if we had four teams, the list [[True,False,False,False],[False,True,False,False],[False,False,True,False],[False,False,False,True]] would represent all teams being at home, whereas the list [[True,False,False,False],[True,False,False,False],[False,False,True,False],[False,False,True,False]] would represent both teams 1 and 2 being at team 1’s home and teams 3 and 4 being at team 3’s home.

Similarly, the distance between cities is represented as a list of lists, where the first sublist represents team 1 and the elements of that sublist are the distances from city 1 to the other cities. By default the distance from a city to itself is zero.

Lastly, in order to keep track of which teams need to play against each other, a list of lists of numbers is used to represent how many games each team has left to play at home. The first sublist represents team 1 and the elements of that list are how many times team one has to play the other teams. By default, a team plays itself zero times. For example, team 1’s sublist might look like: [0,1,3,2], meaning it has to play team two once at home, team three 3 times at home and team four twice at home.

In the top level call of the code, it looks through every possible pairing of teams and recursively solves for all of those pairings. Then, it takes the minimum distance out of all the recursive calls. In particular, when you have 4 teams there 12 ways to initially pair them all up:

* A@B and C@D
* A@B and D@C
* B@A and C@D
* B@A and D@C
* A@C and B@D
* A@C and D@B
* C@A and B@D
* C@A and D@B
* A@D and B@C
* A@D and C@B
* D@A and B@C
* D@A and C@B

Now, the code will return a number and a locations list. The number is equal to the total distance all the teams traveled after playing all 6 of their games *and* flying home after their last game. The locations list represents the locations of the teams on the *first day*, i.e. it tells you the first day of the schedule. In order to figure out the entire schedule, the locations list and games played list must be manually updated and plugged back into the functions. This must be done as many times as there are days. For the four-team problem this only amounted to 6 days which is not too bad.

Here is a complete example:

There are 4 teams: Boston (1), Tampa Bay (2), Baltimore (3) and New York (4)

The list of distances looks like this:

[[0,1340,400,215],[1340,0,967,1153],[400,967,0,187],[215,1153,187,0]]

The list of who has played who starts out like this:

[[0,1,1,1],[1,0,1,1],[1,1,0,1],[1,1,1,0]]

And the list of locations starts out with everyone at home:

[[True,False,False,False],[False,True,False,False],[False,False,True,False],[False,False,False,True]]

Now, when the code is run it returns the following:

[[True, False, False, False], [True, False, False, False], [False, False, False, True], [False, False, False, True]]

10889

This can be interpreted as following: after 6 days of playing the total travel distance of all teams sums to 10,889 miles. And on the first day of the schedule, Tampa Bay plays @ Boston and Baltimore plays @ New York.

The first schedule turns out to be:

Day 1: Tampa@Bos, Balt@NY

Day 2: Balt@Bos, Tampa@NY

Day 3: NY@Bos, Tampa@Balt

Day 4: NY@Balt, Bos@Tampa

Day 5: Bos@Balt, NY@Tampa

Day 6: Bos@NY, Balt@Tampa

Then all teams fly home.

Although there is no good way of testing the code, it is reasonable to believe that the schedule it produces is correct for the following reason: if we trace Tampa Bay’s schedule, we see that Tampa plays all of its away games first, starting in Boston. Then it plays all of its home games in a row on the last 3 days. This makes sense because Tampa Bay is the farthest city from all the others, so by playing all its away games in a row it minimizes the number of times it has to fly long distances going back and forth from home.

**Limitations of the 4-team method**

The four team problem can be solved for any arbitrary distances and an arbitrary number of games played by each team. However, the code is limited in that its run-time increases exponentially, meaning only small problems can be solved by the computer. We chose to solve the problem using a six day schedule simply because it produced quick results and allowed for a balanced schedule. Potentially the code could solve, say, a 12 day schedule just by changing the input values, but doing so takes the computer a few minutes. Anything above that could not be solved in a reasonable amount of time.

Futhermore, the code is limited in that it can *only* produce a schedule for 4 teams; this aspect, unlike the distances and games played, is hard-coded in. Therefore the code is limited in the sense that if you wanted to produce a schedule for more than 4 teams, the code would have to be totally restructured. But again, the 4-team problem was formulated not because we planned on extending it to the full 30-team problem, but rather to just show that the scheduling problem can in fact be solved.

\*Please see Appendix-1 for the code.

**Runtime (General)**

The runtime is based off of two things: the number of teams (n) and the number of days (d) in the schedule. In each top-level call, the computer must calculate the recursive solution for all the number of ways to pair up the teams. The number of ways to pair up all the teams in the top-level call is along the lines of n!/(n/2)! because the ways to pair them up can be thought of as writing out the teams in a line and then finding the number of ways to order them. For example, if you had 4 teams A,B,C,D the ordering ABCD would represent A@B and C@D whereas the ordering BACD would represent B@A and C@D. However, this is overcounting because the ordering ABCD represents the same scheduling as CDAB. In order to compensate for this we divide by (n/2)! But really this does not matter because we will just say that in the worst case it is order n!, or in big-O notation O(n!).

Now, the depth of the recursive calls is on the order of d. So if we picture the recursive calls branching out exponentially with a depth d, we get that the runtime is:

O((n!)^d)

This is extremely high for say, 30 teams and 162 days. Hence why the full problem cannot actually be solved even with a computer.

**Second Approach: Integer Programming**

In this approach, we will model the MLB regular season schedule with some simplifying assumptions. During the regular season, each team in MLB plays 162 games in total, usually in a form of three-game series (one game each day), occasionally, two- or four-game series. If a game is postponed, it is possible to have a five-game series or a doubleheader (two teams play against each other twice in a day). In formulating the integer programming problem, we will assume that there is no doubleheaders and that games are only played in a form of three-game series. In short, each team plays 54 series in a season with one series lasting three days.

Teams do not play everyday in the season; they sometimes get one or two days off a week. In calculating the total number of days in the season, we will use the data from the 2013 MLB schedule. The season lasted a total of 177 days from Mar. 31 to Sep. 29, excluding the 4 days leading up to and after the All Star game. On average, each team rested 16.3 days, which means the 54 series are played throughout 160.7 days. To simplify the numbers, we’ll round the total number of days up to 162 so it is a multiple of 3. Then we can formulate the problem where each team plays 54 series in 54 three-day periods. Note there is no break between series, but that is not the issue at hand and it is relatively easy to put in days for the athletes to rest.

Integer variable: xi,j,k=1 if it is a home game for Team i against Team j in period k, otherwise 0

for 1i, j30, 1k54

Constraints:

For any given team on any given day, it can only be playing against one team, either home or away:

i,k jixi,j,k+xj,i,k=1

For the whole season, each team has to play 54 series:

i ji, kxi,j,k+xj,i,k=54

Objective function:

minimizei, ji, kxi,j,k di,j,k , where di,j,k is a function that returns the distance from where Team j

is located on day k-1 to Team i's home city.

We observed that it is impossible to solve this problem using this method, therefore, we will be only formulating the integer programming version of the problem.

**Limitations of Integer Programming**

As much as it is complicated, it is virtually impossible to solve integer programming. Even if it is, it will take a very long time and require a complicated calculation. Hence, we create methods that are more solvable by simplifying our problems even further.

**Third Approach - An Improved Solution (but not optimal)**

This method attempts to improve a previous year’s schedule by swapping days. Although the same assumptions hold, this approach is perhaps the most realistic out of all the methods we have discussed. We create a table for 2013 regular season schedule with teams in row header and the days in column header. In each cell, we record rival teams that the team on a corresponding row has played against. Cell is green if it was a home game for the team on a corresponding row and is red if it was an away game. In order to satisfy the third assumption that all teams play every day, we compress the schedule such that the regular season lasts only 162 days, instead of 6 months. We further simplify the table by reducing all of the three game series to one such that each game actually represents three games. After all our modifications, we end up with a table that consists of 30 rows and 54 columns. Please see Appendix-3 for a sample schedule table.

The algorithm is fairly simple. First, we calculate total distance travelled each day by summing each column. Then, we choose two columns with the largest total distances. These are the days where teams had to travel the most collectively. We swap these days by swapping which teams to play. After the swap, we calculate new total distances for the swapped days and the days that fall right after those days. We compare our result to the previous result to see if there is an improvement. If there is an improvement, we conclude that our new schedule is more optimal than the last year’s schedule. If there is no improvement, we repeat the process until we find a more optimal schedule.

For instance, let’s say that Day 7 and Day 10 were the days where there has been most travelling. Let’s also say that Team A played Team B on Day 7 and Team C on Day 10 according to the last year’s schedule. Then, our swapping method will look at a modified schedule with Team A playing Team C on Day 7 and Team B on Day 10. For this new schedule, we will need to calculate new total distances for Day 7, 8, 10 and 11.

**Limitations of the Swapping Method**

Although swapping method presents a swift and easy way to optimize traveling, it doesn’t ultimately calculate the most optimal solution. In addition, the method requires to have an already built-in schedule as it only attempts to improve an already existing solution. Hence, it is impossible to build a completely new schedule using the swapping method.

**Conclusion**

As you can see, finding an optimal schedule that will minimize travelling is a very complicated task. Not only does it take such a long time, but also requires four main assumptions that you will not see in a real regular season. The three approaches we have discussed have their pros and cons: 4-team problem gives an accurate optimal schedule, but crashes easily for a large number of games; integer programming considers all MLB teams, but is virtually impossible to solve; and swapping method provides an easy way to reduce travelling, but does not immediately calculate the most optimal solution.

Although these methods have limitations, it will certainly be useful to implement one of these methods to find a better schedule for next year’s regular season. Reducing the distance teams have to travel will improve the quality of game.

**Appendix-1**

Python code for 4-team problem

import copy

#Helper functions

#Finds the smallest number in a list

def listMin(L):

   if L == []:

       return None

   else:

       a = (L.pop(0))[0]

       for i in xrange(len(L)):

           b = (L.pop(0))[0]

           if b < a:

               a = b

       return a

#In a list of pairs, finds the smallest number out of the first elements

#of the pairs

def listMin2(L):

   if L == []:

       return None

   else:

       a = (L.pop(0))

       c = a[0]

       for i in xrange(len(L)):

           b = (L.pop(0))

           d = b[0]

           if d < c:

               a = b

               c = d

       return a

##########    TESTS    ##########

#print listMin([9,2,3])

#print listMin([2])

#C = [(20,[[False,False,True,False],[False,False,False,True]]),(23,[[True,False,False,False],[False,True,False,False]]),(26,[[]])]

#C = [(2,1)]

#print listMin2(C)

#Determines if first team has played second team @ first team

def hasPlayedHome(homeTeam,awayTeam,played):

   if (played[homeTeam][awayTeam] == 0):

       return True

   return False

#Determines if first team has played second team @ second team

def hasPlayedAway(homeTeam,awayTeam,played):

   if (played[awayTeam][homeTeam] == 0):

       return True

   return False

#Extracts the index representing the input team's location

def extract(team,locations):

   for i in xrange(len(locations[team])):

       if locations[team][i] == True:

           return i

   return

#Determines the first other team not playing team A

def other1(n):

   if n == 1: return 2

   else: return 1

#Determines the second other team not playing team A

def other2(n):

   if (n == 1 or n == 2): return 3

   else: return 2

#distances = [[aa,ab,ac,ad],[ba,bb,bc,bd],[ca,cb,cc,cd],[da,db,dc,dd]]

#locations = [[true,false,false,false],[false,true,false,false],[false,false,true,false],[false,false,false,true]]

#played = [[0,1,1,1],[1,0,1,1],[1,1,0,1],[1,1,1,0]]

#distances[0] = list of distances from A to other teams

#locations[0] = list of truth values, where true is where the team is at

#played[0] = how many times team A has to play the other teams at home

#            a zero means they don't play anymore

#Solution function

def minDistance(distances,locations,played,day):

   distanceList = []  #used to collect the minimum of each recursive call

   #base case

   if day == 0:

       distance = 0

       for i in xrange(len(locations)):

           for j in xrange(len(locations[i])):

               if locations[i][j] == True:

                   #add up distance for each team to travel home on last day

                   distance += distances[i][j]

       return distance

   #loop through the following initial possibilities:

   #1 - team A plays team B at home, C plays D at home

   #2 - team A plays team B at home, D plays C at home

   #3 - team A plays team C at home, B plays D at home

   #4 - team A plays team C at home, D plays B at home

   #5 - team A plays team D at home, B plays C at home

   #6 - team A plays team D at home, C plays B at home

   #...same for A playing the other teams away

   for i in xrange(len(distances)):

       #if team A has not played team i at home:

       if not(hasPlayedHome(0,i,played)):

           #make copies of the original inputs so when we loop again they are unchanged

           DISTANCES = copy.deepcopy(distances)

           LOCATIONS = copy.deepcopy(locations)

           PLAYED = copy.deepcopy(played)

           DAY = copy.deepcopy(day)

           PLAYED[0][i] -= 1  #indicates that they have now played at A

           M = extract(i,LOCATIONS)

           P = extract(0,LOCATIONS)

           for k in xrange(len(LOCATIONS[i])):

               LOCATIONS[i][k] = False

           LOCATIONS[i][0] = True  #adjust team A's location as @A

           for k in xrange(len(LOCATIONS[0])):

               LOCATIONS[0][k] = False

           LOCATIONS[0][0] = True  #adjust team i's location as @A

           if not(hasPlayedHome(other1(i),other2(i),PLAYED)):

               #Now do the same updates for the other two teams, other1 and other2

               #here, they play @ other1

               PLAYED[other1(i)][other2(i)] -= 1

               N = extract(other2(i),LOCATIONS)

               Q = extract(other1(i),LOCATIONS)

               for j in xrange(len(LOCATIONS[other2(i)])):

                   LOCATIONS[other2(i)][j] = False

               LOCATIONS[other2(i)][other1(i)] = True

               for j in xrange(len(LOCATIONS[other1(i)])):

                   LOCATIONS[other1(i)][j] = False

               LOCATIONS[other1(i)][other1(i)] = True

               Lcopy = copy.deepcopy(LOCATIONS)

               #add up the travel distance of the four teams, then add the recursive call

               distanceList.append((DISTANCES[0][M]+DISTANCES[other1(i)][N]+DISTANCES[0][P]+DISTANCES[other1(i)][Q]+

                                   minDistance(DISTANCES,LOCATIONS,PLAYED,DAY-1),Lcopy))

           #repeat for same combination of teams playing, but with other1 and other2

           #now playing @ other2 instead of @ other1

           DISTANCES = copy.deepcopy(distances)

           LOCATIONS = copy.deepcopy(locations)

           PLAYED = copy.deepcopy(played)

           DAY = copy.deepcopy(day)

           PLAYED[0][i] -= 1

           M = extract(i,LOCATIONS)

           P = extract(0,LOCATIONS)

           for k in xrange(len(LOCATIONS[i])):

               LOCATIONS[i][k] = False

           LOCATIONS[i][0] = True

           for k in xrange(len(LOCATIONS[i])):

               LOCATIONS[0][k] = False

           LOCATIONS[0][0] = True

           if not(hasPlayedAway(other1(i),other2(i),PLAYED)):

               PLAYED[other2(i)][other1(i)] -= 1

               N = extract(other1(i),LOCATIONS)

               Q = extract(other2(i),LOCATIONS)

               for j in xrange(len(LOCATIONS[other1(i)])):

                   LOCATIONS[other1(i)][j] = False

               LOCATIONS[other1(i)][other2(i)] = True

               for j in xrange(len(LOCATIONS[other2(i)])):

                   LOCATIONS[other2(i)][j] = False

               LOCATIONS[other2(i)][other2(i)] = True

               Lcopy = copy.deepcopy(LOCATIONS)

               distanceList.append((DISTANCES[0][M]+DISTANCES[other2(i)][N]+DISTANCES[0][P]+DISTANCES[other2(i)][Q]+

                                   minDistance(DISTANCES,LOCATIONS,PLAYED,DAY-1),Lcopy))

       #now, if team A has played all teams at home, repeat the same process except with

       #team A playing all the other teams away

       elif not(hasPlayedAway(0,i,played)):

           DISTANCES = copy.deepcopy(distances)

           LOCATIONS = copy.deepcopy(locations)

           PLAYED = copy.deepcopy(played)

           DAY = copy.deepcopy(day)

           PLAYED[i][0] -= 1

           M = extract(0,LOCATIONS)

           P = extract(i,LOCATIONS)

           for k in xrange(len(LOCATIONS[0])):

               LOCATIONS[0][k] = False

           LOCATIONS[0][i] = True

           for k in xrange(len(LOCATIONS[i])):

               LOCATIONS[i][k] = False

           LOCATIONS[i][i] = True

           if not(hasPlayedHome(other1(i),other2(i),PLAYED)):

               PLAYED[other1(i)][other2(i)] -= 1

               N = extract(other2(i),LOCATIONS)

               Q = extract(other1(i),LOCATIONS)

               for j in xrange(len(LOCATIONS[other2(i)])):

                   LOCATIONS[other2(i)][j] = False

               LOCATIONS[other2(i)][other1(i)] = True

               for j in xrange(len(LOCATIONS[other1(i)])):

                   LOCATIONS[other1(i)][j] = False

               LOCATIONS[other1(i)][other1(i)] = True

               Lcopy = copy.deepcopy(LOCATIONS)

               distanceList.append((DISTANCES[i][M]+DISTANCES[other1(i)][N]+DISTANCES[i][P]+DISTANCES[other1(i)][Q]+

                                   minDistance(DISTANCES,LOCATIONS,PLAYED,DAY-1),Lcopy))

           DISTANCES = copy.deepcopy(distances)

           LOCATIONS = copy.deepcopy(locations)

           PLAYED = copy.deepcopy(played)

           DAY = copy.deepcopy(day)

           PLAYED[i][0] -= 1

           M = extract(0,LOCATIONS)

           P = extract(i,LOCATIONS)

           for k in xrange(len(LOCATIONS[0])):

               LOCATIONS[0][k] = False

           LOCATIONS[0][i] = True

           for k in xrange(len(LOCATIONS[i])):

               LOCATIONS[i][k] = False

           LOCATIONS[i][i] = True

           if not(hasPlayedAway(other1(i),other2(i),PLAYED)):

               PLAYED[other2(i)][other1(i)] -= 1

               N = extract(other1(i),LOCATIONS)

               Q = extract(other2(i),LOCATIONS)

               for j in xrange(len(LOCATIONS[other1(i)])):

                   LOCATIONS[other1(i)][j] = False

               LOCATIONS[other1(i)][other2(i)] = True

               for j in xrange(len(LOCATIONS[other2(i)])):

                   LOCATIONS[other2(i)][j] = False

               LOCATIONS[other2(i)][other2(i)] = True

               Lcopy = copy.deepcopy(LOCATIONS)

               distanceList.append((DISTANCES[i][M]+DISTANCES[other2(i)][N]+DISTANCES[i][P]+DISTANCES[other2(i)][Q]+

                                   minDistance(DISTANCES,LOCATIONS,PLAYED,DAY-1),Lcopy))

       else: continue

   DL = copy.deepcopy(distanceList)

   #For the top-level call, extract the minimum solution and print the team locations for that solution

   X = listMin2(DL)

   if day == 6:

       print X[1]

   return listMin(distanceList)

**Appendix-2**

Example for 4-team problem. First table shows the distance between four teams. Second table illustrates the optimal solution calculated by the 4-team problem code in Appendix-1. Red denotes away games and green denotes home games. The total distance for the optimal solution is 10,889.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Boston** | **Baltimore** | **New York** | **Tampa Bay** |
| **Boston** | 0 | 400 | 215 | 1340 |
| **Baltimore** |  | 0 | 187 | 967 |
| **New York** |  |  | 0 | 1153 |
| **Tampa Bay** |  |  |  | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Day 1** | **Day 2** | **Day 3** | **Day 4** | **Day 5** | **Day 6** |
| **Boston** | TB | Orioles | NY | TB | Orioles | NY |
| **Orioles** | NY | Boston | TB | NY | Boston | TB |
| **NY** | Orioles | TB | Boston | Orioles | TB | Boston |
| **TB** | Boston | NY | Orioles | Boston | NY | Orioles |

**Appendix-3**

A sample of what the previous schedule looks like before the swap.

****