Chess League Cup Pairings in the Netherlands

In this article we investigate the pairings for the Chess League Cup in the Netherlands. I have some doubts about the classification of the groupings. We use different techniques to solve this optimization problem:

* Power BI Visualization & Comparison
* R (Optimization using OMPR & Visualization)
* SQL Server (Geocoding address Chessclubs)

## Chess League Cup Explanation

As with football there is a League Cup and the regular national competition in the Netherlands. Though the regular competition is a single round robin event of 10 teams. Just 9 games leaves plenty of time to play individual tournaments or multiple foreign league competitions. In the regular competition you play with 10 players in the highest 2 classes and with 8 players per team in the lower classes.

In the League Cup you play with 4 players often during the week on the regular chess club evening of the home playing team. The League cup is based on a knockout system where around 100 teams participate. The teams are divided in 4 groups and the winners of each group go to the finals where the cup winner will be decided.

The 4 groups are divided in equal sized groups based on strength (based on level in national competition) and secondly on distance. So, one group in the North, East, West and South, right? Well not exactly, the Dutch chess teams are more or less divided according to population density in the country. The league cup games are played in the evening and can take over 4 hours. Short travelling distances are preferable. The teams in the North, South and East have to travel a lot as these areas are too small to form a single group. And yes, I happen to live and play chess in one of these regions…

## The Problem

But the distance penalty seemed to be a bit unfair with long driving’s (ok perspective of the Netherlands…). Of course an idea is to make 8 groups instead of 4 groups. Then play 3 games in the final weekend or 2 games in the final weekend and one battle with a neighboring group winner to qualify for the finals. The latter solution makes sense, but there are traditions for a reason.

Enough talk, let’s see whether the solution was optimized or whether my gut feeling was correct.

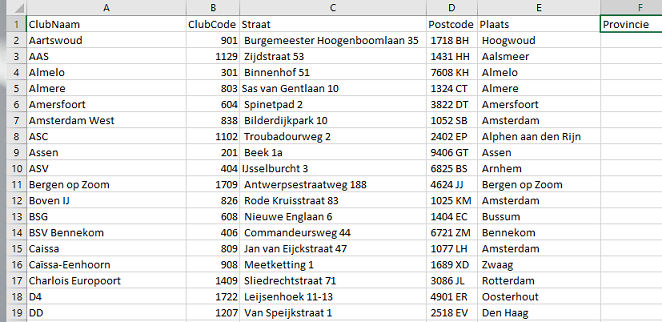
What do we need:

* the clubs and their playing venue location
* the national competition league of the club (for strength assessment)
* the average rating of the teams (for strength assessment)
* the group in the League Cup

As usually I work with the Microsoft stack, that’s where we start.

The first thing is to get the data of the chess clubs and optionally put it in a database, sql server. Unfortunately there is no webservice or something to get this data easily. I might could have asked for this data but I extracted the data from the website (at that point still [schaakbond.nl](www.schaakbond.nl) instead of the new website [schaken.nl](www.schaken.nl)).

I did some data wrangling/munging, sounds cooler than the just plain good old ETL/ELT. In this case it was mainly fixing small errors from the parsing. In the end I extracted the addresses of the 150 clubs in the national league and I looked up the addresses of the 10 champions from the regions. These are all the potential clubs which can participate in the league cup, though second teams are excluded.

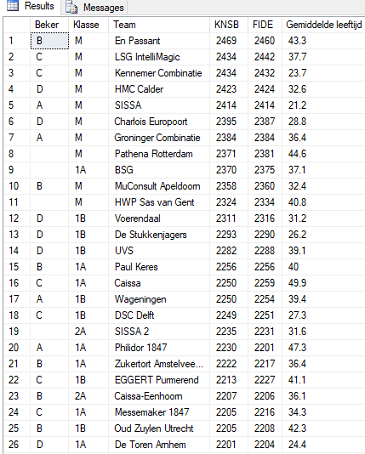


Address Chessclub

In the next figure the team information is presented.

Team information

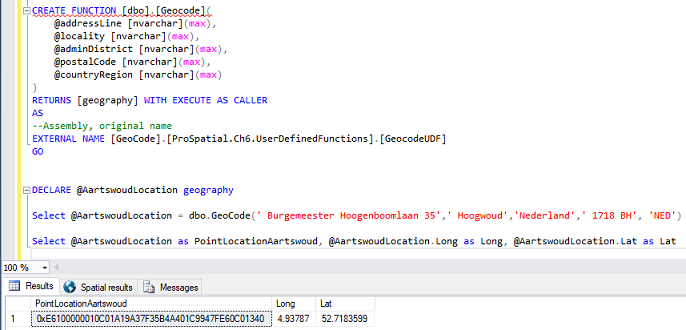
|  |  |
| --- | --- |
| Dutch name | Translation |
| Klasse | National league |
| Beker | Cup group |
| Team | TeamName (2 indicates second team of the club) |
| KNSB and FIDE | Average strength of the team |
| KNSB | National rating |
| FIDE | International rating |
| Gem. leeftijd | Average age of the team. As a BI-er I am very well aware of our habit to add data in the pipeline nobody needs… |



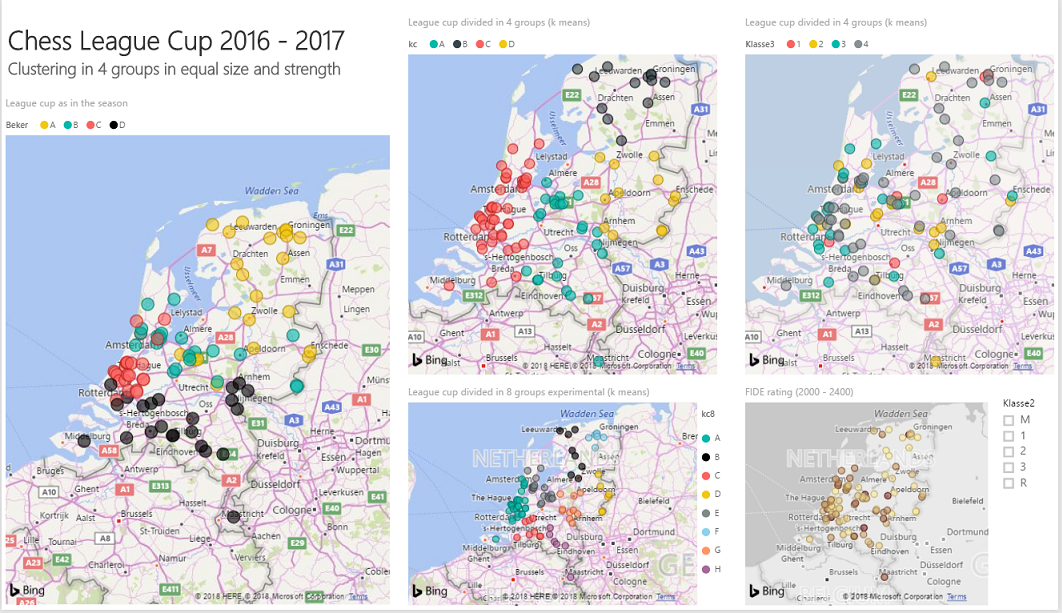
Chessclub Information

Not every team participates in the league cup, therefor the blanks. There can be different reasons for not participating.

The next step is to calculate the location (latitude, longitude) of the club based on the playing venue address of the club. I use the method illustrated by Alastair Aitchison in his book *Pro Spatial with SQL Server 2012*, <https://alastaira.wordpress.com/2012/05/04/geocoding-in-sql-server-with-the-bing-maps-locations-api/>. So a typical call to get the sql server geography object looks like this.



GeoCode SQL Server using Bingmaps

There are different ways to get the latitude longitude coordinate, but it is not our intention to discuss this in detail. Now we import the data in Power BI Desktop and create some initial visualization to get a rough idea about the data. 

The brief EDA gave some clues that improvement does not seem impossible. On the left visualization we see the actual 4 league cup groups as played in the season 2016-2017. In the middle we see two versions of the k-means clustering with 4 and 8 clusters (groups), using R within Power BI. In this example strength and equal sized cluster are not considered, but it gives an idea about the data. On the right visualizations we see the division of strength over the club locations. The bottom right visualization is based on FIDE rating, while the upper visualization is based on the national league class.

Once again, my personal annoyance was that each year we play in group south, including some pretty far western clubs, while living in the east. More concretely, is the league cup already the optimal solution or is there a solution which is better in all relevant aspects. So what should we take into consideration:

* average strength (based on national league class)
* group size (equal length over the groups)
* avg distance and avg squared distance in the group

Unfortunately, there is no magic button in Power BI to get the answer for this problem. Clustering algorithms have a different purpose, similarity, and cannot be used for this problem.

The solution can be found in linear optimization type of problems. Math is certainly not my expertise area, but let’s see whether we can come up with some results. As a Microsoft guy nowadays you need to be savvy with R, so lets fire up R studio!

I found a nice looking package from Dirk Schumacher, *OMPR*, Optimization Modelling Package. More information can be found [here](https://dirkschumacher.github.io/ompr/index.html). In general in works a bit like the package *Caret*, for Regression and Classification type of problems. It provides a universal front-end for many different optimization algorithms.

## R

After importing the data in R, we do a little preprocessing before building the model. In the specific year we have 93 clubs participating in the league cup. We set the number of groups to 4, and the size of the group to a minimum of 23 clubs and a maximum of 24 clubs. We use the same weight setting as in Power BI. But we add the maximum and minimum weight for a group to ensure equal strength groups.

At development time I used a subset, for example 20 clubs and 4 groups, to quickly test the solution. I played a bit with the *OMPR* package and tried different approaches to solve the problem.

Some less successful attempts have been tried with different models. They are within R script *Optimize ChessLeagueCupGroupings.R* in GitHub. One of the attempts was the famous TSP, Traveling Salesman Problem. It wasn’t particularly successful as the problem space grew massively. But the idea was to have 4 chains of clubs equal in size and strength. Unfortunately, testing it with just 20 cities already took too long, while with 10 cities and 2 groups it was ok.

Here we look into the final model. But first the preprocessing.

rm(list=setdiff(ls(), "mapNL"))  
  
library(ompr)  
library(ROI.plugin.glpk)  
library(ompr.roi)  
library(ROI)  
library(geosphere)  
library(tidyr)  
library(knitr)  
library(ggplot2)  
library(ggrepel)  
library(ggmap)

## Preprocessing

In the preprocessing we load the data from the already geocoded chessclubs, csv file, and set a few variables for the groupings.

source("Preprocessing ChessLeagueCup.R")

Create a map of the Netherlands (once).

load("output/mapNL.rda")

## Optimization Model

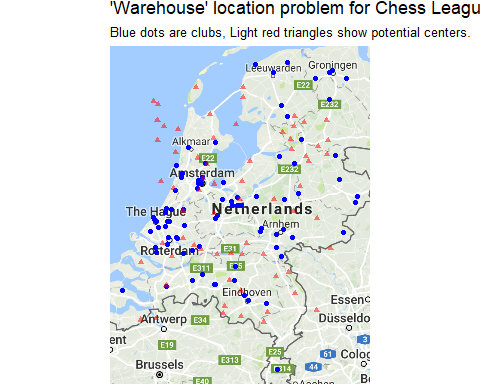
As a reminder we need to take into account similar strength and equal group sizes and reduce the overall distance between the clubs assigned to the 4 groups.

For our final solution a number of random points are choosen within the studyarea (added with the best points from previous runs). Then we choose the 4 most optimized points to create the 4 groups with a minimal (squared) distance from the clubs to the grouppoint (centroid approximation). Taking into account the constraints for equal strength and equal group sizes. The proposed solution is related to the following type of problem: finding the most optimal warehouse location to serve customers.

#Study area extent based on locations of the chessclub  
minLat <- min(ChessClubs20162017$Lat)  
maxLat <- max(ChessClubs20162017$Lat)  
minLong <- min(ChessClubs20162017$Long)  
maxLong <- max(ChessClubs20162017$Long)  
  
deltaLat <- maxLat - minLat  
deltaLong <- maxLong - minLong  
  
centroidLoc <- 96  
  
set.seed(12345)  
centroid\_locations <- data.frame(  
 IdCentroid = 1:centroidLoc,  
 LongCentroid = runif(centroidLoc) \* deltaLong + minLong, #x  
 LatCentroid = runif(centroidLoc) \* deltaLat + minLat #y  
)  
  
#Add top centroid from previous run  
df1 <- data.frame(IdCentroid = centroidLoc + 1, LongCentroid = 4.933878, LatCentroid = 52.38954)  
df2 <- data.frame(IdCentroid = centroidLoc + 2, LongCentroid = 6.404083, LatCentroid = 52.64013)  
df3 <- data.frame(IdCentroid = centroidLoc + 3, LongCentroid = 5.487537, LatCentroid = 51.88348)  
df4 <- data.frame(IdCentroid = centroidLoc + 4, LongCentroid = 4.409887, LatCentroid = 51.88037)  
  
centroid\_locations <- rbind(centroid\_locations, df1)  
centroid\_locations <- rbind(centroid\_locations, df2)  
centroid\_locations <- rbind(centroid\_locations, df3)  
centroid\_locations <- rbind(centroid\_locations, df4)  
  
#it was a bit too ambitious to have the study area crossing the North Sea, Belgium and Germany.  
#Lets trim it a bit more  
centroid\_locations <- centroid\_locations %>%  
 filter(LongCentroid > 4.1) %>%  
 filter(LongCentroid < 6.6) %>%  
 filter(LatCentroid > 51.2) %>%  
 filter(LatCentroid < 53.1)

We have now our random centroid locations. Let’s visualize our data.

p <- ggmap(mapNL, extent = 'device') +  
 geom\_point(data = ChessClubs20162017, aes(Long, Lat), color = "blue") +   
 geom\_point(data = centroid\_locations, aes(x = LongCentroid, y = LatCentroid), color = "red", alpha = 0.5, shape = 17) +  
 scale\_x\_continuous(limits = c(minLong, maxLong)) +  
 scale\_y\_continuous(limits = c(minLat, maxLat))   
  
p +   
# geom\_label\_repel(data = centroid\_locations, aes(x = LongCentroid, y = LatCentroid, label=as.character(IdCentroid)), box.padding #= 0.25, point.padding = 0.3, segment.color = 'grey50') +  
 ggtitle("'Warehouse' location problem for Chess League Cup", "Blue dots are clubs, Light red triangles show potential centers.")



The suggested centroid locations look reasonable.

## DistanceMatrix

We are almost ready, to start crafting the optimization model. But it is a handy to make an initial Distance Matrix and not calculate again and again in the model optimization function. A squared distance is used as a distance measure. So it tries too limit larger distances. The distance function is not the driving distance though (this could be sort of *geocoded* as well).

centroidLoc <- nrow(centroid\_locations)  
m1 <- as.matrix(ChessClubs20162017[,c("Long", "Lat")])  
m2 <- as.matrix(centroid\_locations[,c("LongCentroid", "LatCentroid")])  
md2 <- distm(m1, m2)  
  
distanceCost2 <- function(i, j) {  
 md2[i,j]^2  
}  
  
centroid\_locations$Id <- row\_number(centroid\_locations$IdCentroid) #As some point locations are removed, create a new Id field  
cl <- centroidLoc #a small variable names helps too read the model  
  
rm(m1, m2, centroidLoc, df1, df2, df3, df4, deltaLong, deltaLat) #cleanup

## Model

Finally! The model is easy to read and understand. It was one of the reasons for choosing this package.

model4 <- MIPModel() %>%  
 # 1 if i gets assigned to centroid\_location j otherwise 0  
 add\_variable(x[i, j], i = 1:n, j = 1:cl, type = "binary") %>%  
   
 #the centroid locations (dummy variable)  
 add\_variable(u[j], j = 1:cl, type = "binary") %>%  
   
 #exactly 4 groups are allowed  
 add\_constraint(sum\_expr(u[j], j = 1:cl) == 4) %>%   
   
 #minimize squared distance cost  
 set\_objective(sum\_expr(distanceCost2(i, j) \* x[i, j], i = 1:n, j = 1:cl), "min") %>%  
  
 # every chessclub needs to be assigned to exactly one group  
 add\_constraint(sum\_expr(x[i, j], j = 1:cl) == 1, i = 1:n) %>%   
  
 # each of the 4 groups has to be of equal size  
 add\_constraint(sum\_expr(x[i, j], i = 1:n) >= gMin \* u[j], j = 1:cl) %>%  
 add\_constraint(sum\_expr(x[i, j], i = 1:n) <= gMax \* u[j], j = 1:cl) %>%  
   
 #each of the 4 groups has to be of equal strength  
 add\_constraint(sum\_expr(x[i, j] \* weightLevel[i], i = 1:n) >= minTotalWeight \* u[j], j = 1:cl) %>%  
 add\_constraint(sum\_expr(x[i, j] \* weightLevel[i], i = 1:n) <= maxTotalWeight \* u[j], j = 1:cl)  
  
#Lets inspect the model created  
model4

## Mixed linear integer optimization problem  
## Variables:  
## Continuous: 0   
## Integer: 0   
## Binary: 6298   
## Model sense: minimize   
## Constraints: 362

Now we run the model using the glpk solver

result4 <- solve\_model(model4, with\_ROI(solver = "glpk", verbose = TRUE))

## <SOLVER MSG> ----  
## GLPK Simplex Optimizer, v4.47  
## 362 rows, 6298 columns, 31490 non-zeros  
## 0: obj = 0.000000000e+000 infeas = 9.700e+001 (94)  
## \* 253: obj = 1.236354690e+012 infeas = 2.274e-015 (1)  
## \* 500: obj = 8.712538744e+011 infeas = 0.000e+000 (0)  
## \* 1000: obj = 4.987494456e+010 infeas = 6.297e-016 (0)  
## \* 1170: obj = 3.919634122e+010 infeas = 4.441e-017 (0)  
## OPTIMAL SOLUTION FOUND  
## GLPK Integer Optimizer, v4.47  
## 362 rows, 6298 columns, 31490 non-zeros  
## 6298 integer variables, all of which are binary  
## Integer optimization begins...  
## + 1170: mip = not found yet >= -inf (1; 0)  
## + 2031: >>>>> 2.471864739e+011 >= 4.263027948e+010 82.8% (62; 0)  
## + 2922: mip = 2.471864739e+011 >= 4.538587812e+010 81.6% (115; 8)  
## + 3556: mip = 2.471864739e+011 >= 4.554063146e+010 81.6% (181; 9)  
## + 3992: >>>>> 1.727665160e+011 >= 4.554063146e+010 73.6% (215; 9)  
## + 4562: mip = 1.727665160e+011 >= 4.795663865e+010 72.2% (225; 94)  
## + 5207: mip = 1.727665160e+011 >= 4.865716627e+010 71.8% (290; 95)  
## + 5900: mip = 1.727665160e+011 >= 5.055920563e+010 70.7% (346; 97)  
## + 6340: >>>>> 1.696382545e+011 >= 5.055920563e+010 70.2% (391; 97)  
## + 7088: mip = 1.696382545e+011 >= 6.817468142e+010 59.8% (407; 135)  
## + 8305: mip = 1.696382545e+011 >= 6.906003480e+010 59.3% (463; 141)  
## + 9976: mip = 1.696382545e+011 >= 7.245957448e+010 57.3% (537; 147)  
## + 11925: mip = 1.696382545e+011 >= 7.448739326e+010 56.1% (583; 158)  
## + 13616: mip = 1.696382545e+011 >= 8.402818222e+010 50.5% (629; 173)  
## Time used: 60.0 secs. Memory used: 8.7 Mb.  
## + 15499: mip = 1.696382545e+011 >= 8.766373127e+010 48.3% (689; 182)  
## + 16904: mip = 1.696382545e+011 >= 8.858068210e+010 47.8% (753; 187)  
## + 18524: mip = 1.696382545e+011 >= 9.075606456e+010 46.5% (836; 194)  
## + 20138: mip = 1.696382545e+011 >= 9.194304204e+010 45.8% (927; 200)  
## + 21077: mip = 1.696382545e+011 >= 9.272320757e+010 45.3% (992; 203)  
## + 22282: mip = 1.696382545e+011 >= 9.440250899e+010 44.4% (1049; 211)  
## + 23491: mip = 1.696382545e+011 >= 9.510877569e+010 43.9% (1116; 216)  
## + 23916: >>>>> 1.684059921e+011 >= 9.510877569e+010 43.5% (1152; 216)  
## + 24915: mip = 1.684059921e+011 >= 9.555496191e+010 43.3% (1203; 235)  
## + 26387: mip = 1.684059921e+011 >= 9.623257085e+010 42.9% (1267; 243)  
## + 27587: mip = 1.684059921e+011 >= 9.665408087e+010 42.6% (1324; 247)  
## + 28975: >>>>> 1.639851247e+011 >= 9.675488984e+010 41.0% (1394; 252)  
## + 30420: mip = 1.639851247e+011 >= 9.696464215e+010 40.9% (1421; 335)  
## Time used: 120.0 secs. Memory used: 13.3 Mb.  
## + 32108: mip = 1.639851247e+011 >= 9.752051108e+010 40.5% (1511; 340)  
## + 33413: mip = 1.639851247e+011 >= 9.775433322e+010 40.4% (1591; 345)  
## + 34678: mip = 1.639851247e+011 >= 9.818190382e+010 40.1% (1648; 353)  
## + 35711: mip = 1.639851247e+011 >= 9.847701723e+010 39.9% (1689; 358)  
## + 36013: >>>>> 1.614383733e+011 >= 9.847701723e+010 39.0% (1713; 358)  
## + 37110: mip = 1.614383733e+011 >= 9.870858723e+010 38.9% (1731; 478)  
## + 38455: mip = 1.614383733e+011 >= 9.944889530e+010 38.4% (1811; 488)  
## + 39744: mip = 1.614383733e+011 >= 9.985416170e+010 38.1% (1896; 495)  
## + 41245: mip = 1.614383733e+011 >= 1.003874466e+011 37.8% (1988; 502)  
## + 43417: mip = 1.614383733e+011 >= 1.009366198e+011 37.5% (2114; 513)  
## + 45836: >>>>> 1.606941897e+011 >= 1.017297278e+011 36.7% (2205; 528)  
## + 47689: mip = 1.606941897e+011 >= 1.021944294e+011 36.4% (2257; 589)  
## + 49187: >>>>> 1.554849223e+011 >= 1.022479093e+011 34.2% (2364; 594)  
## + 49957: mip = 1.554849223e+011 >= 1.022833152e+011 34.2% (2188; 1000)  
## Time used: 180.0 secs. Memory used: 20.8 Mb.  
## + 51395: mip = 1.554849223e+011 >= 1.030384324e+011 33.7% (2265; 1009)  
## + 53637: mip = 1.554849223e+011 >= 1.036169461e+011 33.4% (2331; 1025)  
## + 55227: mip = 1.554849223e+011 >= 1.041166757e+011 33.0% (2393; 1035)  
## + 56391: mip = 1.554849223e+011 >= 1.043157819e+011 32.9% (2448; 1041)  
## + 58461: mip = 1.554849223e+011 >= 1.046022576e+011 32.7% (2512; 1058)  
## + 60184: mip = 1.554849223e+011 >= 1.051108828e+011 32.4% (2609; 1068)  
## + 61512: mip = 1.554849223e+011 >= 1.054012509e+011 32.2% (2650; 1078)  
## + 62327: mip = 1.554849223e+011 >= 1.054522680e+011 32.2% (2685; 1082)  
## + 63601: mip = 1.554849223e+011 >= 1.057714949e+011 32.0% (2732; 1091)  
## + 65103: mip = 1.554849223e+011 >= 1.063379951e+011 31.6% (2787; 1103)  
## + 66495: mip = 1.554849223e+011 >= 1.065841518e+011 31.5% (2840; 1114)  
## + 67759: mip = 1.554849223e+011 >= 1.068237909e+011 31.3% (2899; 1123)  
## Time used: 240.0 secs. Memory used: 26.8 Mb.  
## + 68502: mip = 1.554849223e+011 >= 1.069482586e+011 31.2% (2946; 1126)  
## + 69522: mip = 1.554849223e+011 >= 1.070962505e+011 31.1% (2986; 1134)  
## + 70481: >>>>> 1.553848283e+011 >= 1.072214176e+011 31.0% (3032; 1138)  
## + 71942: mip = 1.553848283e+011 >= 1.075968841e+011 30.8% (3066; 1166)  
## + 73241: mip = 1.553848283e+011 >= 1.078764035e+011 30.6% (3112; 1176)  
## + 74974: mip = 1.553848283e+011 >= 1.083367423e+011 30.3% (3163; 1190)  
## + 76564: mip = 1.553848283e+011 >= 1.090389626e+011 29.8% (3205; 1208)  
## + 77532: mip = 1.553848283e+011 >= 1.092263095e+011 29.7% (3276; 1214)  
## + 79207: mip = 1.553848283e+011 >= 1.100547247e+011 29.2% (3323; 1238)  
## + 81232: mip = 1.553848283e+011 >= 1.103763412e+011 29.0% (3376; 1253)  
## + 83827: mip = 1.553848283e+011 >= 1.109787078e+011 28.6% (3435; 1274)  
## + 85084: mip = 1.553848283e+011 >= 1.111471771e+011 28.5% (3510; 1280)  
## Time used: 300.0 secs. Memory used: 35.4 Mb.  
## + 86436: mip = 1.553848283e+011 >= 1.113853677e+011 28.3% (3575; 1292)  
## + 88997: mip = 1.553848283e+011 >= 1.123147695e+011 27.7% (3631; 1317)  
## + 91337: mip = 1.553848283e+011 >= 1.130049725e+011 27.3% (3681; 1339)  
## + 93469: mip = 1.553848283e+011 >= 1.136999468e+011 26.8% (3757; 1357)  
## + 94824: mip = 1.553848283e+011 >= 1.139443486e+011 26.7% (3810; 1366)  
## + 96126: mip = 1.553848283e+011 >= 1.142506630e+011 26.5% (3834; 1381)  
## + 97574: mip = 1.553848283e+011 >= 1.144623689e+011 26.3% (3871; 1399)  
## + 99494: mip = 1.553848283e+011 >= 1.149795559e+011 26.0% (3899; 1425)  
## +101373: mip = 1.553848283e+011 >= 1.154568496e+011 25.7% (3940; 1446)  
## +103451: mip = 1.553848283e+011 >= 1.157636978e+011 25.5% (3978; 1462)  
## +105769: mip = 1.553848283e+011 >= 1.162518370e+011 25.2% (4012; 1483)  
## +106963: mip = 1.553848283e+011 >= 1.165883901e+011 25.0% (4055; 1497)  
## Time used: 360.0 secs. Memory used: 45.6 Mb.  
## +108910: mip = 1.553848283e+011 >= 1.170705245e+011 24.7% (4083; 1523)  
## +110559: mip = 1.553848283e+011 >= 1.173438975e+011 24.5% (4123; 1535)  
## +111755: mip = 1.553848283e+011 >= 1.176096484e+011 24.3% (4154; 1549)  
## +113511: mip = 1.553848283e+011 >= 1.179893102e+011 24.1% (4199; 1576)  
## +114955: mip = 1.553848283e+011 >= 1.183835063e+011 23.8% (4214; 1601)  
## +116946: mip = 1.553848283e+011 >= 1.185077799e+011 23.7% (4254; 1623)  
## +119340: mip = 1.553848283e+011 >= 1.190824451e+011 23.4% (4290; 1650)  
## +121185: mip = 1.553848283e+011 >= 1.193658125e+011 23.2% (4328; 1677)  
## +121734: mip = 1.553848283e+011 >= 1.193954750e+011 23.2% (4368; 1680)  
## +122627: mip = 1.553848283e+011 >= 1.195599517e+011 23.1% (4398; 1689)  
## +123834: mip = 1.553848283e+011 >= 1.198451432e+011 22.9% (4402; 1711)  
## +125361: mip = 1.553848283e+011 >= 1.200672885e+011 22.7% (4421; 1733)  
## Time used: 420.0 secs. Memory used: 52.8 Mb.  
## +127408: mip = 1.553848283e+011 >= 1.203687891e+011 22.5% (4456; 1751)  
## +129594: mip = 1.553848283e+011 >= 1.207225134e+011 22.3% (4473; 1779)  
## +132590: mip = 1.553848283e+011 >= 1.213044660e+011 21.9% (4508; 1821)  
## +133130: mip = 1.553848283e+011 >= 1.214555497e+011 21.8% (4564; 1828)  
## +134280: mip = 1.553848283e+011 >= 1.214901120e+011 21.8% (4608; 1836)  
## +135511: mip = 1.553848283e+011 >= 1.218082965e+011 21.6% (4617; 1854)  
## +136211: mip = 1.553848283e+011 >= 1.219380593e+011 21.5% (4652; 1862)  
## +138190: mip = 1.553848283e+011 >= 1.220171694e+011 21.5% (4720; 1877)  
## +139761: mip = 1.553848283e+011 >= 1.221683034e+011 21.4% (4754; 1895)  
## +141687: mip = 1.553848283e+011 >= 1.225602105e+011 21.1% (4775; 1919)  
## +143804: mip = 1.553848283e+011 >= 1.226442635e+011 21.1% (4815; 1940)  
## +145937: mip = 1.553848283e+011 >= 1.229762187e+011 20.9% (4863; 1963)  
## Time used: 480.0 secs. Memory used: 61.0 Mb.  
## +148726: mip = 1.553848283e+011 >= 1.232676599e+011 20.7% (4920; 1995)  
## +152869: mip = 1.553848283e+011 >= 1.236831919e+011 20.4% (4950; 2045)  
## +156126: mip = 1.553848283e+011 >= 1.241630019e+011 20.1% (4973; 2091)  
## +158492: mip = 1.553848283e+011 >= 1.244146265e+011 19.9% (5047; 2116)  
## +161051: mip = 1.553848283e+011 >= 1.246639119e+011 19.8% (5135; 2139)  
## +164840: mip = 1.553848283e+011 >= 1.251062689e+011 19.5% (5175; 2196)  
## +168545: mip = 1.553848283e+011 >= 1.253762573e+011 19.3% (5261; 2236)  
## +172319: mip = 1.553848283e+011 >= 1.259064366e+011 19.0% (5328; 2298)  
## +177920: mip = 1.553848283e+011 >= 1.264647315e+011 18.6% (5343; 2392)  
## +178334: >>>>> 1.544210002e+011 >= 1.264647315e+011 18.1% (5393; 2393)  
## +182388: mip = 1.544210002e+011 >= 1.269527979e+011 17.8% (5326; 2641)  
## +188133: mip = 1.544210002e+011 >= 1.274890451e+011 17.4% (5356; 2736)  
## +193928: mip = 1.544210002e+011 >= 1.279462696e+011 17.1% (5391; 2825)  
## Time used: 540.0 secs. Memory used: 73.5 Mb.  
## +198782: mip = 1.544210002e+011 >= 1.282897155e+011 16.9% (5471; 2885)  
## +204337: mip = 1.544210002e+011 >= 1.287042796e+011 16.7% (5526; 2963)  
## +209375: mip = 1.544210002e+011 >= 1.291392939e+011 16.4% (5599; 3037)  
## +215146: mip = 1.544210002e+011 >= 1.295322576e+011 16.1% (5645; 3127)  
## +220248: mip = 1.544210002e+011 >= 1.299841889e+011 15.8% (5699; 3203)  
## +225801: mip = 1.544210002e+011 >= 1.304615799e+011 15.5% (5738; 3299)  
## +231478: mip = 1.544210002e+011 >= 1.308889577e+011 15.2% (5763; 3403)  
## +237205: mip = 1.544210002e+011 >= 1.312169871e+011 15.0% (5806; 3489)  
## +239938: mip = 1.544210002e+011 >= 1.314731715e+011 14.9% (5817; 3542)  
## +241928: mip = 1.544210002e+011 >= 1.315930367e+011 14.8% (5834; 3572)  
## +243192: mip = 1.544210002e+011 >= 1.315930367e+011 14.8% (5863; 3588)  
## +244937: mip = 1.544210002e+011 >= 1.316703369e+011 14.7% (5910; 3608)  
## Time used: 600.0 secs. Memory used: 89.6 Mb.  
## +246593: mip = 1.544210002e+011 >= 1.318159149e+011 14.6% (5928; 3638)  
## +249731: mip = 1.544210002e+011 >= 1.320118165e+011 14.5% (5936; 3696)  
## +250477: mip = 1.544210002e+011 >= 1.320133353e+011 14.5% (5982; 3702)  
## +252612: mip = 1.544210002e+011 >= 1.321203528e+011 14.4% (6054; 3728)  
## +255586: mip = 1.544210002e+011 >= 1.322937001e+011 14.3% (6101; 3772)  
## +258847: mip = 1.544210002e+011 >= 1.323919239e+011 14.3% (6164; 3817)  
## +262723: mip = 1.544210002e+011 >= 1.325661054e+011 14.2% (6205; 3874)  
## +265634: mip = 1.544210002e+011 >= 1.326615411e+011 14.1% (6274; 3911)  
## +266826: mip = 1.544210002e+011 >= 1.327986933e+011 14.0% (6330; 3927)  
## +269017: mip = 1.544210002e+011 >= 1.328143609e+011 14.0% (6389; 3961)  
## +270624: >>>>> 1.540105216e+011 >= 1.329591942e+011 13.7% (6430; 3986)  
## +273766: mip = 1.540105216e+011 >= 1.330629414e+011 13.6% (6370; 4169)  
## Time used: 660.0 secs. Memory used: 98.7 Mb.  
## +276471: mip = 1.540105216e+011 >= 1.332830944e+011 13.5% (6358; 4222)  
## +279765: mip = 1.540105216e+011 >= 1.334140309e+011 13.4% (6429; 4274)  
## +283838: mip = 1.540105216e+011 >= 1.336950182e+011 13.2% (6503; 4330)  
## +287282: mip = 1.540105216e+011 >= 1.338510511e+011 13.1% (6539; 4399)  
## +292652: mip = 1.540105216e+011 >= 1.342027238e+011 12.9% (6521; 4515)  
## +296889: >>>>> 1.480105129e+011 >= 1.344432148e+011 9.2% (6543; 4590)  
## +299389: mip = 1.480105129e+011 >= 1.346597125e+011 9.0% (4808; 7858)  
## +303079: mip = 1.480105129e+011 >= 1.348895240e+011 8.9% (4784; 7982)  
## +306375: mip = 1.480105129e+011 >= 1.350962339e+011 8.7% (4789; 8076)  
## +310483: mip = 1.480105129e+011 >= 1.353990926e+011 8.5% (4782; 8180)  
## +312082: mip = 1.480105129e+011 >= 1.355305797e+011 8.4% (4743; 8237)  
## +313607: mip = 1.480105129e+011 >= 1.356668224e+011 8.3% (4742; 8282)  
## Time used: 720.0 secs. Memory used: 103.6 Mb.  
## +316663: mip = 1.480105129e+011 >= 1.358061327e+011 8.2% (4722; 8375)  
## +320169: mip = 1.480105129e+011 >= 1.361640869e+011 8.0% (4683; 8482)  
## +323108: mip = 1.480105129e+011 >= 1.363082227e+011 7.9% (4627; 8580)  
## +325958: mip = 1.480105129e+011 >= 1.365733770e+011 7.7% (4595; 8697)  
## +330670: mip = 1.480105129e+011 >= 1.369557757e+011 7.5% (4510; 8877)  
## +334310: mip = 1.480105129e+011 >= 1.372091286e+011 7.3% (4497; 9012)  
## +338870: mip = 1.480105129e+011 >= 1.376595582e+011 7.0% (4458; 9145)  
## +342884: mip = 1.480105129e+011 >= 1.379834627e+011 6.8% (4386; 9336)  
## +346994: mip = 1.480105129e+011 >= 1.382135732e+011 6.6% (4331; 9499)  
## +352104: mip = 1.480105129e+011 >= 1.386554643e+011 6.3% (4198; 9753)  
## +355110: mip = 1.480105129e+011 >= 1.388595754e+011 6.2% (4148; 9870)  
## +358198: mip = 1.480105129e+011 >= 1.392721340e+011 5.9% (4051; 10047)  
## Time used: 780.0 secs. Memory used: 103.6 Mb.  
## +361190: mip = 1.480105129e+011 >= 1.395125216e+011 5.7% (3979; 10172)  
## +365702: mip = 1.480105129e+011 >= 1.398563388e+011 5.5% (3829; 10405)  
## +370284: mip = 1.480105129e+011 >= 1.402236368e+011 5.3% (3665; 10649)  
## +373416: mip = 1.480105129e+011 >= 1.404360543e+011 5.1% (3561; 10837)  
## +375490: mip = 1.480105129e+011 >= 1.406767209e+011 5.0% (3507; 10955)  
## +377621: mip = 1.480105129e+011 >= 1.408396190e+011 4.8% (3448; 11065)  
## +379787: mip = 1.480105129e+011 >= 1.410247626e+011 4.7% (3359; 11197)  
## +381866: mip = 1.480105129e+011 >= 1.411524345e+011 4.6% (3276; 11341)  
## +383926: mip = 1.480105129e+011 >= 1.412618554e+011 4.6% (3225; 11428)  
## +385745: mip = 1.480105129e+011 >= 1.414277376e+011 4.4% (3157; 11538)  
## +389319: mip = 1.480105129e+011 >= 1.417664934e+011 4.2% (3020; 11770)  
## +392551: mip = 1.480105129e+011 >= 1.422127514e+011 3.9% (2831; 12122)  
## Time used: 840.0 secs. Memory used: 103.6 Mb.  
## +396078: mip = 1.480105129e+011 >= 1.426753033e+011 3.6% (2637; 12449)  
## +398422: >>>>> 1.475556012e+011 >= 1.430664428e+011 3.0% (2476; 12759)  
## +400973: mip = 1.475556012e+011 >= 1.434209561e+011 2.8% (2087; 13629)  
## +404207: mip = 1.475556012e+011 >= 1.438148623e+011 2.5% (1839; 14269)  
## +406823: mip = 1.475556012e+011 >= 1.443737779e+011 2.2% (1589; 14879)  
## +409473: mip = 1.475556012e+011 >= 1.447683607e+011 1.9% (1366; 15486)  
## +411826: mip = 1.475556012e+011 >= 1.452863623e+011 1.5% (1126; 16343)  
## +413929: mip = 1.475556012e+011 >= 1.457911344e+011 1.2% (846; 17379)  
## +415440: mip = 1.475556012e+011 >= 1.462347258e+011 0.9% (636; 18365)  
## +417215: mip = 1.475556012e+011 >= 1.467560840e+011 0.5% (362; 19968)  
## +418064: >>>>> 1.472613739e+011 >= 1.471252795e+011 < 0.1% (188; 21463)  
## +418316: mip = 1.472613739e+011 >= tree is empty 0.0% (0; 24309)  
## INTEGER OPTIMAL SOLUTION FOUND  
## <!SOLVER MSG> ----

#choosen centroid locations  
selected\_centroid <- result4 %>%   
 get\_solution(u[j]) %>%  
 filter(value > 0) %>%  
 select(j)  
  
  
#Take group assignments  
matching <- result4 %>%   
 get\_solution(x[i,j]) %>%  
 filter(value > 0) %>%   
 select(i, j)

Now we integrate the results and visualize the results.

plot\_assignment <- matching %>%   
 inner\_join(ChessClubs20162017, by = c("i" = "Id")) %>%   
 inner\_join(centroid\_locations, by = c("j" = "Id"))  
  
p +   
 geom\_segment(aes(x = LongCentroid, y = LatCentroid, xend = Long, yend = Lat), data = plot\_assignment) +   
 geom\_point(aes(x = LongCentroid, y = LatCentroid), data = centroid\_locations[centroid\_locations$Id %in% selected\_centroid$j, ], color = "red", size = 3, shape = 17) +  
  
 ggtitle("Optimal groups for the Chess League Cup",  
 "Based on the possible centroid locations.   
Big red triangles show the 4 selected centroid locations.   
Light red are unused centroid locations, blue dots represent chessclubs.")

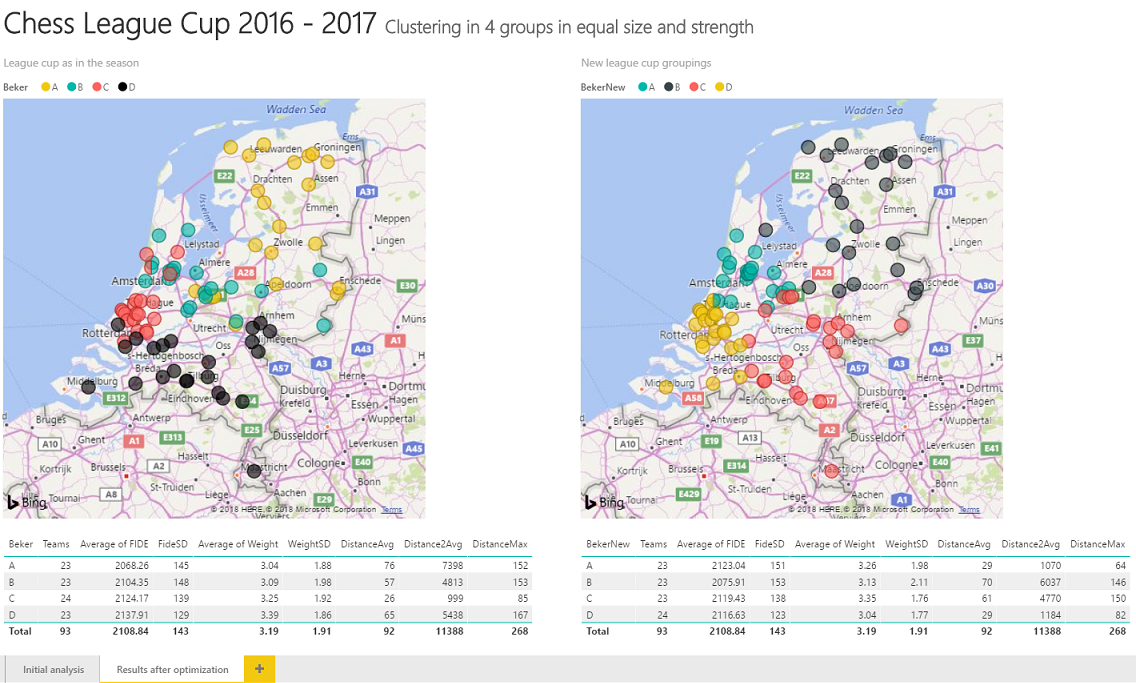


*And no, we still don’t have mountains here…*

The results look like an improvement. Let’s compare the results in Power BI. Therefor we first need to export the results. Export the data to csv.

#Some reordering and export  
ChessClubs20162017New <- matching %>%   
 inner\_join(ChessClubs20162017, by = c("i" = "Id"))  
  
ChessClubs20162017New <- rename(ChessClubs20162017New, Id = i)  
  
jGroups <- sort(unique(ChessClubs20162017New$j))  
ChessClubs20162017New$BekerNew <- mapvalues(ChessClubs20162017New$j, from = jGroups, to = c("A", "B", "C", "D"))  
ChessClubs20162017New$j <- NULL  
  
write.csv(ChessClubs20162017New, file = "output\\leagueChessClub.csv")

## Power BI

The results in Power BI 

Well it seems my gutfeeling was about right. The average distance dropped with almost 10 km (20 km for roundtrip). Ok, the club in province North Holland need to use some bridges, but that seems acceptable. Everything appears better, apart from the SD Weight. It is a little higher for 1 group. This indicates there are more strong and weak teams compared tp the other groups. I think this is acceptable as well.

These statistics are created using DAX (measures), the most interesting is the DistanceAvg and Distance2Avg (squared) measures. They provide the average distance between the selected clubs (within the group in this case). In DAX we can do this in the following way:

DistanceAvg = AVERAGEX(   
 FILTER(  
 CROSSJOIN(  
 SELECTCOLUMNS ( ChessLeagueCup;   
 "Lat\_1"; ChessLeagueCup[Lat];   
 "Long\_1"; ChessLeagueCup[Long];   
 "ClubNaam1"; ChessLeagueCup[ClubNaam]);  
 SELECTCOLUMNS ( ChessLeagueCup;   
 "Lat\_2"; ChessLeagueCup[Lat];   
 "Long\_2"; ChessLeagueCup[Long];   
 "ClubNaam2"; ChessLeagueCup[ClubNaam])  
 );  
 [ClubNaam1] <> [ClubNaam2]  
 );   
 6371 \*((2\*ASIN(SQRT((SIN((RADIANS([Lat\_1])-RADIANS([Lat\_2]))/2)^2)+COS(RADIANS([Lat\_1]))\*COS(RADIANS([Lat\_2]))\*(SIN((RADIANS([Long\_1])-RADIANS([Long\_2]))/2)^2)))))  
)

It would have been nice if this can be added in a kind of UDF as now the formula is copied a couple of times.

After all these analysis, reporting and writing, it is time to do some real calculations again and bring home that Cup ones more!

You can view the Power BI report [here](https://app.powerbi.com/view?r=eyJrIjoiZWIzOTBiYmEtMzc5OC00NmE0LWJhMWItMmQyMDAwNzlkMTgwIiwidCI6Ijg3NGM1MzA1LWI0MDktNGU5Ni04ODhiLTQ4ODViNWQ0ZDYwNiIsImMiOjl9).

All the code can be found on GitHub [ChessLeagueCup](https://github.com/Ruud-Janssen/ChessLeagueCup).