Project Algorithms 2

Σκορδούλης Κωνσταντίνος 1115 2016 00155

Main Programs: cluster.cpp

Source Programs (with their header files): Vector.cpp, Item.cpp, Curve.cpp, Vector_Node.cpp, Cluster.cpp, LSH_HT.cpp, Curve_LSH.cpp, Initialization.cpp Assignment.cpp, Update.cpp, Evaluation.cpp, Utilities.cpp, Utilities2.cpp, Utilities3.cpp

Compilation: Υπάρχει διαθέσιμο **Makefile**, οπότε με την εντολή **make**, γίνεται γρήγορα το compilation.

ΣΗΜΑΝΤΙΚΟ: Το πρόγραμμα υλοποιήθηκε με C++ ➔ **(C++11)**. Χρησιμοποιήθηκαν αρκετές συναρτήσεις/δομές της STL, όπως **vectors** (είναι σαν arrays , αλλά με πολλές παραπάνω λειτουργίες και μεταβλητό μέγεθος).

ΣΗΜΑΝΤΙΚΟ2: Επειδή η αγγλική μετάφραση του διανύσματος είναι Vector, καταλαβαίνω μπορεί να γίνει πονοκέφαλος με τους vectors(STL). Στο κώδικα μου όταν μιλάω για **vectors ➔ STL**, ενώ όταν λέω **Vectors➔Object ➔ Διάνυσμα**. Διαφορά αν είναι uppercase ή lowercase.

ΣΗΜΑΝΤΙΚΟ3: Για όλες τις κατανομές, χρησιμοποιήθηκαν έτοιμες βιβλιοθήκες της C++ ➔ std::normal_real_distribution, std::uniform_real_distribution, std::uniform_int_distribution.

ΣΗΜΑΝΤΙΚΟ4: Το **cluster.cpp** και το **Cluster.cpp** είναι διαφορετικά. Το $1^{ο}$ είναι η **main** μας, το $2^{ο}$ η **δομή** της **συστάδας**. **ΠΡΟΣΟΧΗ!!!!!** Στα **windows**, δεν αναγνωρίζει την διαφορά των αρχείων , όποτε θα προσπαθήσει **να κάνει overwrite στον προορισμό ➔ διαγράφοντας 1 από τα 2 αρχεία!!**

ΣΗΜΑΝΤΙΚΟ5: Το **unit testing** γίνεται στον υπό-φάκελο **CUnit**. Τέσταρα κάποιες απλές συναρτήσεις από τα Utilities source files, όπως **Mod(), Max(), Min()**.

ΣΗΜΑΝΤΙΚΟ6: Στην main, χρησιμοποιώ ένα **vector<Item*>* items**, το οποίο στην αρχή περιέχει όλα τα αντικείμενα, αλλά μετά το **Initialization** περιέχει μόνο τα **non-centroid items**!!!!!!!!!! (πάντα)

ΣΗΜΑΝΤΙΚΟ7: Το ζητούμενο του προβλήματος είναι να επιλέξουμε την κατάλληλη θέση των **centroid**, έτσι ώστε να πετύχουμε το καλύτερο δυνατό clustering , συμφώνα με το silhouette. Τα centroids αλλάζουν μόνο σε 2 περιπτώσεις:

1) Μετά από **Update** (στην περίπτωση του PAM ala Lloyd's, μπορεί να προκύψουν τα ίδια κέντρα ➔ No more updates => exit).
2) Ύπαρξη **outliers** μετά το Assignment. Πάμε να αντιμετωπίσουμε αυτό το πρόβλημα επιλέγοντας καινούργια centroids ➔ καινούργια clusters.

Και στις 2 περιπτώσεις , χρησιμοποιούμε την **Break_Assignments()** ➔ **Utilities3.cpp**, για να σπάσουμε τους δεσμούς αντικειμένων-clusters και να ξαναρχίσουμε αργότερα το Assignment.

ΣΗΜΑΝΤΙΚΟ8: Στο πρόβλημά μας, μπορεί να προκύψουν και **outliers**, δηλαδή αντικείμενα/**centroid** που σχηματίζουν αντικείμενα clusters με [**πλήθος αντικειμένων < 2]** (αντικείμενα ➔ σημεία/καμπύλες από το input και όχι φτιαγμένες από k-means]). Αυτοί οι outliers δημιουργούν προβλήματα στο **Silhouette** ➔ για κάθε cluster , **cluster->size() > 2**, οπότε έπρεπε να αντιμετωπίσουμε τέτοιες περιπτώσεις.

- Ο κ. Χαμόδρακας είχε προτείνει στο forum μια λύση ➔ από τον outlier/centroid, **επίλεξε το πιο μακρινό σημείο του** και αυτό θα είναι το καινούργιο σου centroid. Προσέθεσε στο μάθημα, ότι αν δεν πετύχει τον πιο μακρινό, δοκίμασε να ψάξεις κάποιον στη μισή απόσταση κοκ. Στην εξής περίπτωση μπορεί να έχουμε πρόβλημα: Α Β Γ
    - Α και Β είναι τα 2 κέντρα μας, όπου το Α και το Γ ισαπέχουν από το Β, και υπάρχει μεγάλη συσσώρευση αντικειμένων γύρω από το Β ➔ Α είναι **outlier.** Με την προηγούμενη τακτική , θα επιλέγαμε το Γ σαν καινούργιο κέντρο και από κει το Α, και μετά πάλι Γ . Δηλαδή **Α ➔ Γ ➔ Α ➔ Γ ➔Α ➔Γ ……** (Ping Pong)
- **(ΠΟΛΥ ΣΗΜΑΝΤΙΚΟ)** Σκέφτηκα μια άλλη τακτική. Ο λόγος που υπάρχουν **outliers** είναι επειδή υπάρχουν centroid/clusters τα οποία συσσωρεύουν μεγάλο πλήθος στοιχείων. Το σκεπτικό είναι να επιλέξω ένα από τα αντικείμενα του **πιο «γεμάτου» cluster**, με σκοπό να γίνει **partition**(διαίρεση) cluster στα 2 (ή περισσότερα από 2 αν έχουμε παραπάνω από 1 outlier). Σκοπός μας είναι σε αυτό το «τσουβάλι» να χωρέσουμε όσο το δυνατόν περισσότερους outliers, έτσι ώστε τα καινούργια κέντρα να έχουν **τουλάχιστον 2 αντικείμενα**. Πως αποφασίζουμε ποια από τα υπαρκτά αντικείμενα του cluster θα γίνουν τα νέα κέντρα? ➔ **Initialization++.** Δηλαδή ακολουθούμε τον ίδιο αλγόριθμο, με τη διαφορά ότι το $1^{ο}$ centroid δεν επιλέγεται τυχαία , αλλά είναι το ήδη υπάρχον centroid (τα υπόλοιπα υπολογίζονται κανονικά) .
- Αυτό που έχω περιγράψει παραπάνω βρίσκεται στην **Check_Assignment()** και **Fix_Assignment() ➔ Assignment.cpp**. Το $1^{ο}$ τσεκάρει αν έχουμε πρόβλημα και το $2^{ο}$ το επιλύει.

VERY IMPORTANT!!!!!!!!!: Σαν παράμετρο εισόδου (console) έχω προσθέσει την **-combi** xxx , με την οποία αποφασίζουμε ποιο συνδυασμό θα εκτελέσουμε , πχ **112** σημαίνει **Random Initialization ➔ Lloyd's assignment ➔ Mean Vector/Curve.** Οι πιθανοί συνδυασμοί είναι **111,112,121,122,211,212,221,222 ➔** 8 συνδυασμοί.

Περιγραφή cluster(main):

Καταρχάς κάνω parse τις πληροφορίες από την κονσόλα και διαβάζω το **config** καθώς και το **input**. Για το **input των καμπυλών** έχω να κάνω 2 σχόλια:

1) Ο κ. Χαμόδρακας είχε πει, πως επειδή μπορεί να συμβεί σε 1 καμπύλη να υπάρχουν ίδια διαδοχικά σημεία (ίδιες συντεταγμένες) (στην ίδια καμπύλη) ➔ delete/μη

εισαγωγή duplicate διαδοχικών σημείων ➔ **Input_Curve()**. Αν δεν ισχύει αυτό, τότε μπορείτε να απασχολιάσεται (uncomment) την **Input_Curve1()**, η οποία δεν κάνει αυτόν τον έλεγχο.

2) Όταν εκτέλεσα το **input_projection_6.csv**, βρήκα αντικείμενα στο input που είχαν **απόσταση_0** !!!! Δηλαδή **duplicate καμπύλες,** που εννοείται δημιουργούν προβλήματα. Για αυτό χρησιμοποιώ την **Remove_Duplicates() ➔ Utilities3.cpp**, που αφαιρεί αυτές τις καμπύλες.

Αφού γίνουν όλα αυτά, ελέγχω αν θέλω αργότερα να κάνω **Range Search**. Αν ναι, αρχικοποιώ τις δομές LSH. Και μετά αρχίζω τη διαδικασία:

1) **Initialization:** Επιλέγω ποιο initialization θέλω να εκτελέσω.
2) **1st Assignment:** Εκτελώ το assignment που μου έχουν ορίσει.
3) **Check_Assignment():** Δες ότι δεν υπάρχουν outliers , αλλιώς **Fix_Assignment() ➔ Break_Assignment()** και **loop until the problem is fixed** ➔ go back to step 2.
4) **BIG LOOP**:
    a. **Update:** εκτελώ το update που μου έχουν ορίσει. Αν δεν αλλάξουν τα κέντρα ή υπάρξει αμελητέα αλλαγή ➔ Freedom.
    b. **2nd Assignment:** Τα ίδια και εδώ, αν υπάρξει πρόβλημα με βγαίνουμε από το **nested loop** μέχρι να λυθεί.
5) **Compute_2nd_best:** this is a sanitary check, δηλαδή ότι όλα τα αντικείμενα έχουν βρει το καλύτερο γείτονα τους (χρειάζεται στο silhouette).
6) **Silhouette:** Κατά τα γνωστά, υπολογίζουμε τις τιμές που απαιτούνται
7) **Print_Output:** Γράψε τα αποτελέσματα μας στο αρχείο.

Περιγραφή source files

Χρησιμοποίησα αρχεία της προηγούμενης εργασίας (1 έως 7), οπότε αφήνω αυτά που έγραψα στο προηγούμενο PDF:

1. **Curve.cpp**: Αυτή είναι η δομή των καμπυλών μας. Όσον αφορά τα σημεία των καμπυλών, αποφάσισα να χρησιμοποιήσω **vector< vector<double>* >** , δηλαδή έναν vector που περιέχει δείκτες σε vector<double>. Καλύτερη εξήγηση ➔ δείτε το Curves.hpp.
2. **Curves_LSH**: Εδώ έχουμε την δομή του **LSH_grid**. Για κάθε grid δημιουργείται ένα **LSH_HT ➔ LSH hashTable** ή **Hypercube**(**htc**).
    a. Τα grid points που προκύπτουν (για το input) αποθηκεύονται (στο εσωτερικό του αντικειμένου) σε vectors<>, και μόλις γίνει το **padding ➔ (max_coord + 1) of grid points** ,εισάγονται στην αντίστοιχη δομή για να γίνει το hashing.
    b. Επιπλέον, οι συναρτήσεις που μετατρέπουν **Curve ➔ Grid Curve** και **Grid Curve ➔ Grid points**, είναι η **Curve2Gcurve()** και **Gcurve2Curve() .**
    c. Η συνάρτηση που κάνει το padding λέγεται **Update**.
3. **LSH_HT**: ➔ **LSH_HashTable**. Η αγαπημένη μας μέθοδο LSH, εδώ έχουμε το HashTable της μεθόδου. Τα **number of buckets**
    a. Παράγουμε το **double** s** ➔ (k x d). Τα στοιχεία του προέρχονται από την uniform_real_distribution. Το s φτιάχνεται κατά τη δημιουργία του Hash_Table

b. **Hash()**. Εδώ παράγουμε τα **ai**, σύμφωνα με την Manhattan metric. Το **m = 2^32** . Για το mod() χρησιμοποίησα δικιά μου συνάρτηση ➔**Mod()** (στη **Utilities.cpp**). Επιπλέον για το **modular exponentiation**, έφτιαξα την **alt_pow()**. Μόλις παραχθούν όλα τα h(x), τα μετατόπιζα στην κατάλληλη θέση χρησιμοποιώντας std::**sll() ➔ shift left logical.** Τον παραγόμενο αριθμό **(32 bit)** τον έκανα **string**.
    i. **ΣΗΜΑΝΤΙΚΟ**: τα g(x) (αφού τα υπολόγιζα με τον παραπάνω τρόπο σε 32 bit int) τα αποθηκεύω σαν **STRING ➔** δεν αλλάζει καθόλου το περιεχόμενο του αλγορίθμου (απλά συγκρίνω με strings αντί για integers)
c. **Insert()**. Δέχομαι τα **input Vectors**. Βρίσκω την g(x), και φτιάχνω για κάθε Vector μια νέα δομή **Vector_Node**(more on this later). Αυτή η δομή θα είχε αποθηκευμένη την τιμή του g(x) για να γίνουν οι απαραίτητες συγκρίσεις αργότερα. Εισάγεται ως Vector_Node στο κατάλληλο bucket.
d. **Search()**. Αφού βρούμε το g(x) , και μας οδηγήσουν στο κατάλληλο bucket, ψάχνουμε 1-1 τα Vector_Nodes του bucket. Αν έχουμε **ίδιο g(x)**, υπολογίζουμε την **Manhattan Distance**, και συγκρίνουμε κάθε φορά με τον προ-υπολογισμένο κοντινότερο ➔ if we found a new approximate nearest neighbor. Έχω βάλει και **άνω φράγμα για τον έλεγχο στοιχείων στο bucket**, το οποίο μπορείται να τροποποιήσετε αν θέλετε.

4. **Utilities.cpp**: Εδώ περιέχονται διάφορες χρήσιμες συναρτήσεις για το $1^{ο}$ κομμάτι της εργασίας που αφορά **Vectors**. ➔ **Mod() , alt_pow()** κλπ
5. **Utilities2.cpp**: Εδώ περιέχονται διάφορες χρήσιμες συναρτήσεις για το $2^{ο}$ κομμάτι της εργασίας που αφορά **Curves**.
6. **Vector.cpp:** Η δομή μας για τα **διανύσματα**
7. **Vector_Node:** Ουσιαστικά είναι **Vectors**, τα οποία έχουν extra πληροφορίες:
    a. **g(x)** , που χρησιμεύει σε οτιδήποτε έχει σχέση με **LSH σημείων**.
    b. **Curve* Real**, δείκτη στην πραγματική καμπύλη, για να μπορέσουμε να κάνουμε **DTW**.
    c. **Curve* Grid** (scrapped Idea). Δείκτη στην Grid καμπύλη που προκύπτει από την Curve_LSH. Η original ιδέα είναι ότι στην Search() που κάνεις στη curve_grid_lsh, να συγκρίνεις αυτά που έχουν ίδιο g(x) αλλά και ίδιο Grid Curve.
    d. **(NEW) Item* item,** σε ποιο αντικείμενο αντιστοιχεί αυτό το vector_node
8. **Utilities3.cpp:** Διάφορες χρήσιμες συναρτήσεις (που προέκυψαν για το project2) κυρίως για class Item και class Cluster
9. **Item.cpp**: Η δομή μας για τα αντικείμενα (υπαρκτά/input και μη)
10. **Cluster.cpp**: Η δομή μας για τις συστάδες.
11. **Initialization.cpp**: Εδώ περιέχονται 2 μέθοδοι αρχικοποίησης: **Random_Initialize() ➔ (1) , Initialization() ➔(2)** (Initialization++). Να αναφέρω 2 πράγματα για την Initialization++
    a. Καθόλη τη διάρκεια του αλγορίθμου, κρατάμε έναν πίνακα D ➔ **vector<double>* D**. Αυτόν τον πίνακα τον διαλύω μόνο στο τέλος του αλγορίθμου, και τον κρατάω για όλη τη διάρκεια του αλγορίθμου, ανανεώνοντας τον όποτε πρέπει.
        i. Συγκεκριμένα υπολογίζω τις αποστάσεις των non-centroids από το $1^{ο}$ κέντρο και της αποθηκεύω στον D.

ii. Υπολογίζω το καινούργιο $2^{ο}$ κέντρο, αφαιρώ το $2^{ο}$ κέντρο από τον **vector<double>* D** (και από το Item* items), και υπολογίζω τις αποστάσεις όλων των υπόλοιπων non-centroid από το καινούργιο κέντρο (δεν χρειάζεται να ξαναυπολογίσω για τον/τους προηγούμενους) και ανάλογα κάνω update τον πίνακα D (κρατάει τις min αποστάσεις σαν ποσότητες , δεν μας ενδιαφέρει προς ποιον).
iii. Κάνουμε normalize τις τιμές του D (αλλά σε καινούργιο πίνακα D1 or D2)
iv. Γρήγορα μπορούμε να δούμε πως [**P(r) = P(r-1) + D(r)^2**]
v. Τέλος χρησιμοποιώ δικιά μου **Binary Search() ➔ Utilities3.cpp**

12. **Assignment.cpp:** Πάλι 2 μέθοδοι:
    a. **Lloyd's Assignment():** ➔ Brute Force. Απλά υπολογίζω και τον κοντινότερο γείτονα ➔ 2^{nd} best cluster (για τη silhouette).
    b. **Range Search():** Έχοντας ήδη αρχικοποιήσει τις δομές (πριν την Initialization) , ετοιμάζουμε τα queries ➔ **Initialize_query_Vector() , Initialize_query_Curve().** Η διαδικασία λοιπόν έχει ως εξής:
        i. Βρίσκω την αρχική ακτίνα αναζήτησης.
        ii. Ξεκινάει το $1^{ο}$ query (1^{st} player) και κάνει range Search σε όλα τα hash_tables/grids και κάνει **CLAIM (σημαντικό)**, τα αντικείμενα τα οποία πληρούν τις προυποθέσεις του range Search (δες **LSH_HT.cpp κάτω κάτω**).
        iii. Συνεχίζουν και οι υπόλοιποι κατά τον ίδιο τρόπο, αν υπάρξει **CONFLICT** ➔ το αντικείμενο γίνεται **CLAIMED** από το κοντινότερο cluster (κάνοντας update και το (2^{nd} best cluster) field του αναφερόμενου item ➔ ο previous γίνεται 2^{nd} best).
        iv. Μόλις τελειώσουν όλοι, και πριν διπλασιάσουμε την ακτίνα ➔ **END OF ROUND** ➔ εκτελούμε την **Commit_Assignment_Items()** → Assignment.cpp. Κοιτάζει όλα τα στοιχεία τα οποία έχουν γίνει claimed από κάποιο cluster , και τα κάνει **ASSIGNED** στο cluster (εισάγοντας τα επιτέλους στα αντίστοιχα cluster). Αλλάζει το **assigned = true** (για να το αγνοήσουμε στον επόμενο γύρο) και υπολογίζει τον 2^{nd} best cluster, σε όσα αντικείμενα δεν είχαμε conflict (δεν μας διαβεβαιώνει κανείς ότι σε επόμενο γύρο θα υπάρξει conflict ➔ do it now)
            1. **ΣΗΜΑΝΤΙΚΟ: Claimed** ➔ temporary, **Assigned** ➔ **COMMIT**
        v. Τέλος διπλασιάζω την ακτίνα και επαναλαμβάνω την διαδικασία

13. **Update.cpp:** Οι 2 μέθοδοι μας **Update_Lloyd(), Update_Vector()/Update_Curve().**
    a. **Update Lloyd():** Υπολογίζω τα αθροίσματα για όλα αντικείμενα που βρίσκονται μέσα στο cluster, επιλέγω σαν νέο κέντρο εκείνο με το ελάχιστο άθροισμα αποστάσεων.
    b. **Update_Vector():** Εύκολο, βρίσκουμε απλά τον **average Vector**
    c. **Update_Curve():** Εκτελούμε τον αλγόριθμο, απλά επειδή αυτή τη φορά χρειαζόμαστε **DTW Backtracking** (για τα index pairs) , χρησιμοποιώ την **DTW1() και την DTW_Backtracking()** ➔ Utilities3.cpp.

14. **Evaluation.cpp:** Εδώ περιέχεται η **Silhouette**.

Σχολιασμός Αποτελεσμάτων

Το πρόβλημα αυτό, είναι πρόβλημα βελτιστοποίησης, καθώς προσπαθούμε να βρούμε τις βέλτιστες θέσεις των centroids, για να έχουμε το βέλτιστο clustering. Θα πάρω κάθε στάδιο ξεχωριστά και θα συγκρίνω τις μεθόδους:

1. **Initialization:** Η $2^{η}$ μέθοδος (**Initialization++**) είναι ξεκάθαρο ότι είναι καλύτερη. Όχι μόνο μας δίνει **καλύτερη ακρίβεια** , αλλά δεν υπάρχει αισθητή διαφορά στον χρόνο εκτέλεσης των 2 αλγορίθμων. Ο μόνος λόγος για να επιλέξεις την $1^{η}$ μέθοδο είναι για να έχεις κάποιες λιγότερες πράξεις, αλλά θα έχεις **penalty** αργότερα, γιατί θα χρειαστείς περισσότερες επαναλήψεις **Update-Assignment()** για να βρεις τα κατάλληλα centroids, αλλά είναι πιθανό να προκύψουν και **outliers** (extra καθυστερήσεις) ➔ περισσότερο **clustering time.** Στο δικό μου πρόγραμμα που έχω **δεδομένο αριθμό επαναλήψεων**(7 ή 10) , τότε **επηρεάζεται αρνητικά το silhouette**. Οπότε σχεδόν πάντα **Initialization++.**
2. **Assignment:** Για το $1^{ο}$ δεν έχω να πω πολλά (**Brute force**), οπότε έχουμε 100% ακρίβεια στην ανάθεση , αλλά αρκετά πιο αργό συγκριτικά με το $2^{ο}$. Τώρα για τον $2^{ο}$ (**Range Search with LSH**):
    a. (+) **Χρόνος**. Αισθητά πιο γρήγορος από την $1^{η}$ μέθοδο. Όπως την έχω φτιάξει εγώ, εκτελώ 10 γύρους **range search**(δηλαδή initial radius, μέχρι και (2^9) * initial radius). Τα υπόλοιπα αντικείμενα που απομένουν και δεν έχουν γίνει assigned , χρησιμοποιούμε Brute force για να βρούμε το cluster τους. Τα τελευταία είναι συνήθως λίγα, οπότε έχουμε **αισθητή βελτίωση χρόνου**.
    b. (-) **Ακρίβεια**. Είναι πιθανό κάποιο αντικείμενο να βρίσκεται αρκετά κοντά σε έναν cluster A και αρκετά μακριά από έναν cluster B. Υπάρχει περίπτωση, κατά την Range Search() να μην εντοπιστεί από τον A (σε κανένα bucket) , αλλά να εντοπισθεί από τον B αργότερα. Επομένως, πρέπει να έχουμε υποψιν την πιθανότητα σφάλματος ➔ **penalty in silhouette.**
    c. **Final Thoughts:** Δύσκολο. Αν μας ενδιαφέρει μόνο το **silhouette** ➔ **PAM**, για 100% ακρίβεια ανάθεσης. Τώρα αν βάλουμε και το **clustering time** μέσα, τότε αξίζει να χρησιμοποιήσουμε το $2^{ο}$, **clustering time** ➔ **RangeSearch()**.
3. **Update**: Θα συγκρίνω αυτούς τους 2 , από θέμα ακρίβειας και ταχύτητας/πράξεων.
    a. **Ακρίβεια**: ο $1^{ος}$ (**PAM ala Lloyd**) είναι αρκετά καλύτερος από τον άλλο, γιατί απαιτεί πολύ λιγότερες επαναλήψεις για να συγκλίνει στη βέλτιστη λύση.
    b. **Ταχύτητα**: ο $2^{ος}$ (**K-means**) είναι πιο γρήγορος από τον $1^{ο}$, γιατί απαιτεί λιγότερες πράξεις, αλλά το γεγονός ότι έχει χειρότερη σύγκλιση από τον $1^{ο}$ εξακολουθεί να υπάρχει. Δηλαδή έχουμε γρηγορότερη εκτέλεση του Update, αλλά περισσότερες επαναλήψεις σε σύγκριση με τον $1^{ο}$.
    c. **Final Thoughts**: Αν μας ενδιαφέρει μόνο **Silhouette**, τότε ο $1^{ος}$ αλγόριθμος είναι σχετικά μονόδρομος, γιατί είμαστε σίγουροι ότι θα συγκλίνει στο βέλτιστη λύση (αναφέρθηκε και κάτι παρόμοιο στο μάθημα από τον κ. Χαμόδρακα ➔ local minimum?). Τώρα αν θέλουμε να κάνουμε και λίγη εξοικονόμηση χρόνου, έχοντας κάποιο penalty στο Silhouette , επιλέγουμε τη $2^{η}$. Επομένως :
        i. **PAM ala Lloyd: (+) Silhouette**, **(-) Χρόνος**
        ii. **Kmeans: (-) Silhouette, (+) Χρόνος**

iii. Δηλαδή υπάρχει **tradeoff**, μεταξύ των 2 παραμέτρων.

iv. Αν δεν έχουμε περιορισμό στο χρόνο, εγώ θα επέλεγα το 1^{ο}.

4. **Ποιος από τους 8 συνδυασμούς:**
   a. Αν δώσουμε βάρος καθαρά στο **Silhouette** ➔ **211**
   b. Αν μας ενδιαφέρει το **clustering time** ➔ **221**
      i. **Όχι 222**, γιατί έχουμε αρκετό penalty στο **Silhouette**