Estudi del cost per la cerca del veí més proper en arbres k-dimensionals

Grau A – Q1 Curs 2023-2024

Arnau Claramunt, Ferriol Falip, Giancarlo Morales i Martí Puerta

Octubre 2023

Índex

1 Informe tècnic

L'objectiu d'aquest treball és implementar els arbres k-dimensionals juntament amb algunes variants del mateix, com són els arbres k-dimensionals relaxats i els arbres k-dimensionals *squarish*, tot això juntament amb la implementació d'un algorisme que permeti realitzar consultes per cercar el veí més proper en aquests arbres. I, amb els resultats obtinguts, estimar experimentalment el valor de ζ en funció de k i del tipus d'arbre tenint en compte el cost mitjà de les consultes i el nombre de nodes dels arbres.

Les implementacions han estat fetes amb C++ i cada arbre està separat en un fitxer *.hh*. Que després s'inclouen en el programa principal *main{tipusArbre}.cpp* i allà es genera tots els resultats pels experiments.

1.1 Disseny i implementació dels arbres *k*-dimensionals

Els arbres k-dimensionals són estructures de dades utilitzades per gestionar i cercar eficaçment punts en espais multidimensionals. Aquests arbres són útils en moltes aplicacions, com ara bases de dades espacials, cerca de veïns més propers i algorismes de cerca espacial.

1.1.1 Arbres *k*-dimensionals estandards

Per tal d'implementar els arbres k-dimensionals estàndards hem utilitzat una estructura de dades basada en nodes que apunten altres nodes, és a dir un node guarda la seva informació i qui són els seus fills.

```
struct Node {
    int disc;
    vector<double> coordenades;
    Node* left;
    Node* right;
};
```

Tota l'estructura de Node està dins la classe *KdTree*, juntament amb un atribut enter que representa la dimensió dels punts de l'arbre, mètodes privats i públics.

A més hem creat diferents funcions per tal de crear, inserir nodes i esborrar l'arbre.

- La creadora simplement crea un arbre amb un node null.
- Per tal d'inserir nous nodes hem implementat la següent funció, descrita en pseudocodi

```
insert(coords_node)
    insertRec(primer_node, coords_node, 0)
end

insertRec(current, coords_node, discrim)
    if current == NULL then
        Nou_node := Nou Node
        Nou_node.disc := discrim
        Nou_node.coordenades := coords_node
        Nou_node.left := NULL
        Nou_node.right := NULL
        current := Nou_node
    else if current.coordenades[discrim] >= coords_node[discrim] then
        insertRec(current.left, coords_node, (discrim + 1) mod k)
    else
        insertRec(current.right, coords_node, (discrim + 1) mod k)
end
```

- Per tal d'esborrar un arbre hem implementat la seguent funcó descrita també en pseudocodi:

```
borrar_arbre()
    esesborra_node_arbre(primer_node)
end

esborra_node_arbre(m)
    if m is not NULL then
        esborra_node_arbre(m.left)
        esborra_node_arbre(m.right)
        delete m
    end if
end
```

Finalment hem implementat una funció que s'encarrega de generar un arbre k-dimensional amb n nodes. La funció és bàsicament un bucle que genera n punts aleatoris de dimensió k i els va inserint en un arbre inicialment buit. A continuació posem la funció en pseudocodi:

```
generar_arbre(n: integer)
    for i := 0 to n-1 do
        coord: array of doubles
        for j := 0 to k-1 do
            x := rand() / RAND_MAX
            coord[j] := x
        end for
        insert(coord)
    end for
end
```

1.1.2 Arbres *k*-dimensionals relaxats

En els arbres k-dimensionals relaxats el discriminant és escollit de forma aleatoria, el que resulta en que siguin més inconsistents i la cerca sigui més costosa. Tot i això són útils en casos on on no es requereix una recerca exacta o on les dades són massa complexes per a una divisió uniforme.

Els arbres relaxats també ofereixen la possibilitat d'eliminar i inserir nodes sense complicació, cosa que els altres arbres no permeten degut a la rigidesa d'assignació de discriminants.

Per implementar aquests arbres, l'únic canvi que vam fer a la implementació original dels arbres estàndards és en l'insert, on ara el valor del discriminant de cada node es va calculant aleatòriament, en la funció insert i insertRec.

1.1.3 Arbres k-dimensionals *squarish*

Les *bounding boxes* es fan servir per organitzar i buscar eficaçment objectes geomètrics en espais multidimensionals, creant així "caixes" que divideixen l'espai on treballem. En els *squarish k-d trees* el discriminant per fer les particions és escollit segons el costat de la *bounding box* més llarg amb l'objectiu de dividir l'espai de la manera més uniforme possible.

Per aquest motiu hem afegit al *struct Node*, un *pair* de vectors de nombres reals que delimiten la bounding box del propi node, representant el punt mínim i màxim d'aquesta.

Per tal de trobar el costat més llarg de la bounding box de un punt hem implementat la funció *get_maxEdge*, que retorna el costat pel qual hem de fer la següent divisió, uitiltzant l'insert recursiu enviant-hi el discriminant obtingut.

```
get_maxEdge(bound_box)
    pos_max := 0
    max := 0
    for i := 0 to k-1 do
        aux := abs(bound_box.second[i] - bound_box.first[i])
        if aux > max then
            max := aux
            pos_max := i
        end if
    end for
    return pos_max
end
```

1.1.4 Programa principal

Hi ha tres programes principals, un per cada arbre (*mainEstandar.cpp mainRelaxed.cpp mainSquarish.cpp*). Centrant-nos només en el programa principal *mainEstandar.cpp* s'encarrega de definir les constants dels experiments tals com: T, n, Q, nombre de punts de la gràfica. A més a més, s'instancia les classes dels arbres per generar-los un a un, omplir-los de coordenades aleatòries entre [0, 1] i es fa la cerca del veí més proper amb ells, guardant en un comptador el nombre de nodes visitant, ja que serà tractat com el cost de la cerca. En acabar es crida la destructora i així no s'ocupa memòria i es pot generar el següent arbre.

1.1.5 Programa per tractar els experiments

Un cop tenim les dades recollides dels costos mitjans (Cn) per una certa n (nombre de nodes) en una k determinada, les dades estan guardades en fitxers *dadesK*{*númeroDimensió*}{*tipusArbre*}*.txt*, on el número de dimensió va entre 2 i 6 i el tipus d'arbre: _ (no s'indica, pels estàndards), R (*relaxed*), S (*squarish*). Un exemple seria: *dadesK3S.txt*, on s'indica que els resultats obtinguts s'han fet en la dimensió 3 amb el tipus d'arbre squarish. Un exemple de 3 línies del fitxer amb els resultats és aquest:

```
500                     // n
21 17 ... 14            // 500 valors separats per espais, que son
                        // el cost obtingut de cada una de les 100
                        // consultes fetes en els 5 arbres
18.985                  // cost mitja
```

Posteriorment en el treball, s'explicarà en detall el procés d'obtenció dels resultats experimentals i les conclusions obtingudes.

A més a més, per tal de tractar aquestes dades, vam crear un fitxer *compararKs.py* en el llenguatge de programació Python, que a trets generals llegeix línia a línia les dades del fitxer, les tracta i es poden visualitzar amb gràfiques usant la llibreria matplotlib. Concretament veient el creixement del cost mitjà (Cn) vs. el nombre de nodes (n) de l'arbre en cada dimensió pels arbres estandar.

A més a més, tenim un altre fitxer *plotComplet.py* que conté 4 gràfiques. Una del creixement Cn contra n amb la variància en arbres estandar i la dimensió que s'especifica com a entrada. Una altra que mostra aplicant una transformació logarítmica i addicionalment una regressió lineal per tal que quedi una recta a cada tipus d'arbre i mostra l'equació de la recta amb el pendent i el *R-squared*. Una altra que aplica regressió logarítmica a les gràfiques originals amb els 3 tipus d'arbres, per a una k donada i una última gràfica que divideix cada valor per n^{ζ} per veure que la relació és constant.

1.2 Algorisme del veí més proper

Hem implementat un mètode per trobar dins d'un arbre (sigui estàndard, relaxat o *squarish*) quin és el node que té la menor distància euclidiana a un punt hipotètic Q. Després de provar unes primeres versions de com fer aquest algoritme, la més eficient que hem trobat segueix la següent idea. L'algoritme es divideix principalment en dos parts:

1. Primera ubicació del punt Q dins l'arbre: Volem fer una hipotètica inserció del que seria un node amb les coordenades del punt Q, per saber en quina fulla exacta de l'arbre recauria el punt. Es fa ja que és una bona aproximació a priori de la zona on podria estar el node més proper a Q: gràcies a l'estructura binària dels arbres i usant el mecanisme dels discriminants per decidir la branca que es "baixa". Durant aquest recorregut, a cada node calculem la distància euclidiana de les coordenades del node actual A i el punt Q, i sempre anem guardant la distància mínima i la informació del node A, que durant tot el funcionament de l'algorisme és el candidat a ser el veí més proper X.

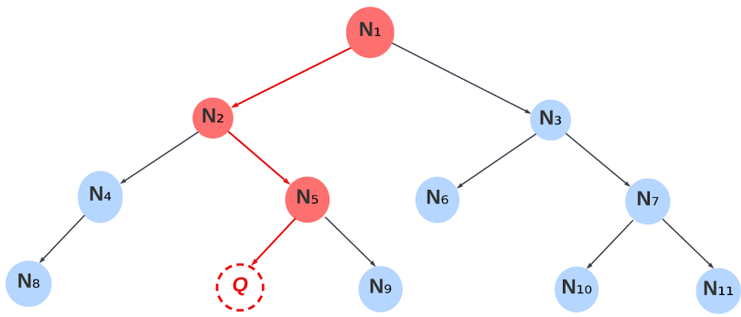


Figura 1: Primer recorregut de l'arbre

2. Cerca exhaustiva de nodes alternatius (*Backtracking*): Fet el primer recorregut, podem assegurar que hem trobat algun node relativament proper a Q, però no sabem si és el més proper de tots els nodes de l'arbre. Per tant, el que fem és, partint des del node fulla hipotètic B que hem trobat en el pas 1), considerem, en cas que existeixi, el node germà B' (això és, si existeix el node pare P de B, B' seria l'altre fill de P). Comparem la distància només entre la coordenada del discriminant del punt Q i del node B', i en cas que aquesta distància sigui menor a la del candidat a ser el veí més proper, ens interessa tot el subarbre encapçalat per B', que en el primer recorregut no haviem explorat. Per tant, tornem a fer la crida recursiva amb aquest subarbre i acaba fent el mateix recorregut inicial, com al punt 1), per ubicar el punt Q dins del subarbre (que té B com a arrel). I quan s'arriba al final (fulla) es repeteix el pas 2 del backtrack i s'aprofita la desigualtat triangular de la distància per evitar recórrer subarbres que ja no milloren la distància mínima i no hi haurà cap candidat a ser el veí més proper. A grans trets estem recorrent l'arbre des de baix cap a dalt, i recorrent a més a més els subarbres que ens quedaven explorar amb la simulació de l'insert, que amb sort es poden anar descartant ("podant").

   A l'hora del *backtracking*, quan estem decidint (*bound*) si hem de continuar explorant un subarbre, mirem la distància: distDiscriminant des del node actual A, només entre la component del discriminant d'aquest cap a Q i la comparem amb la distància mínima candidata obtinguda

fins al moment. Només mirem la distància al discriminant, ja que no cal considerar totes les dimensions del node actual a Q. El discriminant ens dicta la dimensió sobre la qual la distància és més significativa: i el seu valor (representat com a un segment en 1D) el podem valorar si es troba dins o fora el radi que s'estableix entre Q i el punt candidat trobat fins ara a ser el més proper. Si es troba dins, ens interessa aquest node A i continuar mirant als seus subarbres.

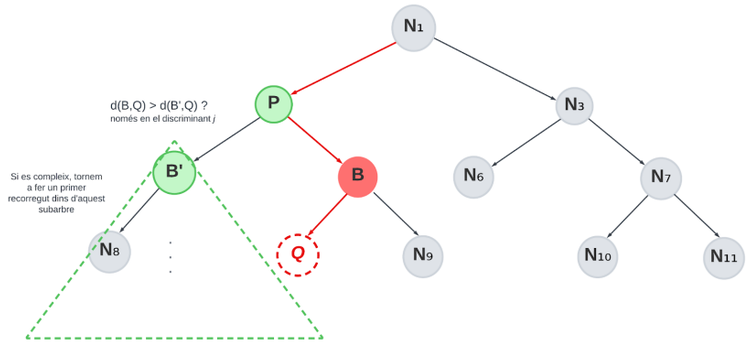


Figura 2: Backtraking

El pseudocodi de l'algoritme emprat és el següent:

```
findNeighbour(current, coordsQ, min)

    if current == nullptr then
        return void
    end if

    dist := distancia(coordsQ, current->coordenades)

    if dist < min.first then
        min.first := dist
        min.second := current.coordenades
    end if

    subarbreQueFaltaExplorar := nullptr //Node

    //Primer recorregut
    if coordsQ[current.disc] < current.coordenades[current.disc] then
        subarbreQueFaltaExplorar := current.right
        findNeighbour(current.left, coordsQ, min)
    else
        subarbreQueFaltaExplorar := current.left
        findNeighbour(current.right, coordsQ, min)
    end if

    //Backtrack - distancia nomes entre components del discriminant
    distDiscriminant := coordsQ[current.disc] - current.coordenades[current.disc]
    distDiscriminant *= distDiscriminant; //totes les distances al quadrat

    // Continuem buscant si estem dins radi de Q amb la min dist
    if distDiscriminant < min.first  then
        findNeighbour(subarbreQueFaltaExplorar, coordsQ, min)
    end if
end
```

1.2.1 Cost de l'algorisme

L'anàlisi del cas mitjà de les cerques pel veí més proper en arbres binaris k-dimensionals estàndard no s'ha fet formalment, però una conjectura diu que el cost és de la forma a $\theta(n^{\zeta} + \log n)$, amb n el nombre de nodes de l'arbre.
Basant-nos en l'algoritme implementat, es pot veure que en mitjana quan fem el pas 1 de simular una inserció de Q, el cost és $\log_2(n)$. Idealment, si ja haguéssim trobat el veí més proper el cost general seria logarítmic, però encara falta explorar algunes branques candidates, fent *backtrackig*, afegint un cost n^{ζ}. Com que s'aprofita la desigualtat triangular i anem descartant moltes branques, tenim un cost molt petit que més endavant es comprova experimentalment.

Per tant en queda una funció amb aquesta forma:

$$f(n) = n^{\zeta} + \log_2(n)$$

On $f(n)$ seria la mitjana de nodes visitats (Cn), que s'usa com a cost de la cerca pel veí més proper.

1.3 Experimentació

1.3.1 Obtenció dels resultats

L'objectiu de l'experimentació és determinar, a través d'experiments, el cost teòric $\theta(n^{\zeta} + \log n)$, on n representa el número de nodes de l'arbre. Aquí, $0 \leq \zeta < 1$ és un coeficient que varia en funció de la dimensió k de l'arbre i que s'haurà d'estimar de manera empírica mitjançant l'ús del pendent de la regressió lineal basat en els resultats obtinguts.

Per tal de recopilar els costos en mitjana i l'evolució segons la n hem recollit un total de 21 valors per dibuixar el cost de la cerca. Començant per $n = 500$, ja que ens ha semblat un bon punt per tal que es vegi el creixement que fa el cost amb n petites i seguint amb $n = 5000$ anant-la incrementant de 5000 en 5000 fins a 100000. A més, hem fet això per diferents dimensions ($2 \leq K \leq 6$), ja que per k superiors a 6 el cost augmenta molt i tampoc considerem que aporti uns resultats rellevants per l'experimentació.

Per cada un dels punts creem aleatòriament 5 arbres de la mateixa mida i realitzem 100 consultes generades també aleatòriament a cada un dels arbres, és a dir, 500 consultes en total. Després calculem la mitjana i la variança de les 500 consultes.

Creem 5 arbres i realitzem 100 consultes degut a que el cost de generar els arbres és molt més elevat que realitzar una consulta i els resultats que obtenim generant molts més arbres i realitzant menys consultes fa variar de manera reduïda els resultats obtinguts. D'aquesta manera, hem trobat adient utilitzar aquests valors, ja que, la reducció de temps a l'hora de realitzar els experiments és rellevant.

1.3.2 Anàlisi dels resultats

Un cop fet tot el conjunt d'experiments, hem pogut recopilar totes les dades necessàries per fer l'anàlisi de resultats.

Observem primer a la *Figura 3* com efecta la dimensió (k) amb el cost. Es pot observar com per k més grans el cost és molt més elevat però segueix una tendència similar, és per això que a continuació mostrarem les gràfiques i les dades per només $k = 2$. Deixem a l'apendix algunes gràfiques i dades que mostren els resultats per altres dimensións, i, com aquests són similars als obtinguts per $k = 2$.

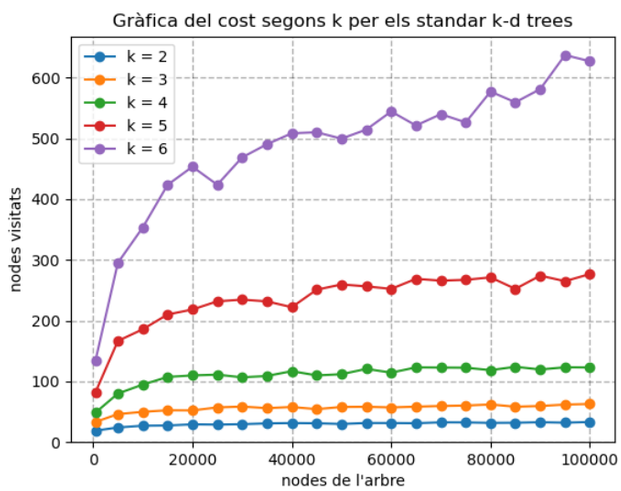


Figura 3: Gràfic del cost segons k per els arbres k-dimensionals estàndards

A la *Figura 4* veiem l'evolució del cost mitjà i la seva variància segons el nombre de nodes per a un arbre de 2 dimensions. Es pot observar un creixement sublineal gairebé logarítmic.

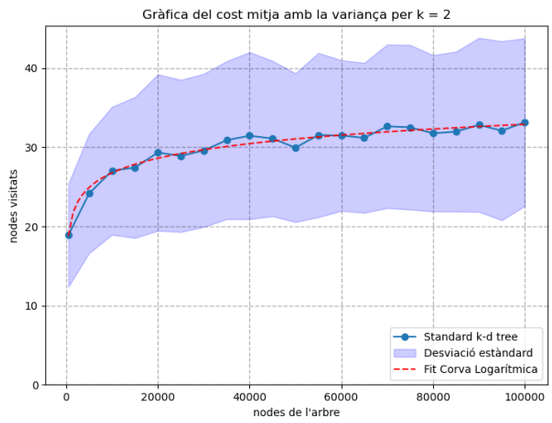


Figura 4: Gràfic del cost mitjà amb la variança per $k = 2$

Fem ara una transformació logaritmica a les dades que ens servirà per analitzar si hi ha una relació lineal entre aquestes, com que no forma una recta perfecta degut a errors experimentals, és necessari aplicar una regressió lineal per tal que quedi una recta i, per tant, es pugui obtenir el seu pendent.
El pendent de la regressió lineal, que és una estimació del comportament del cost segons el número de nodes de l'arbre, correspon amb el valor experimental ζ. Una de les raons és perquè en la funció del cost sabem que n^ζ correspon a la part del *backtracking* de l'algoritme i en cas pitjor, teoricament acabem visitant tots els nodes (n^1, cost lineal). Però al saber que ens estalviem branques, el valor de ζ en promig està llavors entre 0 i 1. Al fer moltes proves amb diverses consultes i arbres, la funció del cost (nodes visitats) contra el número de nodes de l'arbre ens està dient com d'extens o breu és el *backtracking* en funció de n. Per aquesta raó, al obtenir el pendent d'aquesta funció sabem quin comportament té en promig el *backtracking* i podem assignar aquest valor a la variable ζ. Observem aquests resultats a la *Figura 5* on veiem el pendent dels arbres estàndards, relaxats i *squarish*. Si les rectes tinguessin un penden 0, serien horitzontals i per tant el cost de l'algorisme seria logarítmic.

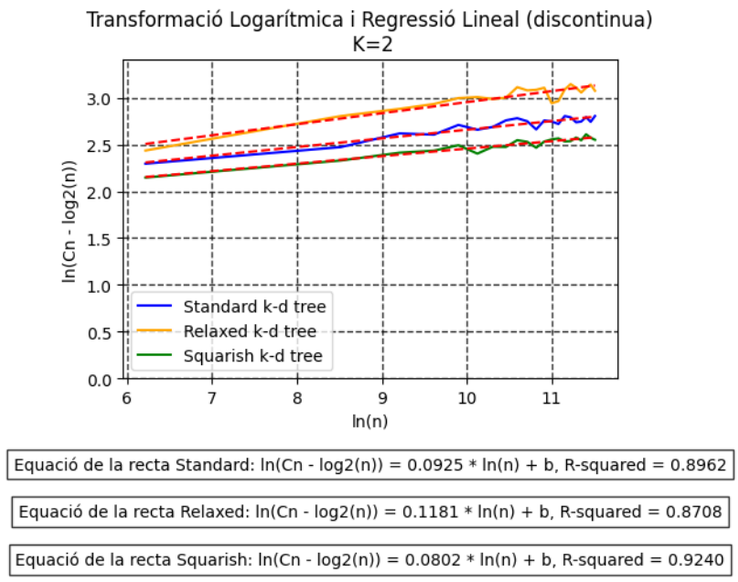


Figura 5: Gràfic de la transformació logaritmica del cost mitjà per $k = 2$, pels arbres k-dimensionals estàndards, relaxats i *squarish*

Així doncs, tenim que $\zeta_e \approx 0.0925$, pels arbres estàndards, $\zeta_r \approx 0.1181$, pels arbres relaxats i $\zeta_s \approx 0.0802$ pels arbres *squarish*. Ja que n^ζ és pròxima a 0 podem practicament negligir la part n^ζ del cost $\theta(n^\zeta + \log n)$ i, com hem vist a la *Figura 4*, els valors segueixen practicament una funció logarítmica, com es pot veure després de dibuixar-ne una fita. Els valors dels *R-squared* ens indiquen la correctesa de la fita de la recta, i per tant, si els valors són porpers a 1, hi ha poc error experimental i la recta ha quedat ben ajustada.

Finalment observem que si agafem $n^{0.0925}$ i ho dividim per tots els resultats ens queda la gràfica de la *Figura 6* on podem veure com ens queden uns valors constants.

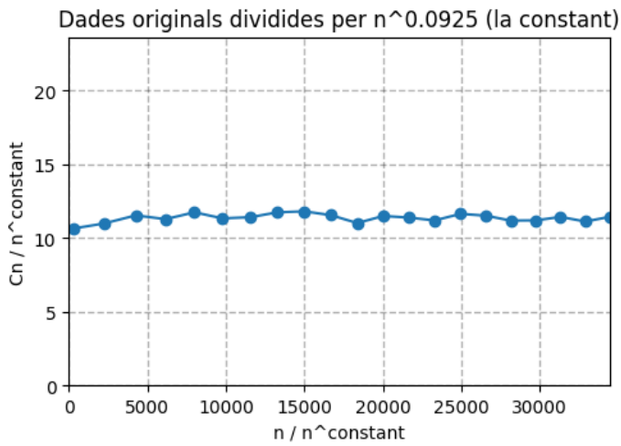


Figura 6: Gràfic del cost dividit per n^{ζ} trobada experimentalment, per arbres k-dimensionals estàndards

1.4 Conclusions

En conclusió, a partir dels resultats obtinguts hem pogut estimar el valor de l'exponent ζ que buscàvem, el qual es troba dins de un marge raonable d'entre 0 i 1. Per tant, podem garantir que el comportament del nostre algoritme és sublineal. Aquest comportament suggereix que l'algoritme té creixement asimptòtic menor al lineal, cosa que comporta una millora d'eficiència a mesura que augmenta el tamany de l'entrada en comparació d'un cost lineal.

A partir dels valors de l'exponent ζ veiem que $\zeta_r > \zeta_e > \zeta_s$ i per tant podem conclore que els arbres k-dimensionals relaxats es comporten pitjor que la resta i els *squarish* són els que tenen més bon comportament repecte el cost. Això últim és lògic segons la naturalesa teórica d'aquests arbres respecte els arbres estàndards i relaxats donat que, com hem dit a l'apartat 1.1, la estructura dels *squarish* és diferent al considerar el discriminant no segons l'altura de l'arbre, sinò en cada moment quin és el costat més llarg de la *bounding box* d'aquell node. Fent així que la partició de *bounding boxes* de tots els nodes sigui més repartida uniformement en tot l'espai k-dimensional (en k=2, hi ha una predominancia de *bounding boxes* quadrades, en comptes de llargues i estretes).

2 Descripció i valoració del procés d'autoaprenentage

2.1 Metodologia

Per organitzar la feina, inicialment vam quedar tots junts per començar i dividir-nos les tasques. Això ens va permetre treballar cadascú pel seu compte des de casa, i d'aquí i en endavant, ens vam anar trobant per parlar del que encara havíem de fer i de tot el que ja havíem fet.

Per tal de comunicar-nos hem utilitzat un xat de grup, una aplicació per fer trucades de veu i, per tal de compartir la feina feta, una carpeta de Google Drive on hem anat penjant totes les actualitzacions i progressos.

2.2 Valoració del procés d'autoaprenentage

El fet d'utilitzar eines desconegudes com són LaTeX i la llibreria de Python Matplotlib ha estat una experiència enriquidora.

En primer lloc, l'ús de LaTeX per a la creació de l'informe ofereix una plataforma molt potent per a la comunicació acadèmica. Hem après a estructurar documents de manera més eficient i professional i hem millorat la nostra capacitat per presentar resultats d'investigació de manera clara i concisa.

Per altra banda, l'ús de la llibreria Matplotlib en Python ha estat crucial per a la visualització de les dades i resultats del treball. Hem après a fer-la servir, així de cara al futur, si tornem a necessitar-la, ja ens resultarà familiar.

Cal destacar també que ens ha sorprès com l'ús de tipus de dades tan semblants com són els 3 tipus d'arbres poden portar a uns resultats diferenciats en realitzar els experiments, simplement pel fet de realitzar petits canvis en les implementacions de les estructures.

Aquest projecte ens ha aportat una manera diferent de pensar a l'hora de fer un algoritme, plantejant una estratègia per fer-lo més eficient, millorant la primera opció ineficient que et ve a la ment de recorrer tots els valors i quedar-te amb el mínim.Mai deixant de banda l'eficiència, per intentar fer codis més ràpids d'executar quan es treballa per exemple amb mides d'arbres grans, fent petites millores, com no calcular l'arrel quadrada en la distància euclidiana, sinó treballar amb totes les distàncies al quadrat i evitar fer una operació costosa.

En resum, el procés ha estat recompensant i ha augmentat el nostre conjunt de coneixements i habilitats en l'àmbit de la recerca i la ciència de dades.

3 Bibliografia

Referències

[1] Documentation. overleaf, online latex editor [online]. [Accessed 17 October 2023]. Available from: https://www.overleaf.com/learn/latex.

[2] Kd-trees - cmu school of computer science [online]. [Accessed 6 October 2023]. Available from: https://www.cs.cmu.edu/ ckingsf/bioinfo-lectures/kdtrees.pdf.

[3] N-dimensional euclidean distance [online]. [Accessed 25 September 2023]. Available from: https://hlab.stanford.edu/brian/euclidean_distance_in.html.

[4] HAYES Adam. What is variance in statistics? definition, formula, and example. investopedia [online]. [Accessed 8 October 2023]. Available from: https://www.investopedia.com/terms/v/variance.asp.

[5] The Matplotlib development team. Quick start guide - matplotlib 3.8.0 documentation [online]. [Accessed 17 October 2023]. Available from: https://matplotlib.org/stable/users/explain/quick_start.html.

[6] CRESPO Maria Mercè Pons. Design, analysis and implementation of new variants of kd-trees [online], 2010. [Accessed 17 October 2023]. Available from: https://upcommons.upc.edu/bitstream/handle/2099.1/11309/MasterMercePons.pdf?sequence=1.

[7] KRUMMEL Michelle. Latex tutorials. michelle krummel - mathematics resources [online]. [Accessed 10 October 2023]. Available from: https://www.michellekrummel.com/tutorials.

[6, 7, 5, 3, 4, 1, 2]

4 Apèndixs

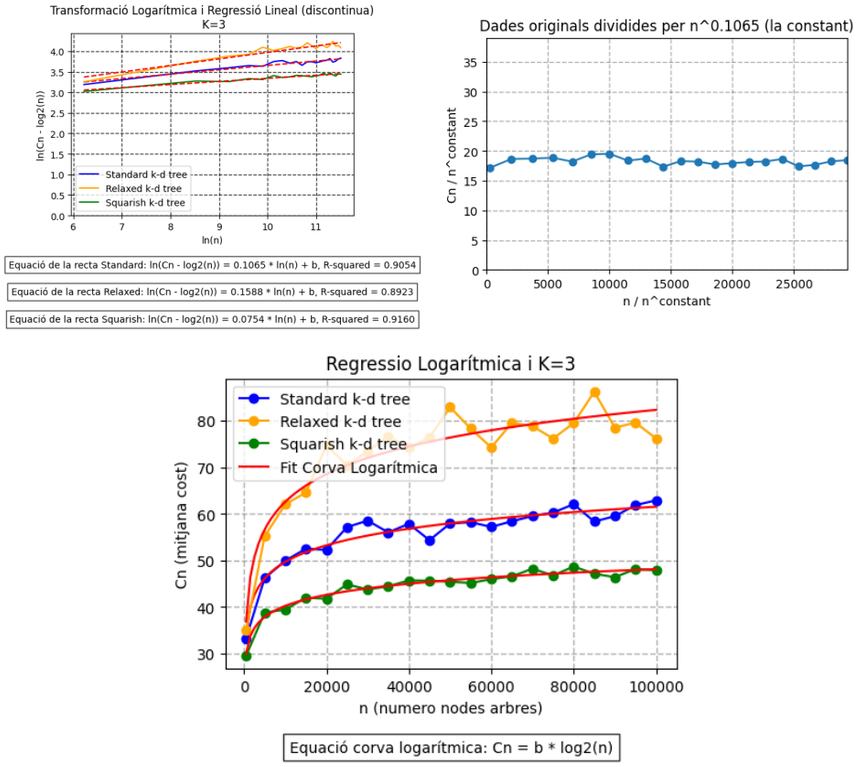


Figura 7: Gràfiques per $k = 3$

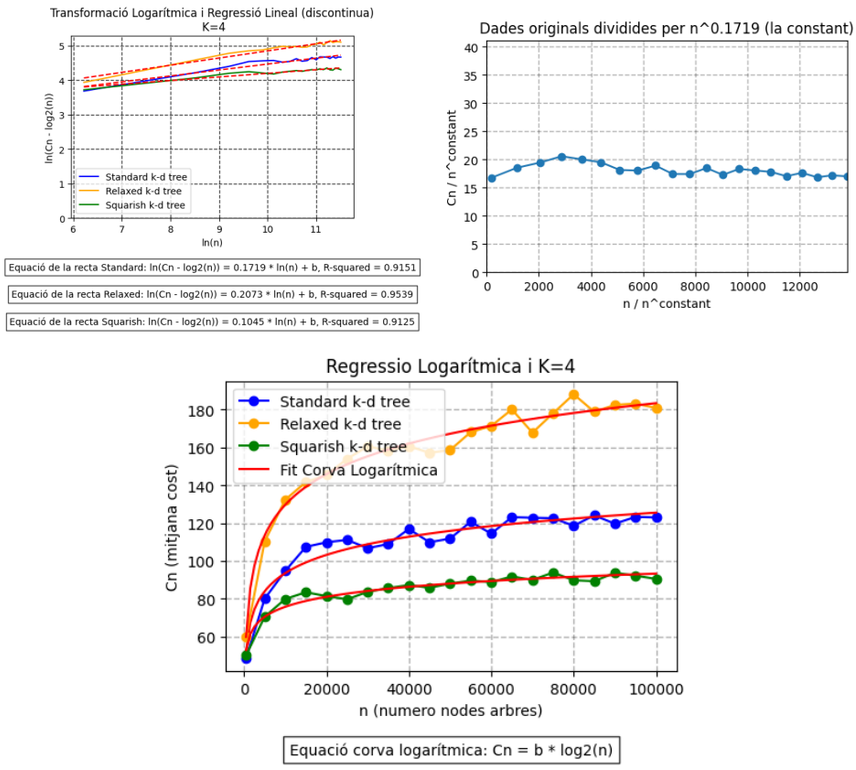
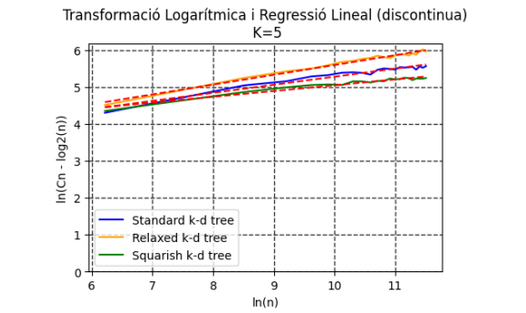
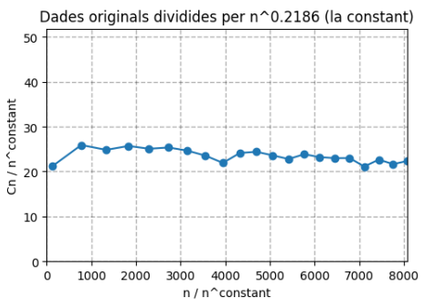


Figura 8: Gràfiques per $k = 4$

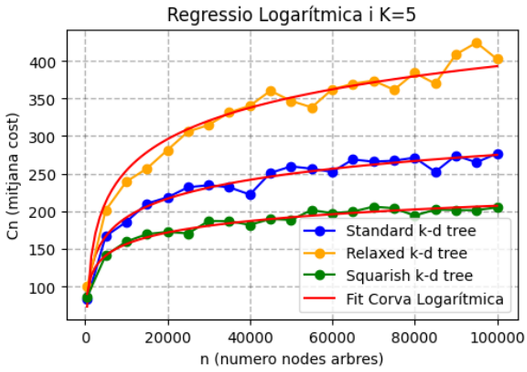




Equació de la recta Standard: ln(Cn - log2(n)) = 0.2186 * ln(n) + b, R-squared = 0.9464

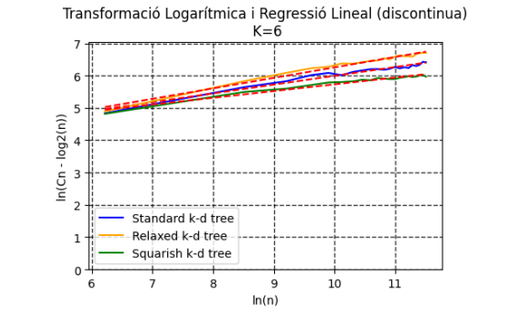
Equació de la recta Relaxed: ln(Cn - log2(n)) = 0.2652 * ln(n) + b, R-squared = 0.9822

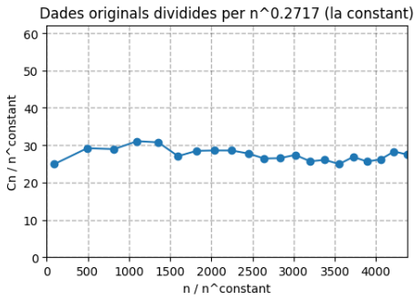
Equació de la recta Squarish: ln(Cn - log2(n)) = 0.1565 * ln(n) + b, R-squared = 0.9484



Equació corva logarítmica: Cn = b * log2(n)

Figura 9: Gràfiques per $k = 5$

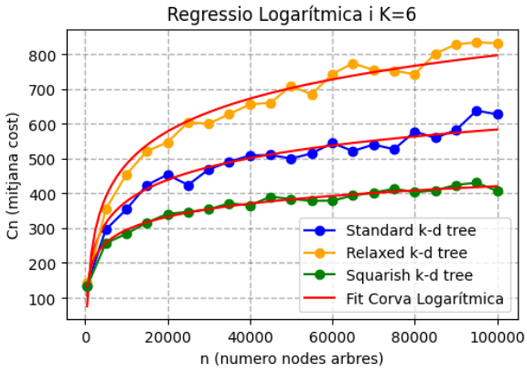




Equació de la recta Standard: ln(Cn - log2(n)) = 0.2717 * ln(n) + b, R-squared = 0.9655

Equació de la recta Relaxed: ln(Cn - log2(n)) = 0.3239 * ln(n) + b, R-squared = 0.9797

Equació de la recta Squarish: ln(Cn - log2(n)) = 0.2083 * ln(n) + b, R-squared = 0.9623



Equació corva logarítmica: Cn = b * log2(n)

Figura 10: Gràfiques per $k = 6$