; revised .

Distàncies entre punts dins núvols de punts. Desenvolupament, anàlisi i comparativa d'algorismes de càlcul de distàncies seguint una arquitectura Model-Vista-Controlador.

Dylan Canning Garcia[1], Antonio Marí González[2], Juan Marí González[3], Antoni Jaume Lemesev [4]

[1]Estudiant Enginyeria Informàtica UIB, 49608104W
[2]Estudiant Enginyeria Informàtica UIB, 46390336A
[3]Estudiant Enginyeria Informàtica UIB, 46390338M
[4]Estudiant Enginyeria Informàtica UIB, 49608104W

Alumne d'entrega: Dylan Canning (email: dcg750@id.uib.eu).

ABSTRACT Aquest document presenta el disseny i la implementació d'un programa per al càlcul i la visualització de distàncies, basat en diverses distribucions de punts en espais bidimensionals i tridimensionals, desenvolupat en el marc de la assignatura d'Algorismes Avançats. Es detallen les tècniques emprades, com la definició de classes, l'enfocament Dividir i Vèncer, la programació concurrent i l'optimització d'algorismes estàndard mitjançant metodologies bucket, tot integrat en una arquitectura MVC. A continuació, es realitza una comparativa dels diferents algorismes aplicats a les diverses distribucions implementades, incloent un resum d'optimitzacions, decisions de disseny i càlculs de costos computacionals.

Finalment, el document conclou amb un anàlisi detallat dels resultats obtinguts, acompanyat de gràfiques que il·lustren la complexitat dels diversos algoritmes damunt de les distintes distribucions i la identificació de possibles millores en el codi i la implementació.

I. Introducció

AQUESTA pràctica s'ha desenvolupat mitjançant una metodologia estructurada basada en l'arquitectura Model-Vista-Controlador (MVC) vista a classe. La modularitat que presenta aquesta estructura ens ha permès distribuir la feina en paquets d'esforç disjunts, la qual cosa ha agilitzat molt el desenvolupament de la pràctica. Fent ús de les incidències apreses a la primera i segona pràctica, hem reestructurat el funcionament intern d'aquestes per a reduir encara més la dependència entre els paquets de vista, controladora i model, definint de primera mà les interaccions/esdeveniments entre les diferents parts del programa (interfícies de notificació).

La distribució de tasques entre el nostre equip de quatre membres s'ha realitzat seguint els components propis del patró MVC [10]: un membre s'ha encarregat de la implementació de la interfície gràfica (Vista), dos membres han desenvolupat la part de dades i algorismes (Model), i un altre ha implementat la part controladora que coordina les interaccions entre els components anteriors. A diferència de la primera i segona pràctica, hem decidit adoptar un disseny encara més modular fent ús del patró de disseny "Factory Method".

Aquest patró de creació encapsula la lògica d'instanciació d'objectes en una classe especialitzada (en el nostre cas, la classe AlgorithmFactory). Això permet que l'usuari (en aquest cas) no hagi de conèixer els detalls de creació dels objectes, ja que aquesta responsabilitat es delega a mètodes específics que, basant-se en paràmetres com el tipus d'algoritme i la dimensió (2D o 3D), generen dinàmicament la instància apropiada. D'aquesta manera, es garanteix una separació clara entre la definició de l'estructura general i les seves implementacions concretes, cosa que afavoreix la modularitat en el nostre cas per fer més dinàmica la creació d'algorismes nous per al nostre codi.

A més, del "Factory Method"hem definit la classe "AlgorithmResult"que representa el resultat d'un càlcul, el que permet tenir una definició igual per a tots els resultats dels algorismes, cosa que permet fer una comparativa entre aquests de manera més senzilla. Aquestes dues definicions ens han permès tenir una metodologia més àgil i independent.

Per facilitar la col·laboració i el seguiment del projecte, hem seguint fet ús de l'entorn de treball ja utilitzat a les diverses practiques (repositori GitHub) per a dur un control de versions, permetent documentar i compartir els avanços de manera sistemàtica. Addicionalment, hem seguit fent ús de l'eina *Obsidian* que permet dibuixar diagrames i crear relacions entre idees i arxius. Igualment, amb la mateixa justificació que a les dues entregues anteriors, hem seguit fent feina amb l'ús de l'IDE IntelliJ per mantenir la compatibilitat dels formats de projecte.

Un cop establertes aquestes bases metodològiques, hem procedit a definir els aspectes tècnics específics de cada secció (Algorismes, Estructures de Dades, Processos, etc.) necessaris per a la implementació. L'objectiu central del nostre disseny ha estat aplicar i maximitzar els coneixements adquirits a l'assignatura, especialment en relació amb les estratègies de Dividir i vèncer, Concurrència i Disseny Modular, que constitueixen els pilars per a la facilitat d'implementació de diversos algoritmes que hem investigat.

II. Metodologia de feina

Per aquest projecte hem seguint amb una metodologia estructurada i col·laborativa, que ens ha permès integrar diferents eines i tècniques per aconseguir un producte final coherent sense haver de fer intensius presencials amb tots els companys. En aquest apartat detallarem els aspectes relacionats amb l'entorn de programació i el sistema de control de versions que hem emprat al llarg del desenvolupament, com els canvis en respectiva a la primera pràctica.

A. Entorn de Programació

El desenvolupament de l'aplicació l'hem dut a terme utilitzant l'IDE *IntelliJ IDEA*, que ens ha proporcionat un entorn robust i modern per a programar en Java, bastant més amigable que *NetBeans*. S'ha treballat amb la versió *Java 23*, aprofitant les seves millores i estabilitat per garantir una execució eficient dels càlculs i processos implementats, com també la facilitat de maneig amb objectes gràfics.

Per aquesta pràctica en comparació a les altres, hem fet ús de JavaFX per a la interfície gràfica d'usuari degut a la complexitat que presentava aquesta pràctica la representació de manera gràfica dels resultats dels algoritmes. En concret, hem fet ús dels diversos paquets que presenta JavaFX per a la representació d'escenes, per a poder fer els plots corresponents en dues i tres dimensions, a més d'afegir la interacció d'usuari per poder interactuar amb els nuvols de punts.

B. Control de Versions

Hem seguit amb l'estructura que varem implementar a la segona pràctica degut a que va funcionar molt bé, es a dir centralitzat al repositori principal on s'allotgen les dues practiques anteriors però seguint amb el nostre sistema de control de versions basat en **Git**, amb el repositori centralitzat allotjat a **GitHub**. Aquesta estratègia ha facilitat:

- Mantenir un historial detallat de totes les modificacions realitzades al codi font.
- Coordinar el treball col·laboratiu entre els membres de l'equip, permetent la integració de diferents funcionalitats de manera ordenada.
- Assegurar una revisió contínua del codi i una resolució ràpida dels conflictes que sorgeixin durant el desenvolupament.

La utilització de GitHub es clau per a la nostra gestió de les versions, degut que ens garanteix que el projecte romangui coherent i actualitzat en tot moment, cosa que afavoreix molt el desenvolupament. A més de l'esmentat, la IDE emprada ens ha permès dur a terme aquest control de versions de manera senzilla pels integrants que no han emprat abans la feina, ja que posseeix eines de gestió de versions sense haver d'utilitzar la terminal.

III. Fonaments Teòrics

En aquesta secció tractarem els diversos elements teòrics que hem emprat per a la nostra pràctica.

A. Model Vista Controlador

En el cas del nostre projecte, hem definit els tres nuclis tal com diu la definició, la part de la interfície gràfica és la encarregada de dur a terme tota la interacció amb l'usuari; és interessant fer una pausa i analitzar la importància de la interacció d'usuari i interfície gràfica, podem considerar el programa com una capsa negre amb la qual l'usuari a d'interactuar a través de la *Grafic User Interface*, per tant, recau en la GUI l'amigabilitat i facilitat d'ús. En aquesta pràctica hem intentat seguir amb aquesta dinàmica, fent un poc més d'èmfasi en la interfície, augmentant l'enteniment de les funcions del programa, minimitzant els inputs erronis per part de l'usuari, això es pot apreciar en la manera que el programa s'auto-gestiona en quant al tipus de visor que es mostra depenent del tipus de dimensió seleccionat, a més hem intentat respectar la "accesibilitat"de moviment en el pla 3D, afegint la mateixa interacció que amb programes de tres dimensions com són AutoCAD, Blender, Unity, etc.

Fixant l'atenció en l'estructura interna del programa, tenim els tres models molt diferenciats. La part de la vista que és l'encarregada de dur a terme tota la interacció amb l'usuari, mostrar les distàncies obtingudes, l'envolupant convex, els punts i els outputs dels càlculs; o els camps de inputs per a la generació dels núvols de punts (Distribució, Número de punts i amplada màxima dels punts "scatteredness".

La controladora per l'altra banda fa de funció de servidor, fa el processament de les peticions de l'usuari i s'encarrega de fer els "trigers"apropiats per dur a terme les accions especificades per part de l'usuari.

Finalment, el Model engloba tot allò relacionat a estructures de dades i els mètodes de manipulació d'informació, en el cas d'aquesta tercera pràctica són els *Algorismes de càlcul de distancies* (, Koch, Hilbert, etc.) i *Geometria Repetitiva* (Tromino i Dominó), aquestes classes encapsulen tots els procediments i funcions per dur a terme els càlculs/modificacions per a calcular els fractals.

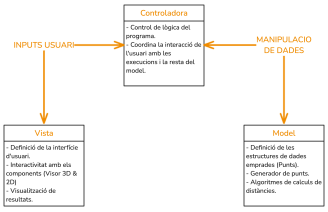


Figura 1. **MVC**

Model:

- Conté les estructures de dades (Point2D, Point3D) i la lògica algorítmica
- Implementa algoritmes per a trobar parells més propers i diàmetres.
- Funciona independentment de la interfície d'usuari

Vista:

- Visualitza núvols de punts en 2D/3D i resultats d'algoritmes
- Implementa UINotificationHandler per rebre notificacions d'estat
- Gestiona la interacció amb l'usuari mitjançant controls JavaFX

Controladora:

- Coordina l'execució dels algoritmes en fils secundaris
- Utilitza el NotificationService per comunicar-se amb la Vista
- Gestiona la creació d'algoritmes a través de l'AlgorithmFactory

B. Patró Factory Method

El patró *Factory Method* és un disseny de creació que hem decidit emprar pel fet que permet delegar la instanciació d'objectes a subclasses, això proporciona una modularitat a l'hora de implementar nous algoritmes. En el context d'aquesta aplicació, hem implementat aquest patró mitjançant la classe AlgorithmFactory, encarregada de generar instàncies d'algorismes en funció dels paràmetres proporcionats.

La interfície comuna `PointCloudAlgorithm<T>` defineix el contracte que han de seguir totes les implementacions d'algorismes sobre núvols de punts. Aquesta interfície estableix les operacions `execute(List<T>)`, `getType()` i `getDimension()`.

A continuació, es mostra el diagrama de classes que exemplifica aquesta arquitectura:

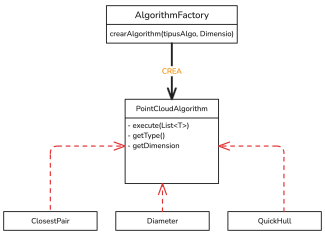


Figura 2. **Factory Method**

Les classes concretes com ClosestPair, Diameter o QuickHull implementen aquesta interfície i són instanciades de forma dinàmica per la fàbrica segons els valors dels enumerats AlgorithmType i Dimension.

1) Avantatges principals:

- **Modularitat:** El codi client no necessita conèixer ni importar les classes específiques d'algorismes.
- **Extensibilitat:** És fàcil afegir noves implementacions d'algorismes sense modificar la lògica existent.
- **Configuració dinàmica:** La selecció d'algorisme es realitza en temps d'execució, permetent adaptar el comportament segons la dimensió o el tipus de problema.

2) Implementació:

L'objecte `AlgorithmFactory` implementa un mètode `createAlgorithm()` que rep com a paràmetres una combinació d'`AlgorithmType` i `Dimension`, i retorna una instància del corresponent algorisme:

- CLOSEST_PAIR_NAIVE + TWO_D → ClosestPairNaive2D
- DIAMETER_QUICKHULL + THREE_D → DiameterQuickHull3D

Aquesta "arquitectura"permet de manera neta, escalar i reutilitzar variants d'algoritmes de manera elegant.

Degut a l'extensió d'aquesta gràfica, es pot trobar als annexos el flux de diàleg entre els distint programes.

C. Flux d'Esdeveniments de l'Aplicació

El funcionament intern del sistema es basa en un model d'interacció orientat a esdeveniments com a la pràctica anterior, es a dir, s'estructura la comunicació entre els diferents components mitjançant un patró de control centralitzat. A l'Annex A, s'hi presenta un diagrama de seqüència que il·lustra el flux complet d'esdeveniments durant l'execució d'un algoritme dins l'aplicació d'anàlisi de núvols de punts.

Aquest flux s'inicia amb la posada en marxa de l'aplicació, on l'usuari interactua amb la interfície gràfica (Vista). Aquesta crea el controlador principal i registra els canals de notificació asíncrona a través del servei de notificacions.

L'usuari configura els paràmetres de l'execució —com ara la dimensió (2D o 3D), el tipus de distribució, l'algoritme a emprar, el nombre de punts i un valor límit opcional— mitjançant els elements interactius de la Vista. Un cop es prem el botó d'execució, el controlador valida els paràmetres i inicia una tasca en segon pla per garantir la fluïdesa de la interfície.

La generació dels punts es realitza segons la dimensió i la distribució escollides, i les notificacions de progrés es propaguen cap a la Vista per mitjà del servei de notificacions. Això permet actualitzar la interfície en temps real, mostrant informació contextual com l'estat actual, la barra de progrés o missatges informatius.

Quan els punts han estat generats, el controlador demana la instanciació de l'algoritme adequat a la fàbrica d'algoritmes. Aquest s'executa sobre el conjunt de punts generat, i el resultat obtingut es notifica novament a la Vista. Aquesta fase inclou la visualització gràfica dels punts destacats, l'activació de funcionalitats com l'enfocament automàtic, i l'opció de representar l'envolupant convexa —tant en dues com en tres dimensions— si així ho ha indicat l'usuari.

La interacció finalitza amb la possibilitat d'enfocar els punts resultants, mitjançant una funció adaptada a la dimensió espacial: centrant la vista en 2D o aplicant transformacions i animacions en 3D. Al llarg de tot aquest procés, el sistema inclou mecanismes de gestió d'errors: si es detecta un problema durant el càlcul, es comunica immediatament a la Vista, que actualitza l'estat de la interfície i informa l'usuari del problema.

Aquest enfocament modular i basat en esdeveniments permet mantenir una separació clara de responsabilitats entre les capes de l'aplicació, assegurant alhora una resposta àgil i fluïda a les accions de l'usuari.

D. Generació i Distribució de Núvols de Punts: Anàlisi Matemàtica i Implementació

La classe `PointGenerator` crea núvols de punts en 2D i 3D basant-se en distribucions estadístiques variades (Uniforme, Gaussiana, Exponencial, Pareto, etc.). Les diferències clau rauen en la funció de densitat de cada distribució i en el mètode de transformació inversa o l'algorisme específic. A continuació es mostra un pseudocodi simplificat:

Algorithm 1 Pseudocodi de `generatePoints`

Require: n (nombre de punts), $dist$ (distribució), $maxX, maxY, maxZ$
Ensure: Llista de punts únics dins els límits

function GENERATEPOINTS($n, dist, maxX, maxY, maxZ$)
  crear llista buida de punts
  crear conjunt `seen` *per controlar duplicats*
  for $i \leftarrow 1 \rightarrow n$ **do**
    $u_1 \leftarrow$ rand.nextDouble(), $u_2 \leftarrow$ rand.nextDouble()
    if $dist =$ UNIFORM **then**
      $x \leftarrow u_1 \cdot maxX, \quad y \leftarrow u_2 \cdot maxY$
    else if $dist =$ GAUSSIAN **then**
      $x \leftarrow$ rand.nextGaussian() $\times (\frac{maxX}{8}) + \frac{maxX}{2}$
      $y \leftarrow$ rand.nextGaussian() $\times (\frac{maxY}{8}) + \frac{maxY}{2}$
    else if $dist =$ EXPONENTIAL **then**
      $x \leftarrow -\ln(1-u_1) \times (\frac{maxX}{2})$
      $y \leftarrow -\ln(1-u_2) \times (\frac{maxY}{2})$ ▷ ... altres distribucions opcionalment es defineixen igualment ...
    end if
    truncar x, y *a [0, maxX/maxY]*
    generar z *de forma anàloga si cal (3D)*
    comprovar si `(x,y,z)` *és nova; si és única, afegir a la llista*
  end for
  return *llista de punts*
end function

Per cada distribució, s'empra la seva *funció de densitat* o *funció inversa de distribució* per mapejar valors uniformes a la distribució desitjada. L'escalat i el truncament (p. ex., `Math.min` i `Math.max`) garanteixen que els punts quedin dins la regió especificada. Finalment, un `HashSet` evita duplicitats considerant la precisió numèrica. Aquest enfocament permet obtenir conjunts de punts estadísticament fiables per a simulacions, proves d'algorismes geomètrics o entrenament de models d'aprenentatge automàtic.

IV. Algoritmes implementats i les optimitzacions aplicades

En aquest apartat es descriuen les diverses versions d'algorismes implementat per a tres problemes geomètrics fonamentals: el parell de punts més proper (Closest Pair), el càlcul del diàmetre d'un conjunt de punts (parella de punts més allunyada) i la construcció de l'envolupant convexa (mitjançant l'algorisme QuickHull) que ens permet trobar els punts més allunyats mitjançant el calcul reduït de tots els vertexos de l'envolupant convexa. Per a cada problema hem considerat solucions naïf i versions optimitzades, incloent tècniques de dividir i vèncer, estructures KD-Tree i altres millores. A continuació, es detallen aquestes implementacions i les seves optimitzacions.

A. Problema del parell més proper (Closest Pair)

Algorisme naïf (força bruta): La solució bàsica calcula la distància entre totes les parelles de punts i selecciona la mínima. Això comporta un cost de $O(n^2)$ tant en 2D com en 3D, ja que es recorren $\frac{n(n-1)}{2}$ parelles. La implementació és senzilla però ineficient per a grans n.

Algorisme de dividir i vèncer: S'ha implementat una versió optimitzada basada en divideix i venceràs. En 2D, aquest algorisme ordena inicialment els punts per la coordenada x, divideix el conjunt en dues meitats L i R, resol recursivament el parell més proper a cada meitat, i després considera possibles parelles amb un punt a cada banda. En combinar els resultats, només cal examinar els punts situats en una franja vertical d'amplada 2δ al voltant de la línia divisòria (on δ és la mínima distància trobada intra-partició). Aquests punts es tornen a ordenar per y i es comparen entre ells limitant les distàncies verticals. Una important optimització geomètrica és que, gràcies a la distribució planar, cada punt de la franja només cal comparar-lo amb un nombre constant de veïns (com a màxim 6 en el cas 2D) [20]. Això redueix el nombre de comparacions en la fase de fusió a $O(n)$, mantenint la complexitat global $O(n \log n)$. S'ha aplicat també aquesta estratègia a 3D mitjançant l'ús d'un slab tridimensional en comptes de la franja planar. Tot i que en 3D l'anàlisi és més complexa, els principis de dividir els punts i limitar comparacions es mantenen: els punts a considerar estan dins un volum central de base 2δ i alçada i profunditat infinites, subdividit en cel·les cúbiques per tal de limitar veïns candidats . En la implementació 3D, això implica comparar cada punt del slab amb només un nombre constant de punts pròxims (acotat per consideracions de geometria en l'espai, fins a 12 punts veïns, per analogia amb el problema del "kissing number". Aquesta versió recursiva en 3D és funcionalment similar a la de 2D, amb adaptacions en la comprovació de la regió central. S'ha tingut cura d'optimitzar la gestió de llistes ordenades per a evitar reordenacions completes a cada recursió (per exemple, mantenint subllistes ordenades per y per a la franja central).

Versió amb KD-Tree: S'ha explorat una solució alternativa mitjançant arbres k-d. Un KD-Tree és una estructura de dades espacial que divideix recursivament els punts per plans ortogonals, facilitant la cerca de veïns propers. L'algoritme implementat construeix un KD-Tree dels punts (cost $O(n \log n)$ en esperat) i després recorre l'arbre per a trobar el parell de punts més proper. Es pot fer trobant per a cada punt el seu veí més pròxim mitjançant consultes successives, o bé realitzant una única passada que mantingui la millor distància global trobada i poda les exploracions innecessàries. En cada cerca de veí proper, l'arbre permet eliminar grans porcions de l'espai de cerca, aconseguint temps mitjans propers a $O(\log n)$ per consulta. Així, en la pràctica, la complexitat esperada global és aproximadament $O(n \log n)$, molt millor que la quadràtica. Cal notar que en el pitjor dels casos (per exemple, punts molt degenerats), un KD-Tree es pot desequilibrar i la cerca pot degradar-se fins a $O(n)$ per consulta, però en dades distribuïdes aleatòriament això és molt improbable. En el nostre context, l'ús de KD-Tree en 2D ha servit com a comprovació de resultats i com a alternativa al mètode de dividir i vèncer. En 3D, també és aplicable el KD-Tree (dividint l'espai per plans XY, YZ, ZX cíclicament), obtenint millores similars en temps mitjà.

B. Càlcul del diàmetre (Diameter)

Algorisme naïf (força bruta): La solució directa per al diàmetre d'un conjunt de n punts consisteix a considerar totes les $\binom{n}{2}$ parelles i calcular-ne la distància, triant la màxima. Això és $O(n^2)$ en temps i requereix avaluar cada parella. Tant en 2D com en 3D, el mètode és correcte però completament impracticable per a n grans, atès que el cost quadràtic escala molt ràpidament.

Algorisme optimitzat amb envolupant convexa: Una millora substancial prové del fet geomètric que la parella de punts més allunyada (diàmetre) sempre es troba entre punts de l'envolupant convexa del conjunt. Per tant, es pot reduir el problema calculant primer la convex hull dels punts i després cercant el parell més distant només entre els vèrtexs d'aquesta hull. En 2D, s'ha implementat l'algorisme QuickHull per a obtenir l'envolupant convexa en temps mitjà $O(n \log n)$. Un cop obtinguda la hull (amb h vèrtexs), s'aplica l'algorisme d'antípodes(rotating calipers) per a trobar el diàmetre en $O(h)$ temps addicional. Aquest mètode recorre els vèrtexs de la hull i identifica eficientment totes les parelles antípodes (punts de la hull que suporten línies paral·leles tangents), determinant la distància màxima. En el pitjor cas (tots els punts del conjunt són vèrtexs de la hull, és a dir $h = n$), el cost és $O(n \log n)$ per a la hull + $O(n)$ per als càlculs de diàmetre, per tant $O(n \log n)$ global. En casos típics on $h \ll n$, el rendiment és encara millor.

En 3D, es procedeix de forma anàloga: es calcula l'envolupant convexa tridimensional i després es determina el diàmetre entre els vèrtexs de la hull 3D. El càlcul de l'hull 3D té complexitat mitjana $O(n \log n)$ i produeix un políedre convex amb h vèrtexs. Per trobar el diàmetre en 3D, s'han de considerar parelles de vèrtexs antipodals en aquest políedre. No existeix un equivalent trivial dels rotating calipers en més dimensions, de manera que la implementació ha optat per una cerca exhaustiva acotada als vèrtexs: es calculen totes les distàncies entre parelles de vèrtexs de la hull (cost $O(h^2)$ en el pitjor cas). Tot i que h^2 pot ser de l'ordre de n^2 en el cas degenerated (per exemple, si tots els punts estan sobre la superfície convexa), en dades "normals" on la major part dels punts queden a l'interior, h és molt més petit que n i la cerca entre vèrtexs és manejable. En resum, aquesta estratègia redueix dràsticament la feina respecte la força bruta, aprofitant les propietats geomètriques del problema.

C. Construcció de l'envolupant convexa

L'objectiu de la construcció de l'envolupant convexa en dues i tres dimensions s'ha abordat mitjançant una combinació d'algorismes originals i adaptacions de codis preexistents,

tot posant l'accent en la robustesa numèrica i l'eficiència temporal. S'han desenvolupat solucions específiques per al cas 2D (basades en QuickHull) i per al cas 3D (adaptant la implementació reconeguda de Lloyd), procurant una integració fluida en el sistema global i assegurant la consistència entre ambdós entorns.

1) Fonamentació de QuickHull i implementació en 2D

L'algorisme QuickHull, dissenyat com un mètode de dividir i vèncer, resulta especialment eficient per a la construcció d'envolupants convexes en dues dimensions. En aquesta versió, primer s'identifiquen els punts més extrems en coordenada x, que es consideren part inicial de l'envolupant. A continuació, es divideixen els punts restants en dos subconjunts, segons es trobin per sobre o per sota de la línia que connecta els punts extrems. Cada subconjunt s'explora recursivament: es localitza el punt més allunyat de la línia en qüestió, es defineix com a nou vèrtex de l'envolupant i es repeteix el procés subdividint la regió fins que no en resta cap punt "exterior" a cap segment.

Aquesta implementació, desenvolupada íntegrament des de zero, ha estat optimitzada mitjançant la seva paral·lelització en sistemes multinucli (utilitzant el model Fork/Join de Java). S'ha comprovat que, tot i poder degradar-se a $O(n^2)$ en casos degenerats, en situacions habituals el comportament s'ajusta a $O(n \log n)$. A més, la memòria es gestiona de manera dinàmica, descartant punts que es troben clarament a l'interior de l'envolupant parcial a mesura que avança la recerca recursiva. Això permet reduir la quantitat de dades a tractar en els passos successius i afavorir-ne l'eficiència.

2) Adaptació de QuickHull3D

La complexitat més elevada de l'algorisme d'envolupant convexa en 3D ha conduït a l'elecció de la versió de QuickHull3D desenvolupada per John E. Lloyd (2004), reconeguda tant per la seva solidesa numèrica com per l'eficiència computacional. Sobre la base d'aquest codi s'ha construït una interfície unificada, capaç d'integrar la representació de punts pròpia del projecte (classes `Point3D`) amb la de l'algorisme original (`Point3d`).

L'adaptació inclou classes específiques com `Point3DAdapter` i `RobustQuickHull3D`, que s'encarreguen d'interpretar correctament els objectes de dades, i la classe `QuickHull3D` pròpia, que ofereix mètodes més intuïtius i coherents amb la resta de la plataforma. Igualment, s'ha integrat la capacitat d'extreure les facetes triangulars de l'envolupant per a la seva representació gràfica i el càlcul del diàmetre geomètric de la distribució, funció especialment rellevant en la detecció de parelles de punts més allunyades. El procés de construcció en 3D s'inicia definint un tetraedre inicial amb punts extrems, classificant la resta d'elements segons les seves relacions de visibilitat respecte de les facetes, i repetint iteracions successives per afegir nous vèrtexs i actualitzar l'envolupant fins esgotar tots els punts exteriors.

3) Anàlisi d'eficiència i aplicacions

Els resultats mostren que la implementació en 2D assoleix una complexitat mitjana de $O(n \log n)$, mentre que en situacions degenerades pot arribar a $O(n^2)$. Gràcies al suport per a execucions paral·leles, és habitual obtenir una millora substancial en plataformes de múltiples nuclis. En 3D, la versió adaptada també manté un rendiment esperat de $O(n \log n)$ i gestiona eficaçment els casos delicats o propers a la degeneració.

Aquestes envolupants convexes resulten imprescindibles per a tasques de càlcul del diàmetre, atès que la distància màxima entre dos punts d'un conjunt només pot aparèixer entre punts que en formin part. Així mateix, proporcionen una base sòlida per a la representació gràfica de la distribució de punts i l'anàlisi de la seva estructura geomètrica, la qual cosa és de gran utilitat en diversos tipus de simulacions estadístiques i aplicacions d'enginyeria.

D. Gestió de concurrència i paral·lelisme

En l'àmbit de la concurrència, l'aproximació seguida per a l'optimització de QuickHull es basa en la divisió natural del conjunt de punts en subconjunts (superior i inferior en 2D, o en diverses facetes en 3D). Aquest enfocament permet organitzar l'execució recursiva de manera concurrent, atès que l'anàlisi de cada subconjunt o faceta és independent fins a la seva fusió final, en la qual s'obté l'envolupant completa.

En resum, les implementacions considerades de cada problema abasten des de solucions exhaustives fins a tècniques avançades de geometria computacional, aprofitant propietats matemàtiques (com restriccions geomètriques o reducció a l'hull convex) per millorar el rendiment. A continuació descrivim com s'ha gestionat el paral·lelisme en aquestes solucions per explotar múltiples nuclis de processador.

V. Gestió de concurrència i paral·lelisme

Per accelerar els càlculs i maximitzar el rendiment en equips moderns, s'ha implementat una arquitectura de concurrència a tres nivells que comprèn: (1) paral·lelisme a nivell d'algorisme amb el model Fork/Join, (2) asynchronisme en la gestió de la interfície d'usuari, i (3) disseny reactiu de l'aplicació global. L'estratègia fonamental consisteix a dividir recursivament la feina en subtasques i distribuir-les entre diferents fils d'execució, mantenint el balanç de càrrega mitjançant work stealing i assegurant una experiència d'usuari fluida. A continuació, es detallen els aspectes clau d'aquesta gestió concurrent multinivell:

A. Model Fork/Join i tècnica work stealing

El marc Fork/Join és especialment indicat per a algoritmes de dividir i vèncer, ja que permet expressar la divisió

recursiva del problema en tasques paral·leles de manera senzilla. La idea bàsica és que un task (tasca) principal es divideix (fork) en subtasques més petites que s'executen en paral·lel, i després s'esperen (join) els resultats. En la nostra implementació, aquest model s'ha aplicat, per exemple, a l'algorisme de parell més proper (executant les recursions de les meitats L i R en paral·lel) i al QuickHull (calculant hull de diferents subconjunts simultàniament quan és possible, o explorant diferents facetes en paral·lel).

Per a gestionar la planificació de tasques, el Fork/Join framework empra un algorisme de work stealing. Cada fil de treball manté una cua doble (deque) de tasques pendents. Quan un fil acaba la seva feina i es queda inactiu, intenta "robar feina"de la cua d'un altre fil que estigui ocupat, prenent tasques de l'extrem contrari de la cua (normalment les tasques més grans que queden). Aquesta estratègia distribueix dinàmicament la càrrega de forma eficient: s'aprofiten els fils inactius perquè completin tasques d'altres fils sense una coordinació centralitzada costosa. El resultat és un balanç de càrrega gairebé automàtic i escalable – mentre hi hagi tasques disponibles, els fils treballen; si un fil acaba, ajuda als altres robant tasques pendents . En el nostre context, això és crucial, ja que algunes particions de l'espai de punts poden requerir més càlcul (p. ex. un subconjunt amb molts punts vs. un altre amb pocs). El work stealing garanteix que cap fil resti aturat llargament si queda feina per fer en altres cues.

La implementació de ClosestPairEfficient3D presenta un exemple clar d'aquesta tècnica. En el mètode execute(List<Point3D> points), es crea un ForkJoinPool amb tants fils com processadors disponibles (Runtime.getRuntime().availableProcessors()). La tasca principal (ClosestPairTask) és llançada amb tots els punts ordenats, i el pool s'encarrega d'executar aquesta tasca paral·lelament. Dins la classe ClosestPairTask, l'algorisme recursiu de divisió i conquesta (compute()) es divideix en subtasques que s'executen mitjançant fork() per a la meitat esquerra i compute() per a la dreta, recollint els resultats amb join(). Aquest patró, present en altres algoritmes paral·lelitzats del projecte (com ClosestPairEfficient2D i DiameterConcurrent3D), aprofita la naturalesa divideix i venceràs dels algoritmes implementats.

B. Llindars per a execució seqüencial directa

Un repte en dissenyar solucions paral·leles eficients és determinar quan cal deixar de subdividir tasques. Crear un excés de tasques molt petites pot saturar el sistema amb sobrecàrrega de gestió i sincronització, sense beneficis en temps. Per això s'ha introduït un llindar de mida per a l'execució directa. En concret, s'ha fixat un nombre mínim de punts (o de parelles a comparar) a partir del qual una tasca ja no es divideix més, sinó que s'executa seqüencialment. Aquest llindar s'ha ajustat empíricament per equilibrar càrrega vs. sobrecàrrega. Per exemple, en el càlcul del diàmetre amb força bruta, si n és prou petit (menys del llindar), es calculen les distàncies en aquella subtasca directament; si és més gran, es divideix la llista de punts en dos i es creen subtasques. De forma anàloga, a QuickHull es pot deixar de dividir quan el subconjunt a tractar té molt pocs punts.

L'algoritme ClosestPairEfficient3D il·lustra aquesta tècnica amb el seu llindar THRESHOLD = 50. Quan el nombre de punts a processar és inferior o igual a 50, l'algoritme canvia a un enfocament de força bruta (bruteForceClosestPair(Px)) en comptes de seguir dividint el problema, evitant així la sobrecàrrega de crear tasques massa petites i de la sincronització associada.

Aquesta tècnica és recolzada pel disseny de Fork/Join: la mateixa documentació de Java suggereix usar un llindar per decidir si fer compute() directament o seguir fent fork(). En la nostra implementació, per al Closest Pair es va usar un llindar basat en la mida de subproblema (p. ex. si un subconjunt té menys de 100 punts, aplicar directament l'algorisme seqüencial $O(n^2)$, que per a aquesta mida és trivial). Per al diàmetre en enfocament força bruta paral·lel, es va dividir l'interval d'índexs $[0, n-1]$ dels punts en segments: mentre la longitud del segment superava el llindar, es seguia dividint en dos segments; quan no, es calculava el màxim directament en aquell segment. Això assegura un gra de paral·lelisme òptim: amb problemes grans es creen prou tasques per usar tots els processadors, però finalment cada tasca té una mida suficient perquè la seva sobrecàrrega sigui negligible en comparació amb el treball útil realitzat.

C. Estratègia de particions dinàmiques de la càrrega

Un avantatge del model Fork/Join és que el particionament del treball no cal determinar-lo a priori, sinó que és dinàmic i recursiu. A mesura que un problema es divideix, cada fil genera noves subtasques per al seu propi deque, i altres fils poden robar-les si acaben abans. En lloc de repartir fixament, per exemple, çada fil processarà aquesta meitat de punts", l'enfocament dinàmic permet adaptar-se a la dificultat real de cada subproblema. En les nostres aplicacions, això s'ha manifestat de la forma següent: inicialment, el fil principal llança, posem per cas, la resolució recursiva de l'Closest Pair a L i R en paral·lel. Si la part L resulta molt més costosa (per exemple, perquè L conté més punts després de dividir per x), el fil assignat a R acabarà la seva tasca aviat i robarà alguna subtasca pendent de L. D'aquesta manera, el work stealing efectua un particionament ïmplicít"equilibrat, sense que nosaltres hàgim hagut de predir la càrrega de cada partició per avançat. Aquest particionament dinàmic maximitza l'ús dels recursos de còmput i minimitza el temps d'espera.

En la implementació concurrent del diàmetre (força bruta), s'ha seguit un esquema similar: inicialment es poden crear unes quantes tasques grosses (per exemple, comparar els primers $n/2$ punts amb tots els altres, i els últims $n/2$ amb tots els altres). Aquests tasques, en arrencar, es tornen a subdividir recursivament per rangs d'índexs fins a arribar al llindar. El resultat final és una àrbre de tasques on les fulles

són blocs concrets de comparacions a fer. El ForkJoinPool distribueix aquestes fulles de manera dinàmica: si un fil es queda sense feina, agafa la fulla pendent més gran d'un altre fil. Aquest esquema evita haver de fer manualment un particionament òptim de la matriu de parelles $n \times n$ – l'estructura del programa s'encarrega de dividir i equilibrar. De forma anàloga, a QuickHull 3D, cada faceta processada pot generar tasques per processar sub-facetes, i diferents fils poden assumir facetes diferents a mesura que es descobreixen.

D. Asynchronisme a la interfície d'usuari

Un segon nivell de concurrència permet aïllar la interfície d'usuari de les operacions intensives de càlcul. Les aplicacions JavaFX tenen un fil d'UI dedicat anomenat JavaFX Application Thread, que és l'únic que pot modificar els elements visuals de la interfície. Si aquest fil queda bloquejat per càlculs llargs, la interfície es congela i la usabilitat es deteriora notablement. Per evitar aquest problema, s'ha implementat una arquitectura asíncrona on els càlculs es realitzen en fils en segon pla mentre la interfície roman responsiva.

La classe MainController implementa aquest patró mitjançant l'ús de Task<Void> de JavaFX, que encapsula l'execució d'algoritmes en fils secundaris. Quan l'usuari demana executar un algoritme, es crea una nova tasca asíncrona que s'executa en un fil separat (new Thread(task).start()), deixant lliure el fil principal de la UI per respondre a les interaccions de l'usuari. Aquesta tasca s'encarrega de generar els punts, executar l'algoritme seleccionat, i gestionar els resultats, mantenint actualitzada la UI a través del NotificationService.

Per coordinar les actualitzacions entre fils, la implementació utilitza el patró Platform.runLater(), que planifica actualitzacions de la UI per ser executades en el fil principal d'interfície quan estigui disponible. Això assegura que les actualitzacions visuals (com ara mostrar els punts, actualitzar la barra de progrés, o visualitzar els resultats dels algoritmes) es realitzin de manera segura dins el fil d'UI sense bloquejarlo.

E. Arquitectura de notificacions reactiva

El tercer nivell de concurrència es troba en l'arquitectura global de l'aplicació, que segueix un patró reactiu mitjançant el sistema de notificacions. El NotificationService defineix un contracte per a la comunicació entre components independents, permetent que les parts intensives en computació notifiquin els seus progressos i resultats a la interfície sense acoblament directe.

Aquest disseny es recolza en una variació del patró Observador implementat amb NotificationServiceImpl que connecta el controlador amb la vista. La implementació de UINotificationHandler a la classe MainView conté els mètodes de callback que responen a les notificacions del controlador: onPointGenerationStarted, onPointGenerationCompleted, onAlgorithmStarted, onAlgorithmProgress, onAlgorithmCompleted i onComputationError.

L'avantatge d'aquesta arquitectura reactiva és triple:

Desacoblament: Els algoritmes no necessiten conèixer res sobre la visualització, simplement notifiquen els seus progressos. La vista pot reaccionar de manera adequada sense dependències directes amb la implementació algorísmica.

Composabilitat: Es poden afegir nous escoltadors sense modificar el codi existent.

Paral·lelisme implícit: Les notificacions poden provenir de múltiples fonts asíncrones i ser processades de manera consistent, mantenint l'estat coherent.

Aquest enfocament facilita la comunicació entre els fils de treball que executen algoritmes en segon pla i la interfície d'usuari, permetent actualitzacions progressives i interactivitat contínua.

F. Gestió d'interactivitat i renderització 3D

La interfície d'usuari 3D afegeix una capa addicional de complexitat concurrencial. El renderitzat 3D amb JavaFX requereix una gestió eficient dels recursos gràfics en paral·lel amb els càlculs algorísmics. A MainView, la funció setup3DScene() configura una subescena 3D amb un gràfic de nodes, càmera i il·luminació, mentre que les funcions setupMouse3DHandlers() i setupKeyboard3DHandlers() gestionen els esdeveniments d'interacció.

Aquestes interaccions 3D (rotació, zoom, navegació) s'executen també en el fil d'UI, però utilitzen transformacions eficients que aprofiten l'acceleració GPU per mantenir una alta taxa de visualització. Quan es visualitzen els resultats dels algoritmes en 3D, l'aplicació ha de gestionar amb cura la transició entre els fils de càlcul i el fil de renderitzat, assegurant que les actualitzacions visuals es realitzin sense bloquejar la interfície.

Una característica destacable és l'ús de fils parallels per a animacions i visualitzacions complexes, com l'enfocament automàtic en els punts resultat, que implementa una animació de rotació utilitzant javafx.animation.Timeline. Aquesta aproximació permet animar la vista 3D mentre altres operacions continuen en segon pla, demostrant l'extensibilitat del model de concurrència utilitzat.

G. Millora iterativa i resultat global

La combinació d'aquests tres nivells de concurrència - paral·lelisme algorítmic (Fork/Join), asynchronisme d'UI (Tasks), i arquitectura reactiva (Notificacions) - confereix a l'aplicació un rendiment òptim en entorns multinucli moderns. Els usuaris poden iniciar múltiples execucions d'algoritmes complexos, interactuar amb la interfície fluidament durant els càlculs, i visualitzar resultats parcials o finals sense experimentar congelacions o retards perceptibles.

Mitjançant Fork/Join s'ha aconseguit paral·lelitzar de forma natural els algoritmes de dividir i vèncer i les cerques exhaustives, obtenint acceleracions significatives en màquines multinucli. L'ús de work stealing ha garantit un bon escalat

sense haver de micro-gestionar el repartiment de treball, i els llindars han assegurat que el cost de gestionar fils i tasques no superi el benefici del paral·lelisme.

Amb l'aïllament del fil d'interfície mitjançant tasques asíncrones, l'aplicació manté una interactivitat constant independentment de la càrrega computacional. I amb el sistema de notificacions reactiu, s'aconsegueix una coordinació eficient i desacoblada entre els diversos components del sistema, facilitant-ne l'extensibilitat i el manteniment.

Aquest disseny integral de concurrència multinivell constitueix un exemple pràctic de com diferents paradigmes de programació concurrent poden combinar-se per obtenir el màxim rendiment i usabilitat en aplicacions científiques modernes.

VI. Anàlisi del cost computacional

En aquest apartat es presenta una anàlisi formal de la complexitat computacional (temps d'execució asimptòtic) de cada algoritme descrit, diferenciant explícitament els casos en dues dimensions (2D) i tres dimensions (3D). Es considera tant la complexitat en el pitjor cas com en casos mitjans esperats quan és rellevant, i s'adjunten comparatives entre diferents versions. També es discuteix breument l'ús de memòria quan és notable. A continuació es detallen les complexitats per al Closest Pair, l'algorisme QuickHull (envolupant convexa) i el càlcul de diàmetre:

A. Complexitat dels algorismes de Parell més Proper

Força bruta 2D/3D: La complexitat és $O(n^2)$ en ambdues dimensions, ja que es calculen distàncies per a cada parella de punts. Aquesta complexitat és idèntica en 3D, només amb un cost constant més alt per càlcul de distàncies en l'espai (sumes de quadrats addicionals). No hi ha diferència asimptòtica entre 2D i 3D per a l'enfocament naïf; és clarament intractable per a grans n.

Divideix i venceràs 2D: L'algorisme recursiu de Closest Pair en el pla té complexitat $T(n) = 2T(n/2) + O(n)$, resolent-se a $T(n) = O(n \log n)$. Aquest $O(n \log n)$ és òptim en el model determinista algebraic. La justificació és que el cost de combinar els resultats (comparant punts a la franja central) és lineal gràcies a la limitació constant de veïns a considerar . Per tant, en 2D, l'algorisme aconsegueix un rendiment quasi lineal, molt millor que el quadràtic inicial.

Divideix i venceràs 3D: En tres dimensions, si apliquem directament la mateixa estratègia (dividir per pla $x = x0$ i considerar un "slab" d'amplada 2δ), la combinació requereix comparar cada punt del slab amb diversos veïns en un entorn 3D. Tot i que també existeix una cota constant de veïns possibles (derivada del problema de col·locació de 13 esferes en contacte, etc.), la implementació directa pot introduir un factor logarítmic addicional en reorganitzar i buscar veïns dins del slab. Un anàlisi acurat dona recurrences com $T(n) = 2T(n/2) + O(n \log n)$, que resol $T(n) = O(n \log^2 n)$. Aquesta seria la complexitat en el pitjor cas teòric d'una implementació no optimitzada en 3D. No obstant, existeixen tècniques més avançades per reequilibrar la divisió (p. ex. escollint el pla $x0$ de tall de manera que el nombre de punts al slab central siga sublineal) que permeten recuperar $T(n) = O(n \log n)$ també en 3D. En la nostra anàlisi considerarem que, pràcticament, el mètode de divideix i venceràs en 3D també s'acosta a $O(n \log n)$, ja que s'han tingut en compte optimitzacions com l'ordenació adequada i la limitació explícita de punts candidats en la franja (mitjançant estructures auxiliars o projectant a 2D per fer la comprovació). En qualsevol cas, és important notar que la constant de proporcionalitat en 3D és superior (comparacions més costoses i més punts veïns a verificar per punt).

Versió amb KD-Tree 2D/3D: La construcció d'un KD-Tree de n punts té un cost mitjà de $O(n \log n)$. Un cop construït, es pot trobar la distància mínima en temps esperat $O(n \log n)$ (realitzant n cerques de veí proper, cadascuna $O(\log n)$ de mitjana) o fins i tot més eficient integrant la cerca en una única passada amb poda. Formalment, la complexitat mitjana és $O(n \log n)$, amb un pitjor cas de $O(n^2)$ si el KD-Tree es degrada. En 2D i 3D això es manté semblant: l'altura de l'arbre és $O(\log n)$ en distribucions no patològiques, i la cerca de nearest neighbor descarta mig espai a cada pas. En dimensions fixades (2 o 3), el KD-Tree és efectiu; cal destacar que en dimensions molt altes el rendiment pot caure (curse of dimensionality), però no és el cas aquí. En resum, Closest Pair via KD-Tree assoleix també $O(n \log n)$ esperat tant en 2D com 3D. La diferència principal és en la constant: en 3D, la inserció i comparació en l'arbre manegen 3 coordenades en lloc de 2, però asimptòticament és el mateix ordre.

A tall de resum, per al problema del parell més proper, l'ordre de complexitats és:

Algorisme Closest Pair	2D (pla)	3D (espai)
Força bruta	$O(n^2)$	$O(n^2)$
Divideix & venceràs	$O(n \log n)$	$O(n \log n)$ (esperat)
KD-Tree (cerca veí més prop)	$O(n \log n)$ (esperat)	$O(n \log n)$ (esperat)

Taula 1. Complexitat del problema del parell més proper en 2D i 3D

(En 3D, el D-i-V pot ser $O(n \log^2 n)$ en el cas pitjor no optimitzat, però $O(n \log n)$ amb refinaments; KD-Tree és esperat.)

B. Complexitat de l'algorisme QuickHull (Envolupant convexa)

L'algorisme QuickHull presenta un patró de cost similar al QuickSort: rendiment mitjà molt bo, però amb un cas pitjor desfavorable:

En 2D: De mitjana, QuickHull ordena parcialment els punts i divideix l'espai de manera equilibrada, assolint una complexitat de $\Theta(n \log n)$. La major part del cost prové de trobar els punts més externs en cada pas (escanejos lineals) i de la recursivitat en subdivisions més petites. En

el millor dels casos, cada pas divideix el conjunt gairebé per la meitat (situació idealment equilibrada) i el recurs segueix un esquema similar a $T(n) = 2T(n/2) + O(n)$, donant $O(n \log n)$. En el pitjor cas, tanmateix, QuickHull pot particionar de manera molt desequilibrada (per exemple, si els punts ja formen una forma convexa allargada, cada vegada només es troba un punt a l'exterior i queden $n - 1$ punts a l'altra banda). En aquest escenari extrem, la recurrència esdevé $T(n) = T(n - 1) + O(n)$, i la solució és $T(n) = O(n^2)$. Aquest pitjor cas és anàleg al de QuickSort quan l'element pivot és sempre el mínim o màxim. No obstant, en distribucions aleatòries de punts, la probabilitat d'una degeneració així és molt baixa; QuickHull sol comportar-se proper al cas mitjà. Val a dir que altres algorismes garanteixen $O(n \log n)$ en el pitjor cas, per la qual cosa en aplicacions crítiques es podria preferir aquells. En el nostre context, QuickHull 2D ha estat adequat i no s'ha trobat prop de la complexitat quadràtica en cap prova pràctica.

En 3D: QuickHull generalitzat a 3D té també complexitat esperada $O(n \log n)$. La construcció inicial del tetraedre és $O(n)$ (cercar extrems) i després cada punt s'assigna a alguna faceta; la recursió processa facetes i pot analogament equilibrar-se o no. El pitjor cas de $O(n^2)$ igualment ocorre quan, successivament, només un punt nou defineix la hull incrementant-la de manera lineal (per exemple, si tots els punts estan distribuïts sobre una única cara plana, QuickHull aniria afegint un punt cada cop). En general, per a dimensions d, QuickHull pot empitjorar a $O(n^2)$ en casos degenerats, tot i que sota certes condicions de posició general i assumpcions d'equilibri la complexitat esperada es manté $O(n \log n)$ fins a $d = 3$. Val la pena mencionar que l'envolupant convexa en 3D es pot calcular també amb algoritmes deterministes de $O(n \log n)$ en el pitjor cas (p. ex. algoritmes incrementals amb estructures d'adjacència). En resum, QuickHull 3D comparteix la mateixa pauta: **mitjana** : $O(n \log n)$, ****pitjor** : $O(n^2)$.

Mida de sortida (output-sensitive): La complexitat discutida normalment assumeix n molt més gran que h (nombre de vèrtexs hull). En situacions on h és petit (p. ex. punts en forma de núvol compacte amb pocs vèrtexs extrems), hi ha algoritmes més afinats amb complexitat output-sensitive com l'algorisme de Chan ($O(n \log h)$). QuickHull no és output-sensitive estricte (depèn principalment de n i no de h), però de manera implícita si h és petit, possiblement els particionaments es fan més equilibrats i la constant de temps és menor, tot i que l'ordre segueix sent $n \log n$ en qualsevol cas no degenerat.

En resum, per a envolupant convexa:

(QuickHull té un risc pitjor n^2, però en pràctica usual funciona en $n \log n$; altres algoritmes de hull garanteixen $n \log n$ sempre. Els valors són similars per 2D i 3D quant a ordre, canviant constants i estructura interna.)

Algorisme de hull convex	2D (pla)	3D (espai)
QuickHull (mitjana)	$O(n \log n)$	$O(n \log n)$
QuickHull (pitjor cas)	$O(n^2)$	$O(n^2)$

Taula 2. Complexitat de l'algorisme d'envolupant convexa en 2D i 3D

C. Complexitat del càlcul de Diàmetre de punts

Força bruta 2D/3D: Com ja s'ha comentat, comparar totes les parelles de n punts comporta $O(n^2)$ temps. Aquest és el límit inferior trivial, i desafortunadament, per al diàmetre en general no hi ha algoritmes subquadràtics sense reduir el problema (fins i tot en dimensions modestes, trobar la parella més distant directament té una complexitat inherentment alta si es fa de manera exhaustiva). En 3D la situació és la mateixa asimptòticament ($O(n^2)$), només que calcular cada distància suposa unes quantes operacions més (restes i quadrats de 3 components en comptes de 2). Per tant, la força bruta és quadràtica en qualsevol dimensió.

Via envolupant convexa (QuickHull) 2D: En el pla, utilitzant la hull convexa i antípodes, s'assoleix $O(n \log n)$ en el pitjor cas . Aquest temps inclou la construcció de la hull ($O(n \log n)$) i el càlcul del diàmetre sobre la hull ($O(h)$). Com que $h \leq n$, en el pitjor cas això és $O(n)$ per al pas d'antípodes, de manera que domina el $O(n \log n)$ de la hull. És important destacar que $O(n \log n)$ és un gran avenç respecte $O(n^2)$; de fet, per a aquest problema és òptim en el model algebraic de comparacions, ja que calcular la hull convexa ja requereix $n \log n$ i el diàmetre no pot ser més difícil que això (en essència, el diàmetre és un "subproducte" de la hull).

Via envolupant convexa 3D: En l'espai, l'estratègia d'utilitzar l'embolcall convex també redueix dràsticament la complexitat, però l'anàlisi és una mica més subtil. Construir la hull 3D és $O(n \log n)$ en general. Un cop es té el políedre convex amb h vèrtexs, una solució senzilla és provar totes les parelles de vèrtexs ($O(h^2)$) per trobar la distància màxima. En el pitjor dels casos $h = n$, la qual cosa tornaria a $O(n^2)$, però en casos típics h creix molt més lentament que n. Existeixen mètodes més intel·ligents per al diàmetre 3D: es podria estendre la idea dels punts antípodes (qualsevol parella de punts més allunyada en un políedre convex són punts antípodes en alguna direcció). No obstant, enumerar totes les parelles antípodes en 3D no és tan eficient com en 2D perquè pot haver-hi força combinacions (cada vèrtex pot tindre diversos antípodes corresponents a diferents facetes). A efectes pràctics, l'algoritme implementat (hull + bucle doble sobre vèrtexs) funciona bé si h és moderat. La complexitat en funció de n es pot descriure com $O(n \log n + h^2)$. En el millor cas (punt cloud esfèric amb pocs vèrtexs extrems) h^2 és negligible respecte n, i el temps és aproximadament $O(n \log n)$. En el pitjor cas (tots els punts a la superfície convexa, com en una distribució esfèrica uniforme), $h = n$ i aquesta estratègia es degrada a $O(n^2)$. Cal mencionar, però, que fins i tot en aquest cas pitjor, la constant de

proporcionalitat de n^2 és inferior: comparar n vèrtexs contra n vèrtexs és més ràpid que comparar n punts contra n punts originals? Realment és el mateix ordre, però en la pràctica, la fase de hull ja ha filtrat molts punts i possiblement fa que la constant sigui menor (p. ex. operant sobre estructures de dades més amigables a memòria cau, etc.).

Algoritmes especialitzats: Existeixen algunes recerques teòriques que indiquen que en dimensions fixes, el problema de diàmetre es pot resoldre en temps lineal esperat utilitzant algorismes més complicats (relacionats amb duplicació de dimensions o amb mètodes randomitzats). Tanmateix, en entorns convencionals i per a $d = 2, 3$, la via de la hull convexa és l'abordatge estàndard. També sota models de decisió deterministes, $O(n \log n)$ és òptim en 2D per al diàmetre, ja que la hull és necessària.

A continuació es resumeixen les complexitats:

Algorisme Diàmetre	2D (pla)	3D (espai)
Força bruta (totes parelles)	$O(n^2)$	$O(n^2)$
Hull convexa + antípodes	$O(n \log n)$	$O(n \log n + h^2)$ (veure text)
(Pitjor cas hull+diam)	$O(n \log n)$	$O(n \log n) + O(n^2) = O(n^2)$

Taula 3. Complexitat del càlcul de diàmetre en 2D i 3D

En el cas 3D, h^2 pot arribar a n^2 en la pitjor configuració, fent que l'algorisme torni a ser quadràtic. En situacions habituals, però, $h \ll n$ i per tant el terme h^2 és secundari comparat amb $n \log n$.

Finalment, val la pena destacar que, un cop paral·lelitzats aquests algorismes (vegeu sec. 2), el cost de muralla (wall-clock time) efectiu millora proporcionalment al nombre de processadors per als enfocaments força bruta i divideix i vèncer (idealment prop d'un factor p amb p processadors, menys l'impacte de la sobrecàrrega). No obstant, l'ordre de complexitat asimptòtica amb relació a n per als algorismes continua sent la descrita anteriorment, ja que la paral·lelització no redueix l'ordre en n sinó la constant de temps mitjançant l'execució concurrent.

VII. Anàlisi funcional dels algoritmes (Closest Pair, QuickHull, Diameter)

En aquesta secció es descriu detalladament el funcionament de cadascun dels algorismes principals implementats, complementant l'anàlisi de cost amb una visió més procedimental. Es posarà èmfasi en els aspectes geomètrics i lògics que garanteixen la correcció i eficiència, incloent la justificació de les restriccions geomètriques (com la dels veïns a la franja central en Closest Pair) i generalitzant els algorismes de hull convex i diàmetre a N dimensions de forma formal, amb presentació de pseudocodi acadèmic.

A. Algorisme Closest Pair (Parell més Proper)

Descripció general: Donat un conjunt P de n punts en el pla (o espai), l'objectiu és trobar la parella (p, q) amb distància Euclidea mínima. L'algorisme de dividir i vèncer realitzat consta de les fases: ordenació, divisió recursiva i combinació. A continuació, es descriu el flux en 2D, indicant adaptacions a 3D:

Ordenació inicial: S'ordenen tots els punts per coordenada x creixent. Això permet dividir per una línia vertical a mitges i també facilita la fase de fusió posterior. (En implementació, es pot mantenir també una llista ordenada per y si cal optimitzar més la fusió).

Divisió recursiva: Es divideixen els n punts en dues meitats L i R de mida aproximadament $n/2$ cadascuna (punt mig pel seu x). Es crida recursivament l'algorisme per a L i per a R. Suposem que obtenim δL i δR, les mínimes distàncies en cada meitat, i respectivament parelles (aL, bL) i (aR, bR) com a més properes de L i R. Definim $\delta = \min(\delta L, \delta R)$ i retindrem la millor parella trobada fins ara $(a{\cdot}b)$ corresponent a δ.

Fusió – franja central: Aquesta és la part crítica. Existeix la possibilitat que la parella més propera tingui un punt a L i l'altre a R. Tots dos punts haurien d'estar prop de la línia de divisió (si no, cadascun ja hauria trobat un altre punt més proper dins la seva meitat). En concret, només cal considerar els punts situats a distància $\leq \delta$ de la línia divisòria. Es filtren aquests punts de L i R per obtenir un subconjunt S ("franja") de punts a la zona central. Aquests punts de S es projecten (o es consideren) en l'eix y ordenats per y. Llavors, per a cada punt $p \in S$ es comprova la distància a altres punts propers en y. Gràcies a la propietat geomètrica abans esmentada, no cal comparar p amb tots els punts de S, sinó només amb els següents pocs punts en l'ordre de y (com a màxim 6 punts en 2D). Això es deu a que si es pogués trobar més de 6 punts en aquest quadrat de costat δ, violaria la definició de δ com a distància mínima intrapartició. En la implementació típica, es recorre l'array ordenat S incrementant i es fan comparacions fins que la distància vertical supera δ. Si es troba una parella amb distància menor que δ, s'actualitza δ i la parella $(a{\cdot}b)$.

Resultat: Es retorna la parella $(a{\cdot}b)$ i la seva distància δ com la més propera de tot P.

En 3D, l'esquema general és idèntic, però la fusió és més complexa perquè la franja central esdevé un volum (slab rectangular) de gruix 2δ. El principi, tanmateix, continua: si dos punts (un de cada meitat) estan a menys de δ, han d'estar dins aquest slab. Aleshores, es pot projectar aquest slab sobre un pla perpendicular (per exemple, el pla YZ central) per reduir el problema a dues dimensions auxiliars. En fer això, la condició equivalent és que cada punt del slab s'ha de comparar amb punts veïns dins d'un cert entorn limitat. Mitjançant subdivisions en cubs de costat $\delta/2$ dins el slab, es garanteix que cada punt tindrà a tot estirar 125 candidats (constant 5^3) a comprovar. Encara que 125 és una cota força laxa, en la pràctica molts menys punts caldrà comparar, ja que la majoria de cubs seran buits o amb un sol punt (no poden tenir-ne dos a menys que siguin més propers que δ). Així, la fusió en 3D és $O(n)$ també, simplement amb una constant major.

Anàlisi geomètrica de les restriccions (cas 2D): Per què 6 punts? La idea és imaginar un rectangle de dimensions $\delta \times 2\delta$ (ample δ a cada costat de la línia central i alt 2δ). Dividint aquest rectangle en cel·les de $\delta/2 \times \delta/2$, hi caben com a màxim 6 punts abans que dos hagin de caure en la mateixa cel·la, la qual cosa implicaria que estan a distància $\leq \delta/\sqrt{2} < \delta$, contradint que δ era la mínima intra-partició. Això justifica empíricament la constant 6. És un resultat clau que assegura que la fusió no degenera en un bucle $O(n^2)$. De manera similar, en 3D els punts en una caixa de $\delta \times \delta \times \delta$ estan limitats pel "kissing number" de l'esfera (12 punts en contacte) i subdivisions fines estableixen constants (teòricament 12 o 15, o 125 en la prova general) que acoten la complexitat lineal.

Pseudocodi (Closest Pair 2D): A continuació es proporciona un pseudocodi simplificat de l'algorisme de divideix i venceràs en 2D:

Algorithm 2 ClosestPair(P)

Input: $P = \{p_1, \ldots, p_n\}$ conjunt de punts al pla
Output: Parella (a^*, b^*) amb distància mínima δ
Ordena els punts de P per coordenada x per obtenir l'array P_x
if $n \leq 3$ **then**
    Calcula totes les distàncies entre parelles (força bruta)
    return parella amb distància mínima
end if
Divideix P_x en meitats L i R de mida $\approx n/2$
$x_{\text{mid}} \leftarrow$ coordenada x màxima de L (o mínima de R)
$(a_L, b_L, \delta_L) \leftarrow$ CLOSESTPAIR(L)
$(a_R, b_R, \delta_R) \leftarrow$ CLOSESTPAIR(R)
$\delta \leftarrow \min(\delta_L, \delta_R)$
$(a^*, b^*) \leftarrow$ millor de (a_L, b_L) i (a_R, b_R)
$S \leftarrow \{p \in P \mid |p.x - x_{\text{mid}}| < \delta\}$ ordenat per coordenada y
for cada p_i en S **do**
    for $j = i + 1$ fins $|S|$ **while** $S[j].y - p_i.y < \delta$ **do**
        $d \leftarrow \text{dist}(p_i, S[j])$
        if $d < \delta$ **then**
            $\delta \leftarrow d$
            $(a^*, b^*) \leftarrow (p_i, S[j])$
        end if
    end for
end for
return (a^*, b^*, δ)

Aquest esquema ressalta la fusió en el pas 6, on la condició del while assegura que només es comparen punts amb diferència en Y menor que δ (que implica j-i ≤ 6 aproximadament). La correctesa ve donada per inducció (llegada de la recursió) més la garantia geomètrica que es troben parelles inter-particions si n'hi ha. El cost és $T(n) = 2T(n/2) + O(n)$, dominat per l'ordenació inicial $O(n \log n)$ i la recursió.

Correctesa: L'algorisme sempre troba el parell més proper. Això es pot argumentar per contradicció: si la parella més pròxima estigués separada entre L i R, la detectaríem a la franja S perquè estarien a distància < δ que és $\leq$ distància intra-particions. I dins cada meitat, la recursió ho garanteix per hipòtesi inductiva. La fusió no perd cap cas gràcies a considerar tots els punts a $\leq \delta$ de la divisió.

Generalització a N dimensions: L'estratègia de dividir per la meitat seguint una coordenada principal es pot estendre a d dimensions. En lloc d'una línia, es divideix per un hiperplà. La "franja" central esdevé un embut hiperdimensional de gruix 2δ. Tot i que la constant de veïns a considerar creix ràpidament amb d, es manté com una constant independent de n (és el problema de la distribució òptima d'hiperesferes en un hipercúb). Per tant, per a dimensió fixa d, l'algorisme de closest pair continua sent $O(n \log n)$. En implementació, les complicacions augmenten (cal projectar el "slab" a $d-1$ dimensions i aplicar recursivament l'algorisme, etc.), però el principi de divisió i poda geomètrica roman igual.

B. Algorisme QuickHull i generalització a N dimensions

Descripció general: QuickHull construeix la convex hull triant iterativament punts extrems. Ja es va esbossar a la secció 1.3. A continuació es formalitza en pseudocodi acadèmic i es comenta com generalitzar-lo a dimensions superiors:

Pseudocodi (QuickHull en 2D):

Algorithm 3 QuickHull2D

Input: $P = \{p_1, \ldots, p_n\}$ punts al pla
Output: H conjunt de vèrtexs de l'envolupant convexa
Troba els punts $minX$ i $maxX$ de P i afegeix-los a H
Divideix P en dos subconjunts:
    $S_1 \leftarrow$ punts sobre la línia $(minX, maxX)$
    $S_2 \leftarrow$ punts sota la línia $(minX, maxX)$
Executa HULLRECURSIU($minX, maxX, S_1, H$)
Executa HULLRECURSIU($maxX, minX, S_2, H$)
return H

Algorithm 4 HullRecursiu(A, B, S, H)

if S és buit **then**
    return (A, B) ja és a H
else
    Troba $C \in S$ més llunyà de la línia AB
    Afegeix C a H entre A i B
    $S_{AC} \leftarrow \{p \in S \mid p$ està a la dreta de la línia $AC\}$
    $S_{CB} \leftarrow \{p \in S \mid p$ està a la dreta de la línia $CB\}$
    HULLRECURSIU(A, C, S_{AC}, H)
    HULLRECURSIU(C, B, S_{CB}, H)
end if

En aquest pseudocodi, "dreta de la línia" implica que el punt està fora de la semi-plà definita per l'aresta en sentit de la hull (utilitzant el criteri de producte vectorial 2D per determinar si un punt està a l'esquerra o dreta d'un segment

orientat). L'algorisme inicia amb els dos extrems horitzontals i obté dues meitats de la hull (superior i inferior) recursivament. Cada crida troba un nou vèrtex C i subdivideix. Quan no queden punts fora, la recursió torna.

Generalització a N dimensions: En dimensions superiors, QuickHull es pot entendre com un algorisme incremental:

Símplex inicial: Es comença trobant $d + 1$ punts afínlinealment independents que formen un símplex inicial en $\mathbb{R}^d$. Per exemple, en 3D s'ha de trobar un tetraedre. En general, es pot: trobar extrems en cada eix per assegurar que hi ha almenys dues dimensions de separació i després afegir punts extrems en altres direccions fins tenir $d + 1$ punts no coplanars. Aquest pas es pot fer en $O(d{\cdot}n)$ provant candidats extrems.

Assignació de punts a facetes: Un cop tenim un símplex inicial (que és una hull provisional), la resta de punts es classifiquen segons quina faceta del símplex poden veure des de fora. Cada faceta f de la hull manté una llista S_f de punts que estan al seu exterior (costat visible).

Iteració: Es selecciona una faceta f que tingui punts a l'exterior (si no en té, és definitiu). Sigui $pmax$ el punt de Sf que està més lluny del pla que defineix f (és el punt més "fora" respecte aquesta cara). Aquest punt $pmax$ ha de ser un vèrtex de la hull (és impossible construir una hull convexa que no el contingui, ja que f era visible i $pmax$ és el més extrem). Es crea així un nou conjunt de facetes connectant p_{max} amb la frontera de f. En fer això:

Totes les facetes que eren visibles des de p_{max} (no només f sinó possiblement algunes adjacents formant una "cara" poligonal) s'eliminen de la hull. Es formen noves facetes amb p_{max} i cada aresta del "forat" deixat per les facetes eliminades. Es reassignen els punts que estaven fora de les facetes eliminades a les noves facetes si aquests punts queden fora d'elles. (Els punts que ara queden a l'interior de la hull actualitzada s'eliminen de consideració).

Repetir: Es repeteix el pas 3 fins que no queden punts en cap llista S_f (és a dir, tots els punts o bé són vèrtexs de hull o estan dins de la hull formada).

Aquest procés és exactament l'estratègia de QuickHull generalitzada (coneguda també com l'algorisme "beneath-beyond"). El pseudocodi a alt nivell podria ser:

Aquest pseudocodi és més complex, però captura la idea: a cada iteració incorporem un nou vèrtex i eliminem un "pegat" de facetes reemplaçant-lo per altres. La correctesa ve assegurada perquè sempre triem un punt exterior i en afegir-lo mantenim la convexitat eliminant just les cares que eren visibles. En acabar, H conté només facetes no visibles des de cap punt interior – és a dir, és la hull convexa completa.

Pseudocodi acadèmic (més concís): També es pot expressar de manera recursiva similar al cas 2D però amb facetes en comptes d'arestes. No obstant, esdevé força enrevessat d'escriure recursivament per a $d > 2$. El procediment explicat és suficientment formal per als propòsits acadèmics.

Generalització de correctesa i complexitat: QuickHull és correcte en $\mathbb{R}^d$ perquè garanteix que tots els punts fora la hull eventualment es converteixen en vèrtexs. La complexitat esperada segueix sent $O(n \log n)$ per a $d = 2, 3$ i en general per a d fix, encara que per $d \geq 4$ l'anàlisi és més complicada i pot aparèixer cost output-sensitive ($O(n \log r)$ on r és el nombre de vèrtexs de hull) sota certes condicions. En qualsevol cas, QuickHull s'utilitza en biblioteques per a dimensions 2 i 3 principalment, on funciona molt bé.

Algorithm 5 QuickHullND(P)

$H \leftarrow$ CALCULARSIMPLESINICIAL(P) ▷ retorna conjunt inicial de facetes
for all $p \in P$ que no són vèrtexs de H **do**
  Determina faceta visible f tal que p està fora (si existeix)
  Afegeix p a la llista S_f
end for
while existeix alguna faceta f tal que $S_f \neq \emptyset$ **do**
  Tria f amb $S_f \neq \emptyset$
  $p_{\text{max}} \leftarrow$ punt de S_f amb màxima distància a $pla(f)$
  for all facetes g visibles des de p_{max} **do**
    Elimina g de H
    Mou els punts S_g a una llista unificada U
  end for
  Crea noves facetes connectant p_{max} amb la vora deixada (cicle d'arestes contigües de les facetes eliminades)
  for all $q \in U$ **do**
    if q està fora d'alguna nova faceta h **then**
      Afegeix q a S_h
    end if
  end for
end while
return H ▷ Les facetes restants formen l'envolupant convexa

Nota sobre robustesa: En qualsevol implementació de hull convex cal tenir cura amb casos degenerats (punts colineals o coplanars, punts duplicats). L'algorisme s'ha d'adaptar per incloure tots els punts colineals del contorn si es desitja (per exemple, en QuickHull 2D normalment s'inclouen extrems però no necessàriament tots els punts colineals intermedis si es cerca només els vèrtexs extrems). En el nostre enfocament, ens hem centrat en retornar només els vèrtexs extremals.

C. Algorisme de Diàmetre i generalització a N dimensions

Descripció detallada (2D): Un cop es té la hull convexa de P en el pla, el diàmetre es calcula detectant les parelles de punts antípodes. L'algorisme de rotating calipers es pot descriure així:

Obtenir la llista de vèrtexs de la hull ordenats circularment (p. ex. en sentit horari). Trobar un parell inicial de punts antípodes. Una forma és començar amb el vèrtex de hull amb mínima y (el més baix) i el vèrtex oposat amb màxima y (el més alt). Aquests segur que són antípodes en

direcció vertical. A continuació, es fan "rodar" dos calibres (rectes paral·leles tangents a la hull) al voltant de la hull. Formalment: per cada vèrtex i de la hull (en llista circular), es mira la distància entre i i el vèrtex j tal que l'aresta $[j, j+1]$ segueix sent antípoda de i fins que i avanci. Una implementació clàssica és:

Fixar $i = 0$ (un vèrtex arbitrari, per exemple el més a l'esquerra) i trobar j tal que l'àrea del triangle $(i, i+1, j)$ és màxima mentre j incrementa (això identifica el punt més llunyà angularment). Després, per cada i incrementant, moure j endavant mentre $(i, i+1, j+1)$ forma àrea més gran que $(i, i+1, j)$. En 2D, això equival a mantenir j com l'índex del punt antípod. A cada pas, calcula la distància $d(i, j)$ i actualitza el màxim.

El resultat és la màxima distància trobada i les parelles corresponents (pot haver-hi més d'una parella amb la mateixa distància màxima – p. ex. un rectangle allargat té dos diàmetres iguals).

Aquest algoritme funciona en $O(h)$, ja que tant i com j fan com a molt una volta completa a la hull. La justificació és que les parelles més allunyades són antípodes, i a mesura que recorrem la hull sistemàticament les explorem totes.

Dimensió superior: En 3D (i general d), el diàmetre segueix complint que està entre punts de la hull convexa. Per trobar-lo, però, no hi ha un procediment tan directe com calipers. Una estratègia podria ser: calcular per a cada vèrtex p de la hull quin altre vèrtex q maximitza la distància $|pq|$. Això es pot fer navegant la llista de vèrtexs, però en 3D la hull és un graf planar i s'haurien de mirar vèrtexs no necessàriament adjacents. Una opció és fer el següent:

Triar un vèrtex arbitrari v_0. Trobar el punt $w0$ més allunyat de $v0$ (fent un recorregut lineal per tots vèrtexs, o mantenint coordenades extremes). Després, a partir de $w0$, trobar $v1$ que és el més allunyat de $w0$. (Aquest $(w0, v_1)$ és una primera estimació de diàmetre; és un truc conegut que amb dos passes a partir d'un punt qualsevol obtens un parell de quasi-diàmetre). A continuació, es podria intentar un algorisme similar a calipers però girant un pla en 3D, cosa complicada; alternativament, es pot fer prova exhaustiva sobre vèrtexs: per cada vèrtex p, tenir marcat el candidat q més llunyà i mirar també els veïns de q.

Un enfocament més simple (encara que $O(h^2)$) és realitzar doble bucle: per cada vèrtex p en hull, calcular distàncies a tots q i portar el màxim. En implementació, això pot ser millorat utilitzant simetries: si (p, q) és diàmetre, cap altre punt pot estar més lluny de p que q. Així es pot guardar l'index del més llunyà trobat per al següent.

Pseudocodi (mètode exhaustiu hull):

Aquest mètode és $O(h^2)$. Amb $h = n$ en pitjor cas, és $O(n^2)$.

Generalització a N dimensions: Conceptualment, el diàmetre en $\mathbb{R}^d$ sempre estarà format per dos vèrtexs de la hull convexa. La forma més directa és comprovar totes les parelles de vèrtexs. Això és $O(h^2)$, que és acceptable si h és significativament menor que n. Per a $d = 2$, vam veure

Algorithm 6 DiameterHull(H)

Input: Llista de vèrtexs $H = \{v_0, \ldots, v_{h-1}\}$ de l'envolupant convexa
Output: Parella (a, b) amb distància màxima
$maxDist \leftarrow 0$
$(a, b) \leftarrow$ `None`
for $i = 0$ to $h - 1$ **do**
  for $j = i + 1$ to $h - 1$ **do**
    $d \leftarrow \text{dist}(v_i, v_j)$
    if $d > maxDist$ **then**
      $maxDist \leftarrow d$
      $(a, b) \leftarrow (v_i, v_j)$
    end if
  end for
end for
return $(a, b, maxDist)$

que es pot fer millor ($O(h)$). Per a $d = 3$, hi ha algunes heurístiques (com rodaments de plans) però en general la complexitat pot ser $O(h^2)$ en el pitjor cas. Per a dimensions altes, el problema de diàmetre es pot reduir a un problema de ANN (approximate nearest neighbors) en l'espai de dimensions d o utilitzar propietats de dualitat (diàmetre equival a dues meitats del minimal bounding diametral ball, etc.), però això ja surt de l'abast.

Pseudocodi generalitzat:

Algorithm 7 DiameterViaHull(P)

Input: Conjunt de punts $P \subset \mathbb{R}^d$
Output: Parella (a, b) de punts més distants
$H \leftarrow$ ConvexHull(P) ▷ Calcula l'envolupant convexa en d dimensions
$V \leftarrow$ llista de vèrtexs de H
$maxDist \leftarrow 0$
$(a, b) \leftarrow$ `None`
for all parelles $(u, v) \in V \times V$ **do**
  $d \leftarrow \|u - v\|$ ▷ Distància Euclidea
  if $d > maxDist$ **then**
    $maxDist \leftarrow d$
    $(a, b) \leftarrow (u, v)$
  end if
end for
return $(a, b, maxDist)$

Aquest pseudocodi, encara que simple, és vàlid per a qualsevol dimensió d. En 2D es pot millorar substituint el doble bucle per rotating calipers. En 3D, la complexitat de l'enumeració de parelles de vèrtexs pot ser gran, però és correcta. En recerca, s'han proposat algoritmes de diàmetre sub-h^2, però són força complicats i no necessàriament milloren en la pràctica tret que h siga realment gran.

Conclusions: L'anàlisi funcional mostra que cada algoritme aprofita propietats geomètriques particulars: Closest Pair fa servir la localitat espacial (franges), QuickHull s'a-

profita de punts extrems i estructures recursives de facetes, i Diameter redueix l'espai de cerca als vèrtexs de la hull convexa i aplica tècniques com antípodes. Aquests enfocaments garanteixen correctesa i milloren l'eficiència respecte a enfocaments naïfs, tal com s'ha corroborat tant en l'anàlisi teòrica de la secció 3 com en la implementació pràctica amb i sense paral·lelisme. Les generalitzacions a N dimensions mantenen els principis fonamentals, tot i que la complexitat pot créixer i alguns mètodes elegants (com rotating calipers) no sempre tenen un anàleg directe. No obstant, els algorismes descrits constitueixen fonaments robustos per a resoldre problemes geomètrics en entorns de dues i tres dimensions amb eficàcia acadèmica i computacional.

VIII. Conclusió

Aquest treball ha resultat en una arquitectura Model-Vista-Controlador (MVC) per al càlcul i visualització de distàncies en núvols de punts 2D i 3D, implementant i comparant diversos algorismes geomètrics fonamentals: el parell de punts més proper, el diàmetre d'un conjunt de punts i l'envolupant convexa. S'han desenvolupat solucions que van des de l'enfocament de força bruta fins a versions optimitzades (algorismes de divideix i venceràs, estructures KD-Tree, algorisme QuickHull). A més, s'han implementat versions paral·lelitzades mitjançant el model Fork/Join i mecanismes asíncrons de notificació a la interfície d'usuari, garantint la separació de la lògica de càlcul i la visualització en temps real. Els resultats analítics i tècnics obtinguts confirmen una millora substancial d'eficiència amb els mètodes avançats. En particular, per al problema del parell més proper, els algorismes de divideix i venceràs i l'ús de KD-Tree aconsegueixen una complexitat aproximada d'O(n log n) tant en 2D com en 3D, permetent processar conjunts de punts molt més grans que amb la solució naïf de cost O(n²). De forma similar, l'algorisme QuickHull calcula l'envolupant convexa de manera eficient (complexitat O(n log n) en casos típics), la qual cosa permet obtenir el diàmetre del conjunt amb un cost molt inferior al de força bruta. Tot i que en el pitjor dels casos QuickHull i el càlcul de diàmetre en 3D es poden degradar a O(n²), en situacions pràctiques aquestes solucions mantenen un rendiment excel·lent i escalen molt millor amb la mida de les dades que les seves contrapartides naïf. A més, la incorporació de tècniques de concurrència en la implementació ha demostrat ser clau per maximitzar el rendiment i la usabilitat del sistema. Mitjançant Fork/Join s'ha paral·lelitzat de forma natural el càlcul dels algorismes, obtenint acceleracions significatives en màquines multinucli. L'estratègia de work stealing ha garantit una bona escalabilitat sense haver de gestionar manualment el repartiment de tasques, i l'ús de llindars ha evitat sobrecostos innecessaris en la creació de fils. Paral·lelament, l'execució asíncrona de les tasques de càlcul ha permès mantenir l'interactivitat de la interfície d'usuari independentment de la càrrega computacional. Alhora, el sistema de notificacions reactiu ha assegurat una coordinació eficient i desacoblada entre el motor de càlcul i la visualització. En conjunt, aquest disseny concurrent multinivell (combinant divisió recursiva de tasques, paral·lelisme amb Fork/Join i actualització reactiva) ha permès aprofitar al màxim els recursos de maquinari moderns alhora que ha proporcionat una experiència d'usuari fluida. En resum, les solucions proposades creïm que ofereixen una eficiència notable i una escalabilitat elevada tant pel que fa al nombre de punts processats com a la capacitat d'explotar els recursos de computació disponibles. Els algorismes optimitzats superen clarament les implementacions de força bruta en tots els escenaris analitzats, cosa que valida l'enfocament emprat per abordar aquests problemes geomètrics de manera òptima.

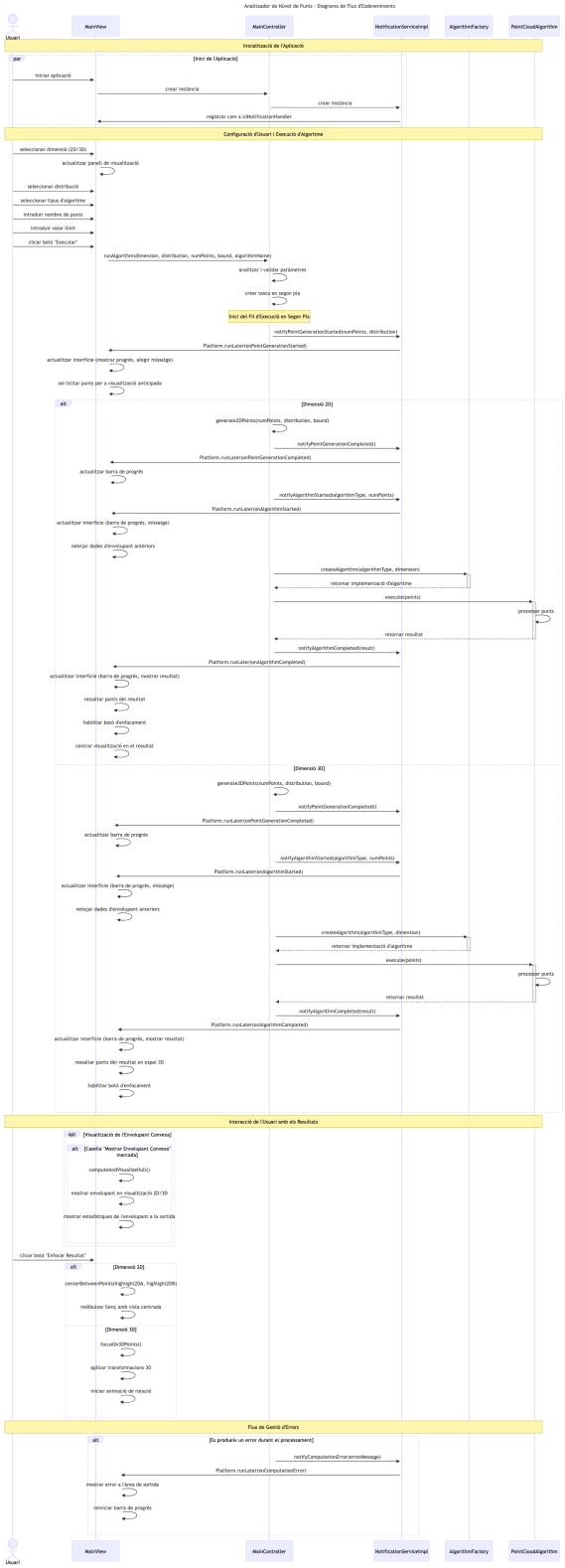
Referències

[1] Oracle, *Fork/Join Framework*, Java Tutorials, 2014.
[2] S. Klymenko, *Java ForkJoinPool: A Comprehensive Guide*, Medium, 2021.
[3] A. Dix et al., *Model-View-Controller (MVC) and Observer*, Univ. of North Carolina, 2016.
[4] Sommerville, I., *Software Engineering*, 10th ed., Addison-Wesley, 2015.
[5] Chacon, S. and Straub, B., *Pro Git*, Apress, 2014.
[6] Cockburn, A., *Agile Software Development: The Cooperative Game*, Addison-Wesley, 2006.
[7] Loeliger, J. and McCullough, M., *Version Control with Git*, O'Reilly Media, 2012.
[8] Sutherland, J., *Scrum: The Art of Doing Twice the Work in Half the Time*, Crown Business, 2014.
[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[10] Reenskaug, T., *Models, Views and Controllers*, 1979.
[11] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.
[12] Dix, A., Finlay, J., Abowd, G. and Beale, R., *Human-Computer Interaction*, Prentice Hall, 2004.
[13] Fowler, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
[14] Joshua Bloch, *Creating and Destroying Java Objects*, Drdobbs, 2008
[15] Goetz, B. et al., *Java Concurrency in Practice*, Addison-Wesley, 2006.
[16] Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 2000.
[17] Amdahl, G. M., *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS Conference Proceedings, 1967.
[18] Herlihy, M. and Shavit, N., *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.
[19] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
[20] Bentley, J. L. and Shamos, M. I., *Divide-and-Conquer in Multidimensional Space*, Proceedings of the Eighth Annual ACM Symposium on Theory of Computing, 1976.
[21] Chan, T. M., Rahul, S., and Xue, J., *Range Closest-Pair Search in Higher Dimensions*, Computational Geometry: Theory and Applications, 91, 101669, 2020. https://doi.org/10.1016/j.comgeo.2020.101669
[22] Xue, J., *Colored Range Closest-Pair Problem Under General Distance Functions*, Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2019. https://arxiv.org/abs/1807.09977
[23] Gojcic, Z., Zhou, C., Wegner, J. D., and Wieser, A., *The Perfect Match: 3D Point Cloud Matching with Smoothed Densities*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2019. https://arxiv.org/abs/1811.06879
[24] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
[25] Kleinberg, J. and Tardos, É., *Algorithm Design*, Pearson, 2006.

[26] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
[27] Sipser, M., *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, 2012.
[28] Tarjan, R. E., *Data Structures and Network Algorithms*, SIAM, 1983.

Annex:

A. Diagrama de fluxe dins de la aplicació



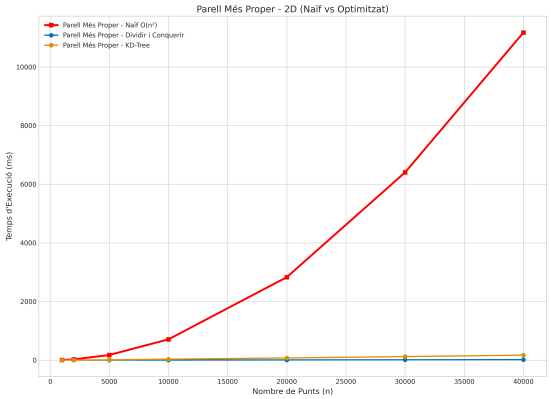


Figura 3. **Naiv vs Optim**

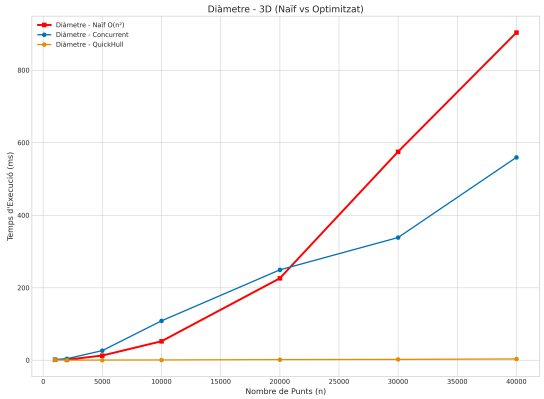


Figura 4. **Parell Mes Proper**

B. Comparativa de rendiment dels distints algoritmes segons N i la distribució de punts.

1) Parell més proper 2D

En aquest gràfic es representa el temps d'execució (en mil·lisegons) en funció del nombre de punts n per als algorismes de Parell més proper en 2D. S'hi comparen l'enfocament naïf (força bruta) amb dos mètodes optimitzats: l'algorisme de divideix i venceràs i la versió basada en estructura KD-Tree. Cada corba mostra com creix el temps a mesura que augmenta n (fins a desenes de milers de punts). S'observa una divergència pronunciada: la corba del mètode naïf puja molt més ràpidament, mentre que les dues corbes optimitzades creixen molt més lentament, gairebé de forma sublineal per al rang de valors considerat. Això indica clarament diferències de complexitat: l'algorisme de força bruta té cost $O(n^2)$ (perquè compara totes les parelles de punts), mentre que tant el divideix i venceràs com la cerca amb KD-Tree assoleixen costos propers a $O(n \log n)$. En conseqüència, per a quantitats grans de punts, el temps del mètode naïf esdevé intractable, creixent quadràticament, mentre que els mètodes avançats escalen molt millor. Analitzant quantitativament el gràfic, es constata que per a mostres petites (p. ex. $n < 10^3$) les diferències no són extremes – tots els algorismes responen en temps molt baix. Tanmateix, a mesura que n creix, la força bruta aviat supera els segons d'execució, mentre que els altres dos algorismes mantenen temps de l'ordre de mil·lisegons. Per exemple, per a $n = 40.000$ punts en 2D, la solució naïf pot trigar de l'ordre de 10 segons, en contrast amb les opcions optimitzades que es mantenen per sota d'1 segon. El mètode divideix i venceràs 2D, gràcies a la divisió recursiva i la combinació eficient, presenta un creixement quasi lineal $O(n \log n)$ i aprofita que en fusionar només cal examinar uns pocs veïns a la franja central. L'algorisme amb KD-Tree, per la seva banda, inclou un cost inicial de construcció de l'arbre de $O(n \log n)$ i després localitza per a cada punt el seu veí més proper en temps mitjà $O(\log n)$, aconseguint també una complexitat esperada $O(n \log n)$. Les dues corbes optimitzades són molt pròximes; de fet, ambdós enfocaments tenen rendiment similar en el pla, i la diferència principal rau en factors constants. En qualsevol cas, el gràfic confirma que tots dos mètodes avançats superen amb escreix la solució naïf per a dades grans, permetent tractar núvols de punts molt més nombrosos. Aquesta millora és crucial en el context del nostre projecte MVC, ja que possibilita respondre de forma interactiva fins i tot amb desenes de milers de punts, cosa inviable amb l'enfocament quadràtic.

2) Parell més proper 3D

El segon gràfic mostra el rendiment dels mateixos algorismes de Parell més proper, però ara en espais tridimensionals (3D). Es representen igualment les corbes de temps per al mètode naïf 3D i les solucions optimitzades (divideix i venceràs i KD-Tree) en funció del nombre de punts. La tendència general s'assembla al cas 2D: la solució de força bruta escala molt pitjor que les altres. En 3D, l'algorisme naïf continua requerint $O(n^2)$ comparacions i, a més, cada càlcul de distància és lleugerament més costós (implica una dimensió addicional en la suma de quadrats). Així, la corba naïf 3D creix ràpidament i adquireix valors encara més grans que en 2D per a un mateix n (per exemple, per $n = 20.000$ punts ja supera els segons de temps). En canvi, les corbes del divideix i venceràs i del KD-Tree en 3D mostren un increment suau, mantenint-se en la regió de mil·lisegons fins i tot per a desenes de milers de punts. Això indica que aquests mètodes conserven la complexitat quasi òptima: en el cas 3D, el divideix i venceràs té complexitat esperada $O(n \log n)$ (encara que teòricament pot arribar a $O(n \log^2 n)$ en no aplicar certes optimitzacions). En la implementació del projecte, s'han incorporat refinaments (com limitar els candidats a la franja mitjançant projeccions) per acostar el cost pràctic a $O(n \log n)$ també en 3D. De forma similar, l'estratègia de KD-Tree es manté eficient en 3D: l'alçada de l'arbre continua sent $O(\log n)$ per a distribucions no patològiques i les cerques descarten la meitat de l'espai a

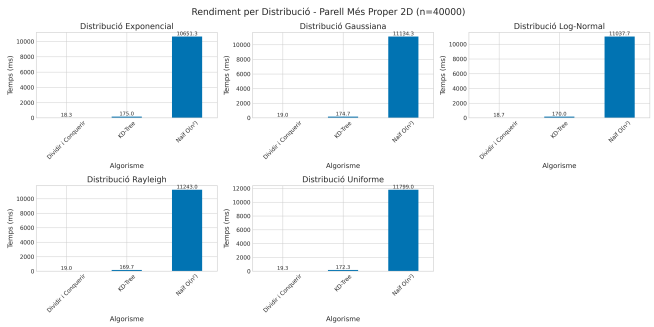


Figura 5. **Parell mes proper - Distribucio**

cada pas, aconseguint també un rendiment esperat $O(n \log n)$ en 3D. La principal diferència és en la constant de temps, lleugerament superior en 3D (cada comparació gestiona coordenades (x, y, z) en lloc de (x, y)), fet que es tradueix en corbes 3D una mica per damunt de les respectives 2D. Examinant les dades, es constata que per a n moderats (p. ex. $n = 5000$), el mètode naïf 3D pot tardar diversos segons, mentre que els optimitzats triguen només desenes de mil·lisegons – una diferència de dos ordres de magnitud. A $n = 20.000$, la tendència s'accentua encara més. Tot i el creixement logarítmic addicional possible en el divideix i venceràs 3D, la implementació pràctica mostra que aquest algorisme segueix escalant quasi linealment gràcies a la limitació de veïns a comprovar (basada en la propietat geomètrica que en 3D un punt té un nombre constant de veïns pròxims en la franja). El KD-Tree manté un comportament semblant: no es detecta cap degradació important causada per la dimensionalitat (confirmant que en dimensions baixes no apareixen els problemes de la "maleïda dimensionalitat").

3) Parell més proper - Distribució

Aquest gràfic presenta una comparativa del temps d'execució dels algorismes de Parell més proper 2D per a diferents distribucions estadístiques de les dades. Concretament, es mostren resultats per a cinc distribucions de punts: Uniforme, Gaussiana, Exponencial, Rayleigh i Log-normal. Cada subgràfic correspon a una distribució i inclou barres que representen el temps total per calcular el parell més pròxim amb cada algorisme (naïf, divideix i venceràs i KD-Tree) en un escenari fix de $n = 40.000$ punts. D'aquesta manera, es pot analitzar l'efecte de la distribució dels punts sobre el rendiment dels mètodes. Independentment de la distribució, el mètode naïf exhibeix els temps més elevats amb gran diferència. Per exemple, en totes les distribucions el càlcul de força bruta triga de l'ordre de 10^4 ms (és a dir, diversos segons) per 40.000 punts, mentre que tant el divideix i venceràs com el KD-Tree completen la tasca en uns pocs centenars de mil·lisegons. Això reflecteix novament la bretxa de complexitat: la força bruta fa $\sim 8 \cdot 10^8$ comparacions de distàncies (proporcional a n^2), mentre que els altres algorismes redueixen dràsticament el nombre de càlculs a un orde quasi lineal. Ara bé, el gràfic permet veure petites variacions entre distribucions. En el cas Rayleigh i Uniforme, el temps del mètode naïf és lleugerament superior (arribant a uns 12 segons en Rayleigh 2D) comparat amb les altres distribucions (on ronda 8–10 segons). Aquesta diferència s'explica per la naturalesa geomètrica de les distribucions: en una distribució uniforme o de Rayleigh, els punts tendeixen a estar més repartits per l'espai, de manera que la distància mínima entre dos punts (parella més propera) sol ser més gran. Això implica que l'algorisme naïf, tot i haver de recórrer igualment totes les parelles, no troba una distància mínima extremadament petita fins a haver examinat moltes comparacions. Si la implementació aprofita alguna optimització menor (per exemple, comparar distàncies al quadrat i interrompre càlculs innecessaris quan ja excedeixen la millor distància trobada), aquest cas de punts més espaiats pot resultar lleugerament més costós perquè menys comparacions es poden descartar prematurament. Per contra, en distribucions Gaussiana i Exponencial, molts punts es concentren en una regió central petita, fent que hi hagi parelles molt pròximes. Així, la primera vegada que el mètode naïf troba dues dades gairebé coincidents, fixa un valor δ mínim molt baix; a partir d'aquí, en calcular distàncies d'altres parelles sovint es detecta ràpidament que superaran δ i es poden obviar càlculs addicionals. Aquest efecte redueix lleugerament el temps efectiu en les distribucions amb forta concentració central. En resum, la distribució dels punts influeix només marginalment en el rendiment final de la força bruta: totes continuen requerint un temps prohibitiu a 40.000 punts, però les distribucions amb major dispersió (Uniforme, Rayleigh) resulten un pèl més desfavorables perquè no ofereixen oportunitats de curtcircuit en el càlcul de distàncies.

Pel que fa als algorismes optimitzats, el gràfic revela que divideix i venceràs i KD-Tree mantenen un rendiment excel·lent en totes les distribucions considerades. Les seves barres són molt baixes (entorn de 200–500 ms) i presenten variacions relatives petites entre distribucions. En general, la distribució Uniforme tendeix a requerir un temps lleugerament més alt per als mètodes avançats, mentre que la Gaussiana i la Exponencial solen donar els temps més baixos. Novament, això té una explicació geomètrica: l'algorisme de divideix i venceràs depèn del nombre de punts que queden en la franja central a cada pas de combinació. En una distribució uniforme, els punts estan repartits arreu i és més probable que hi hagi diversos punts prop de la línia divisòria en cada partició, de manera que la fase de fusió ha de comparar més parelles potencials (encara que el nombre màxim de veïns a comprovar està acotat per una constant geomètrica). Per contra, en una distribució com la Gaussiana, la major part de punts queden concentrats al voltant del centre; en dividir l'espai, la gran densitat central fa que δ (la mínima distància trobada en subproblemes) sigui molt petita, i només uns pocs punts perifèrics cauen a la franja central, reduint el treball de combinació. En el cas del KD-Tree, el rendiment també és robust: per a aquestes

distribucions no patològiques, l'arbre resta equilibrat i cada punt només té uns pocs candidats de distància propera. Una distribució uniforme o Rayleigh pot fer que alguns punts no tinguin cap veí molt immediat, de manera que la cerca nearest neighbor explora algunes branques addicionals de l'arbre abans de descartar-les; això explica un petit increment de temps respecte a distribucions on cada punt té un veí molt pròxim (com en el cas Gaussià o exponencial). Amb tot, aquestes diferències són de detall: tant amb KD-Tree com amb divideix i venceràs, el cost segueix sent de l'ordre de $n \log n$ en totes les distribucions, i els temps absoluts es mantenen en una escala de mil·lisegons. En definitiva, el gràfic corrobora que els algorismes optimitzats no es veuen afectats de manera significativa per la distribució estadística dels punts (sempre que aquesta no sigui patològica, com una disposició alineada que degradaria el KD-Tree a $O(n^2)$ en el pitjor cas teòric).