Pràctica 3 - CS:LS

Entrega Final

Adrià Pajares **adrian.pajares**

Lídia Figueras **lidia.figueras**

Martí Ejarque **marti.ejarque**

Victor Xirau **victor.xirau**

Enginyeria Informàtica

La Salle, URL

Índex

1 Explicació detallada dels algoritmes

1.1 Fase 1

L'algorisme a implementar en aquesta fase es l'algorisme de Dijkstra. Aquest algorisme té com a finalitat trobar el camí menys car entre el node inicial i el node final, en cas que aquest existeixi. En el nostre cas, el concepte de çar"ve condicionat per la probabilitat de trobar-se a un enemic en funció del camí que escollim. Al jugador li interessarà anar per aquells camins que tinguin una possibilitat més baixa de trobar-se a un enemic. Dit això, sorgeix una complicació. No és el mateix anar de A a B i que tingui un cost de probabilitat de 100, que anar de A a B passant per C i que també tingui un cost de 100, ja que de A a C el cost és de 60 i de C a B de 40. El cost total és el mateix, però la probabilitat de trobar-se amb algú al primer cas és del 100%, en canvi en el segon cas és del 50%. Per tal d'aconseguir que l'algorisme funcionés adequadament, hem seguit el següent procés:

Primerament, hem creat un array de Probabilitats (classe Probability, la qual conté la probabilitat de trobar-se a un enemic i si és final o no). En aquest array, hem col·locat la probabilitat de l'habitació inicial a 0, mentres que tota la resta de probabilitats han estat posades a un nombre molt gran (infinit).

A continuació, el que farem és iniciar el recorregut des de l'habitació inicial. Observarem les habitacions adjacents a l'actual i encara no visitades pel nostre algorisme, i guardarem la probabilitat de trobar-se un enemic caminant en aquella direcció. A mesura que anem avançant, anirem guardant i acumulant les probabilitats com si de pesos es tractés, de tal forma que al arribar al final, per a cada node tindrem una probabilitat acumulada segons el camí que hagueem escollit. Un cop arribem a l'habitació final, i havent visitat i contemplat tots els valors de les probabilitats de les habitacions anteriors, tindrem l'array de probabilitats plena.

Finalment, el que farem és buscar un camí des de l'habitació final fins a la inicial, ajudant-nos a escollir el camí correcte mitjançant les probabilitats. El que farem és restar la probabilitat total de l'habitació total amb la probabilitat guardada a l'habitació corresponent del graf que estiguem mirant, i si el resultat d'aquesta resta coincideix amb la probabilitat guardada a l'array de probabilitats, voldrà dir que l'habitació de la qual hem restat la probabilitat es troba en el camí correcte(el més curt). D'aquesta manera, anem restant les probabilitats de l'habitació a la qual ens trobem amb les probabilitats de les habitacions adjaçents, i la habitació de la qual la resta sigui igual al valor d'aquella habitació en l'array de probabilitats serà l'habitació a escollir per aconseguir el camí més curt.

1.2 Fase 2

1.2.1 Arbres BST

Al fer la cerca d'un objecte pel preu a la botiga hem implementat l'arbre BST. De manera que introduint un preu o un nom, aquest farà la cerca de l'objecte corresponent. Per tal de crear l'arbre, s'ha fet ús d'una funció anomenada "inserir", la qual funciona de la següent manera: Per saber a quina posició s'ha d'inserir el node, aquest compara el seu preu amb el de l'arrel de l'arbre. Depenent de si aquest preu és més petit o més gran, es col·locarà a l'esquerra o la dreta d'aquest. Si l'arrel no presenta cap fill dret ni esquerre, aquest node és directament inserit. Però, en el cas de que ja presenti algun fill, es torna a cridar la funció "inserir" recursivament, per mirar a quina posició del fill de l'arrel s'ha de col·locar. Per tal d'eliminar un node de l'arbre, s'ha fet ús d'una funció anomenada "deleteKey", la qual funciona de la següent manera: Aquesta funció rep el node a eliminar, i aquest és passat per paràmetre a una funció anomenada deleteRec, juntament amb l'arrel. Quan s'ha eliminat un node, es resta una unitat de la nostre variable "totalNodes". La funció deleteRec s'encarregarà d'esborrar el node de l'arbre recursivament. Primer rebrà l'arrel i el valor del preu del node que volem esborrar. Es compararà aquest valor amb el de l'arrel i es tornarà a cridar de forma recursiva en funció de si el valor del preu és més gran o més petit que el de l'arrel. D'aquesta manera, s'anirà cridant de forma recursiva a la funció, fins que el node trobat tingui un valor de preu igual al valor que li hem passat. En aquest cas, voldrà dir que hem trobat el node que volem esborrar, i aquest serà esborrat, tot tornant l'arrel actual. Finalment, si el node a esborrar tenia fills, s'haurà de seleccionar un per tal de substituir al pare; fent ús de la funció minValue, la qual va buscant a través dels fills esquerres del fill dret del node esborrat, fins trobar un valor adequat. Per últim, s'ha implementat la funció "inordre", el qual en aquesta s'ha creat una variable anomenada result, la qual guarda l'ordre dels nodes en els que s'ha visitat. Primer recorrent el subarbre esquerre, i seguidament el subarbre dret.

1.2.2 Arbres R

El plantejament per realitzar aquest arbre ha seguit el següent procés: Primerament s'han descarregat tots els nodes del Json, per tal de tenir-los en un array. Un cop s'ha creat aquest array del tipus ObjectesMapa, procedim a la creació de l'arbre amb aquests objectes.

Abans de veure el procediment de creació, cla mencionar que l'arbre estarà guardat en un array del tipus RectanglePare, on aquesta classe té una variable que representa el teu pare, és a dir, l'array de rectangles que t'inclou, un array de nodes, en el cas que tinguis nodes com a fills, un array de pares, en el cas que tinguis pares com a fills, i coordenades, per saber com és de gran aquest rectangle en qüestió.

Un cop vist aquesta clase, procedim a la creació de l'arbre. A continuació, assignarem a cada node un pare, el qual serà ell mateix. Després de fer que tots els nodes (els objectes) tinguin un pare (ells mateixos), començarà el procediment d'unió entre objectes. Buscarem per cada pare de cada node un altre pare (el que sigui més proper) i els unirem, per tal de formar grups de dos pares. En cas de que la quantitat d'objectes que tinguem sigui parell, tots els objectes (els nodes) agrupats en grups de dos, i procedirem a formar grups més grans.

En el cas que els nodes siguin imparells, tindrem tot parelles menys un node. Doncs el que farem serà buscar de quins rectangles que hem creat (que tenen dos nodes, per tant és un rectangle nou més gran) és el més pròxim a aquest node solitari. Un cop s'ha trobat aquesta parella més propera, li afegim aquest node i creem un array de tres nodes. Això ens genera un seguit de parelles (del tamany de l'array d'objectes tende menys un) i un trio.

Un cop tenim els pares agrupats per parelles, i un trio si es dóna el cas, comencem a ajuntar aquests grups, fins que ens queden com a màxim 3 agrupacions. Aquest procés es fa igual que amb nodes, amb l'única diferència que ara estarem buscant pares més propers entre si. Ens trobarem amb la mateixa casuística, i és que si tenim pares (agrupacions de nodes) imparells, s'aniran agrupant per parelles i quedarà un pare sol. Aquest cas es soluciona com hem vist anteriorment, és a dir, buscant l'agrupació de pares més propera al pare solitari.

Aquest procés s'anirà repetint fins que les agrupacions d'agrupacions de pares quedin com a molt de mida 3 i com a mínim de mida 2. Un cop s'ha completat aquest procés, se li diu a aquesta agrupació resultant que té com a pare una variable anomenada root, que és el RectanglePare que ho engloba tot. Aquest rectangle pare serà el que utilitzarem per explorar tot l'arbre en el moment que vulguem trobar els punts.

Un cop esta l'arbre muntat, per tal de trobar si en el punt que ens passa l'usuari es troba un objecte, anem recorrent agrupació a agrupació, fins trobar un objecte que contingui el punt

trobat. Si l'ha trobat, fem el hit i s'elimina, sinó fem miss i tornem a demanar un punt Un cop l'arbre ha estat creat, haurem de comprovar si el punt introduït per l'usuari ha fet un HIT o no. El que farem per tal de detectar això és recórrer totes les estructures de rectangles, fins arribar al rectangle que conté el punt introduït. Per tal d'eliminar l'objecte trobat, el que es farà és eliminar l'objecte del conjunt d'objectes que tenim inicialment i es tornarà a crear l'arbre per tal de que els nodes estiguin agrupats de forma correcta. Si el HIT trobés més d'un objecte solapat, s'eliminarà el primer objecte que es trobi al recórrer l'array d'objectes.

1.3 Fase 3

En aquesta fase se'ns ha demanat buscar els nostres amics pel seu nom. Així doncs se'ns ha proposat implementar Hash Map, la manera més ràpida (aproximadament un cost de O(1)) de buscar un element seguint claus úniques.

El Hash Map implementat ha estat seguint el criteri d'open hashing, en el que per cada jugador nou que es volia guardar en l'array R (on cada jugador te una casella assignada, que pot estar ocupada o no) el fem pasar per la nostre funció de Hash. En concret, aquetsa funció de Hash funciona amb la següent lògica:

Agafaem el número d'aquest jugador, ja que en le fitxer JSON, els noms dels jugadors estan designats de la següent manera: Player<num>, sent num el número del jugador en qüestió. Llavors, un cop agafat el número, sumem cada un dels seus dígits i el multipliquem per 7. La multiplicació per 7 es fa ja que, al ser un número primer, aquest genera númeors tals que permet una millor distribució en els grups de hash. Dit d'una altra manera, com menys divisors té un número, menys probabilitats de que es porodueixin colisions de hash, i per tant tindrem una millor distribució dels elements que es guarden en l'array R. Tot i que implementem open hashing, no ens interessa que hi hagi moltes col·lisions, ja que això provocaria la creació de llistes llargues d'objectes en una mateixa posició de l'array R, cosa que no ens interessa. Un cop obtingut el codi de la funció de hash, es fa la divisió modular entre el número de nodes que ja hem inserit, per així generar números relativament petits pel que fa les posicions d'un array.

Un cop tenim tots els elements guardats, ja podem buscar aquell jugador en qüestió. Per obtenir el jugador, obtenim una altra vegada el codi que jenera el número d'aquest jugador, i anem passant en la llista que hi ha en aquesta posició del array R (en cas que hi hagués més d'un jugador assignat a aquetsa cassella) i mirem quin té la mateixa clau que la que ens ha introduit l'usuari. Així, aconseguim obtenir una cerca per clau única prou efectiva, o si més no, més efectiva que la búsqueda d'aquest jugador en un array qualsevol.

2 Mètode de proves utilitzat

2.1 Fase 1

Per tal de comprovar que els resultats són aparentment correctes, s'ha printat el recorregut total desde el node inicial fins al final. Al llarg de la creació de l'algorisme, també hem mostrat tots els successors pels que decidia anar el programa, per tal de comprovar si decidia correctament el camí a seguir. Per comprovar que aquests eren els desitjats, hem creat un dataset de prova, i hem fet un dibuix per tal d'observar el recorregut de forma manual, i així poder comprovar que realitzava el recorregut de forma correcta, i extrapolar els resultats als altres datasets.

2.2 Fase 2

Per tal de poder trobar els errors que anaven apareixent en el codi, hem emprat el debugger que ens ofereix IntelliJ IDEA. Ja que hem pogut anar seguint el codi linia a linia, tot mirant com canvien els valors de les nostres variables. Per exemple, ha estat molt útil en el cas de trobar si els nodes s'havien inserit de forma correcta a l'arbre.
També s'han utilitzat diferents datasets, sobretot per l'arbre R, ja que volíem comprovar el correcte funcionament del codi si el nombre d'objectes era tant parell com imparell. Això ens ha ajudat a detectar varies errades a l'hora de considerar un número imparell de nodes.

2.3 Fase 3

Per comprovar el correcte funcionament de la funció de Hash, hem utilitzat varis mètodes de proves. Primerament, hem comprovat el temps que tardava en realitzar una cerca, envers el temps que tardava buscant en un array "normal". Aquest fet ens ha fet veure que la utilització de Hash Tables és més ràpida que no pas la cerca en un array normal. Després, hem comparat el temps que la nostre funció de hash tardava en generar la clau i tribar el jugador amb la pròpia generació d'una clau de Hash que té Java. La funció de Java va tardar un segon menys en realitzar la cerca, per tant vam concloure en que la nostre funció de Hash era prou efectiva.

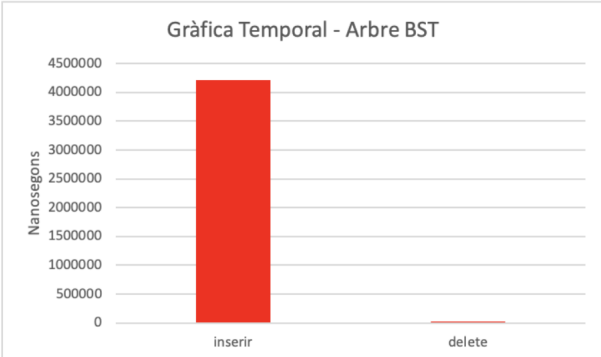
3 Comportament dels algoritmes

3.1 Fase 1

(La implementació del nostre algorisme de Dijkstra no ens ha permès obtenir els resultats dels altres datasets)

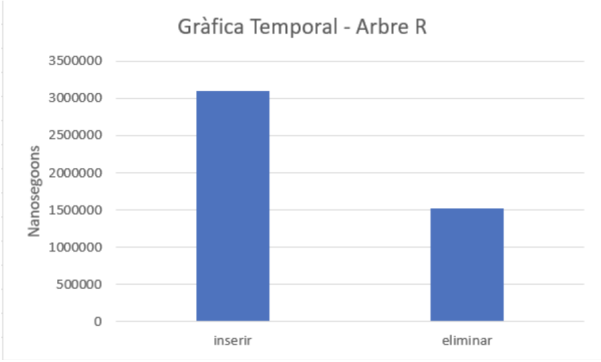
3.2 Fase 2

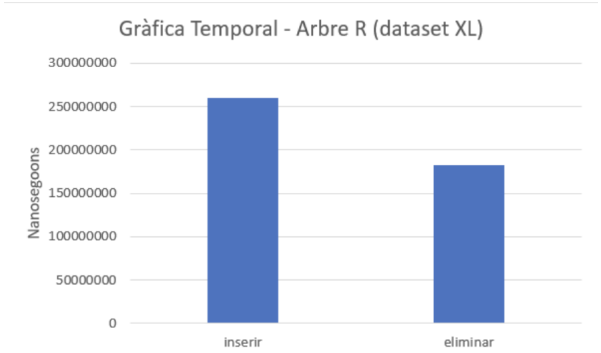
3.2.1 Arbre BST



Com podem comprovar, el temps d'execució de la funció inserir és molt més alt que el del temps de la funció delete. Com és lògic, ja que per tal d'inserir s'ha de recórrer tot l'arbre per tal d'estudiar a on s'ha d'inserir. En canvi, per tal d'eliminar només s'ha de fer una cerca fins a trobar el node que volem eliminar, però no es fa una cerca total, és a dir, no es recorre tot l'arbre.

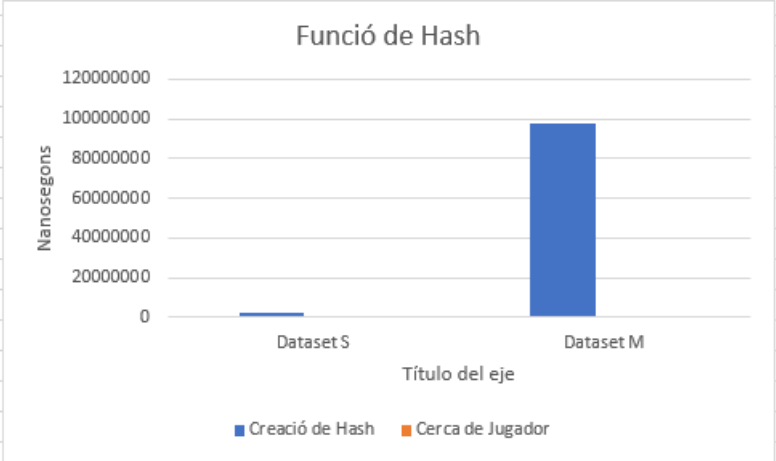
3.2.2 Arbre R





En aquest cas, observem que el temps d'inserció és aproximadament del doble que del d'eliminació. Aquest temps variarà en funció del que es trigui en trobar el node, i quant més temps es trigui a trobar el node a eliminar, més trigarà la funció d'eliminació. A les captures observem que això es compleix tant el dataset proporcionat a la pràctica com el dataset creat per nosaltres, el qual contenia 457 nodes.

3.3 Fase 3



Com podem observar, tot i que la creació de la funció de hash triga una mica més pel dataset M, estem tractant amb valors de temps molt baixos(treballem amb nanosegons). Cal destacar que el temps de cerca dels jugadors quasi no varia entre el dataset S i el M, el que reforça la nostra idea de que la funció de hash està implementada de forma òptima.

4 Anàlisi dels resultats

4.1 Fase 1

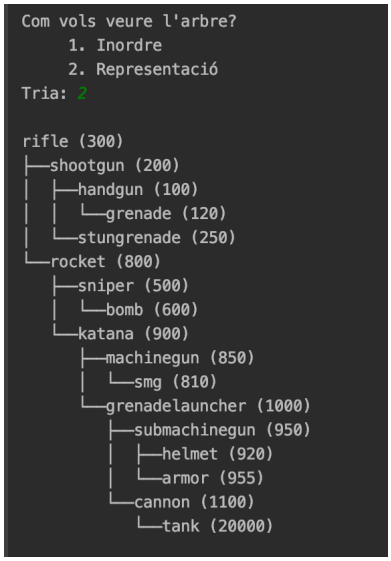
```
Introdueix l'habitació inicial:
1
Introdueix l'habitació final:
127
La millor ruta des de l'habitació 1 fins l'habitació 127 és:
1) - Habitació número 1
2) - Habitació número 2
3) - Habitació número 3
4) - Habitació número 11
5) - Habitació número 20
6) - Habitació número 28
7) - Habitació número 37
8) - Habitació número 38
9) - Habitació número 46
10) - Habitació número 55
11) - Habitació número 71
12) - Habitació número 87
13) - Habitació número 103
14) - Habitació número 111
15) - Habitació número 127
L'algorisme ha trigat 14 mil·lisegons.

Process finished with exit code 0
```

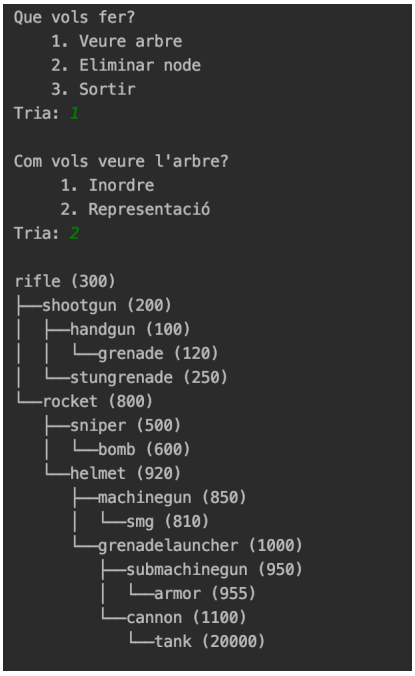
El resultat a observar es tracta del camí amb menys probabilitats de trobar-se a un enemic de l'habitació 1 a la 127. Observant aquest resultat, podem deduïr que el nostre algorisme és correcte, ja que el simple fet de que l'habitació 1 i la 127 siguin les que més allunyades entre si; l'algorisme és capaç d'encaminar-les en tant sols 15 habitacions, incluides ambdós.

4.2 Fase 2

Com podem comprovar a les següents captures, tant el procés d'eliminació d'un objecte mitjançant l'arbre BST amb un arbre R funcionen de manera correcta.

4.2.1 Eliminació mitjançant arbres BST



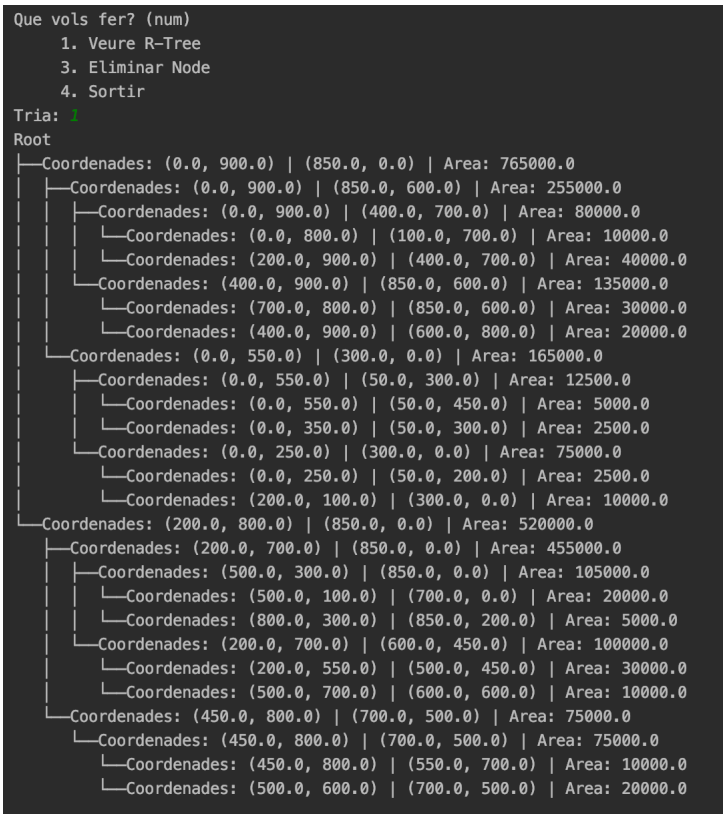


Com podem comprovar, en la primera imatge observem tot l'arbre desde l'arrel. Per tal de crear aquest arbre s'ha dut a terme un procés d'inserció de nodes, tal i com hem explicat anteriorment. Després de buscar l'objecte per nom "katana", podem comprovar com en la imatge 2, aquest objecte ha estat eliminat correctament. Pel que podem deduir, que la funció inserir i delete funcionen de forma correcta.

4.2.2 Crear un objecte al mapa (Arbre R)

Com es pot observar, al introduir un punt al nostre programa, es recorrerà el nostre arbre R per tal de comprovar si hem fet HIT o MISS. En el primer exemple, podem observar que el punt introduït ha fet HIT i, per tant, ha trobat un objecte a eliminar del nostre arbre. Si observem l'arbre abans i després, podrem comprovar que efectivament s'ha eliminat l'objecte. El segon exemple mostra com el punt que hem introduït no ha tocat cap objecte, i per tant la llista d'elements mostrada no presenta cap modificació.

4.2.3 Abans d'eliminació



4.2.4 Després d'eliminació

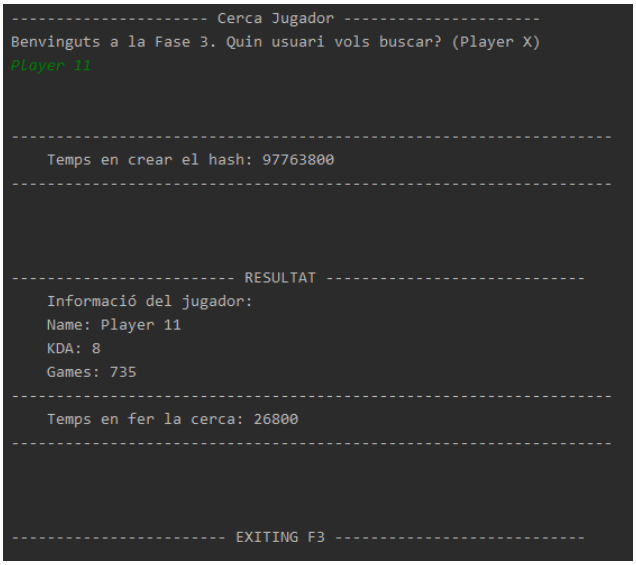
```
En quina posició es troba el teu objecte? (x1,y1)
750,700

HIT!

TEMPS ELIMINACIÓ: 1009330
S'ha eliminat l'objecte: 9

Que vols fer? (num)
     1. Veure R-Tree
     3. Eliminar Node
     4. Sortir
Tria: 1
Root
├──Coordenades: (0.0, 900.0) | (700.0, 300.0) | Area: 420000.0
│  ├──Coordenades: (0.0, 900.0) | (700.0, 500.0) | Area: 280000.0
│  │  └──Coordenades: (0.0, 800.0) | (100.0, 700.0) | Area: 10000.0
│  │  └──Coordenades: (200.0, 900.0) | (400.0, 700.0) | Area: 40000.0
│  │  └──Coordenades: (500.0, 600.0) | (700.0, 500.0) | Area: 20000.0
│  └──Coordenades: (0.0, 550.0) | (50.0, 300.0) | Area: 12500.0
│     └──Coordenades: (0.0, 550.0) | (50.0, 450.0) | Area: 5000.0
│     └──Coordenades: (0.0, 350.0) | (50.0, 300.0) | Area: 2500.0
├──Coordenades: (0.0, 300.0) | (850.0, 0.0) | Area: 255000.0
│  ├──Coordenades: (0.0, 250.0) | (300.0, 0.0) | Area: 75000.0
│  │  └──Coordenades: (0.0, 250.0) | (50.0, 200.0) | Area: 2500.0
│  │  └──Coordenades: (200.0, 100.0) | (300.0, 0.0) | Area: 10000.0
│  └──Coordenades: (500.0, 300.0) | (850.0, 0.0) | Area: 105000.0
│     └──Coordenades: (500.0, 100.0) | (700.0, 0.0) | Area: 20000.0
│     └──Coordenades: (800.0, 300.0) | (850.0, 200.0) | Area: 5000.0
└──Coordenades: (200.0, 900.0) | (600.0, 450.0) | Area: 180000.0
   ├──Coordenades: (200.0, 700.0) | (600.0, 450.0) | Area: 100000.0
   │  └──Coordenades: (200.0, 550.0) | (500.0, 450.0) | Area: 30000.0
   │  └──Coordenades: (500.0, 700.0) | (600.0, 600.0) | Area: 10000.0
   └──Coordenades: (400.0, 900.0) | (600.0, 700.0) | Area: 40000.0
      └──Coordenades: (400.0, 900.0) | (600.0, 800.0) | Area: 20000.0
      └──Coordenades: (450.0, 800.0) | (550.0, 700.0) | Area: 10000.0
```

4.3 Fase 3

4.3.1 Dataset S

```
---------------------- Cerca Jugador ----------------------
Benvinguts a la Fase 3. Quin usuari vols buscar? (Player X)
Player 5


------------------------------------------------------------------------
    Temps en crear el hash: 2142300
------------------------------------------------------------------------



------------------------- RESULTAT -----------------------------
    Informació del jugador:
    Name: Player 5
    KDA: 5
    Games: 175
------------------------------------------------------------------------
    Temps en fer la cerca: 21900
------------------------------------------------------------------------
```

4.3.2 Dataset M



5 Problemes observats

5.1 Fase 1

Primerament, la creació del graf ens va resultar mitjanament fàcil, però el problema va venir quan ens vam donar compte que dues habitacions podien tenir més d'un camí. Problema que vam solucionar agafant la menor probabilitat de trobar-se a un enemic.
Un altre problema va ser el fet d'estudiar les probabilitats dels nodes. En un principi, vam plantejar-ho de manera que comparàvem la probabilitat menor només amb les dels nodes adjacents del node actual que estudiàvem. Més tard ens vam adonar que d'aquesta manera, el nostre programa no funcionava de forma correcta, així que vam decidir recòrrer l'array de probabilitats i quedar-nos amb la més petita d'aquestes (en el cas de que encara no hagués estat seleccionada anteriorment).

5.2 Fase 2

Respecte l'arbre BST, no ens ha suposat un gran repte, si més no ha estat dificultós a l'hora d'interpretar el pseudo-codi dels apunts i pasar-ho a llenguatge Java. Un cop s'ha tingut l'arbre muntat i s'ha procedit a l'eliminació, ens vam adonar que només havíem eliminat el nom de l'objecte, i no pas el node complert. Per tant el preu dels productes es canviava entre productes. Com veurem a les conclusions, hauriem d'haver aplicat un Red Black Tree, que si hagués estat un repte més complex.
En quant a l'arbre R, el principal problema ha estat raonar la implementació d'aquest, ja que l'ús de rectangles i punts feia el codi més tediós del normal. Després de raonar la implementació, hem trobat diverses dificultats per tal de poder crear les diferents agrupacions de rectangles, ja que hem hagut de controlar moltes variables per tal de poder generalitzar el codi per poder obtenir rectangles cada cop més grans i que agrupéssin més rectangles. Trobar el rectangle adequat per crear els grups més òptims també ha estat difícil, ja que, tal i com hem comentat a l'apartat d'explicació del codi, hem hagut de treballar amb les àrees i distàncies entre els rectangles i els punts que els formaven.

5.3 Fase 3

Pel que fa els problemes que s'han tingut, hem observat com en les primeres versions del codi la fució de Hash retornava valors molt alts, que traduit a posicions d'un array, resultaria un problema. Per solucionar aquest, es va decidir fer el mòdul entre aquest número i el nombre de jugadors inserits fins el moment. Així, es va aconseguir obtenir números més adients per a la representació de posicions d'un array. Aquest número tant gran ve donat pel fet que multipliquem per 7 el resultat obtingut de la suma dels digits del número de cada jugador. Encara que ens doni un número molt gran, ens compensa ja que al multiplicar per un número primer, la distribució dels jugadors es veu afavorida, és a dir, es produiexien menys col·lisions.

6 Conclusions

6.1 Fase 1

L'objectiu d'aquesta fase ha estat, mitjançant la implementació d'un graf i l'algorisme de Dijkstra, trobar els camins més curts (en aquest cas, els camins amb menys probabilitats de trobar un enemic) des d'una habitació(node) fins a una altra.
Després d'haver realitzat la pràctica, hem observat que una implementació amb llistes d'adjacència hauria estat molt més eficient, ja que la matriu utiltzada per representar el graf tenia molts espais buits, i moltes posicions. Utilitzant una llista d'adjacències, hauríem pogut reduir els recorreguts de forma dràstica, fent així que el nostre algorisme ens permetés obtenir resultats a partir dels datasets més grans.
També hem raonat que una forma més eficaç d'implementar l'algorisme hauria estat recórrer el graf a mesura que es va executant l'algorisme de dijkstra (és a dir, que el graf es vagi creant a mesura que el dijkstra es va executant), i així evitar recórrer el graf al principi per tal de crear-lo i més tard tornar-lo a recórrer per tal d'executar el dijkstra.

6.2 Fase 2

L'objectiu d'aquesta fase ha estat, mitjançant la implementació d'un arbre BST i un arbre R, implementar algorismes de cerca d'objectes en funció d'uns criteris. Cal destacar que la realització d'aquesta pràctica ha estat satisfactòria,i els conceptes clau sobre arbres han quedat molt clars degut a la implementació detallada d'aquests algorismes.
A l'arbre BST, trobem un cost asimptòtic de O(n) com a pitjor cas, ja sigui quan s'insereix, com quan s'elimina o es cerca un node. Encara que presentin un comportament logarítmic, el pitjor cas segueix sent de cost O(n), per tant aquest arbre acostuma a ser menys eficient que un arbre AVL o Red Black Tree, degut a que no esta balancejat. Amb un AVL o RB, existeix el factor de balanceig, i per tant la consulta d'aquests dos es torna d'uns 1,44 O(log(n)) en el cas de AVL, i de 2 O(log(n)) en el cas del RB. En aquest cas, el que hauriem d'haver aplicat hauria estat un RB Tree, ja que tenim el mateix número de consultes que d'eliminacions, i per tant és més útil i eficient utilitzar un Red Black Tree. L'AVL queda descartat ja que, encara que les cerques siguin una mica menys cares, no ens recompensa ja que tenim el mateix número de consultes que d'eliminacions.
D'altra banda, l'arbre R, en relació als costs asimptòtics, trobem que per la creació dels primers rectangles que envolten els objectes el cost asimptòtic és de O(n), ja que recorrem el conjunt d'ojectes un sol cop i anem atribuint directament els rectangles a cada objecte. A

l'hora de crear els rectangles pare pels rectangles creats a cada objecte (la primera part de la lògica de creació de rectangles de la nostra implementació) i els rectangles generals, trobem un cost aproximat de $O(n^2)$, ja que trobem els dos bucles principals per crear els pares i el while que controla si l'assignació és correcte. En cas que quedi un objecte $O(n^2)$, per tal de trobar el rectangle òptim amb què s'hauria d'agrupar.

6.3 Fase 3

Aquesta fase final ha tingut com a objectiu la implementació d'una cerca mitjançant la creació d'una funció de hash. Després d'haver obtingut els resultats, hem comprovat que, tot i que la nostra funció de hash no és tan ràpida com altres (com per exemple la funció nativa de Java), segueix tenint una velocitat i rendiment molt elevats. Així, hem arribat a la conclusió de que les funcions de hash ens permeten fer cerques molt ràpides, fent així que per a cerques puntuals on coneixem una clau, siguin una molt bona opció (a diferència de recórrer un array, per exemple).

També hem deduït que per tal d'evitar i reduïr el nombre de col·lisions, el factor de carga (nº de claus/nº de posicions de l'array) de la nostra funció de hash no hauria de ser superior al 60%. En el moment en que aquest valor es superi, haurem d'augmentar el nombre d'slots, per tal d'evitar el màxim nombre de col·lisions possibles.

7 Bibliografia

Dijkstra

Implementació Dijkstra [consulta: 7 de Març 2020]

https://www.youtube.com/watch?v=4I7W5WUQQQI

Arbre BST

Árbol binario de búsqueda [consulta: 1 de Maig 2020]

https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda

Binary Search Tree [consulta: 3 de Maig 2020]

https://www.geeksforgeeks.org/binary-search-tree-data-structure/

Rtree

rtree [consulta: 5 de Maig 2020]

https://github.com/davidmoten/rtree

R-Tree Implementation Java [consulta: 5 de Maig 2020]

https://stackoverflow.com/questions/8456240/r-tree-implementation-java

RTree.java [consulta: 6 de Maig 2020]

http://www.spf4j.org/jacoco/org.spf4j.base/RTree.java.html

Rtree Java Source Code [consulta: 6 de Maig 2020]

https://www.programcreek.com/java-api-examples/?code=gegy1000%2FEarth%2FEarth-master%2Fsrc%2F

Hash

Hash Tables and Hash Functions [consulta: 17 de Maig 2020]

https://www.youtube.com/watch?v=KyUTuwz$_b7Qt = 479s$

$WhatareHashFunctionsandHowtochooseagoodHashFunction?[consulta : 17deMaig2020]$

https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/

Hash function [consulta: 17 de Maig 2020]

https://en.wikipedia.org/wiki/Hash_function