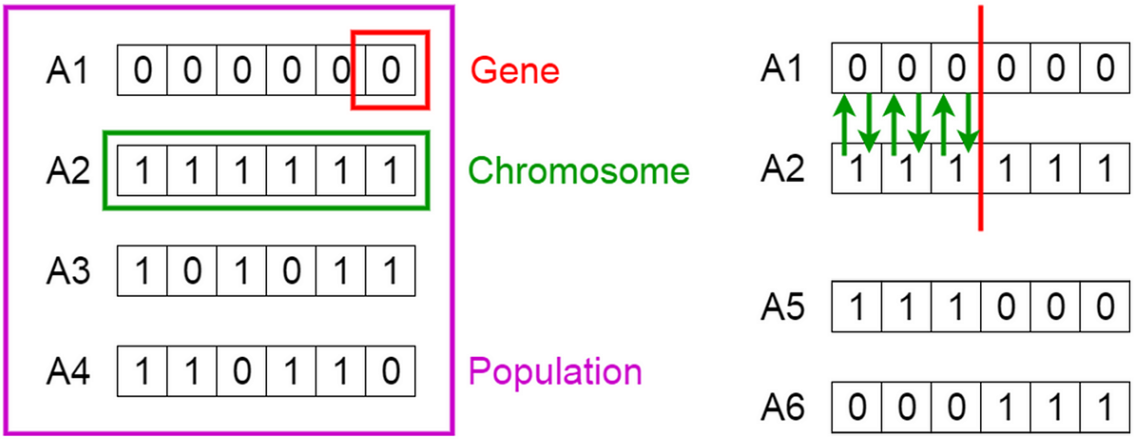
ED&ALG
Funcionalitat Ppal



Grup 11.3

Joel Corredor Casares
Rosa Hortelano Bronchal
Susanna Jané Ramón
Nicola Scognamillo

Índex

Estructures de dades

HashMap

Per l'algoritme de la classe FiveGuess hem decidit utilitzar HashMap per millorar el cost de la cerca exhaustiva. Tot i que amb el seu ús augmenta el cost en espai es reduirà el cost en temps. L'explicació de com l'hem implementat es troba en l'apartat de l'algoritme FiveGuess en les següents pàgines.

Els costos d'utilitzar un HashMap són:

- Ús de memòria: HashMap utilitza una estructura de taula de hash per emmagatzemar les parelles clau-valor. El nombre d'elements que pot emmagatzemar no està limitat per una grandària fixa inicial. A mida que s'afegeixen més elements, HashMap pot créixer i requerir més memòria per emmagatzemar les dades. El factor de càrrega de HashMap determina quan s'ha d'augmentar la grandària interna de la taula de hash per mantenir un rendiment òptim. Això pot implicar un increment en l'ús de memòria.
- Inserció i eliminació: Són operacions en promig de temps constant O(1). En el cas de les col·lisions, on dues claus diferents es mapegen a la mateixa ubicació, pot ser necessari realitzar operacions addicionals per gestionar-les.
- Cerca i accés: HashMap permet una cerca eficient d'un valor a partir de la clau associada. Les operacions de cerca i accés són en promig de temps constant O(1). En cas de col·lisions el rendiment pot disminuir lleugerament, ja que s'ha de gestionar aquesta situació amb una estructura de dades addicional.

ArrayList

Per tal de representar vectors de patrons i de correccions hem decidit utilitzar ArrayList. Aquesta classe ens permet utilitzar funcions bastant útils, com poden ser *empty()* per veure si les llistes estan buides sense tenir que accedir a elles.

Els costos són els següents:

- Ús de memòria: ArrayList utilitza un array subjacent per emmagatzemar els elements, el que significa que s'ha d'assignar una quantitat fixa de memòria per a aquest array. Tot i que la grandària de l'array pot augmentar automàticament a mesura que s'afegeixen elements, això pot conduir a un malbaratament de memòria, especialment si s'assigna inicialment una grandària molt gran.
- Inserció i eliminació d'elements: Són operacions de temps constant en promig O(1), hi ha casos en què poden requerir una còpia d'elements, cosa que augmenta el seu cost.
- Cerca i accés: La cerca pot ser costosa, especialment si es realitza una cerca seqüencial. En el pitjor dels casos, la cerca d'un element pot requerir recórrer tot l'ArrayList, el que resulta en un temps d'execució proporcional a la grandària de la llista O(n).

Classes Patro i CorreccioLinea

Per tal de representar els patrons i les correccions hem creat les classes Patro i CorreccioLinea respectivament, aquestes classes permeten emmagatzemar arrays i disposen de mètodes pel tractament específic que requereixen. Hem decidit crear aquestes classes per seguir el principi de disseny d'alta cohesió, aquestes dues classes tenen unes responsabilitats altament relacionades que no són de mida excessiva.

Stub Jugador

Per tal de poder crear i executar els tests unitaris de la classe Jugador ha estat necessari crear la classe StubJugador degut al fet que aquesta és abstracta. StubJugador conté implementacions trivials de tots els mètodes abstractes de Jugador.

Five-guess algorithm

Que és?

L'algorisme Five Guess es tracta d'una cerca exhaustiva, mitjançant la tècnica de minimax. Endevina la combinació proposada pel Codebreaker en una mitjana de 4.3411 torns pel cas habitual de 4 posicions i 6 colors, en el pitjor cas serien 6 voltes. Aquest algorisme va ser demostrat per Donald Knuth en 1977.

La tècnica minimax és un algorisme recursiu per a triar el següent moviment en un joc de n-jugadors, normalment en un joc de 2 jugadors. Un valor es associat a cada posició o estat del joc. Aquest valor esta decidit per una funció que avalua les posicions i indica que tan bona pot ser per al jugador arribar a aquesta posició. El jugador després fa un moviment que maximitza el mínim valor de les possibles posicions on podrà moure el rival.

Com l'hem implementat?

Per implementar aquest algoritme, hem creat una classe anomenada "*FiveGuess*". Aquesta és una extensió de la classe "*Algoritme*".

Funcions principals:

Hem creat 8 funcions diferents per a poder implementar l'algoritme. Aquestes funcions són les següents *setCorrecció(), trobarCombinacions(), trobarCombinacionsSenseRepeticions(), primerIntent(), actualitzaCombinacions(), maxim(), getIntent() i solve().*

L'algoritme se centra en la funció *getIntent()*, que va cridant a la resta. En crear una partida nova com a *codemaker,* amb 4 o 6 boles (aquesta última només sense repeticions de colors), es fa la crida a la funció per primer cop. Si aquesta té la llista de possibilitats buida, crida a una funció per trobar totes les combinacions. Depenent si la partida permet colors repetits o no, es cridarà una funció o un altre. Ho hem fet així per reduir la quantitat de possibilitats i fer l'algoritme més eficient. Les operacions encarregades de fer això són *trobarCombinacions()* i *trobarCombinacionsSenseRepeticions(),* que es basen a fer una cerca exhaustiva (algoritme de cerca utilitzant la força bruta). Les combinacions generades es guarden en un vector de patrons (ArrayList). Fem una còpia de la llista i la guardem com a intents per provar.

Després es crida a la funció primerIntent() que retorna un patró específic segons el nombre de boles de la partida i si es poden repetir colors o no. En el cas de 4 boles amb repeticions serà el patró {0, 0, 1, 1}, si són 6 o 8 boles s'encarregaria un altre algoritme (genètic). D'aquesta forma eliminen la màxima quantitat de combinacions possibles i les següents iteracions són més fàcils. Sense repeticions serien les següents: 4 boles {0, 1, 2, 3}, 6 {0, 1, 2, 3, 4, 5} i amb 8 no es faria amb aquest algoritme.

Un cop provat aquest intent, la partida retorna la correcció del patró enviat i s'actualitzen les possibles combinacions amb la funció *actualitzaCombinacions()*. Aquesta operació s'encarrega de comparar les diferents possibilitats de la llista amb l'últim intent (combinació provada) i si són pitjors que la solució proporcionada pel jugador descartar-les. Això ho fa cridant a les funcions *setCorreccio()* i *comparaIntent(),* aquesta última es troba a la classe Algoritme. A través de la classe algoritme s'agafa la correcció feta per l'usuari utilitzant *setCorreccio()*. Amb *comparaIntent()* es recorren els patrons per obtenir la seva correcció (quantitat de pius negres i blancs) i es compara amb la correcció donada pel jugador.

Després d'actualitzar les possibilitats, s'avaluarà quina combinació serà la que elimini més patrons en la següent iteració. Per fer això s'ha de buscar per cada combinació possible el mínim de patrons que s'eliminarien per després obtenir el màxim entre els mínims. Per a fer això, hem de calcular el màxim d'encerts de cada patró, amb l'operació *maxim()*. Aquesta compara (amb *comaparaIntent())* el patró passat amb la resta de combinacions de la llista, i calcula el màxim d'encerts d'aquesta.

L'algoritme original fa una cerca exhaustiva i recorre per cada patró possible, per cada combinació de pius de CorreccioLinea, cada combinació. D'aquesta forma el cost seria de: $combinacionsNoProvades \times correccioLinea \times combinacionsPossibles$. Per millorar el cost, hem disminuït el cost en temps d'execució, però augmentat el cost en espai fent que només recorri: $combinacionsNoProvades \times combinacionsPossibles$.

Ja que per cada parella de patrons només pot haver-hi una correcció, hem decidit utilitzar un HashMap. D'aquesta forma no hem de recórrer totes les correccions. Aquest HashMap té com a clau la correcció línia i el nombre d'encerts pel patró amb la resta. Cada cop que per dos patrons es dona un control específic, se suma en 1 el valor d'aquest. Mentre es fan aquests càlculs, es va guardant el màxim.

Un cop tenim aquest valor calculem el mínim de patrons eliminats (mínim = possibles combinacions - màxim d'encerts). Amb aquest mínim, el comparem amb el màxim mínim d'encerts de la iteració anterior, si aquest nou mínim és major que l'anterior se substitueix. Si són iguals i l'anterior combinació ja formava part de les possibles combinacions, no es fa res, ja que s'haurà d'agafar el patró més petit. Si el patró anterior no es troba entre les possibles combinacions i l'actual si, es canviarà.

El pròxim intent es decidirà segons la combinació amb el màxim de patrons eliminats. Aquesta combinació l'eliminarem de les combinacions per provar i serà retornada a la classe algoritme per a després continuar fins a arribar a la solució.

Tot el que hem explicat s'ha de fer per cada intent, a partir d'actualitzar les combinacions. En el cas que la combinació sigui de 4 boles s'arribarà a la solució en 5 iteracions. En els altres casos té un màxim de 10 intents per endevinar la combinació, si no ho aconsegueix perdrà la partida.

Genetic algorithm

Que és?

És una cerca heurística. Basada a evolucionar una població d'individus sotmetent-la a accions aleatòries, semblants a les que actuen en l'evolució biològica, així com també a una selecció.

El procés comença amb un conjunt d'individus que es denomina Població. Cada individu és una solució al problema que es vol resoldre. Aquests es caracteritzen per un conjunt de paràmetres denominats Gens. Els gens s'uneixen en una cadena per a formar un cromosoma (solució). El conjunt de gens d'un individu es representa mitjançant una cadena de char/int/float/bits.

Com l'hem implementat?

Per implementar aquest algoritme, hem creat una classe anomenada "*Genetic*". Aquesta és una extensió de la classe "*Algoritme*".

Funcions principals:

Hem creat 21 funcions diferents per a poder implementar l'algoritme. Aquestes funcions són les següents *setCorrecció(), primerIntent(), iniciarPoblacio(), calcularAptitud(), ordenaSegonsAptitud(), ordena(), divide(), swapInteger(), swapPatro(), evolucionaPoblacio(), crossover1(), crossover2(), mutar(), permutar(), invertir(), getPosicioPares(), repeticionsRandom(), esRepeticio(), afegirAPossibles(), getIntent() i solve().*

L'algoritme se centra en la funció *getIntent()*, que va cridant a la resta. En crear una partida nova com a *codemaker,* amb 6 boles (només amb repeticions de colors) o 8 (tant amb repeticions de color com sense), es fa la crida a la funció per primer cop. Si aquesta té les llistes amb correccions negres i blanques buides, es crida a la funció *primerIntent().* Aquesta funció retorna un patró específic segons el nombre de boles de la partida i si es poden repetir colors o no. En el cas de 6 boles i colors repetits serà el patró {0, 0, 1, 1, 2, 3}, en el de 8 i colors repetits serà {0, 0, 1, 1, 2, 2, 3, 4} i en el cas de 8 boles colors sense repeticion {0, 1, 2, 3, 4, 5, 6, 7}. Aquest intent s'afegeix a una llista d'intents provats i es retorna perquè es provi com a primer intent.

En el cas que no es cridi per primer cop a la funció, l'usuari ja haurà fet una correcció i aquesta es pasarà a l'algoritme utilitzant l'operació *setCorreccio()*, que s'encarrega d'afegir el nombre de pius blancs i negres a les llistes pertinents. Per tant, les llistes de pius negres i blancs no estaran buides. Llavors es generarà una població aleatòria i es calcularà l'aptitud per a aquesta població.

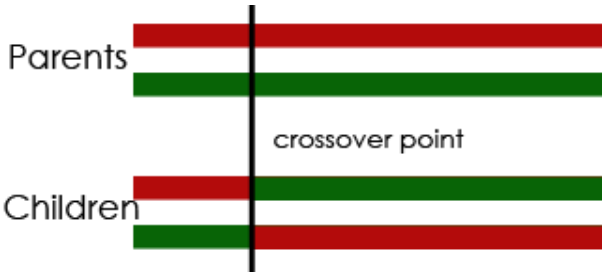
Per a generar una població es fa servir la funció *iniciarPoblacio()* que s'encarrega de generar de forma random totes les combinacions de la població. Un cop generada la població, es calcula l'aptitud de cada patró amb la funció *calcularAptitud()*. Aquesta operació compara tots els patrons de la població amb els intents provats anteriorment. Amb la correccioLinea generada es resta el resultat de pius negres obtinguts amb el que va obtenir l'intent anterior i es fa el mateix amb els pius blancs, per després sumar-ho tot.

Després de calcular totes les aptituds les ordenem amb un quick sort. Aquest sort té un cost de $O(n \log n)$ en el millor cas i en el pitjor de $O(n^2)$. Les funcions encarregades de fer-lo són: *ordenaSegonsAptitud(), ordena(), divide(), swapInteger(), swapPatro().* Aquestes ordenen tant els patrons de la població com la llista d'aptituds.

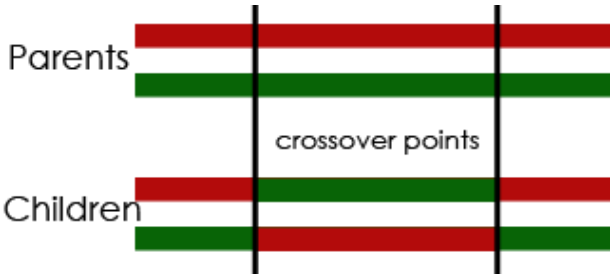
Amb les aptituds ja ordenades podem començar a evolucionar la població. Sempre que no hàgim trobat cap combinació possible i no superem el nombre màxim de cops (en el nostre cas 5000) que vulguem evolucionar la població. Això ho farem amb la funció *evolucionaPoblacio()*. En el moment que trobem algun patró possible (s'explica més endavant com saber si un patró és possible en la funció *afegirAPossibles*) haurem acabat i després que l'usuari introdueixi la correcció farem una nova iteració. Finalment, es guardaria l'intent a la llista d'intents provats i es retornaria l'intent.

La funció *evolucionaPoblacio()*, utilitza el crossover d'1 i 2 punts entre patrons aleatoris. També fa ús de mutacions, permutacions i inversions per generar noves poblacions (evolucionar la generada anteriorment). Per generar la nova població, un de cada dos està generat amb un crossover (50% de ser d'1 punt i 50% de ser de 2 punts). Això ho aconseguim amb les funcions *crossover1(), crossover2()* i amb *getPosicioPares()* que escull aleatòriament dos patrons de la població.

crossover1() genera un número random per escollir la posició a partir d'on es canviaran els colors entre dos patrons. En la imatge següent es pot veure un exemple:

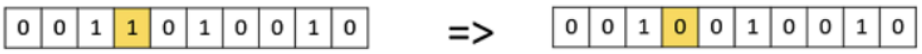


crossover2() fa el mateix que crossover1 però amb dues posicions de separació.

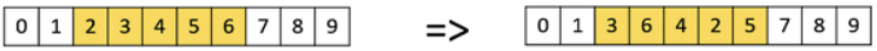


Sense tenir en compte el valor de la nova població es fa una mutació amb una probabilitat de 0,03, una permutació amb un 0,003 i una inversió amb el 0,02. Tot això ho fem amb les funcions *mutar(), permutar(), invertir().*

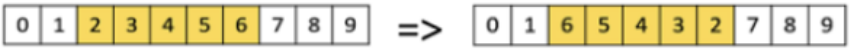
mutar() genera una posició random i intercanvia el color de la posició per un altre.



permutar() genera dues posicions random i barreja i canvia d'ordre de forma random els colors del tros de patró que es troba entre les dues posicions abans generades.



invertir() genera dos posicions random i inverteix l'ordre dels colors en el tros del patró que es troba entre les dues posicions abans generades.



En cas de que es generi un patró duplicat, aquest el canviem per una combinació de colors random i actualitzem la població. Ho podem comprovar amb la funció *repeticionsRandom().* Aquesta cridarà a la funció *esRepeticio()* per cada combinació de la població. Si es repeteix el patró *esRepeticio()* tornarà un boolea amb true, llavors *repeticionsRandom()* generarà una combinació random nova.

Per a saber si un patró es pot afegir als possibles intents que siguin solució s'utilitzarà la funció *afegirAPossibles().* Aquesta operació compara cada element de la població amb tots els intents que s'han provat anteriorment i mira que la seva correcció tingui els mateixos valos de pius negres i pius blancs. Si es compleix i la llista no està plena ni tenia anteriorment el patró s'afegeix a la llista.

Tot el que hem explicat s'ha de fer per cada intent, té un màxim de 10 intents per endevinar la combinació, si no ho aconsegueix perdrà la partida.

Cerca Lineal

Que és?

És un un simple algorisme que bàsicament recorre tots els elements fins trobar l'element buscat. En el nostre cas, utilitzem aquest algorisme per recorrer tots els elements d'una estructura per anar canviant els seus elements o anant agafant la informació d'ells.

Com l'hem implementat?

La utilització d'aquest algorisme ha sigut constant i per tot tipus d'estructures. Com s'ha mencionat en l'apartat anterior més bé que per buscar un element concret l'hem utilitzat per recòrrer estructures completes.

Hem implementat cerca lineal tant en vectors, com Patro, com Rankings, com CorreccioLinea, com ArrayList de qualsevol dels anteriors. D'aquesta manera ens asseguravem que recorriem tota la estructura.

Funcions on hem utilitzat:

Per passar d'un tipus d'estructura a un altre, exemples sent: correccioToInt() de CorreccioLinea i correccioToString() de Partida o la lectura de les posicions dels rankings. També ho fem servir quan hem de canviar informació en matrius com són les boles de les correccions de una Partida o les boles dels intents d'una Partida. I l'últim cas seria en el cas de que necessitem recórrer una estructura per les seves dades.
En el cas dels rankings, l'hem utilitzat per a recorre'l sencer llegint totes les posicions per a poder comparar les amb la nova posició candidata a ranking.

Cost:

El cost de la cerca lineal és el nombre de la posició en la que es troba l'element que búsquem. Sent el millor cas que estigués en la primera posició, que el cost seria (1), i el pitjor que fos al final del vector, que el cost seria (n), sent 'n' la mida de l'estructura.

En el nostre cas, sempre el cost serà el pitjor cas, ja que hem d'accedir sempre a totes les posicions, per tant el cost serà (n), sent aquesta la longitud de l'estructura.

Accés a un fitxer

Que és?

És un algoritme que bàsicament escriu sobre un fitxer de text, en el nostre cas, en fitxers .txt. Accedim a fitxers tant per a carregar el contingut al programa com per a guardar certes coses del programa per a poder recuperar-les més endavant. També accedim a l'hora de crear les carpetes i els fitxers que necessitem per al correcte funcionament de la persistència.

Com l'hem implementat?

La utilització d'aquest algoritme ha sigut només en la capa de persistència. L'hem implementat a través de la lectura i escriptura de la classe de Java "File", amb les funcions de la pròpia classe.

Alguns exemples poden ser, mkdir() per a crear carpetes, separator per a crear un separador global en tots els sistemes operatius, bw.write() per a escriure i bw.close() per a tancar el flux de l'escriptura, br.readline() per a llegir i br.close() per a tancar el flux de lectura, entre d'altres.

Hem implementat aquest algoritme en les quatre classes de la capa de dades, les quals serien, PartidaData, RankingCronoData, RankingClassicData i RecordsData.

Funcions on hem utilitzat:

Ho hem utilitzat en les funcions de lectura i escriptura d'aquestes classes, com ara *creaArchives()*, *carregarPartida(), carregarRanking()*, *carregarRecords()*, *guardarPartida(ArrayList)*...

En aquestes funcions, menys a la creadora d'arxius/carpetes, per a triar en quin arxiu volem escriure/llegir, ho hem fet a partir d'un String amb el path de la ruta corresponent, i així poder triar quin fitxer volem. En el cas de la PartidaData i dels RecordsData, la ruta serà sempre la mateixa al només haver-hi un fitxer.

Cost:

El cost de per a accedir a un arxiu pot variar per culpa de diversos factors.

Per exemple, es veu afectat pel temps d'execució, perquè com implica obrir i llegir/escriure en dades del disc, pot ser una operació relativament més lenta en comparació a altres instruccions. El cost dependrà de la mida del fitxer i la velocitat del disc dur.

I un altre factor poden ser els recursos del sistema, com per exemple l'espai en la memòria i la capacitat del processament.

Degut això, nosaltres suposarem que tant la lectura com l'escriptura d'un fitxer és de (1).

Partint d'això, les funcions de creació de carpetes tenen un cost de (1) en tots els casos. El pitjor cas seria el cas de la creació, però al ser (1) és negligible.

En el cas de les funcions de creació d'arxius tenen un cost de (1) en el cas de les classes RecordsData i PartidaData en tots els casos i un cost de (n) en el cas de RankingClassicData i RankingClassicCrono, ja que es creen diversos arxius. (n) serà el

pitjor cas, perquè seria el cas on no està creat cap dels arxius i llavors el programa els haurà de crear tots.

Tant les funcions de guardar dades i de carregar dades en un fitxer tenen un cost de (n), perquè estan formades per un bucle for de N posicions que són les files de l'Array de dades i dins d'aquest for, tenim un accés. Una vegada fora del for, tenim un altre accés, per tant, ens quedaria un cost de (n)+(1) que acaba sent (n).