P2 - RUNNERSLS

Programació Avançada i Estructura de Dades

Grup 8
Marc Valsells Niubó (marc.valsells)
Pol Verdú Oliva (pol.verdu)

Índex

Llenguatge de programació escollit

Donat que aquesta pràctica té certa continuació amb l'anterior s'ha seguit utilitzant Java com a llenguatge de programació per tal de poder aprofitar el codi per llegir les dades del arxiu JSON. A més a més, donat que s'havia de llegir un segon fitxer JSON amb Java i utilitzat la llibreria GSON és una acció que és pot realitzar sense gaire complicació comparat C (l'altre llenguatge que dominen els components del grup) al qual seria necessari anar llegint i passejant manualment tot l'arxiu. Addicionalment Java ja ens proporciona varies classes i interfícies que ens han sigut útils alhora d'implementar els algorismes, especialment amb el Branch and Bound.

Explicació com s'han codificat els algorismes

Backtracking – Cursa Relleus

Per a la primera qüestió de l'enunciat, s'ha implementat aquest algorisme. A la funció de backtracking, se li passa l'array de configuració actual, la posició de l'atleta actual a l'array d'atletes i el nombre màxim d'equips possibles que es poden formar, el qual canviarà depenent del fitxer que s'utilitzi.

Primer s'inicialitza la primera posició de l'array de configuracions a 0, perquè el següent bucle anirà trobant les millors configuracions fins que s'hagi arribat al nombre màxim d'equips. Dins d'aquest bucle i tal i com funciona l'algorisme del backtracking, primer es comprova si la configuració d'atletes actual està complerta (funció: "comprovaEquips"), és a dir, si s'ha trobat una possible solució. Si no és el cas, però la solució podria ser una configuració millor que la millor configuració obtinguda fins el moment, es tornarà a cridar la funció backtracking, i així recursivament fins acabar de trobar la configuració actual. En cas que la configuració no estigui complerta, però ja no és possible que sigui millor que la millor configuració obtinguda fins al moment, es podarà aquella branca i es seguirà buscant configuracions.

El mètode per comprovar si una configuració és millor que la millor obtinguda fins al moment, és el següent. L'objectiu que proposa la qüestió de l'enunciat és que les mitjanes de velocitat de cada equip siguin el més properes possible. Per aquest motiu s'ha creat una funció ("calculDifMitjanes") a la que li passem la configuració i el nombre màxim d'equips, la qual calcularà la suma de totes les diferències de les mitjanes entre tots els equips, per tant la nostra solució serà la que tingui aquest paràmetre més petit.

D'aquesta manera, s'aniran actualitzant la "millorConfig" per trobar la millor solució, és a dir, la que tingui la menor diferència entre les mitjanes de velocitat de tots els equips.

Backtracking – Cursa individual

Aquesta és la segona implementació que s'ha fet del backtracking a la segona qüestió que proposa l'enunciat. Al mètode de backtracking se li passa la informació de la configuració actual, és a dir els valor del esforç en cada tram, quin és el tram que estem provant actualment, el nombre total de trams i quins esforços ja s'han utilitzat.

Inicialitzem a esforç 1 el primer tram de la primera configuració i a continuació executem un bucle el qual ens va generant les possibles configuracions dels esforços pels trams. Un cop entrem el bucle és mira si l'esforç actual ja s'ha utilitzat o no, en cas afirmatiu podem i passem al següent esforç. En cas contrari marquem el esforç com a utilitzat i calculem l'estimació de l'esforç fins el punt actual amb la llibreria proporcionada.

Un cop calculat el l'esforç fins el punt actual mirem si ja estem en el últim tram de la curs, és a dir ja tenim tots els trams amb el seu esforç corresponent i per tant podria ser una possible solució. En aquest cas comprovarem si la solució obtinguda és la millor fins ara o no, en cas que si guardarem la seva informació, i en cas contrari descartem la solució.

Si un cop hem calculat l'esforç estimat encara no hem arribat a una possible solució comprovarem si l'esforç obtingut és més petit de la millor solució obtinguda, en aquest cas cridarem la funció de backtracking de forma recursiva sumant-li 1 al tram actual fins a podar o obtenir una possible solució, en cas contrari podarem i descartarem la branca donat que sempre ens donarà un valor més alt al obtingut anteriorment.

Al finalitzar totes aquestes accions desmarcarem l'esforç actual com a utilitzat per tal de poder-lo utilitzar en un nivell superior i sumarem u a la configuració del tram actual per provar amb el següent tram.

Branch and Bound

Per tal de codificar el Branch and Bound s'ha creat una classe extra CIconfig (Cursed Individuals config) la qual conte informació sobre la configuració en aquell punt específic, en cada objecte d'aquesta classe és guarda la configuració actual, el nombre d'esforços utilitzats i en quin tram ens trobem; a més a més s'ha implementat un mètode el qual donat un punt genera els seus successors. Per implementar el Branch and Bound s'ha utilitzat una PriorityQueue la qual sempre ens proporciona l'element amb l'estimació més baixa, per tal d'aconseguir-ho s'ha implementat la interfície Comparable de Java a la classe CIconfig i s'ha utilitzat la llibreria proporcionada RaceHelper.

Un cop ja tenim la cua de prioritats amb el primer element ens hem basat en fer un bucle el qual és va executant mentre hi hagi elements a la cua. El primer que fem en aquest bucle és agafar el primer element de la cua (el que te l'estimat més baix) i obtenim els seus successors. En el cas de que el successor sigui una possible solució, és a dir que ja tinguem tots els trams assignats, mirem si el estimat és el millor que hem obtingut fins ara, en cas que ho sigui el guardem, en cas contrari el descartem. Per altre banda, si no tenim tots el trams assignats, l'afegim a la cua de prioritats.

Acte seguit tornem al inici del bucle per agafar el objecte CIconfig, és a dir el successor, amb menys estimat, ja sigui una configuració dels últims successors obtinguts o una d'una configuració d'un nivell superior. Un cop la cua estigui buida ja sabrem quina és la configuració amb el estimat més baix.

Greedy

Alhora d'implementar la gestió d'horaris és volia aconseguir en assistir el màxim nombre de curses independent de la seva durada, per tant s'ha optat per agafar primer les curses amb la durada més curta. S'ha utilitzat un array en el qual s'indicaven quines curses ja s'havien utilitzat i quines curses és solapaven amb les utilitzades, la implementació del algorisme consisteix en un bucle que és va executant fins que s'ha utilitat i/o descartat totes les curses.

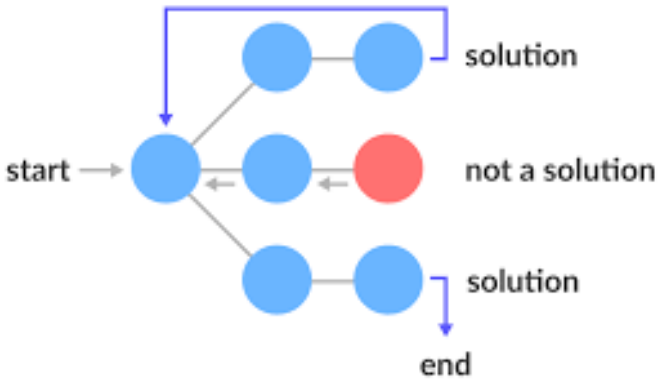
Just abans d'iniciar aquest bucle s'ordenen totes les curses per ordre de durada, agafem la cursa mínima i que ja no hagi sigut descartada, en la primera volta no s'ha descartat cap, a continuació la descartem i indiquem a l'array de configuració que la cursa en aquella posició ha sigut elegida. Acte seguit realitzem un segon bucle, encara dins del primer, per mirar si la resta de curses que ja no han estat descartades anteriorment és solapa amb l'ultima elegida, en cas que sigui així la descartem.

Al final totes les curses hauran sigut descartades i en el array de config tenim marcat amb un 1 les curses les quals hi podem assistir.

Comparativa d'algorismes

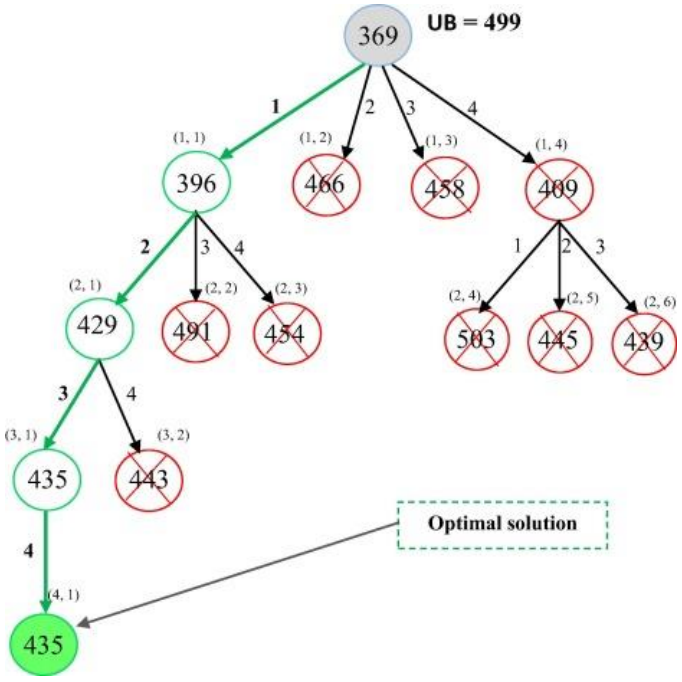
Backtracking

Backtracking és emprat per cercar totes les solucions possibles a una qüestió proposada. Aquest algorisme s'utilitza de forma recursiva per trobar totes les possibles solucions. Aquest algorisme, quan es troba a la situació en que veu que ha fet una mala decisió i per tant aquella branca no és solució, torna enrere i segueix buscant. A diferencia de Branch and Bound, Backtracking és utilitzat per solucionar qüestions de decisions, no d'optimització com és el cas del segon algorisme, i a més a més, és més eficient el Backtracking.



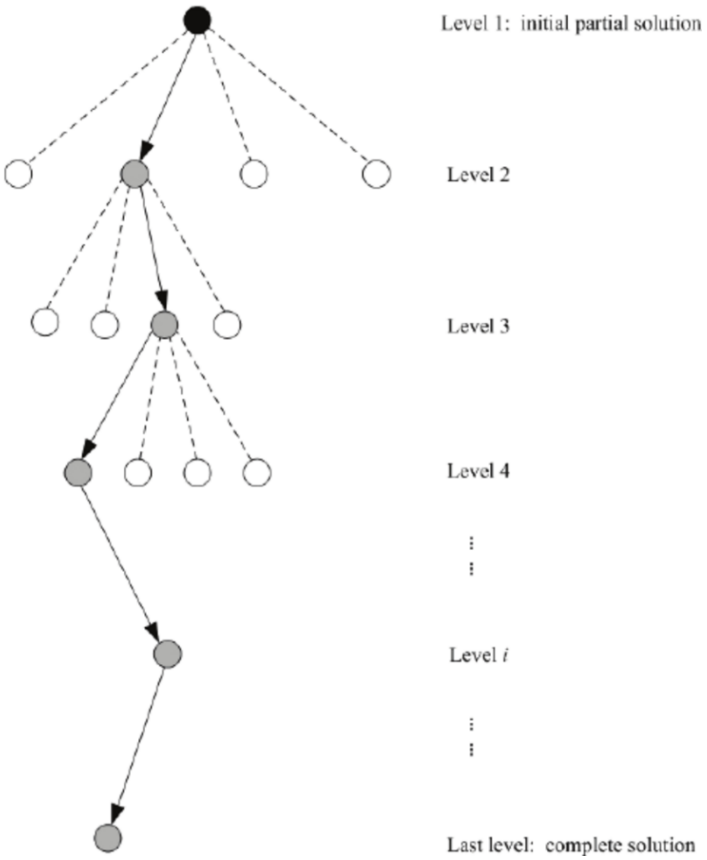
Branch and Bound

Aquest algorisme està dissenyat per resoldre problemes d'optimització combinatòria i optimització matemàtica. Aquest algorisme el que fa es mitjançant l'expressió de la qüestió a tractar com les arrels d'un arbre, analitza cada arrel buscant les solucions més òptimes. Una vegada una solució ja sigui total o parcial, és a dir, encara que no hagi arribat al final de la solució, ja és pitjor que la millor, la descarta i segueix buscant. Aquest algorisme és molt útil per a buscar les millors solucions (optimització) de certes qüestions, això comporta també que sigui menys eficient que el Backtracking.



Greedy

Aquest algorisme, és un tant més peculiar que els altres dos. A diferència del Backtracking i Branch and bound, a mesura que va trobant solucions, aquest algorisme no pot retrocedir per rectificar i aconseguir la millor solució. El que fa molt útil aquest algorisme és la seva extremada eficiència, és a dir, és molt més ràpid que el Backtracking i Branch and Bound. Per aquest motiu aquest algorisme, encara que no doni la millor solució, sinó que dona una possible solució i de manera molt ràpida, és molt útil quan s'utilitza de manera complementaria amb altres algorismes com pot ser Backtracking.



Anàlisis dels resultats

Cursa relleus				
Backtracking	0.085s	//	//	//
Cursa individual				
Numero de trams	Trams: 5	Trams: 10	Trams: 15	Trams: 20
Backtracking	5,638s	6,456s	1,631s	1,441s
Branch and Bound	2,695s	2,913s	3,337s	/ /
Gestió d'horaris				
Mida dataset	Fitxer XS	Fitxer S	Fitxer L	Fitxer XXXL
Greedy	0,002s	0,004s	0,025s	4.014s

Degut a la mida desmesurada dels fitxers S, M, L i XL, només s'ha pogut testejar i comprovar l'eficàcia de l'algorisme amb el fitxer més petit, el qual s'ha modificat per que hi trobi 2 equips en comptes d'un. Al llarg de la implementació de l'algorisme, s'han anat veient els costos de temps que tenia, i el major canvi ha arribat quan s'ha implementat la poda de configuracions que ja no podien ser una millor solució passant de mig segon d'execució aproximadament al temps que veiem a la taula. S'ha pogut comprovar per tant que aquest algorisme és prou ràpid tenint en compte la quantitat de possibles solucions que contempla.

Si s'analitza la taula, salta a la vista la eficiència de la tercera qüestió que proposa l'enunciat, la qual ha sigut implementada amb Greedy. Aquest algorisme és molt ràpid, encara que no proposi la millor de les solucions en tots els problemes, és una de les millors (en el cas de la qüestió proposada per l'enunciat sí que és la millor). I es pot observar que aquest algorisme triga molt poc en executar-se, també pel fet que és més simple que els altres dos.

La segona qüestió de l'enunciat (cursa individual) en la que s'havia de trobar la millor combinació dels esforços per dur a terme la carrera, ha sigut implementada amb dos algorismes: Backtracking i Branch and Bound. Aquesta qüestió és més difícil d'analitzar a nivell de temps d'execució ja la implementació del codi inclou una funció d'una llibreria proposada per l'enunciat que fa que cada execució sigui diferent a la anterior. Com es d'esperar, a mesura que es van afegint trams a la combinació, el temps d'execució augmenta en tots dos algorismes, sobretot en Branch and Bound. Però si s'analitza el temps que triga Backtracking amb 15 trams d'esforços, aquest disminueix, possiblement perquè el número de podes és major a mesura que anem augmentant els trams d'esforç.

Mètode de proves utilitzat

Per anar provant si els algorismes implementats eren funcionals s'ha anat mostrant per pantalla les configuracions i, en el cas de la cursa individual el estimat, per cada punt per veure com s'anaven creant les configuracions i si eren correctes o no. També s'ha utilitzat l'eina del debugger en certs punts, com per exemple en el Branch and Bound per verificar que els successor és creessin correctament.

Per analitzar el codi s'ha utilitzat el mètode System.currentTimeMillis() amb aquest mètode s'ha guardat el temps abans i després d'executar els algorismes per saber quants de temps tardaven en executar i a posteriori poder realitzar les comparacions amb els diferents valors i datasets.

Problemes observats

Curses de relleus

La primera qüestió de l'enunciat era la cursa de relleus, que tal i com s'ha vist ha sigut la que més problemàtica ha portat a la implementació del projecte. Aquesta qüestió era la que faltava per implementar, amb l'algorisme Backtracking. Aquest algorisme no ha resultat difícil d'entendre ja que ja s'havia implementat anteriorment la segona qüestió amb Backtracking i per tant ja es coneixia el seu funcionament. D'altra banda, el major problema del projecte s'ha donat en aquesta qüestió. S'han trobat molts problemes alhora de realitzar els equips i això ha sigut el que ens ha portat a estirar tant la implementació d'aquesta qüestió. Després de varies sessions de dubtes amb el becari hem pogut assolir i implementar correctament els conceptes d'aquest algorisme.

Curses individuals

Per a aquesta segona qüestió proposada a l'enunciat, es va optar primerament per la implementació de l'algorisme de Backtracking, amb el qual es van aconseguir resultats òptims. Aquesta qüestió de l'enunciat va ser la segona en ser portada a terme i per això només faltava la primera qüestió (Cursa de relleus) per implementar amb l'algorisme Branch and Bound. Durant la implementació d'aquesta primera qüestió es va veure que la manera més òptima de implementar tots dos algorismes era utilitzant Backtracking a la primera qüestió i Branch and Bound a la segona. Per tant es va procedir a rectificar la segona qüestió implementant-la amb l'algorisme Branch and Bound. Arribat a aquest punt, es tenia la tercera qüestió (gestió d'horaris) implementada amb Greedy i la segona qüestió implementada amb dos algorismes diferents (Backtracking i Branch and Bound).

Gestió d'horaris

Durant la implementació del tercer problema proposat per l'enunciat (gestió d'horaris) no s'han trobat problemes majors que hagin suposat un gran repte alhora de implementar l'algorisme Greedy. Inicialment es va començar per ordenar totes les curses segons la hora en que es duia a terme però havent vist els resultats s'obtenien del codi, es va optar per una forma d'ordenar les curses de manera alternativa. Aquest problema es va solucionar ordenant les curses segons la seva durada de més petites a més grans, arribat a aquest punt, s'agafarien totes les curses possibles amb la menor durada, descartant totes aquelles curses incompatibles cada vegada que s'agafava una cursa amb la mínima duració.

Conclusions

Tot i que no s'ha pogut dur a terme tots els requeriments necessaris a de la pràctica, s'ha pogut aprendre com funcionen Greedy, Backtracking i Branch and Bound donat que d'una manera o altre s'han acabat implementant tots 3 algorismes. Per tant s'ha pogut

veure el seu funcionament i com en alguns casos és millor un o l'altre, o inclús la combinació dels dos com és el cas de podar el Backtracking amb una configuració anteriorment obtinguda amb Greedy per tal d'optimitzar el màxim.

De cara a la següent pràctica seria millor començar a plantejar-la i implementar-la amb més antelació donat que és va planificar i sintetitzar l'enunciat amb antelació però la part d'implementació s'ha deixat els últims dies i això a jugant en contra alhora d'entregar la pràctica correctament i a temps.

Bibliografia

How do you find the sum of all the numbers in an array in Java? – StackOverflow [en línia] [data de consulta 11 de gener 2021] Disponible a: https://stackoverflow.com/questions/4550662/how-do-you-find-the-sum-of-all-the-numbers-in-an-array-in-java

How to find the minimum value in an ArrayList, along with the index number? (Java) – StackOverFlow [en línia] [data de consulta 11 de gener de 2021] Disponible a: https://stackoverflow.com/questions/15995458/how-to-find-the-minimum-value-in-an-arraylist-along-with-the-index-number-jav

Checking if two time intervals overlap – CodeReview [en línia] [data de consulta 8 de gener 2021] Disponible a: https://codereview.stackexchange.com/questions/206710/checking-if-two-time-intervals-overlap

Greedy Algorithms for Time-Slot Interval Optimization – Youtube [en línia] [data de consulta 7 de gener 2021] Disponible a: https://www.youtube.com/watch?v=BWlXudP7Unk

LocalTime compareTo() method in Java with Examples [en línia] [data de consulta 9 de gener 2021] Disponible a: https://www.geeksforgeeks.org/localtime-compareto-method-in-java-with-examples/

How to check ALL elements of a boolean array are true - StackOverflow [en línia] [data de consulta 9 de gener 2021] Disponible a: https://stackoverflow.com/questions/18631837/how-to-check-all-elements-of-a-boolean-array-are-true/18632446