1. FUNCIÓ ORDENATAULA 2.1.4 (PRÀCTICA 2)

Aquesta part de la pràctica 2 consistia a crear una funció ordenataula que, a partir de la taula alumnes i un segon argument amb una taula alumnesord buida, guardi a la segona taula la mateixa informació que hi ha a la primera, però amb les entrades ordenades per NIU, aquesta funció també necessita que se li entrin el nombre d'alumnes com a paràmetre i hem de mostrar per pantalla aquesta segona taula.

Aquesta funció l'hem hagut de canviar respecte a l'anterior, ja que no era gens eficient el que estàvem fent.

Abans el que fèiem era posar un número mínim i un número màxim que englobés tots els nius i els buscàvem de manera ascendent trobant els nius en ordre i afegint-los a la nova llista juntament amb la seva informació.

És una solució que podria funcionar, però que era molt poc fina i gens optimitzada, ja que comprovàvem cada niu, fins si tot els que no existien. Aquests que no ens trobàvem no els analitzàvem, ja que fèiem un for fins a trobar el número del NIU.

Exemple: comencem amb el valor 1000, el busquem amb alumnes[j]. Niu per tot el fitxer d'alumnes (si es troba a la posició 3000 hem hagut de comprovar 2000 cops) i si existeix guardem aquest niu a la llista ordenada, si no, analitzem el 1001.

Per poder arreglar i fer-ho de manera més ràpida i optimitzada, hem fet dos bucles que recorreran els elements.

En primer lloc el que fem és comprovar si el niu en la posició j+1 és més petit que el niu en la posició j si és així es guarda el valor j a una variable i guardem al nou vector alumnes j+1 i després a la següent posició d'alumnes ordenats guardem alumnes, és a dir, els intercanviem de posició i ho guardem i això es repeteix fins que tot queda ordenat.

```
void ordenataula(Alu *alumnes, Alu *ordenats_aux, int n){
   Alu aux;
   for (int i = 0 ; i < n; i++ ){
                for (int j = 0; j < n; j++){
                        if (alumnes[j+1].niu < alumnes[j].niu){
                                aux = alumnes[j];
                        ordenats_aux[j] = alumnes[j + 1];
                                ordenats_aux[j + 1] = aux;
                                }
```

2. FUNCIÓ ESBORRAR 5.1.3 (PRÀCTICA 5)

En aquest exercici havíem de contemplar l'opció d'eliminar la informació corresponent a un identificador donat mitjançant una funció esborrar.

Per a fer-la, caldrà localitzar el bloc corresponent a l'identificador, refer la llista i alliberar la memòria.

La funció rep com a paràmetre el NIU a esborrar i el llistat d'alumnes.

A partir d'aquí fem una cerca per la llista d'alumnes la qual estarà ordenada i que pararà quan trobem el niu que volem esborrar. Si el NIU es troba a la llista, gaurdarem en una variable l'elemnt Alu * i actualitzem l'apuntador següent al NIU que hi ha després del cercat i alliberem memòria.

En cas que el NIU que es busqui no estigui a la llista acabem amb el programa acompanyat d'un missatge d'error.

3. CERCA DE CAMÍ ENTRADA I SORTIDA (PRÀCTICA 7)

Aquesta part no la vam arribar a fer.

Havíem de programar el recorregut (per nivell o en profunditat) de tots els vèrtexs per tal de descobrir si existeix un camí que uneixi els punts d'entrada i sortida marcats amb les fletxes verdes. Adaptant el programa per tal que pugui mostrar (si existeix) un camí que solucioni el problema.

Per a aquesta pràctica, s'ha programat una cua senzilla amb les funcions de cua d'encuar i desencuar.

En aquest cas, en el laberint només hi havia 4 arestes per node de manera que si la variable de l'estructura que controla quantes arestes s'han visitat és 0, s'encua aquest node, i aquesta variable passa ser 1.

Per saber si hem arribat al final, comprovem si l'element de la cua a l'inici és l'element final, trenquem el bucle i imprimim per pantalla el camí.

4. EXERCICI 8.1.1 (PRÀCTICA 8)

Exercici 8.1.1: Obteniu, mitjançant l'algoritme Dijkstra, el camí més ràpid per a connectar l'ordinador 0 amb cada un dels altres

Aquest exercici no el vam poder arribar a enllestir del tot (es troba a mitges falta de temps), però l'hem i el vam pensar d'aquesta manera.

En aquest cas, com que el graf és petit i volem calcular el temps per a tots els nodes, mantindrem una llista de nodes pendents amb una estimació del recorregut corresponent a cada un d'ells. A cada pas, traurem d'aquesta llista el node amb recorregut més curt, actualitzant l'estimació del recorregut per a tots els nodes que estan connectats amb ell i no hagin sortit de la llista de nodes pendents.

En aquest cas s'hauria de programar l'algorisme dijkstra on, en primer lloc, hem de posar tots els elements de la cua en pendent i inicialitzat el temps a infinit menys el primer de tots que és arrel i que té la variable visitat diferent a 0 indicat que l'hem expandit.

Ara es comprova el temps de cada adjunt comparat amb el de l'actual més el de l'adjunt, si el temps calculat amb la suma és més petit que el temps del node adjunt, aquest temps s'actualitza amb el mínim i assignem com a anterior el node estudiat.

Els nodes s'afegeixen a la cua prioritzant el cost més petit, amb la funció encua, en cas que el node ja s'hagi visitat i tingui la variable que ho indica diferent de 0 es reencuarà actualitzant el seu cost.

Un cop fet, i expandit els nodes necessaris per arribar al node final, es mostren per pantalla

L'objectiu d'aquesta funció, és carregar un vector amb tots els nodes pare a partir del node final, és a dir tindre un vector que comenci pel node d'arribada i acabi al node inicial amb tots els nodes que utilitzem o que impliquem per unir aquests dos punts, els quals uns seran els pares o anteriors dels altres.

Per carregar un vector sense haver de fer un hem de saber quants nodes o elements hi haurà en aquest vector.

Per saber això podem recórrer tots els nodes pare del node final fins que el node anterior dels anteriors del node final sigui el node inicial, i contem quantes iteracions hem fet fins a trobar-ho.

Un cop contats creem aquest vector amb n+1 components, i tornem a començar de nou fent exactament el mateix, però ara a part de contar iteracions per després llegir el vector carregat a la inversa, anem carregant el vector on vector[0] serà el node final i vector[1] el seu node pare i així successivament.

Un cop acabat aquest segon while tenim el vector carregat amb el camí a la inversa, de manera que l'hem de recórrer des del final fins al principi per mostrar el camí en l'ordre correcte.

Per a fer-ho fem un for utilitzant un dels iterats carregats amb el nombre de nodes del camí i per imprimir-los tots, els imprimim restant un al valor del iterador per anar enrere en el for, el qual pararà quan el valor del iterador sigui 0.

5. CARREGAR LA INFORMACIÓ DE CARRERS A LA PRÀCTICA 9 I COM VAM FER L'EXERCICI 9.2.2

El fitxer carrers, és una llista de carrers que concatenate entre ells és a dir els adjacents.

Per a la lectura del fitxer de carrers, el primer que hem fet ha estat inicialitzar una variable amb el nombre de carrers a 0 i hem obert el fitxer de carrers en mode de lectura fent un fopen.

Un cop obert, s'ha realitzat la lectura del fitxer contant quants salts de línia hi ha al fitxer, ja que a cada línia li un carrer amb els seus adjunts.

Un cop contats, fem un rewind per tornar a posicionar-nos al principi del fitxer i creem diferents variables, la primera és un tipus char anomenat carrer_id de 12 component on guardarem el primer carrer, dos unsigneds un actual i un anterior, i un long int on guardarem el valor de la id del node.

Per poder obtenir tota la informació, fem un for que s'anirà repetint fins a arribar a la iteració del nombre de carrers contats abans.

En aquest bucle, agafem el primer carrer, i el guardem a la variable char[12] i inicialitzem la variable anterior a 0. Dintre d'aquest bucle comencem un while que anirà llegint el fitxer de dades de carrers fins que ens trobem un salt de línia.

El primer pas en aquest while és fer un fseek per tirar una posició enrere, això ho vam posar, ja que en fer un fgetc es menjava aquesta mencionada.

Dintre d'aquest while hem de distingir dos casos, el cas on la variable anterior sigui 0 i el cas on no.

Si la variable no és 0, és a dir que el node analitzat no és el primer de la llista, fem dos buscapunts per trobar la id dels dos nodes consecutius i posar-ho a la llista de nodes adjacents dels nodes que estem mirant i sumem un al nombre d'arestes del node dels dos.

Això es fa perquè carrer 1 concatena amb carrer 2 i carrer 2 concatena amb carrer 1 (bidireccional).

Per poder avançar guardem el node actual al node anterior i actualitzem el node actual i tornem a fer el mateix tota l'estona.

Si, en canvi, anterior és 0 de manera que és el primer carrer que analitzem, (primer de les línies de dades), fem que el node actual digui l'anterior i augmentem el nombre d'arestes d'aquest carrer en 1 (abans, per poder accedir a la llista de nodes necessitem la seva la ID i per això abans fem un buscacamí).

A l'exercici 9.2.2 havíem de fer a partir de les dades que tenia, guard, que el programa demani per pantalla la identificació d'un carrer i mostri els seus nodes separats per punt-i-coma (;) en un ordre que hi hagi una aresta entre un node i el següent.

Per a fer aquesta part, ha estat necessària la distinció entre carrers i rotondes.

Per als carrers, es recorren tots els nodes fins a trobar el carrer que estem buscant. A partir d'aquí es recorren totes les arestes que pertanyin al carrer que no passi per un node estudiat i no porti al node final. En cas de no trobar, fa un break i imprimeix les ID's dels nodes recorreguts, ja que hauríem trobat el camí.

En el cas de les rotondes s'ha programat una funció que recorre tots els nodes fins a trobar una que pertanyi a la rotonda i es recorren tots els nodes adjacents a aquest trobat fins a arribar a aquest mateix i s'imprimeixen les ID's dels nodes acabant amb l'inicial.