```
#include "teseus.hpp"
using namespace std;

//IMPLEMENTACIÃM-^S DE LA CLASSE TESEUS.
/*

Per implementar la classe Teseus, ens hem basat en l'algorisme de recorregut d'a
mplada d'un graf (BFS).
Per fer-ho, hem creat un struct que tÃ© com a atributs una posiciÃ³ pos i un boo
leÃ  vis. El primer indica
la posiciÃ³ de la cambra que estem tractant; el segon si ja ha sigut visitada pr
Ã¨viament. Aquest struct
Ã©s un dels eixos principals de la classe, doncs ens permet mantenir gran part d
e la informaciÃ³ recollida
en un mateix array. Aquest array Ã©s el que anomenem arrayPos[] i on juntament a
mb l'array pred[],
de predecessors, desenvoluparem l'algorisme. A mÃ©s, cada vegada que volguem obt
enir les cambres successores
d'una altra cambra, crearem una llista local que les emmagatzemarÃ , serÃ  recor
reguda i posteriorment destruÃ¯da.
Aquesta llista de successors tindrÃ  sempre una mida mÃ xima de 4, una per paret
. Cada vegada que una cambra
sigui explorada, aquesta serÃ  marcada com a visitada, i posarem el seu booleÃ
de info a true. Un cop
trobada la posiciÃ³ destÃ-, si Ã©s que Ã©s accessible, nomÃ©s haurem de recÃ³rre
r l'array pred[] cap a enrrere,
fins arribar a la posiciÃ³ sentinella, la (0,0).

Vam decidir fer servir un struct perquÃ¨ Ã©s una estructura que ens permetria ac
cedir a la informaciÃ³ amb
un cost molt petit, doncs tenint la posiciÃ³ guardada podrÃ-em calcular la seva
ubicaciÃ³ dins de l'arrayPos[]
amb un senzill cÃ lcul, implementat al mÃ¨tode calculIndex(posicio pos, nat cols
). D'aquesta manera,
obtenir la ubicaciÃ³ de cada cambra tindria un cost constant, nomÃ©s requerim d'
accedir a l'atribut pos.
A mÃ©s, la creaciÃ³ de la llista de successors dintre del while ens permet estal
viar costos en memÃ²ria que
d'una altra manera tindriem si l'haguÃ¨ssim afegit dins de l'struct info, tenint
 una llista per cada cambra
del laberint, com tenÃ-em inicialment. Aquesta optimitzaciÃ³ ens ha permÃ©s redu
Ã¯r notablement els temps en
els diferents jocs de prova tant pÃºblics com propis.

L'Ãºs de les dues arrays, arrayPos[] i pred[], fan de matriu d'adjacÃ¨ncia, per
tant, ens fa tenir un cost
ÎM-^X(nÂ²) on n Ã©s el nÃºmero de posicions o total de cambres que tenim. Sent u
na mica mÃ©s pesat que una simple
llista d'adjacÃ¨ncia, que seria ÎM-^X(n+a). Ens vam decidir a fer servir la matr
iu per raons de senzillesa
en el codi i per temps, doncs implementar la llista ens portava a fer servir cue
s de prioritat i no
estÃ vem segurs de si ens donaria temps a fer-les bÃ© en el temps que quedava.

*/

struct info {
  posicio pos;
  bool vis;
};
```



```
nat calculIndex(posicio pos, nat cols);
//Pre: pos es una posiciÃ³ dintre de laberint, cols Ã©s el nÃºmero de columnes d
el laberint.
//Post: Retorna l'index en quÃ¨ es troba la cambra de posiciÃ³ pos.


void successors(std::list<posicio> &lsucc, info &in, const laberint & M, info ar
rayPos[]);
//Pre: in contÃ© informaciÃ³ sobre la cambra escollida, M Ã©s el laberint, array
Pos[] Ã©s no buit.
//Post: Adjunta a in els seus corresponents successors.


// Genera una llista de posicions que contÃ© el camÃ- mÃ-nim. El primer element
// d'aquesta llista serÃ  la posiciÃ³ inici, i l'Ãºltima la posiciÃ³ final.
// Una posiciÃ³ respecte a la seva anterior o posterior ha de ser consecutiva.
// Dues posicions sÃ³n consecutives si i nomÃ©s si la diferÃ¨ncia de les primere
s
// components de les posicions Ã©s en valor absolut 1 i les segones components
// sÃ³n iguals, o si la diferÃ¨ncia de les segones components Ã©s en valor
// absolut 1 i les primeres components sÃ³n iguals. Es produeix un error si no
// hi ha cap camÃ- que vagi des de la cambra inicial del laberint a la final,
// o si la cambra inicial o final no sÃ³n vÃ lides.
// Cost en temps: ÎM-^X((numFiles*numColumes)Â²).  Cost en espai: ÎM-^X((numFile
s*numColumes)Â²)
void teseus::buscar(const laberint & M, const posicio & inici, const posicio & f
inal,
                    std::list<posicio> & L) throw(error)
{

  // Comprovem si la posiciÃ³ Ã©s vÃ lida.
  if (inici.first < 1 or inici.first > M.num_files()
      or inici.second < 1 or inici.second > M.num_columnes())
    throw error (IniciFinalNoValid);

  if (final.first < 1 or final.first > M.num_files()
      or final.second < 1 or final.second > M.num_columnes())
    throw error (IniciFinalNoValid);


  cambra cinici = M(inici),cfi = M(final);
  if (inici != final) {

    if (not cinici.porta_oberta(paret::NORD) and not cinici.porta_oberta(paret::
SUD) and
        not cinici.porta_oberta(paret::EST) and not cinici.porta_oberta(paret::O
EST) )
      throw error (SenseSolucio);

    if (not cfi.porta_oberta(paret::NORD) and not cfi.porta_oberta(paret::SUD) a
nd
        not cfi.porta_oberta(paret::EST) and not cfi.porta_oberta(paret::OEST) )
      throw error (SenseSolucio);
  }

  nat files = M.num_files(), cols = M.num_columnes(), totalVert = files*cols;
  //Creem la matriu d'adjacÃ¨ncia.
  info arrayPos[totalVert];
  posicio pred[totalVert];

  nat k = 0;
  posicio senti(0,0);  // Sentinella
```

```
  for (nat i = 1; i <= M.num_files(); i++) {
    for (nat j = 1; j <= M.num_columnes(); j++, k++) {

      info in;
      posicio p(i,j);
      in.vis = false;
      in.pos = p;

      arrayPos[k] = in;
      pred[k] = senti;
    }
  }
  if (inici == final) {
    posicio res(arrayPos[calculIndex(inici,cols)].pos);
    L.push_back(res);
  } else {

    std::list<info> noVist;     // Lista dels nodes no explorats.

    // Marquem com a visitat el punt inicial.
    nat in = calculIndex(inici, cols);
    arrayPos[in].vis = true;
    noVist.push_back(arrayPos[in]);

    nat indexActual, indexSuc;
    bool cami = false;

    while (not noVist.empty() and not cami) {
      // Explotem el node.

      std::list<posicio> lsucc; //successors.
      successors(lsucc,noVist.front(),M, arrayPos);

      info actual =  noVist.front();
      noVist.pop_front();

      // Recorrem els successors del node explotat. Seran com a mÃ xim 4 iteraci
ons.
      for (list<posicio>::iterator it = lsucc.begin(); it != lsucc.end() and not
 cami; it++) {
        indexSuc = calculIndex(*it, cols);

        if (not (arrayPos[indexSuc].vis)) {
          indexActual = calculIndex(actual.pos,cols);
          arrayPos[indexSuc].vis = true;
          pred[indexSuc] = arrayPos[indexActual].pos;
          noVist.push_back(arrayPos[indexSuc]);

          if (arrayPos[indexSuc].pos == final) {
            cami =  true;
          }
        }
      }
    }

    if (not cami) throw error (SenseSolucio);

    // Prenem el camÃ- mÃ©s curt del que hem explorat.
    posicio previ = final;
    nat indexPrevi = calculIndex(previ,cols);
    L.push_back(previ);
```



```
    while (pred[indexPrevi] != senti) {
      indexPrevi = calculIndex(previ,cols);
      L.push_front(pred[indexPrevi]);
      previ = pred[indexPrevi];
    }
    //Eliminem la sentinella (0,0)
    L.pop_front();
  }
}


// Cost en temps: ÎM-^X(1).  Cost en espai: ÎM-^X(1)
// El cost temporal Ã©s ÎM-^X(1) ja que nomÃ©s es faran quatre consultes, en el
millor dels casos, i no
// afegim cap node. En el pitjor dels casos, nomÃ©s afegim quatre posicions a la
 llista de
// successors.
void successors(std::list<posicio> &lsucc, info &in, const laberint & M, info ar
rayPos[])
{

  cambra c = M(in.pos);
  if (c.porta_oberta(paret::NORD)) {

    posicio posNovaN(in.pos.first-1, in.pos.second);
    info infoNovaN  = arrayPos[calculIndex(posNovaN, M.num_columnes())];

    if (not infoNovaN.vis) {
      lsucc.push_back(posNovaN);
    }
  }

  if (c.porta_oberta(paret::SUD)) {

    posicio posNovaS(in.pos.first+1, in.pos.second);
    info infoNovaS  = arrayPos[calculIndex(posNovaS, M.num_columnes())];

    if (not infoNovaS.vis) {
      lsucc.push_back(posNovaS);
    }
  }

  if (c.porta_oberta(paret::EST)) {

    posicio posNovaE(in.pos.first, in.pos.second+1);
    info infoNovaE  = arrayPos[calculIndex(posNovaE, M.num_columnes())];

    if (not infoNovaE.vis) {
      lsucc.push_back(posNovaE);
    }
  }

  if (c.porta_oberta(paret::OEST)) {

    posicio posNovaO(in.pos.first, in.pos.second-1);
    info infoNovaO  = arrayPos[calculIndex(posNovaO, M.num_columnes())];

    if (not infoNovaO.vis) {
      lsucc.push_back(posNovaO);
    }
  }
}
```

```
// Cost en temps: ÎM-^X(1). Cost en espai: -
nat calculIndex(posicio pos, nat cols)
{

  nat index = cols*(pos.first-1)+(pos.second-1);
  return index;
}
```