Capítol 2 Tècniques de simulació discreta

En aquest capítol veurem com es fan simulacions de processos discrets que depenen d'algun paràmetre aleatori. Per donar la idea de com farem les simulacions, veurem primer un cas concret.

2.1 Un exemple senzill

Suposem que tenim una màquina amb quatre "peces" que cal canviar periòdicament, a causa del seu desgast. Aquesta màquina està sempre funcionant, i només es para per canviar alguna peça, quan s'espatlla.

Considereu ara una altra alternativa, motivada pel fet següent: com que per canviar una de les peces cal "obrir" la màquina, el temps necessari per canviar les quatre peces de cop és molt menor que quatre vegades el temps de canviar-ne una. Ens podem qüestionar, doncs, si és rentable l'estratègia següent: "cada cop que una peça s'espatlla, canviar totes quatre peces". La resposta depèn, evidentment, de quan costa cada peça, de quan perdem (o deixem de guanyar) si la màquina està parada, dels temps de substituir-ne una en relació al temps per substituir-les les quatre, etc.

Per tal de detallar la metodologia, donem valors a aquests paràmetres. Suposem que el temps de canviar una peça es pot modelar amb una llei normal de mitjana 15 minuts i variància 1 minut, i que el temps necessari per canviar-ne quatre es pot modelar per una normal de mitjana 25 minuts i variància 4 minuts. El temps de vida d'una peça el modelem per una llei exponencial de mitjana 500 minuts. Suposem que cada peça costa 50 (la unitat monetària és irrellevant), i que cada minut que la màquina resta aturada costa 10.

Volem avisar que aquest problema, a causa de la seva senzillesa, es pot resoldre analíticament (amb "paper i bolígraf").

Anem a veure com es pot organitzar una simulació d'aquest procés. En primer lloc, necessitem funcions que generin les quantitats aleatòries que ens calen. Anomenem `expo` la funció que genera exponencials de mitjana donada, i `normal` la que genera lleis normals amb mitjana i variància prefixades.

Noteu que, de fet, necessitem dos programes: un per simular la primera estratègia (canviar una peça cada cop) i avaluar-ne el cost, i un altre per a la segona estratègia.

2.1.1 Esdeveniments

Abans de detallar els organigrames, ens cal introduir alguns conceptes i definicions. Dividirem la simulació en una successió d'esdeveniments. Els esdeveniments són canvis en l'estat del sistema que estem simulant (en aquest cas, els esdeveniments són les avaries de les peces). Mentre no succeeix cap esdeveniment, el programa de simulació no ha de fer res. Quan hi ha un esdeveniment, el programa l'ha de "resoldre" (en aquest cas, ha de simular el reemplaçament de la peça espatllada). Per tant, orientarem el nostre programa d'acord amb aquests esdeveniments: per cada esdeveniment, farem les accions necessàries (essencialment, canviar la peça) i *calcularem quan passarà el següent esdeveniment.* Això ho podem fer perquè podem calcular el temps que duren les peces (exponencial de mitjana 500), i ens permetrà passar directament d'un esdeveniment al següent. Amb aquesta tècnica, aconseguirem que el programa sigui un bucle on, a cada passada, es va tractant cada esdeveniment. Noteu que, de fet, tenim quatre esdeveniments: les avaries de cada una de les peces, i que ens cal tenir aquests esdeveniments ordenats pel temps (hem de tractar-los en ordre cronològic).[1]

Un altre detall a remarcar és que els esdeveniments es componen de dues dades: el *que passa* (avaria, etc.) i *quan passa.* Aquest últim és l'hora en què l'esdeveniment succeeix.

Noteu que tota simulació té dos esdeveniments més: l'hora d'inici de la simulació i l'hora d'acabar-la. En l'exemple que estem tractant, l'hora d'inici correspon a engegar la màquina, i l'hora d'acabar-la pot ser o bé l'hora a la qual es para la màquina o bé l'hora a la qual nosaltres decidim que ja hem simulat prou. En aquest cas concret suposem que la màquina no para mai. Agafarem un temps de simulació de, per exemple, 100 hores.

2.1.2 Gestió dels esdeveniments

Per simplificar el programa, una bona opció és escriure unes funcions que s'encarreguin d'emmagatzemar de manera ordenada els esdeveniments futurs de la simulació. Aquest conjunt de funcions s'anomena AGENDA.

La agenda consta, essencialment, de dues funcions: una que introdueix esdeveniments (que anomenarem `posa_agenda`, i una altra que els treu de manera ordenada segons el temps (que anomenarem `treu_agenda`). Quan es posa un esdeveniment a l'agenda, aquest s'afegeix a la llista d'esdeveniments que ja hi ha a l'agenda. Quan es treu un esdeveniment, aquest s'esborra de l'agenda.

Més endavant (en la secció 2.1.4) veurem una implementació en C d'una agenda per a aquest cas concret.

2.1.3 Organigrama

Considerem el cas del primer programa (i.e., només canviem les peces espatllades). Per uniformitzar el programa, comptarem el temps en minuts (és a dir, la durada de la simulació serà 100 hores $\equiv$ 6000 minuts),

L'organigrama de la simulació podria ser el següent:

Pas 1. Posem a l'agenda l'esdeveniment `INICI` a temps 0, que correspon a engegar la màquina amb les quatre peces noves.

[1] En aquest exemple concret es poden fer alguns trucs per estalviar-se l'ordre dels quatre esdeveniments. Nosaltres no considerarem aquests tipus d'optimitzacions perquè no són aplicables en general, i el nostre propòsit és descriure una metodologia aplicable al major nombre possible de casos.

Pas 2. Posem a l'agenda l'esdeveniment `FINAL` a temps 6000, que correspon a acabar la simulació.

Pas 3. Treiem de l'agenda el proper esdeveniment.

Pas 4. Si l'esdeveniment és `INICI`, calculem el temps de vida de les quatre peces, i guardem els quatre esdeveniments (de tipus `AVARIA`) a l'agenda, cadascun amb el seu temps.

Pas 5. Si l'esdeveniment és `AVARIA` a temps `t0`, calculem el temps `tc` necessari per canviar la peça segons una normal de variància 1 minut i mitjana 15 minuts, i calculem el temps de vida (`t1`) de la nova peça segons una exponencial de mitjana 500. Posem a l'agenda l'esdeveniment `AVARIA` a temps `t0+tc+t1`.

Pas 6. Si l'esdeveniment és `FINAL`, la simulació ha acabat.

Pas 7. Anar al Pas 3.

Comentem-lo una mica. El Pas 1 és per inicialitzar el bucle principal del programa. El Pas 2 és per fixar el temps al qual pararem la simulació. El Pas 4 només s'executa un cop, al principi del programa. Arriben així al pas interessant, que és el 5. Aquest pas és el nucli de tota la simulació. Si de l'agenda ha sortit un esdeveniment `AVARIA`, mirem a quin temps passa aquesta avaria (això també ho diu l'agenda). Anomenem aquest temps `t0`. A continuació obtenim el temps necessari `tc` per canviar la peça i també la seva durada `t1`. Per tant, la peça que s'acaba d'installar fallarà a temps `t0+tc+t1`, i per aquest motiu introduïm a l'agenda una avaria a aquesta hora.

Creiem que ara el funcionament de l'algorisme queda clar. Volem fer notar que a l'agenda només hi guardem, per cada peça, el següent temps en què fallarà. No hi guardem tots els temps de fallada. Això fa que sols estem guardant cinc esdeveniments: el `FINAL` i els quatre `AVARIA`.

2.1.4 Implementació en C

Veiem ara com seria una primera versió en C de l'algorisme que acabem de descriure.

L'agenda

En primer lloc, ens cal decidir com guardarem els esdeveniments. Donat que cada esdeveniment consta de dues dades (l'hora i el tipus), usarem una estructura. Per comoditat, podem definir un nou tipus de variable, que podem anomenar `esdev`, que contingui una variable de tipus `float` per al temps, i una variable de tipus `int` per al tipus d'esdeveniment.

Llavors, una implementació per a l'agenda pot ser la següent:

```
#include <stdio.h>
#include <stdlib.h>

#define N 5

typedef struct
{
   float quan;
```

```
   int que;
} esdev;

esdev agenda[N];
int ara=-1;

void posa_agenda(esdev e)
{
   int i;
   ++ara;
   if (ara == N) {puts("error. agenda plena."); exit(1);}
   for (i=ara; i>0; i--)
   {
      if (e.quan <= (agenda[i-1]).quan) break;
      agenda[i]=agenda[i-1];
   }
   agenda[i]=e;
}
int treu_agenda(esdev *e)
{
   if (ara == -1) return (0); /* agenda buida */
   *e=agenda[ara];
   --ara;
   return (1);
}
```

Observeu que hem creat un vector d'esdeveniments (anomenat agenda) de 5 components (ja hem comentat que, en aquest cas, cinc és el màxim nombre d'esdeveniments que tindrem). Aquest vector és global: l'hem declarat fora de qualsevol funció, de manera que és visible des de tot el fitxer. També hem creat una variable entera anomenada ara, i l'hem fet global. Indicarà l'última component plena del vector agenda, i ens servirà per controlar els esdeveniments que hi tenim.

Veiem què fan les funcions. La funció posa_agenda comprova, en primer lloc, si el vector agenda està ple. En aquest cas, la simulació no pot continuar (ens cal guardar un esdeveniment que no podem guardar), i el programa es para. Si això passés en aquest exemple, voldria dir que tenim un error de programació en alguna banda del programa, perquè ja hem dit que 5 esdeveniments són suficients. Si el vector agenda no és ple, llavors es posa aquest esdeveniment dins l'agenda, però de manera ordenada: volem que l'esdeveniment més distant en el temps (el que trigarà més temps a passar) estigui a la component 0 del vector, i que el més proper en el temps (el següent que passarà) estigui a la component més alta. Per això fem una cerca pel vector i insertem l'esdeveniment e allà on toqui. Els detalls d'aquesta ordenació no són difícils, i els deixem per al lector.

La funció treu_agenda retorna el proper esdeveniment de la simulació. Com que la rutina posa_agenda ja ha col·locat els esdeveniments de manera ordenada, aquesta funció és molt senzilla: només ha d'agafar la component ara de l'agenda i retornar-la. En cas que l'agenda fos buida, la funció retorna un zero (altrament, retorna un 1). Noteu que, a diferència de la

rutina posa_agenda, no hem de parar la simulació: el programa pot continuar executant-se i el programa principal ja decidirà si el fet que l'agenda estigui buida és un error o no (més endavant veurem exemples on no és cap error que l'agenda es quedi buida a mitja simulació).

El programa principal

Una versió del programa principal és la següent:

```
#include <math.h>
#include <stdio.h>

#define   INICI  1
#define  AVARIA  2
#define   FINAL  3

void main(void)
{
   typedef struct
   {
      float quan;
      int que;
   } esdev;
   float normal(long int *ap_llavor);
   float expo(float m, long int *ap_llavor);
   void posa_agenda(esdev e);
   int treu_agenda(esdev *e);
   long int llavor;
   float t,cost;
   int fi,j;
   esdev e;
   llavor=-1;
   fi=0;
   e.que=INICI;
   e.quan=0.0;
   posa_agenda(e);
   e.que=FINAL;
   e.quan=6000.0;
   posa_agenda(e);
   cost=0.0;
   while (fi == 0)
   {
      treu_agenda(&e);
      switch (e.que)
      {
         case INICI:
            e.que=AVARIA;
            for (j=0; j<4; j++)
```

```
            {
               e.quan=expo(500,&llavor);
               posa_agenda(e);
            }
            break;
         case AVARIA:
            t=15+normal(&llavor);
            cost += 50+10*t;
            e.quan += t+expo(500,&llavor);
            posa_agenda(e);
            break;
         case FINAL:
            fi=1;
            break;
         default:
            puts("error: esdeveniment desconegut.");
      }
   }
   printf("cost total: %f\n",cost);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
```

Comentem una mica el seu funcionament. En primer lloc, omplim l'agenda amb els dos esdeveniments que coneixem (l'hora d'inici i acabament de la simulació) i inicialitzem la variable `cost` a zero (aquí hi guardarem el cost de l'estratègia que estem simulant).

A continuació ve el bucle principal del programa: es va cridant l'agenda i, per cada esdeveniment que passa anem fent les accions oportunes. Si l'esdeveniment és `INICI`, calculem el temps d'avaria de les 4 peces i els posem a l'agenda. Si és `AVARIA`, calculem el temps usat per canviar la peça, que ens dóna el cost de l'avaria. A continuació calculem el temps de vida de la nova peça amb el qual podem saber l'hora a què s'espatllarà (hora de l'avaria actual + temps de canviar la peça + temps de vida de la nova peça). Per tant, posem aquest esdeveniment a l'agenda. Si l'esdeveniment és `FINAL`, parem la simulació (tot i que encara queden esdeveniments a l'agenda).

Per simplificar una mica el programa, la generació del temps de vida de les peces s'ha posat a part al final, en forma de funció anomenada **expo**.

2.1.5 Comentaris

El propòsit d'aquesta secció ha estat donar una visió general del tipus de tècniques que usarem per simular. Volem subratllar com el fet d'introduir l'agenda ha simplificat la simulació, de manera que el programa principal es redueix a un bucle senzill. Per altra banda, l'agenda tampoc és gens complexa. El gran avantatge és en el fet que, per a moltes simulacions, es pot usar la mateixa agenda (o amb canvis mínims). Es pot dir que l'agenda és el que tenen en comú

totes (bé, gairebé totes) les simulacions. Per tant, és una bona política separar l'agenda de la resta i programar-la de manera general. Això ens simplificarà molt l'escriure els programes corresponents.

En les següents seccions donarem algunes (petites) modificacions a l'agenda que hem vist aquí per fer-la el més general possible. També veurem la manera de fer un grup de rutines (amb la mateixa filosofia de l'agenda) per gestionar cues. Totes aquestes rutines ens seran molt útils en les següents seccions, en les quals construirem simuladors per a alguns problemes concrets.

2.2 Gestió d'agendes

Ja hem explicat el funcionament i propòsit de l'agenda. Ara donarem només unes petites modificacions per fer-la una mica més general.

Un inconvenient que té l'agenda que hem vist és el següent: suposem que volem fer moltes simulacions (amb llavors diferents) del problema de la secció anterior, amb la idea de veure la mitjana i la variància dels resultats. Una manera simple seria posar un bucle més en el programa principal i fer variar la llavor, i anar escrivint els diferents resultats. L'inconvenient és que, després d'haver fet una simulació, ens poden quedar esdeveniments a l'agenda que impedeixen tornar-la a usar (per exemple, en l'exemple de la màquina amb quatre peces ens queden a l'agenda quatre esdeveniments de tipus `AVARIA`). Per tant, afegirem a l'agenda una rutina que s'encarregui de buidar-la.

Una segona versió de l'agenda podria ser la següent:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
   float quan;
   int que;
} esdev;

static esdev *agenda;
static int n_esd,ara;

void ini_agenda(int n)
{
   n_esd=n;
   agenda=(esdev*)malloc(n_esd*sizeof(esdev));
   if (agenda == NULL) {puts("agenda: falta memoria."); exit(1);}
   ara=-1;
}
void posa_agenda(esdev e)
{
   int i;
   ++ara;
   if (ara == n_esd) {puts("error. agenda plena."); exit(1);}
   for (i=ara; i>0; i--)
```

```
    {
        if (e.quan <= (agenda[i-1]).quan) break;
        agenda[i]=agenda[i-1];
    }
    agenda[i]=e;
}
int treu_agenda(esdev *e)
{
    if (ara == -1) return (0); /* agenda buida */
    *e=agenda[ara];
    --ara;
    return (1);
}
void buida_agenda(void)
{
    ara=-1;
}
void llibera_agenda(void)
{
    free(agenda);
}
```

Com podeu veure, els canvis són mínims: hem afegit la rutina `ini_agenda` per poder crear una agenda de qualsevol dimensió des del programa principal. La rutina `buida_agenda` permet posar a zero l'agenda de cara a fer una altra simulació. Finalment, la rutina `llibera_agenda` serveix per alliberar la memòria reservada per `ini_agenda`. Aquesta rutina ens caldrà si volem, dins d'un mateix programa, fer dues simulacions que requereixin agendes de diferent longitud, sense haver de dimensionar sempre a la longitud de l'agenda més llarga.

Més endavant veurem exemples que usen aquesta agenda.

2.3 Gestió de cues

Suposem ara que volem fer una simulació que ens requereix gestionar una cua (per exemple, els caixers d'un banc o supermercat). Ens seria molt útil disposar d'un conjunt de rutines que s'encarreguessin de fer-ho: voldríem rutines que posessin gent a la cua, que ens traiessin el següent de la cua, que ens diguessin quan llarga és la cua, etc. A més, seria molt pràctic que aquestes rutines admetessin la possibilitat de gestionar més d'una cua, ja que hi ha molts casos en què això es requereix. Finalment, voldríem que fos el més independent possible del cas concret que estem simulant, per poder-lo usar en molts problemes diferents.

En aquesta secció veurem maneres d'implementar gestors de cues per satisfer les demandes del paràgraf anterior. Per simplificar la discussió, veurem primer un cas senzill, en què només tenim una cua. Més endavant veurem casos més complexos, on hi ha diverses cues amb règims de funcionament diferent.

2.3.1 Una única cua

Suposem que volem simular l'evolució de les cues que es formen davant d'un caixer automàtic (de moment, suposarem que només n'hi ha un). Estem interessats a obtenir informació com: longitud màxima de la cua, temps màxim i temps mitjà d'espera dels clients, etc. Suposarem que el temps que passa entre l'arribada de dos clients segueix una llei exponencial de 2 minuts de mitjana, i que el temps que passa cada client davant del caixer és de la forma $1 + t$, on t segueix una exponencial d'1 minut de mitjana.

Primera versió

Per a aquesta simulació usarem l'agenda que hem vist a la secció 2.2. A més, ens anirà molt bé disposar d'un joc de funcions que s'encarreguin de gestionar la cua (de la mateixa manera que ho hem fet amb l'agenda). Una possible implementació d'aquest gestor de cues (per al cas d'una única cua) podria ser la que donem tot seguit:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
   float tar;
} el_cua;

static el_cua *cua;
static int max_cua,ini_cua,fin_cua,lon_cua;

void crea_cua(int l)
{
   max_cua=l;
   cua=(el_cua*)malloc(max_cua*sizeof(el_cua));
   if (cua == NULL) {puts("ini_cua: error 1."); exit(1);}
   ini_cua=0;
   fin_cua=0;
   lon_cua=0;
}
int posa_cua(el_cua c)
{
   if (lon_cua == max_cua) return(0);
   ++lon_cua;
   cua[fin_cua]=c;
   ++fin_cua;
   if (fin_cua == max_cua) fin_cua=0;
   return(1);
}
int treu_cua(el_cua *c)
{
   if (lon_cua == 0) return(0);
```

```
    --lon_cua;
    *c=cua[ini_cua];
    ++ini_cua;
    if (ini_cua == max_cua) ini_cua=0;
    return(1);
}
int long_cua(void)
{
    return(lon_cua);
}
void elim_cua(void)
{
    free(cua);
}
```

Comentem el seu funcionament. En primer lloc, definim una estructura anomenada `el_cua` que representa la persona que està fent cua. En aquesta estructura guardem l'hora a la qual la persona ha arribat a la cua (`tar`). Amb això, quan la persona deixi la cua per anar al caixer, podrem saber quant temps ha estat fent cua. El fet d'usar una estructura (en lloc d'una simple variable `float`) permet afegir informació addicional sobre la persona que fa cua. Per exemple, si la cua correspon al caixer d'un supermercat, podrem afegir un nou membre amb el nombre de productes que porta el client, amb el propòsit de calcular quant de temps estarà pagant (el temps de pagament depèn del nombre de productes). Noteu que si afegim més membres a l'estructura `el_cua` no cal modificar el gestor de cues, ja que aquests membres no apareixen enlloc del programa.

Cua circular

A continuació tenim la declaració de l'apuntador `cua`. L'usarem per crear un vector d'elements de tipus `el_cua`, que contindrà la cua que simulem. De moment, usarem una cua de tipus circular:[2] una cua circular és un vector (on guardarem la cua) amb dos índexs que indiquen la primera i l'última component de la cua. Inicialment el vector es buit i els dos índexs (els podem dir `ini_cua` i `fin_cua`, com en el programa) valen zero. Quan arriben els primers clients a la cua, el vector es va omplint i l'índex que indica el seu final es va incrementant de manera adient (és a dir, `ini_cua` continua valent zero, pero `fin_cua` ja no). Quan algú ha de sortir de la cua (naturalment, el primer que surt és el primer que hi ha arribat), l'índex `ini_cua` ens diu quin és el primer que ha de sortir. Per tant, traiem aquest client i incrementem el valor de `ini_cua`. Amb aquest procediment, la cua es "desplaça" al llarg del vector on la guardem. Evidentment, el més natural és que en algun moment l'índex `fin_cua` arribi a l'última component del vector sense que la cua estigui plena (segurament, alguns clients hauran sortit de la cua i `ini_cua` tindrà un cert valor positiu). En aquest moment, si cal afegir un altre client a la cua, "donarem la volta" al vector (com si la primera i l'última component estiguessin connectades) i posarem aquest nou client a la component zero del vector, fent que `fin_cua` valgui també zero. Per tant, la condició de cua plena no serà que `fin_cua` arribi al final del vector, sinó que "atrapi" a `ini_cua`. Naturalment, quan `ini_cua` arribi al final del vector `cua`, també li "donarem la volta" i el farem tornar a començar per zero.

[2] A la secció 2.3.4 veurem un altre sistema per gestionar la cua.

Noteu que l'avantatge d'aquest esquema és la seva rapidesa. Tant posar un client a la cua com treure'l necessita molt poques operacions, que a més són fàcils de programar. Els seu principal inconvenient és que cal saber, al principi de la simulació, quant llarg ha de ser el vector `cua`. De moment, per mantenir la simplicitat dels programes, ens acontentarem a donar "a ull" la longitud màxima d'aquesta cua. En la secció 2.3.4 veurem una altra tècnica de gestió de cues que no presenta aquest inconvenient.

Finalment, hem afegit un altre índex (`lon_cua`) que conté el nombre de persones que hi ha a la cua. Estrictament parlant aquest índex és innecessari, ja que el seu valor pot ser deduït a partir dels valors de `ini_cua` i `fin_cua`. La raó d'incloure'l ha estat que, en la nostra opinió, el programa queda una mica més clar.

Les funcions

La rutina `crea_cua` dimensiona el vector `cua` a la longitud demanada. Al mateix temps inicialitza els diversos índexs del programa: a més dels que acabem de comentar, tenim l'índex `max_cua`, que conté la longitud del vector cua (correspon al nombre màxim de persones que podem tenir fent cua). Naturalment, cal cridar aquesta rutina abans d'usar qualsevol de les funcions que gestionen la cua.

La funció `posa_cua` serveix per afegir un nou client a la cua. El seu funcionament és molt senzill (tenint en compte el que s'ha dit de cues circulars): primer verifiquem que la cua no estigui plena, després posem el client a la component indicada per `fin_cua`, i incrementem aquest índex. Si, en fer això, `fin_cua` és més gran que `max_cua`, vol dir que hem de "donar la volta" al vector i per tant posem `fin_cua` a zero. El valor retornat per la funció és 1 si tot ha anat bé, i 0 si la cua era plena i no hem pogut afegir el client. Noteu que, en aquest últim cas, ens caldrà parar la simulació, ja que hem arribat a una situació "irresoluble": no podem guardar el client (no tenim espai a la cua) i tampoc podem ignorar-lo (la simulació perdria sentit). Generalment, cal tornar a començar amb un vector cua més llarg.

La funció `treu_cua` ens retorna el primer client de la cua. Cal passar-li una estructura de tipus `el_cua` per adreça i ens retornarà el primer client que surt de la cua dins aquesta estructura. Creiem que el seu funcionament hauria de quedar clar amb les explicacions anteriors i, per tant, no el comentem. Només direm que la funció retorna un 0 si la cua es buida i un 1 en cas contrari. Noteu que, a diferència de la funció `posa_cua`, si la cua és buida no tenim cap problema.

Per acabar, tenim les funcions `long_cua` i `elim_cua`. La funció `long_cua` retorna la llargada de la cua, i la funció `elim_cua` allibera la memòria reservada per `crea_cua` al crear la cua.

El programa principal

Veiem ara com seria un programa principal per fer la simulació que estem comentant.

```
#include <math.h>
#include <stdio.h>

#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4
```

```
void main(void)
{
   typedef struct
   {
      float quan;
      int que;
   } esdev;
   typedef struct
   {
      float tar;
   } el_cua;
   void ini_agenda(int n);
   void posa_agenda(esdev e);
   int treu_agenda(esdev *e);
   void buida_agenda(void);
   void crea_cua(int l);
   int posa_cua(el_cua c);
   int treu_cua(el_cua *c);
   int long_cua(void);
   void elim_cua(void);
   float expo(float m, long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmax;
   long int llavor;
   int bn,caixa,j,nca;
   ini_agenda(3);
   crea_cua(100);
   llavor=-1995;
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   caixa=0;
   nca=0;
   tmax=0.0;
   while (treu_agenda(&e) != 0)
   {
      switch (e.que)
      {
         case OBRIR:
            bn=1;
            caixa=0;
```

```
            e.que=ARRIBADA;
            e.quan=expo(2,&llavor);
            posa_agenda(e);
            break;
         case ARRIBADA:
            if (bn == 1)
            {
               t=e.quan;
               if (caixa == 0)
               {
                  caixa=1;
                  e.quan += 1+expo(1,&llavor);
                  e.que=SORTIDA;
                  posa_agenda(e);
               }
               else
               {
                  c.tar=t;
                  j=posa_cua(c);
                  if (j == 0) {puts("error: cua massa petita"); exit(1);}
               }
               e.quan=t+expo(2,&llavor);
               e.que=ARRIBADA;
               posa_agenda(e);
            }
            break;
         case SORTIDA:
            ++nca;
            j=treu_cua(&c);
            if (j != 0)
{
               t=e.quan-c.tar;
               if (t > tmax) tmax=t;
               e.quan += 1+expo(1,&llavor);
               e.que=SORTIDA;
               posa_agenda(e);
}
            else
            {
               caixa=0;
            }
            break;
         case TANCAR:
            bn=0;
            break;
         default:
            puts("error: esdeveniment desconegut.");
```

```
        }
    }
    printf("nombre de clients atesos: %d\n",nca);
    printf("temps maxim de cua: %f\n",tmax);
}
float expo(float m, long int *ap_llavor)
{
    float alea1(long int *ap_llavor);
    return(-m*log(alea1(ap_llavor)));
}
```

Aquest programa utilitza l'agenda de la secció 2.2, el gestor de cues (secció 2.3.1) i la funció `alea1`. El seu funcionament és molt similar al del programa de la secció 2.1.4: en primer lloc definim els tipus d'esdeveniments que tindrem en la simulació: `OBRIR` (posar en marxa el caixer, correspon a l'inici de la simulació), `ARRIBADA` (arriba un client), `SORTIDA` (un client se'n va, després d'haver estat atès) i `TANCAR` (parem el caixer). Naturalment, ens cal definir els tipus `esdev` i `el_cua`.

El programa comença inicialitzant l'agenda (noteu que només hi guardarem 3 esdeveniments alhora), la cua (el valor 100 l'hem posat "a ull"; si és massa petit, el programa ja avisarà) i la llavor per als nombres aleatoris. Després posem els esdeveniments `OBRIR` i `TANCAR` a l'agenda, que indicaran el principi i la fi de la simulació.[3] La variable `bn` és una bandera que usarem per saber si el caixer és obert (valor 1) o no (valor 0). La raó d'incloure aquesta bandera és que, per tancar el caixer, el que farem serà tancar les portes d'accés (per no acceptar més clients) i deixar que acabin les persones que fan cua. Per aquest motiu, la simulació no acabarà amb l'esdeveniment `TANCAR`, sinó que continuarà fins que no quedi ningú per atendre. La bandera `bn` ens indicarà, doncs, si hem d'acceptar arribades o no (tota arribada d'un client quan `bn` sigui 0 no ha de ser considerada). Una altra variable auxiliar és `caixa`, que indica si el caixer és ocupat (val 1) o no (val 0). L'usarem per saber si, quan arriba un client, l'hem de posar al caixer (ho farem si `caixa` val 0) o a la cua (si `caixa` val 1). A més, el programa usa altres variables auxiliars, com `nca` i `tmax`, que usarem per acumular el nombre de clients atesos i el temps màxim que una persona ha estat fent cua.

A continuació ve el bucle principal de la simulació: es van traient esdeveniments de l'agenda i es prenen les corresponents accions. Si l'esdeveniment és `OBRIR`, les accions que cal fer són: posar `bn` a 1, `caixa` a 0, generar l'arribada del primer client i posar-lo a l'agenda.

Si l'esdevenimemt és `ARRIBADA`, mirem si les portes són obertes (és a dir, si `bn` és igual a 1), perquè només llavors tindrem en compte l'arribada. Si aquest és el cas, si el caixer és buit posem la persona al caixer (fem `caixa=1`), calculem el temps que hi serà, i posem a l'agenda un esdeveniment `SORTIDA` a l'hora que marxarà (hora d'arribada + temps al caixer). Si quan arriba el client el caixer està ocupat, el posem a la cua. Noteu que cal guardar l'hora a què ha arribat, perquè quan surti de la cua voldrem saber quan temps hi ha estat. Finalment, i tant si l'hem posat al caixer com a la cua, generem l'arribada del següent client. Noteu que això només es pot fer en aquest punt del programa: com que el que sabem és la llei que segueix el temps entre arribades de clients, quan tenim l'arribada d'un client podem generar l'arribada del següent.

[3] De moment, hem posat un temps de funcionament de 12 hores. Evidentment, no hi ha cap problema a posar un altre valor.

El tractament de l'esdeveniment `SORTIDA` és com segueix: en primer lloc, incrementem en 1 el nombre de clients atesos i mirem si hi ha un client a la cua. Si hi és, mirem a quina hora aquest client ha començat la cua i deduïm el temps que hi ha estat. Això ens serveix per actualitzar la variable que conté el temps màxim que algú ha estat en cua. A continuació, calculem el temps que aquest client estarà ocupant el caixer i posem el corresponent esdeveniment `SORTIDA` a l'agenda. Naturalment, si quan el client se'n va no hi ha ningú fent cua, només cal indicar que el caixer queda buit (`caixa=0`).

Per acabar, si l'esdeveniment és `TANCAR`, posem la bandera `bn` a zero. L'últim `default` és de control: no pot aparèixer cap esdeveniment que no sigui un d'aquests quatre. Si això succeeix, vol dir que hi ha un error de programació.

Deixem com a exercici per al lector modificar aquest programa per mesurar altres paràmetres, com el temps mitja de cua, el percentatge de gent que no ha fet cua, la longitud màxima de la cua, l'evolució de la longitud de la cua respecte del temps, etc. Noteu que per fer això no cal alterar l'algorisme de simulació, només cal afegir (als llocs adequats) variables que "comptin" el que volem observar.

Segona versió

Suposem ara que volem modificar el programa per estudiar una variant d'aquest problema. La variant és la següent: tenim l'oportunitat de comprar caixers d'un nou tipus més ràpid, cosa que implica que els temps de cua seran menors. Volem quantificar aquesta reducció del temps de cua, per decidir si val la pena fer la inversió o no. Per ser precisos, suposem que el temps de servei amb els nous caixers és 45 segons més una exponencial de mitjana 1 minut. Noteu que no costa gens modificar el programa anterior per simular aquest cas: només cal canviar el lloc on es genera el temps d'estada davant el caixer.

Moltes vegades estem interessats a fer comparacions sota les mateixes condicions: en el cas que estem tractant aquí, volem saber la reducció de cua deguda als nous caixers. Si per cada un dels dos casos usem arribades diferents de clients, podria ser que la reducció de cua observada fos deguda a un cert tipus d'arribada de clients que, per casualitat, s'ha produït en un dels casos. Cal dir també que, per obtenir resultats fiables, cal fer moltes simulacions i observar mitjana, variància, etc. dels resultats. Si tenim la mateixa arribada de clients en els dos casos, la convergència de les simulacions és més ràpida que si aquestes arribades són diferents. Quan l'arribada de clients és igual en els dos casos es poden aplicar tècniques de reducció de variància (vegeu, per exemple, [2] o [17]) que permeten obtenir resultats més acurats. Nosaltres no tractarem aquestes tècniques aquí, ja que són més apropiades per a un curs d'estadística. Com ja hem dit, el nostre únic propòsit és donar les eines necessàries per poder fer les simulacions.

Observeu que si modifiquem el programa de la manera que hem dit abans (simplement canviant la generació del temps de servei al caixer) *estem modificant l'arribada de clients*, tot i que usem la mateixa llavor per a les dues simulacions. La raó és que per tenir la mateixa arribada de clients (portant cada client la mateixa "feina" per fer al caixer) cal que es mantingui l'ordre de les crides a les rutines de nombres aleatoris. Si canviem els temps d'estada al caixer, pot passar que un client que abandonava el caixer després d'una certa arribada ara ho faci abans, per tant els nombres aleatoris (de `aleal`) intercanvien els papers. El nombre aleatori que abans s'usava per calcular una nova arribada, ara es fa servir per calcular el temps d'estada al caixer de la següent persona de la cua.

Anem ara a donar una nova versió del programa, modificada per permetre fer les dues simulacions amb la mateixa arribada de clients.

```
#include <math.h>
#include <stdio.h>

#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4

void main(void)
{
   typedef struct
   {
      float quan;
      int que;
   } esdev;
   typedef struct
   {
      float tar;
      float tse;
   } el_cua;
   void ini_agenda(int n);
   void posa_agenda(esdev e);
   int treu_agenda(esdev *e);
   void buida_agenda(void);
   void crea_cua(int l);
   int posa_cua(el_cua c);
   int treu_cua(el_cua *c);
   int long_cua(void);
   void elim_cua(void);
   float expo(float m, long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmax,tserv;
   long int llavor;
   int bn,caixa,j,nca;
   ini_agenda(3);
   crea_cua(100);
   llavor=-1995;
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   caixa=0;
   nca=0;
```

```
tmax=0.0;
while (treu_agenda(&e) != 0)
{
   switch (e.que)
   {
      case OBRIR:
         bn=1;
         caixa=0;
         e.que=ARRIBADA;
         e.quan=expo(2,&llavor);
         posa_agenda(e);
         tserv=1+expo(1,&llavor);
         break;
      case ARRIBADA:
         if (bn == 1)
         {
            t=e.quan;
            if (caixa == 0)
            {
               caixa=1;
               e.quan += tserv;
               e.que=SORTIDA;
               posa_agenda(e);
            }
            else
            {
               c.tar=t;
               c.tse=tserv;
               j=posa_cua(c);
               if (j == 0) {puts("error: cua massa petita"); exit(1);}
            }
            e.quan=t+expo(2,&llavor);
            e.que=ARRIBADA;
            posa_agenda(e);
            tserv=1+expo(1,&llavor);
         }
         break;
      case SORTIDA:
         ++nca;
         j=treu_cua(&c);
         if (j != 0)
{
            t=e.quan-c.tar;
            if (t > tmax) tmax=t;
            e.quan += c.tse;
            e.que=SORTIDA;
            posa_agenda(e);
```

```
    }
            else
            {
               caixa=0;
            }
            break;
         case TANCAR:
            bn=0;
            break;
         default:
            puts("error: esdeveniment desconegut.");
      }
   }
   printf("nombre de clients atesos: %d\n",nca);
   printf("temps maxim de cua: %f\n",tmax);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
```

Abans de comentar aquesta nova implementació, creiem que val la pena explicar la idea de l'algorisme. El problema de la versió anterior venia de generar el temps d'estada al caixer en el moment que el client l'ocupava. Si generem el temps d'estada al caixer en el moment que la persona arriba al sistema, aquest problema desapareix: per cada `ARRIBADA`, calculem el temps que passarà al caixer i calculem l'hora d'arribada del client següent. D'aquesta manera, els nombres aleatòris s'usen amb el mateix ordre: el primer s'usa per calcular l'arribada del primer client, el segon per calcular el seu temps de servei al caixer, el tercer per a l'arribada del segon client, el quart pel seu temps de servei, etc. Si alterem la llei que regeix el temps de servei al caixer, les hores d'arribada no varien.

Aquest mètode ens obliga a "apuntar", per a cada client que és en cua, el seu temps de servei prèviament calculat. Una manera fàcil de fer-ho és afegir un nou camp (diem-li `tse`, per temps de servei) a l'estructura `el_cua`. El fet d'afegir aquest camp pràcticament no altera el gestor de cues: només cal modificar la declaració de `el_cua` a la capçalera del programa perquè inclogui el nou camp `tse`. També necessitem una variable auxiliar (li hem dit `tserv`) per apuntar el temps de servei de la propera `ARRIBADA`: el fet de generar el temps de servei juntament amb l'hora d'arribada provoca que haguem de guardar aquest temps de servei fins que es produeix de manera efectiva aquesta arribada.[4] En aquest moment posem el client al caixer o a la cua (amb el temps de servei `tserv`) i generem l'arribada del següent i el seu temps de servei (que guardem a `tserv`).

Les modificacions del programa són molt poques. Essencialment es tracta d'alterar la gestió de l'esdeveniment `ARRIBADA` de manera que s'encarregui també de la generació del temps d'arribada. Finalment recordem que, per fer funcionar el programa, cal usar el gestor de cues amb la modificació d'incloure el camp `tse` a l'estructura `el_cua`.

[4]Una altra possibilitat és afegir un nou camp a l'estructura esdev per guardar-hi aquest valor `tserv`. Naturalment, aquest nou camp només l'usariem en cas que l'esdeveniment fos una `ARRIBADA`.

2.3.2 Més d'una cua: cues del mateix tipus

Ara volem estendre la simulació anterior al cas en què tenim més d'un caixer, amb una cua davant de cada caixer. Estem interessats a saber com depèn la longitud i el temps mitjà d'espera (a la cua) del nombre de caixers que tenim installats.

L'agenda

En aquesta simulació, el nombre d'esdeveniments possibles no està determinat d'entrada: a més dels `OBRIR`, `TANCAR` i `ARRIBADA`, tenim un esdeveniment del tipus `SORTIDA` per cada un dels caixers. Aquests han de ser diferents, perquè per saber de quina cua hem de treure el client que posarem al caixer cal saber en quin caixer s'ha produït la `SORTIDA`.

Si el nombre de caixers és molt reduït, una solució és distinguir les diferents sortides: per exemple, `SORTIDA1` indicaria una sortida al primer caixer, `SORTIDA2` al segon, etc. Evidentment, aquesta solució és impracticable si el nombre de caixers és gran. Una solució és que "apuntem", per cada esdeveniment `SORTIDA`, el caixer en què s'ha produït. És molt similar a allò que hem fet a la secció anterior amb la cua, però en aquest cas caldrà fer-ho a l'agenda.

El que farem, doncs, serà afegir el camp `on` (de tipus `int`) a l'estructura `esdev`. Noteu que aquest canvi no repercuteix en el funcionament de l'agenda que hem vist a l'apartat 2.2. Aquest nou camp només l'usarem en el cas que l'esdeveniment sigui `SORTIDA`, per apuntar-hi el número de caixer en què aquesta s'ha produït.

Per simplificar la declaració de les variables, hem creat el fitxer `agenda.h`, que conté les declaracions del tipus `esdev` i les funcions de l'agenda:

```
typedef struct
{
   float quan;
   int que;
   int on;
} esdev;

void ini_agenda(int n);
void posa_agenda(esdev e);
int treu_agenda(esdev *e);
void buida_agenda(void);
void llibera_agenda(void);
```

El nou gestor de cues

Aquí és on hi ha els canvis més importants respecte del que ja hem fet. Ara volem un gestor capaç de manejar un nombre arbitrari de cues, amb un funcionament similar al gestor d'una sola cua.

Igual que en el cas de l'agenda, hem introduït el fitxer `cues.h`, amb les declaracions dels tipus de variables i funcions que calen per usar el gestor de cues:

```
typedef struct
{
   float tar;
```

```
   float tse;
} el_cua;

void ini_cues(int n, int l);
int posa_cua(el_cua c, int k);
int treu_cua(el_cua *c, int k);
int long_cua(int k);
int cua_mes_curta(void);
void elim_cues(void);
```

A continuació donem el codi en C d'aquest gestor i després en discutirem el funcionament.

```
#include <stdio.h>
#include <stdlib.h>

#include "cues.h"

static el_cua **cues;
static int *ini_cua,*fin_cua,*lon_cua;
static int num_cues,max_cua;

void ini_cues(int n, int l)
{
   int i;
   num_cues=n;
   max_cua=l;
   cues=(el_cua**)malloc(num_cues*sizeof(el_cua*));
   if (cues == NULL) {puts("ini_cues: error 1."); exit(1);}
   for (i=0; i<num_cues; i++)
   {
      cues[i]=(el_cua*)malloc(max_cua*sizeof(el_cua));
      if (cues[i] == NULL) {puts("ini_cues: error 2."); exit(1);}
   }
   ini_cua=(int*)calloc(num_cues,sizeof(int));
   if (ini_cua == NULL) {puts("ini_cues: error 3."); exit(1);}
   fin_cua=(int*)calloc(num_cues,sizeof(int));
   if (fin_cua == NULL) {puts("ini_cues: error 4."); exit(1);}
   lon_cua=(int*)calloc(num_cues,sizeof(int));
   if (lon_cua == NULL) {puts("ini_cues: error 5."); exit(1);}
}
int posa_cua(el_cua c, int k)
{
   if ((k >= num_cues) || (k < 0)){puts("posa_cua: cua inexistent"); exit(1);}
   if (lon_cua[k] == max_cua) return(0);
   ++lon_cua[k];
   cues[k][fin_cua[k]]=c;
   ++fin_cua[k];
```

```
    if (fin_cua[k] == max_cua) fin_cua[k]=0;
    return(1);
}
int treu_cua(el_cua *c, int k)
{
    if ((k >= num_cues) || (k < 0)){puts("treu_cua: cua inexistent"); exit(1);}
    if (lon_cua[k] == 0) return(0);
    --lon_cua[k];
    *c=cues[k][ini_cua[k]];
    ++ini_cua[k];
    if (ini_cua[k] == max_cua) ini_cua[k]=0;
    return(1);
}
int long_cua(int k)
{
    return(lon_cua[k]);
}
int cua_mes_curta(void)
{
    int j,k;
    j=max_cua;
    for (k=0; k<num_cues; k++) if (j > lon_cua[k]) j=k;
    return(j);
}
void elim_cues(void)
{
    int i;
    free(lon_cua);
    free(fin_cua);
    free(ini_cua);
    for (i=0; i<num_cues; i++) free(cues[i]);
    free(cues);
}
```

Aquestes funcions són una extensió natural de les del gestor de cues anterior. Ara, en lloc d'un vector on guardar la cua, hi tenim un vector de vectors (és a dir, una matriu), anomenat cues. Cada un d'aquests vectors (o sigui, files de la matriu) és una cua (a la secció A.12 s'explica com es crea una matriu d'aquest tipus). De manera anàloga, els enters que ens donaven el primer i l'últim element de la cua i també la seva llargària són reemplaçats per vectors, on cada component fa aquesta funció per a la corresponent cua. Els seus noms són: cues és la matriu de cues, ini_cua és el vector on es guarda el principi de cada cua, fin_cua on es guarda el final i lon_cua indica la seva llargària. Finalment, el nombre de cues es guarda a num_cues i la longitud a la qual dimensionem els vectors cua (és a dir, la seva capacitat màxima) es posa a max_cua.

La rutina ini_cues s'encarrega de dimensionar tots els vectors necessaris per usar la resta de funcions, com també d'inicialitzar les variables globals del fitxer. La funció posa_cua s'encarrega de posar un el_cua a la cua que se li indica. La funció treu_cua treu un cli-

ent de la cua que especifiquem. També tenim dues funcions (`long_cua` i `cua_mes_curta`) que s'encarreguen de retornar la longitud d'una cua prefixada i de determinar quina és la cua més curta. Finalment, per completesa, hem inclòs una rutina que allibera la memòria reservada per a `ini_cues`.

El programa principal

Veiem ara el programa principal que s'encarrega de fer tota la simulació. El seu llistat en C és:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "agenda.h"
#include "cues.h"

#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4

void main(void)
{
   int primer_caixer_buit(int *c, int n);
   float expo(float m, long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmax,tserv;
   long int llavor;
   int bn,*caixa,j,nca,ntc,k;
   llavor=-1995;
   puts("nombre total de caixers?");
   scanf("%d",&ntc);
   ini_agenda(2+ntc);
   ini_cues(ntc,100);
   caixa=(int*)malloc(ntc*sizeof(int));
   if (caixa == NULL) {puts("falta memoria."); exit(1);}
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   nca=0;
   tmax=0.0;
   while (treu_agenda(&e) != 0)
```

```
{
   switch (e.que)
   {
      case OBRIR:
         bn=1;
         for (j=0; j<ntc; j++) caixa[j]=0;
         e.que=ARRIBADA;
         e.quan=expo(2,&llavor);
         posa_agenda(e);
         tserv=1+expo(1,&llavor);
         break;
      case ARRIBADA:
         if (bn == 1)
         {
            t=e.quan;
            k=primer_caixer_buit(caixa,ntc);
            if (k >= 0)
            {
               caixa[k]=1;
               e.quan += tserv;
               e.que=SORTIDA;
               e.on=k;
               posa_agenda(e);
            }
            else
            {
               c.tar=t;
               c.tse=tserv;
               k=cua_mes_curta();
               j=posa_cua(c,k);
               if (j == 0) {puts("error: cua massa petita"); exit(1);}
            }
            e.quan=t+expo(2,&llavor);
            e.que=ARRIBADA;
            posa_agenda(e);
            tserv=1+expo(1,&llavor);
         }
         break;
      case SORTIDA:
         ++nca;
         j=treu_cua(&c,e.on);
         if (j != 0)
{
            t=e.quan-c.tar;
            if (t > tmax) tmax=t;
            e.quan += c.tse;
            posa_agenda(e);
```

```
    }
            else
            {
               caixa[e.on]=0;
            }
            break;
         case TANCAR:
            bn=0;
            break;
         default:
            puts("error: esdeveniment desconegut.");
      }
   }
   printf("nombre de clients atesos: %d\n",nca);
   printf("temps maxim de cua: %f\n",tmax);
}
int primer_caixer_buit(int *c, int n)
{
   int i=0;
   while (i < n)
   {
      if (c[i] == 0) return(i);
      ++i;
   }
   return(-1);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
```

El seu funcionament és anàleg al del programa que ja hem vist per al cas d'una única cua, i per tant només comentarem les diferències principals.

En la declaració de les variables, la principal diferència és en la variable `caixa`, que ara ha passat a ser un vector. La seva funció serà la mateixa d'abans: indicar si un determinat caixer és ocupat (valor `1`) o lliure (valor `0`). Com que el nombre de caixers es llegeix per teclat, hem usat la funció `malloc` per reservar la corresponent memòria.

En tractar l'esdeveniment `ARRIBADA`, necessitem saber si hi ha un caixer lliure i quin és. Per aquest motiu hem creat una petita funció auxiliar (anomenada `primer_caixer_buit`) que explora el vector `caixa` per trobar la primera component igual a `0`. Si la troba, ens retorna quina és aquesta component. Si no hi ha cap component igual a `0` ens retorna `-1` (hem posat aquesta funció al final del programa). La resta és com segueix: si hi ha un caixer buit, posem el client al caixer i introduïm a l'agenda l'esdeveniment `SORTIDA`, que conté l'hora de sortida del caixer *i el número del caixer*. Si tots els caixers són plens, enviem el client a la cua més curta. El càlcul de l'arribada següent és igual que en el cas d'una cua.

Si l'esdeveniment és una SORTIDA, el camp on ens indica de quin caixer ha estat. Per tant, treiem un client de la cua corresponent i el passem al caixer. Si no hi ha ningú fent cua, apuntem (al vector caixa) que aquest caixer queda buit.

Deixem com a exercici modificar aquest programa per mesurar altres paràmetres, com la longitud màxima de les cues o el percentatge de gent que no necessita fer cua. Un altre paràmetre interessant de mesurar és el percentatge de temps d'ocupació de cada un dels caixers.

2.3.3 Més d'una cua: cues de diferents tipus

Considerem ara el cas d'un supermercat que disposa d'un cert nombre de caixers "ràpids" (caixers reservats als clients que porten un nombre reduït de productes, per exemple menys de 10) i de caixers "lents" (caixers oberts a tothom). Volem fer simulacions per comparar el rendiment de diverses configuracions de caixers ràpids i lents. El programa que acabem de veure en l'exemple anterior no serveix per a aquest cas, ja que aquí cal un tractament diferent per als dos tipus de cues.

A partir d'ara anomenarem ncr el nombre de caixers ràpids, i ntc al nombre total de caixers. Identificarem els caixers ràpids amb els números 0, 1, ..., ncr−1. Les modificacions que farem al programa anterior seran, essencialment, tenir en compte que els caixers ràpids són els ncr primers i que la resta són els lents.

Gestió de les cues

El gestor de cues serà bàsicament el mateix que hem usat abans, amb una modificació a la funció cua_mes_curta, que permet buscar la cua lenta més curta i la cua ràpida més curta, de manera separada. La funció modificada és la següent:

```
int cua_mes_curta(int a, int b)
{
   int j,k;
   if (a < 0) {puts("cua_mes_curta: error 1"); exit(1);}
   if (b > num_cues) {puts("cua_mes_curta: error 2"); exit(1);}
   j=max_cua;
   for (k=a; k<b; k++) if (j > lon_cua[k]) j=k;
   return(j);
}
```

La modificació consisteix a poder especificar entre quines cues es fa la cerca de la més curta. El gestor de cues no necessita cap altra modificació tret d'aquesta.

El programa principal

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "agenda.h"
#include "cues_2t.h"
```

```
#define     OBRIR  1
#define  ARRIBADA  2
#define   SORTIDA  3
#define    TANCAR  4

void main(void)
{
   int primer_caixer_buit(int *c, int a, int b);
   float expo(float m, long int *ap_llavor);
   int num_prod(long int *ap_llavor);
   esdev e;
   el_cua c;
   float t,tmcr,tmcl;
   long int llavor;
   int bn,*caixa,j,nca,ntc,ncr,ncl,np,pcr,k;
   llavor=-1995;
   puts("nombre total de caixers?");
   scanf("%d",&ntc);
   puts("nombre de caixers rapids?");
   scanf("%d",&ncr);
   ncl=ntc-ncr;
   puts("nombre de productes maxim per poder usar un caixer rapid?");
   scanf("%d",&pcr);
   ini_agenda(2+ntc);
   ini_cues(ntc,100);
   caixa=(int*)malloc(ntc*sizeof(int));
   if (caixa == NULL) {puts("falta memoria."); exit(1);}
   e.que=OBRIR;
   e.quan=0.0;
   posa_agenda(e);
   e.que=TANCAR;
   e.quan=720.0;
   posa_agenda(e);
   bn=0;
   nca=0;
   tmcr=0.0;
   tmcl=0.0;
   while (treu_agenda(&e) != 0)
   {
      switch (e.que)
      {
         case OBRIR:
            bn=1;
            for (j=0; j<ntc; j++) caixa[j]=0;
            e.que=ARRIBADA;
            e.quan=expo(2,&llavor);
            np=num_prod(&llavor);
```

```
      e.on=(np <= pcr) ? 1 : 0;
      e.ts=0.1*np+0.5+expo(0.5,&llavor);
      posa_agenda(e);
      break;
   case ARRIBADA:
      if (bn == 1)
      {
         t=e.quan;
         if (e.on == 1)
               {k=primer_caixer_buit(caixa,0,ncr);}
            else
               {k=primer_caixer_buit(caixa,ncr,ntc);}
         if (k >= 0)
         {
            caixa[k]=1;
            e.quan += e.ts;
            e.que=SORTIDA;
            e.on=k;
            posa_agenda(e);
         }
         else
         {
            c.tar=t;
            c.tse=e.ts;
            if (e.on == 1)
                  {k=cua_mes_curta(0,ncr);}
               else
                  {k=cua_mes_curta(ncr,ntc);}
            j=posa_cua(c,k);
            if (j == 0) {puts("error: cua massa petita"); exit(1);}
         }
         e.quan=t+expo(2,&llavor);
         e.que=ARRIBADA;
         np=num_prod(&llavor);
         e.on=(np <= pcr) ? 1 : 0;
         e.ts=0.1*np+0.5+expo(0.5,&llavor);
         posa_agenda(e);
      }
      break;
   case SORTIDA:
      ++nca;
      j=treu_cua(&c,e.on);
      if (j != 0)
      {
         t=e.quan-c.tar;
         if (e.on < ncr)
               {if (t > tmcr) tmcr=t;}
```

```
                else
                   {if (t > tmcl) tmcl=t;}
             e.quan += c.tse;
             posa_agenda(e);
          }
          else
          {
             caixa[e.on]=0;
          }
          break;
       case TANCAR:
          bn=0;
          break;
       default:
          puts("error: esdeveniment desconegut.");
    }
  }
  printf("nombre de clients atesos: %d\n",nca);
  printf("temps maxim en cua rapida: %f\n",tmcr);
  printf("temps maxim en cua lenta : %f\n",tmcl);
}
int primer_caixer_buit(int *c, int a, int b)
{
   int i=a;
   while (i < b)
   {
      if (c[i] == 0) return(i);
      ++i;
   }
   return(-1);
}
float expo(float m, long int *ap_llavor)
{
   float alea1(long int *ap_llavor);
   return(-m*log(alea1(ap_llavor)));
}
int num_prod(long int *ap_llavor)
{
   float expo(float m, long int *ap_llavor);
   int n;
   n=1+(int)expo(20,ap_llavor);
   return(n);
}
```

Aquest programa és una (petita) modificació del de la secció 2.3.2. La primera diferència la tenim en la generació de l'esdeveniment `ARRIBADA`: juntament amb el temps d'arribada, generem el nombre de productes i el temps de servei. A partir del nombre de productes ja podem saber

si el client anirà a un caixer ràpid o a un de lent, i apuntem aquest fet al camp on (1 vol dir cua ràpida, 0 cua lenta). Això ens servirà per decidir, quan tractem aquest esdeveniment, a quina cua posem el client.

La diferència fonamental és en el tractament de l'esdeveniment `ARRIBADA`. En primer lloc mirem si hi ha un caixer lliure. La cerca d'aquest caixer ha de tenir en compte si el client ha d'anar a un caixer ràpid o lent, i per això hem modificat la funció `primer_caixer_buit`, per poder especificar el rang de cerca. Si trobem un caixer lliure, la resta és igual que en l'exemple anterior. Si no hi ha cap caixer lliure, llavors hem de buscar la cua més curta d'entre les cues ràpides o lentes, segons correspongui. Per aquest motiu hem modificat la funció `cua_mes_curta`, per poder especificar el rang de cues per a la cerca.

Per simplificar el càlcul del nombre de productes que porta cada client, hem definit la funció `num_prod`. El seu funcionament no hauria de presentar cap problema.[5]

No hi ha més modificacions importants en la simulació. Les altres modificacions es deuen al fet que ara no mesurarem el temps màxim d'estada a les cues en general, sinó que obtindrem el temps màxim per a cues ràpides i lentes. Això ens obliga a introduir uns petits canvis en el tractament de l'esdeveniment `SORTIDA`, que és on es prenen aquestes mesures.

Nauralment, es poden prendre moltes altres mesures, com per exemple l'evolució de la longitud de les cues amb el temps, etc. Deixem com a exercici per al lector la modificació d'aquests programes per obtenir aquesta informació addicional.

Altres extensions del programa

Podriem continuar introduint canvis en el programa amb la intenció de simular sistemes concrets, pero no ho farem. El nostre propòsit ha estat introduir una metodologia, el més senzilla possible, que permeti estudiar molts problemes concrets de simulació discreta. Creiem que el lector no hauria de tenir gaires dificultats a modificar els programes aquí descrits per adaptar-los a una gran quantitat de casos.

També volem comentar l'elecció de les lleis que segueixen els processos simulats aquí (arribada de persones, nombre de productes, etc.). Les lleis usades en aquests exemples són molt simples, donat que la nostra intenció és mostrar els algorismes de simulació. Si es volen efectuar simulacions una mica realistes d'un cert procés, cal canviar aquestes lleis per d'altres que siguin molt més adequades al problema concret. Per exemple, en un problema de cues en un supermercat, el temps que passa entre arribades de clients hauria de dependre de l'hora del dia. Per altra banda, volem remarcar que els programes de simulació descrits aquí no estan afectats per l'elecció de les lleis, perquè aquestes són funcions del programa que poden ser reemplaçades sense cap dificultat. La qüestió és disposar d'una implementació en C d'aquestes lleis. Si la llei és d'un tipus molt general (Gamma, Poisson, etc.) es poden trobar implementacions en molts llocs (per exemple, vegeu la referència [19]). Si la llei és molt complexa, però, no tindrem més remei que implementar-la nosaltres mateixos.

2.3.4 Llistes enllaçades

El propòsit d'aquesta secció és explicar una manera de gestionar cues sense haver-ne d'especificar prèviament una longitud màxima. La tècnica concreta que explicarem es basa en manejar la

[5] La unitat que sumem a la part entera de l'exponencial és per, essencialment, evitar que poguem generar l'arribada d'un client sense productes. Si es vol, també es pot fer servir una llei de Poisson per generar el nombre de productes, sumant-li una unitat per la mateixa raó d'abans.