Escola Politècnica Superior
de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTOL DEL TFC: Borinots: Un algorisme evolutiu multiagent per a problemes d'optimització combinatòria.

TITULACIÓ: Enginyeria Tècnica de Telecomunicació, especialitat Sistemes de Telecomunicació

AUTOR: Jesús Martínez Navarro

DIRECTOR: Francesc Comelles i Padró

DATA: 29 de juny de 2005

Títol: Borinots: Un algorisme evolutiu multiagent per a problemes d'optimització combinatòria.

Autor: Jesús Martínez Navarro

Director: Francesc Comellas i Padró

Data: 29/06/05

Resum

Diversos problemes en telecomunicacions són de naturalesa combinatòria i la seva resolució òptima és en molts casos impossible, ja que el temps de resolució mitjançant algorismes convencionals creix exponencialment amb la mida del problema. Llavors, certs algorismes de naturalesa probabilística com el simulated annealing, algorismes genètics, mètodes multiagent etc. - suggerits per processos naturals evolutius o pel comportament social d'insectes- resulten útils per trobar solucions pròximes a l'òptima en un temps raonable.

En aquest TFC s'ha implementat un nou mètode d'optimització (que hem aplicat a l'acoloriment de grafs, base de l'assignació de freqüències GSM) i que s'inspira en el comportament social d'uns insectes socials peculiars: els borinots. Cada borinot porta codificada una solució al problema d'acoloriment., que evoluciona a mesura que els insectes virtuals segueixen el seu cicle vital (busquen i troben aliment, pateixen mutacions, etc.). D'aquesta manera, i en un procés que simula l'evolució darwinista, s'arriba a trobar solucions bones del problema.

L'eficiència d'aquest mètode s'ha comprovat mitjançant un estudi d'acoloriment en grafs aleatoris i s'ha comparat respecte altres mètodes d'optimització com el d'àngels i mortals i un algorisme genètic, arribant a la conclusió que es tracta d'un mètode vàlid per la resolució

Title: Bumblebees: an evolutionary multiagent algorithm for combinatorial optimization problems.

Author: Jesús Martínez Navarro

Director: Francesc Comelles I Padró

Date: June, 29th 2005

Overview

Many problems in telecommunications are of combinatorial nature and their optimum resolution is impossible in many cases, because the time of resolution by conventional algorithms grows exponentially with the size of the problem. Then, some algorithms of probabilistic nature, like simulated annealing, genetic algorithms, multiagent methods, etc. - suggested by evolutionary natural processes or by the behaviour of insects, are useful to find near optimum solutions in a reasonable time.

In this TFC we have implemented a new method of optimization (applied to tgraph coloring, base of frequency assignment in GSM). The method is inspired by the social behaviour of some peculiar insects: bumblebees. Each bumblebee codifies a solution to the coloring problem, that while the virtual insects continue their life cycle (they seek and find food, suffer mutations, etc). Then, in a process that simulates the darwinist evolution, good solutions of the problem are found.

The efficiency of this method has been verified by means of a study of random graph coloring and has been compared with other optimization methods like *angels and mortals* and a *genetic algorithm*, coming to the conclusion that it is a valid method for the resolution of problems like frequency assignment in GSM networks.

ÍNDEX

INTRODUCCIÓ

L'increment, en les darrers anys, del nombre d'usuaris de telefonia mòbil comporta, per a un millor servei, la necessitat d'aprofitar millor els recursos disponibles.

Concretament, el nombre de freqüències disponibles per a cada operador és limitat, i si s'aconsegueix una bona assignació es pot millorar sensiblement la capacitat de la xarxa.

En aquest TFC proposem un nou mètode d'optimització combinatòria, que anomenem *borinots*, capaç de resoldre un aspecte fonamental de la telefonia mòbil: l'assignació de freqüències. Amb una adequada assignació de freqüències, es pot donar una cobertura màxima amb el mínim de freqüències als usuaris d'una xarxa.

Plantegem el problema fent un paral·lelisme amb l'acoloriment de grafs (conjunt de vèrtexs units per branques): les estacions base són els vèrtexs i les portadores són representades pels diferents colors dels vèrtexs. Les branques representen qualsevol tipus d'interferència cocanal o de canal adjacent. L'objectiu principal és aconseguir que cap branca uneixi dos vèrtexs del mateix color.

Tal i com veurem més endavant, donat que no es tracta d'un problema trivial que es pugui resoldre adequadament amb mètodes tradicionals, els mètodes d'optimització combinatòria són claus per resoldre l'assignació de freqüències. En presentem alguns, amb les corresponents aplicacions.

El TFC se centra en el desenvolupament i estudi d'un nou mètode d'optimització combinatòria basat en el comportament dels borinots. En aquest mètode es fa una simulació per ordinador del seu comportament biològic i social després d'haver associat als individus el problema a resoldre. Talment com passa a la natura, l'evolució de la població optimitza la possible solució. Apliquem aquest mètode a l'acoloriment de grafs, com a versió bàsica de l'assignació de freqüències a GSM i es presenta una taula de resultats on es compara l'eficiència d'aquest nou mètode amb la d'altres algoritmes.

1. Telefonia mòbil GSM

En aquest primer apartat fem referència a la telefonia mòbil GSM com un exemple de problema difícil a telecomunicacions i veiem la seva equivalència en l'acoloriment de grafs, que farem servir per estudiar l'eficiència i efectivitat del nou algorisme.

Podem situar el naixement del sistema GSM (Groupe Special Mobile) a l'any 1982, quan la Conferència d'Administracions de Correus i Telecomunicacions (CEPT) va decidir desenvolupar un conjunt d'estàndards per a una futura xarxa cel·lular de comunicacions mòbils d'àmbit europeu.

1.1. Estructura del GSM

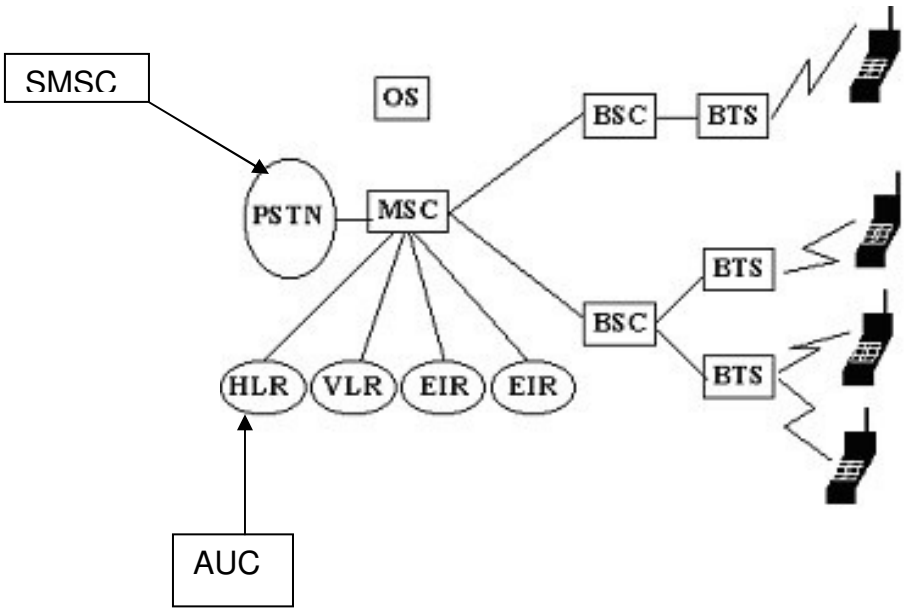

Fig. 1.1: Estructura bàsica d'una xarxa GSM

El sistema s'organitza com una xarxa de cèl·lules radioelèctriques contínues que proporcionen cobertura completa a l'àrea de servei (veure[10]). Cada cèl·lula pertany a una estació base (**BTS**) que opera en un conjunt de canals ràdio diferents als utilitzats en les cèl·lules adjacents i distribuïdes segons un pla cel·lular.

El **BSC** (controlador d'estacions base) està connectat a un grup d'estacions base i s'encarrega d'aspectes com el handover (traspàs del mòbil d'una cèl·lula a una altra) i el control de potència.

Una o vàries BSC`s es connecten a una central de commutació de mòbils (**MSC**). És el cor del GSM, ja que s'encarrega de la inicialització, enrutament, control, financiació i facturació de les trucades.

HLR i **VLR** són els registres de posicions base i de posicions de visitants respectivament. Es tracta de bases de dades on es troba la informació referent als abonats.

El centre d'autentificació (**AUC**) treballa associat a l'HLR. Conté la informació per la qual es comprova l'autenticitat de les trucades per tal d'evitar possibles fraus o la utilització de targetes d'abonat (SIM's) robades.

L'**EIR** (Registre d'identificació d'equips) emmagatzema informació sobre el tipus d'estació mòbil en ús i pot eludir que es realitzi una trucada quan es detecti que pateix algun error que pot afectar negativament a la xarxa.

Finalment cal citar el **SMSC**; el centre de servei de missatgeria (missatges curts SMS, MMS, email's.

1.2. Assignació de freqüències

1.2.1. La reutilització de freqüències

La idea fonamental en què es basen els sistemes de mòbils cel·lulars és la reutilització dels canals mitjançant la divisió del terreny en cel·les contínues que s'il·luminen des d'una estació base amb uns determinats canals.

La reutilització de freqüències no és possible en cèl·lules contigües, però sí en altres més llunyanes. El nombre de cops que un canal pot ser reutilitzat és major quant més petites siguin les cèl·lules.

1.2.2. Caracterització del problema de l'assignació de freqüències

Per caracteritzar el problema que es planteja, cal tenir en compte quatre aspectes: la matriu de restriccions, el vector de requeriments , el vector de transmissors i el vector d'ocupació.

Els elements de la **matriu de restriccions** descriuen la separació freqüencial necessària entre la cel·la fila i la cel·la columna corresponents. Aquestes restriccions venen determinades per un nombre natural: 0 significa que no existeix cap restricció; 1 indica l'existència d' interferència cocanal (freqüències separades per una unitat com a mínim); 2 representa la interferència de canal adjacent, sent necessària una separació de freqüències igual o superior a 2.

Els valors de la matriu de restriccions s'obtindran tenint en compte les característiques de la xarxa GSM que es vol representar (frequency hopping, control dinàmic de potència, transmissió discontínua).

Amb el **vector de requeriments hopping** sabrem el nombre de vèrtexs (freqüències) associats a cada cel·la. Cada transmissor emetrà sempre per una freqüència diferent a la resta de transmissors de la mateixa cel·la. És per això que el nombre de freqüències haurà de ser igual o major al nombre de transmissors per cel·la.

El nombre de transmissors per cel·la és proporcional al trànsit de la zona a tractar (índex de població, tràfic mitjà generat a l'hora punta, grau de servei de la xarxa,...). Aquest nombre de freqüències que es fan servir simultàniament es representa en el **vector de transmissors**.

Finalment, amb el **vector d'ocupació** es mostra el percentatge d'ocupació de cada cel·la.

El problema d'assignació de freqüències a GSM es pot descriure directament com un problema d'acoloriment de grafs. En aquest TFC considerarem la versió més simple equivalent a que la separació freqüencial sigui 1, però el cas general seria fàcilment tractable amb modificacions mínimes respecte el que es descriu en aquesta memòria.

En la següent figura es veu representada l'equivalència entre l'assignació de freqüències i l'acoloriment de grafs. Veiem com, els transmissors de cada cel·la en l'assignació de freqüències s'associen als vèrtexs del graf i les branques del graf representen les possibles interferències entre transmissors. El problema d'assignació de freqüències equivaldria a trobar un acoloriment dels vèrtex del graf amb el mínim nombre de colors i de forma que mai dos vèrtexs amb el mateix color estiguin units per una branca.

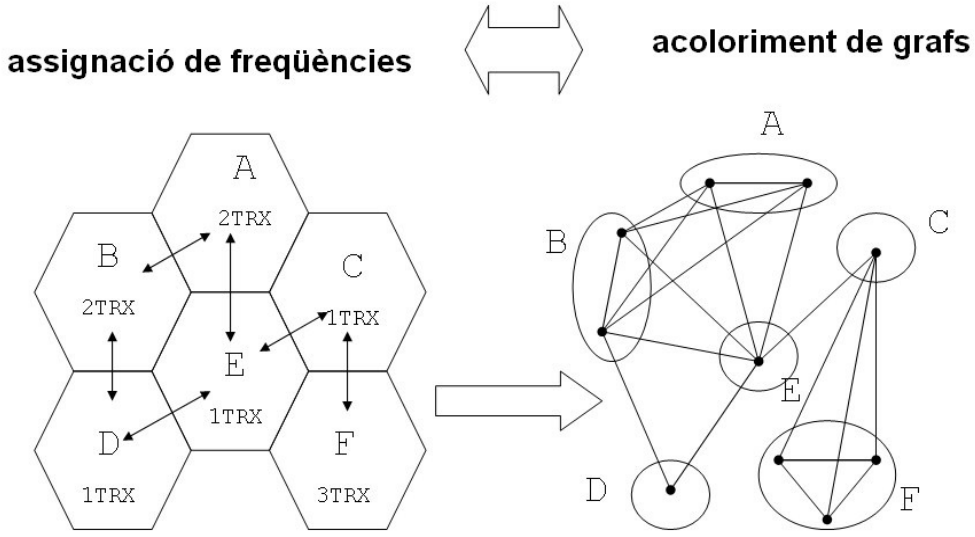

Fig. 1.2: Paral·lelisme entre l'assignació de freqüències i l'acoloriment de grafs

2. Grafs

2.1. Teoria de grafs

Un graf no és més que un conjunt de punts i un conjunt de relacions entre aquests punts. Els punts s'anomenen vèrtexs o nodes i les relacions entre els punts són les branques. Els grafs tenen una estructura topològica perquè establim relacions espacials entre els fenòmens que estem estudiant.

L'ordre del graf és el nombre total de vèrtexs i la mida és el nombre de branques.

D'aquí en endavant, farem referència a un graf com a un parell (V(G), E(G)). V(G) són els vèrtexs i E(G) són les branques.

Un graf **simple** (els que farem servir en tot moment) es caracteritza per no tenir branques repetides (2 vèrtexs units per més d'una branca) ni llaços (branques que enllacen un mateix vèrtex).

Parlem de vèrtexs u i v **adjacents** si hi ha una branca que els uneix (uv). Dues branques (uv i vw) són adjacents si tenen un vèrtex en comú. El nombre de branques que incideixenen en un vèrtex determina el seu grau.

La **densitat d'un graf** es calcula mitjançant el quocient entre el nombre de branques que té i el nombre màxim que en podria tenir.
El nombre màxim de branques és

$$\frac{n(n-1)}{2} \quad , \qquad \textbf{(2.1)}$$

on *n* és el nombre de vèrtexs.
I la densitat de G(V,E) serà:

$$\rho = \frac{2 \cdot E}{n(n-1)} \qquad \textbf{(2.2)}$$

La distància entre dos vèrtexs és el nombre mínim de branques que es necessiten per arribar d'un vèrtex a un altre, i el diàmetre la distància màxima entre dos vèrtexs.

Un graf simple és complet d'ordre n (K_n) si qualsevol parell de vèrtexs està unit per una branca, i és regular de grau r i tots els vèrtexs tenen grau r.

Un graf és bipartit en V_1 i V_2 si $V_1 \cup V_2 = V(G)$, $V_1 \cap V_2 = 0$ i cada branca uneix un vèrtex de V_1 amb un vèrtex de V_2.

2.2. Acoloriment de grafs

L'acoloriment d'un graf es dóna quan totes les branques uneixen vèrtexs adjacents de colors diferents. El mínim nombre de *k* colors amb els quals es pot acolorir el graf es diu nombre cromàtic $\chi(G)$ del graf G.

Evidentment, aquest nombre serà igual o inferior al nombre de vèrtexs que contingui el graf.

2.2.1. Optimització

Intentar resoldre el problema directament de manera exhaustiva (provant primer amb 2 colors, després amb 3,...) comportaria fer una quantitat de càlculs desmesurada.
Suposem, per exemple, un graf d'ordre *n*. Amb *i* colors, les combinacions possibles serien i^n. Aleshores, intentar trobar la solució òptima d'un graf de 50 vèrtexs i un $\chi(G)$=4 suposaria un total de $1.267 \cdot 10^{30}$ operacions.

3. Algorismes: Resolució de problemes difícils

Definim un algorisme com el conjunt d'instruccions necessàries per resoldre un problema. Per resoldre un problema donat poden existir diferents algorismes i aleshores caldrà decidir quin és el més adequat. Per comparar algorismes, s'ha de tenir en compte, a més de l'eficàcia de l'algorisme (que realment trobi una solució útil), tant el temps necessari per la resolució del problema com el nombre de recursos utilitzats.

Per resoldre un problema a l'ordinador, cal introduir la quantitat de dades inicial del problema, n. El nombre d'instruccions necessàries està directament relacionat amb aquestes dades: f(n).

El problema a resoldre en aquest TFC és d'*optimització combinatòria*. Per resoldre aquests problemes, s'avalua la qualitat de cada solució trobada mitjançant una funció de cost (veure [3]).

Un exemple de problema d'optimització combinatòria pot ser el *problema del viatjant,* on es tracta de trobar el recorregut més curt visitat un conjunt de ciutats i acabar en la ciutat d'inici. En cas que no hi hagi moltes ciutats, el problema es pot resoldre calculant la distància de tots els camins possibles. Però, com que el nombre de camins possibles creix factorialment amb el nombre de ciutats, el problema no es pot resoldre de manera exhaustiva quan puja el nombre de ciutats.

Podem classificar els problemes segons la seva complexitat, en tres tipus:

Un problema és de tipus P si *f(n)* és un polinomi en *n* (algorisme resoluble en temps polinòmic).

Els problemes de tipus NP (temps polinòmic no determinista) són aquells que no són de tipus P però podem arribar a trobar una solució en temps polinòmic.

Per últim trobem la classe NP-complets (p.e. el problema del viatjant), els més complicats de resoldre. Els problemes NP-complets es caracteritzen per ser tots iguals, de manera que si es descobreix una solució P per algun d'ells, aquesta solució seria aplicable a la resta. Per a aquests problemes no es coneixen algorismes que garanteixin que la solució òptima pugui trobar-se en un temps raonable d'execució d'un programa (veure[4]).

4. Mètodes d'optimització

Abans de parlar d'uns quants mètodes d'optimització existents, cal aclarir el concepte de **funció de cost o fitness**, necessària en el procés d'optimització.

Aquesta funció ens proporciona informació sobre la qualitat de les solucions que van apareixent abans de trobar una solució definitiva.

L'objectiu és optimitzar aquesta funció. Donada una funció de cost F amb variables x_1, x_2, x_3,..., es tracataria de trobar una combinació d'aquestes variables per tal que la funció sigui màxima o mínima, segons convingui.

Pot donar-se el cas que la funció de cost no existeixi o que l'haguem de calcular polinòmicament o a partir d'altres funcions.

4.1. Mètodes analítics

Aquests mètodes poden fer-se servir sempre que es conegui analíticament la funció de cost. Donat que hem de trobar els màxims absoluts i relatius, aquesta funció ha de ser derivable dues vegades.

4.2. Mètodes exhaustius, heurístics i aleatoris

Aquests tres famílies de mètodes són útils quan no sabem quina és la funció de cost.

Encara que són eficients perquè sempre troben la millor solució, perden velocitat i efectivitat a mesura que el nombre de possibles solucions augmenta.

Tant els mètodes exhaustius com els heurístics investiguen totes les solucions existents, escollint la més favorable. La diferència és que amb els heurístics s'eliminen zones de l'espai que no són prou bones.

D'altra banda, els mètodes aleatoris, com el seu nom indica, exploren l'espai de solucions de forma aleatòria.

4.3. *Hillclimbing*

Aquests algorismes escaladors van avaluant la funció de cost en un o varis punts, passant sempre d'un punt a un altre on el resultat de l'avaluació és superior en el darrer.

El principal problema que trobem en aquests algorismes és que solen quedar-se al pic més proper a la solució inicial. Cal afegir que no són útils en

problemes multimodals, ja que la funció de cost en aquests problemes conté varis possibles òptims.

4.4. Recuita simulada (*Simulated Annealing)*

El nom prové de la manera com s'aconsegueixen certs aliatges en forja: un cop fos el metall, es va refredant de mica en mica per aconseguir una estructura cristal·lina perfecta amb un aliatge dur i resistent.

Es tracta d'un algorisme escalador estocàstic que evita els mínims locals que aconsegueixen els algorismes escaladors. La manera de fer-ho és no acceptant sempre la solució òptima, sinó que es pot agafar una solució menys òptima sempre i quan la diferència entre les dues solucions no sigui major a un valor determinat, que està relacionat amb el paràmetre *temperatura*. A mesura que augmenta la temperatura, hi ha més possibilitats que una solució sigui acceptada.

La probabilitat d'acceptar una solució és major quan menor és la diferència d'energies i quan més alta és al temperatura del sistema. Per arribar a convergir en una solució, ha de baixar la temperatura progressivament en el moment que totes les solucions es troben dins d'aquest marge i s'ha de tornar a buscar solucions que acompleixin el nou marge.

Si el marge de temperatura que s'accepta al principi és molt gran, no serà eficaç perquè l'algorisme acceptarà solucions molt dolentes amb moltes oscil·lacions d'energia. Es necessitaran, per tant, més iteracions per baixar el marge de temperatura i anar millorant l'algorisme.

Un esquema del funcionament podria ser el següent:

Busca una solució inicial a l'atzar
Calcula la funció de cost
Bucle: Genera una nova solució
Calcula la funció de cost *f(j)*
Si és millor que l'anterior → guardar la nova solució
Si és pitjor → guardar la nova solucio amb probabilitat $e^{-\frac{f(j)-f(i)}{T}}$
Si és la millor solució → canviar la millor solució anterior per aquesta.
Si el nombre de iteracions és igual a canvi de temperatura → $T_k = r \cdot T_{k-1}$
Fi fins que nombre iteracions màxim o solució trobada.
Fi

4.5. Algorismes genètics

Els algorismes genètics són mètodes sistemàtics per la resolució de problemes de cerca i optimització que apliquen els tres principis de l'evolució biològica: selecció basada en la població, reproducció i mutació.

En aquest tipus d'algorisme, es dóna paràmetres al problema amb una sèrie de variables (x_1, ...x_n). Aquestes variables són codificades en un cromosoma, sobre el qual s'aplicaran tots els operadors utilitzats per un algorisme genètic. En l'algoritme genètic, a diferència d'altres algorismes evolutius com la programació genètica, va implícit el mètode per resoldre el problema. Donat que un algorisme genètic és independent del problema, es tracta d'un algorisme útil per qualsevol problema, però no especialitzat en cap.

Les solucions codificades en un cromosoma busquen ser la millor solució. L'ambient exercirà una pressió selectiva sobre la població, de manera que només els millors adaptats (els que solucionin millor el problema) sobrevisquin o deixin el seu material genètic a les següents generacions, d'igual forma que passa en l'evolució de les espècies. La diversitat genètica s'introdueix mitjançant mutacions i reproducció sexual.

A la natura, l'únic que s'ha d'optimitzar és la supervivència, i això significa maximitzar alguns factors i minimitzar d'altres. Però un algorisme genètic s'usarà habitualment per optimitzar només una funció, no diverses funcions relacionades entre sí simultàniament.

En resum, un algoritme genètic consisteix en:
- Trobar de quins paràmetres depèn el problema
- Codificar els problemes en un cromosoma
- Aplicar els mètodes de l'evolució amb intercanvi d'informació i alteracions que generen diversitat:
  - Selecció
  - Reproducció sexual
- Mutació

4.5.1. Selecció o adaptació al medi

Aquí s'avalua la qualitat de l'individu o el seu grau d'adaptació al medi. Els millors adaptats tindran la possibilitat de reproduir-se i transmetre el seu material genètic a la descendència. L'adaptació té molt a veure amb l'èxit que vagi tenint al llarg de la seva vida, i depèn de la resta d'individus de la població.

A mesura que passin les generacions, tindran més possibilitats de reproduir-se els individus millor adaptats. Existeixen dues tècniques per escollir la part de la població destinada a reproduir-se: selecció directa i selecció aleatòria. En la **selecció directa** es fa servir un criteri objectiu, mentre que en la **selecció aleatòria** es va servir un criteri aleatori que pot ser equiprobable (tots els individus tenen la mateixa probabilitat) o estocàstic (no tots els individus tenen la mateixa probabilitat: *sorteig, categories, ruleta*).

4.5.2 Reproducció

És una part essencial de l'algorisme genètic, ja que permet que les noves generacions heretin i es combinin les característiques dels seus antecessors.

Podem esmentar, per exemple, cinc tipus d'encreuaments:

- *Encreuament bàsic*: se selecciona un punt a l'atzar de la cadena. La part anterior del punt és copiada del genoma del pare i la posterior del de la mare.

- *Encreuament multipunt*: igual que l'encreuament bàsic, però en aquest cas s'estableix més d'un punt d'encreuament.

- *Encreuament segmentat*: existeix una possibilitat que un cromosoma sigui punt d'encreuament. A mesura que es va formant la nova cadena del descendent, per cada gen, es verifica si allà es produirà un encreuament.

- *Encreuament uniforme*: per cada gen de la cadena del descendent existeix una probabilitat que el gen pertanyi al pare, i una altra que pertanyi a la mare.

- *Encreuament per a permutació*: existeix una família d'encreuaments específics per a problemes de permutació, com per exemple l'*encreuament de mapeig parcial*, l'*encreuament d'ordre* o l'*encreuament de cicle*.

4.5.3. Mutació

En la majoria dels casos les mutacions són letals, però en promig contribueixen a la diversitat genètica d l'espècie.

Un cop establerta la freqüència de mutació, s'examina cada bit de cada cadena just abans de crear la nova criatura a partir dels seus pares. Si un número generat aleatòriament està per sota d'aquesta probabilitat, es canviarà el bit (de 1 a 0 o de 0 a 1). Segons el número d'individus que hi hagi i del número de bits per individu, pot ser que les mutacions siguin molt rares en una sola generació.

Per últim, cal dir que no és necessari abusar de la mutació, ja que redueix l'algorisme genètic a una cerca aleatòria. Resultarà més convenient emprar utilitzar altres mecanismes de diversitat, com per exemple augmentar la població o garantir que la població inicial es genera de forma aleatòria.

4.6. Algorismes multiagent: formigues

Les formigues són insectes socials que viuen en colònies i que, degut a la seva col·laboració mútua, són capaces de mostrar comportaments complexes i realitzar feines difícils des del punt de vista d'una formiga individual. Una de les grans habilitats d'aquests insectes és la capacitat de trobar els camins més curts i les fonts d'alimentació dins del formiguer (tenint en compte que moltes espècies són cegues).

Les formigues segreguen una substància anomenada *ferormona* (que pot olorar-se), molt útil com a guia per seguir un camí determinat. Està comprovat que les formigues mostran més tendència pels camins on hi ha més ferormona.

Per tant, els camins amb menys ferormona, com que seran menys visitats, tendiran a desaparèixer. Mentre que la resta de camins es veuran cada cop més reforçats i d'aquesta manera es trobarà el camí més curt cap a l'aliment.

Els algorismes basats en colònies de formigues (OCH) s'inspiren directament en el comportament de les colònies reials de formigues per solucionar problemes d'optimització combinatòria (veure [9]). En cada generació de l'algorisme, cada formiga construeix una solució al problema recorrent un graf de construcció (graf d'estats). La formiga pot anar a qualsevol dels vèrtexs del graf. Hi ha dos paràmetres dels vèrtexs que influeixen en aquests moviments:

- *Informació heurística*: mostra la preferència heurística que es produeixi moviment d'un vèrtex a un altre.

- *Informació dels rastres de ferormona:* mesura la "desitjabilitat apresa" del moviment entre dos vèrtexs. Aquesta informació es modifica durant l'execució de l'algoritme depenent de les solucions trobades per les formigues.

Una altre família d'algorismes multiagent és la descrita a [1]. Le principal s diferències amb abans, es que no hi ha explicitament rastres de ferormona, i que les formigues és mouen no sobre el graf d'estats, sinó sobre el graf objecte d'estudi. Si apliquem aquest mètode d'optimització a l'acoloriment de grafs, podem reflexar-ho en aquest esquema:

Inici
    Acoloriment del graf aleatòriament
    Distribuir les formigues aleatòriament
    Per a cadascuna de les formigues fer:
        Si hi ha probabilitat P_n → moure al pitjor vèrtex adjacent.
          Si no → moure a un altre vèrtex.
        Si hi ha probabilitat P_c → assigna el millor color
          Si no → assigna qualsevol color
        Actualitzar la funció de cost.
        Si la nova solució menor que l'anterior → guardar nova solució.
    Fi fins que nombre iteracions màxim o solució trobada

Fi

, on P_n és la probabilitat de moviment al pitjor vèrtex adjacent i P_c és la probabilitat d'assignar el millor color.

4.7. Tècniques basades en simulacions de poblacions

Aquesta tècnica, molt utilitzada a la majoria dels algorismes evolutius, pot ser interpretada com una versió de l'algorisme de les formigues o de la recuita simulada (simulated annealing).

Aquí es busca l'òptim canviant continuament les solucions –mutació- (ajuda a no trobar-nos amb mínims locals), però relacionant la qualitat de la solució amb la vida d'un individu d'una població simulada en evolució. A continuació presentem un exemple concret.

4.7.1. Àngels i mortals

El disseny del mètode *d'àngels i mortals* s'origina a partir d'una idea de Sorin Salomon *et al.* de l'any 1999 (veure [7]), que és la següent: De l'estudi de l'evolució de dues poblacions en interacció mitjançant una aproximació contínua, resulta l'extinció dels individus d'una població mentre que amb una aproximació discreta per simulació es comprova que no passa el mateix.

Aquest model discret es simula genèricament com un món ple de caselles semblant a un tauler d'escacs. Sobre aquest món es distribueixen aleatòriament uns individus: per una banda hi ha els àngels, que són immortals; mentre que per l'altra es troben els mortals, la vida dels quals anirà disminuint conforme passin les generacions.

No hi ha cap regla de moviment d'aquest individus (el moviment és aleatori). Només s'ha de tenir en compte que si un mortal es troba a la vora d'un àngel, es multiplicarà.

El model suggerí un nou mètode d'optimització combinatòria simplement associant a cada mortal una possible solució al problema considerat i permetent –ne canvis al llarg de les generacions [CG04].

4.8. Xarxes neuronals

És un sistema compost per un gran nombre d'elements bàsics agrupats en capes i que es troben altament interconnectats. Aquesta estructura posseeix vàries entrades i sortides, les quals seran entrenades per reaccionar d'una manera desitjada als estímuls de l'entrada.

Es pot dir que aquests sistemes simulen el cervell humà. Requereixen aprendre a comportar-se i algú déu encarregar-se d'ensenyar-los o entrenar-los en base a un coneixement previ de l'entorn del problema.

5. Nou mètode d'optimització: Borinots

5.1. Introducció

El borinot de terra o *Bombus terrestris ,* és fàcilment identificable pel seu tamany, i el seu acoloriment particularment "agresiu". És un dels tipus de borinots més utilitzats en l'agricultura intensiva, degut al seu alt nivell de polinització. És de color negre, amb una banda blanca al final de l'abdomen. El tòrax i l'abdomen estan creuats per una banda groga. Sempre estan olorant i tocant qualsevol objecte amb les seves antenes, comprovant si és comestible o no (veure[8]).

Els borinots són insectes socials, és a dir, hi ha generalment cooperació entre individus separats.

Per confeccionar el seu niu, el borinot recull partícules de fusta, que mastega fins a obtenir la pasta amb la que construir les parets del refugi. Les colònies estan formades, normalment, per unes cent obreres i una reina, que és la fundadora i la mare de tots els membres de la colònia.

El cicle biològic del *Bombus terrestris* comença a la primavera quan la reina abandona el seu lloc d'hivernació per buscar un niu adequat. Quan el troba, prepara un lloc per emmagatzemar mel, reunir pol·len i posar els primers ous. Després de néixer les primeres obreres, la reina posa cada cop més ous, mentre que la missió de les obreres és alimentar les larves i recol·lectar aliment, de manera que la colònia augmenta.

La seva alimentació es basa en el pol·len, d'on obtenen proteïnes; i en el nèctar, el qual els proporciona tots els sucres necessaris per l'energia dels borinots.

Un nou estudi suggereix que els borinots tenen preferència innata per les flors blaves, grogues i aquelles que posseeixen ambdós colors. Però, aquestes flors que els borinots poden trobar inicialment irresistibles, deixaran de ser freqüentades pels borinots si no proporcionen pol·len, ja que aquests insectes eviten una altra visita innecessària a la mateixa flor.
El model social i comportament dels borinots ens ha suggerit un nou mètode d'optimització combinatòria, el qual especifiquem amb detall a continuació.

5.2. Disseny del programa

En aquest apartat dissenyem un món virtual amb borinots i simulem el seu comportament. Cadascun d'aquests insectes té codificada una possible solució al problema de l'acoloriment de grafs.

Com ja hem dit anteriorment, els diferents elements del graf representen una xarxa GSM de telefonia mòbil: els vèrtexs són les estacions base, les branques representen els conflictes i els colors representen les freqüències portadores.

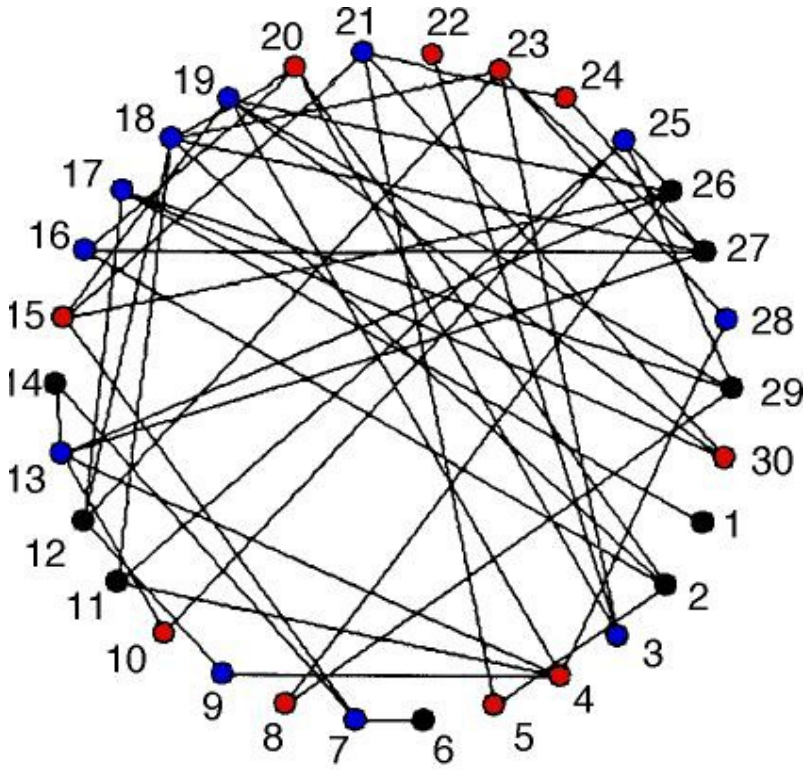

Fig.5.1: Graf de 30 vèrtexs i 50 branques (densitat del 12% aprox.) acolorit amb 3 colors.

Per veure què passa amb diversos tipus de grafs, es realitzaran proves per 2 densitats de branques: 10% i 20%.

Per exemple, si tenim un graf de 40 vèrtexs, el nombre màxim de branques és:

$$\text{Total branques} = \frac{40 \cdot 39}{2} = 780$$

Les proves que realitzaríem serien amb 78 branques (densitat del 10%) i 156 branques (densitat del 20%).

Representem els colors amb lletres per no portar a confusió amb els vèrtexs. Aleshores, si hi ha una solució amb 4 colors (“a, b, c, d”), podria ser representada així: “CDCADABCBBBCABDBDDBAABBBDCDCDA”.

Si el nombre de colors utilitzat no porta a una solució de l’acoloriment, el programa mostra la solució més propera a l’èxit.

5.3. Estructura de l'algorisme

L'estructura de l'algorisme del programa és la següent:

Inici
Per a 20 proves:
- Demanar nombre de branques i nombre de color a l'usuari.
- Generar el graf aleatori.
- Crear el món amb el niu de borinots al centre i el pol·len escampat aleatòriament.
- Generacions
    - Moure els borinots.
    - Fer tornar al niu als borinots que porten pol·len, augmentant-li la vida en 10 unitats.
    - Mutar els borinots.
    - Restar vida als borinots.
    - Buscar el millor individu.
        - Si és la solució o s'ha arribat al màxim de generacions→sortir.
        - Si no és així, tornar a començar generacions.

(Cada 40 generacions, surt al món un altre borinot que prové de la reina)

Fi 20 proves
Calcular la mitja de vegades que ho aconsegueix, i escriure-ho a fitxer.
Fi

5.3.1. Generar el graf aleatori

Es genera un graf de forma aleatòria a partir del nombre de vèrtexs, definit a la capçalera, i el nombre de branques que introdueix l'usuari.

5.3.2. Crear el món

Els borinots es mouen per un món que és una matriu quadrada (20 x 20 cel·les) que poden estar en 3 estats diferents:

- Buida: si no hi és el niu ni hi ha cap borinot.

- Niub: si es troba el niu de borinots. La cel·la té 3 camps:
    - nborinots: indica el nombre de borinots que hi ha al niu en aquell instant.
    - insecte: conté el borinot en qüestió.
    - Reina: conté el borinot amb major pes.

- Borinot: si hi ha un borinot. La cel·la té 6 camps:
    - cromosoma: representa la configuració de l'individu.
    - fitness: mesura quant de bo és un cromosoma.
    - Vida: el nombre de generacions que sobreviurà el borinot.
    - P: identificador de si porta pol·len o no.

- Ruta_x i ruta_y: marquen una guia al borinot per trobar pol·len.

La matriu quadrada on es mouen els borinots descriu una superfície toroidal. D'aquesta manera no hi ha problemes quan es realitzen moviments als extrems i la mobilitat dels individus és més aleatòria.

Cada borinot representa solucions que s'inicialitzen amb una combinació de colors aleatòria. El fitness i la vida de cadascun d'ells depenen de quant de bo és l'individu. Un borinot que acompleixi moltes restriccions tindrà més possibilitats de viure, ja que li seran assignats més fitness i més vida.

5.3.3. Generacions

El bucle de les generacions no finalitza fins que es troba la solució òptima o s'arriba a un màxim de generacions.

5.3.3.1. Moure els borinots

Tots els borinots es troben al niu en l'inici del programa.

En un primer moment es mouen aleatòriament. El grau del moviment és 2, és a dir, es poden moure per les 24 caselles collindans, sempre que alguna d'aquestes no estigui ocupada.

2				
	1			
		B		

Fig. 5.2: Representació del grau del moviment.

Si un borinot troba pol·len, es guarda la ubicació d'aquest (ruta_x i ruta_y) i torna al niu incrementant la seva vida en dues unitats. Quan surti un altre cop del niu, el moviment no serà aleatori, ja que hi haurà més probabilitat a que torni allà on va trobar pol·len.

En cas que s'esgoti el pol·len a la casella on hi era, es regenerarà el pol·len en una altra casella escollida aleatòriament i el borinot no es guardarà les coordenades.

5.3.3.2. Mutació

La mutació dels individus és, realment, la que provoca que hi hagi evolució en el programa perquè ens ha d'aportar el camí cap a la millor solució.

Aquesta acció consisteix en canviar un color per un altre, sempre que l'últim proporcioni alguna millora en l'individu.

El procés de mutació, per cada borinot, és el següent:

- Seleccionem el vèrtex a mutar i guardem el seu color

- Calculem la contribució d'aquest color al total de l'energia de l'individu.

- Escollim un color a l'atzar i calculem la contribució d'aquest.

- Comparem els resultats de les dues contribucions, la nova i la vella.
    - Si el primer color comporta una major energia, no fem cap canvi.
    - Si el segon color millora l'individu, canviem el color. Modifiquem també el fitness de l'individu i augmentem la vida en funció d'aquest.

D'aquesta manera reduïm el temps de càlcul de la CPU notablement, ja que evitem realitzar la comparació amb tots els vèrtexs del graf.

5.3.3.3. Restar vida

Com ja hem dit anteriorment, com més energia tingui l'individu, més possibilitats de viure tindrà.

A cada generació, a part de moure els borinots i mutar-los, el programa resta la seva vida en una unitat.

Quan un borinot arriba a vida 0, desapareix i deixa buit el seu lloc.

El programa també fa desaparèixer els "pitjors" borinots en quant a fitness. D'aquesta manera ens estalviem treballar amb borinots que mai no portaran a qualsevol solució. Així doncs, els que tinguin el 40% de les branques desfavorables, seran també eliminats a cada generació.

5.3.3.4 Treure borinot de la reina

La reina guardarà inicialment les solucions (el fitness i la configuració del cromosoma) dels 5 primers borinots que es creen. Però anirà emmagatzemant les 5 millors solucions d'entre tots els borinots que tornen al niu. Cada 40 generacions, una de les solucions guardada per la reina serà utilitzada per afegir un borinot amb aquest fitness al món de manera aleatòria. Això provoca una millora considerable al programa.

6. Resultats obtinguts

Per tal de poder contrastar els resultats obtinguts amb el mètode dels borinots, hem fet proves amb 2 mètodes més.

El primer és el mètode d'àngels i mortals proposat per Sorin Salomon, del qual ja s'ha parlat en el tercer capítol. Cal dir que es tracta d'una variant del mètode d'àngels i mortals. Les diferències són les següents:

- El món s'inicialitza amb el màxim nombre d'individus, 300, amb un cert grau de mobilitat. Recordem que el nombre inicial d'individus en el mètode d'àngels i mortals era 5.

- Quan un mortal es troba a la vora d'un àngel, se li allarga la seva vida en 6 unitats en lloc de duplicar-lo.

El segon és un algorisme genètic, amb algunes diferències respecte el que s'ha explicat sobre aquests algorismes. Existeix un nou paràmetre, supervivents, que s'encarrega de seleccionar els millors individus per passar-los directament a la següent generació. Aquests individus s'encreuaran i generaran la resta d'individus que falten per arribar al total d'individus de la nova generació. Les característiques d'aquest algorisme són:

- Població: 600 individus.
- Supervivents: 450 individus.
- Repòs (nombre de generacions en les que no es millora el fitness de l'individu): 100.
- Probabilitat d'encreuament: 0.9.
- Probabilitat de mutació: 0.001.

Les proves realitzades són sobre grafs de 30, 50, 70, 100 i 200 vèrtexs. També s'ha provat, per cada graf, densitats del 10% i 20%.

Els paràmetres amb els quals fem la comparació són el nombre de colors necessaris, el nombre de generacions i el nombre de vegades que troba la solució. Però també s'ha afegit una altra columna amb el temps mitjà de les generacions (Això és degut a que les generacions de l'algorisme genètic no comporten el mateix que les generacions d'àngels i mortals i de borinots.

Els resultats s'han obtingut a partir d'una mitjana de 20 simulacions per a cadascun dels casos. És molt probable que no s'obtinguin els mateixos resultats en dues o més simulacions amb els de les mateixes característiques, ja que el moviment que descriuen els borinots depèn de certes probabilitats i la col·locació del pol·len és totalment aleatòria.

Els resultats són el següents:

AM: Variant d'àngels i mortals
GA: Algorisme genètic
B: Borinot

Taula 6.1. Densitat del 10%.

Vèr	Brq.	Colors			Temps (s)			Generac.			%(20)		
		AM	GA	B	AM	GA	B	AM	GA	B	AM	GA	B
30	44	3	3	2	0.017	0.081	0.119	43	22	113	20	10	20
50	123	4	4	2	0.127	0.394	0.377	256	85	306	19	9	20
70	242	5	5	3	0.278	0.902	0.3955	672	159	325	19	3	19
100	495	6	6	4	0.682	2.01	0.5026	1875	206	431	7	2	18
200	1990	12	12	10	1.663	5.25	0.71	3465	201	347	17	5	17

Taula 6.2. Densitat del 20%.

Vèr	Brq.	Colors			Temps (s)			Generac.			%(20)		
		AM	GA	B	AM	GA	B	AM	GA	B	AM	GA	B
30	87	4	4	2	0.056	0.21	0.175	167	84	201	19	8	20
50	245	6	6	3	0.203	0.633	0.329	503	115	280	19	8	19
70	483	7	8	3	0.542	1.11	0.4732	1524	229	522	13	5	19
100	990	10	11	7	0.812	2.06	0.486	2155	166	340	16	8	18
200	3980	19	20	19	2.051	6.19	0.79	4751	390	503	16	6	15

Com es pot observar, la qualitat de resultats amb el nou algorisme dels borinots és millor que la dels altres mètodes: millor acoloriment amb major convergència de resultats.

Tenint en compte que el nombre d'individus de l'algorisme genètic és el triple del nombre màxim d'individus en l'algorisme dels borinots, els resultats obtinguts amb el aquest algorisme genètic haurien de ser millors que els obtinguts amb els altres dos.

Aquest nou mètode ens ofereix uns resultats molt bons. El fet d'anar afegint una còpia d'un dels millors individus evita la convergència en mínims locals. Com que ho fem cada 40 generacions, no es generen molts individus semblants i d'aquesta manera no es redueix la velocitat de convergència.

Els resultats obtinguts amb el mètode d'àngels i mortals i amb el mètode dels borinots són millors que en l'algorisme genètic. Per una banda el món d'*AM* es crea directament amb 300 individus que es van eliminant a mesura que s'acaba

la seva vida o no són favorables. En els borinots, el món és de 200 individus, però van sortint del niu progressivament de manera que mai hi ha 200 individus al món. El fet d'anar afegint borinots contraresta aquest factor.

7. Conclusions

En aquest treball fi de carrera s'ha dissenyat un nou mètode d'optimització adequat pe a molts problemes combinatoris. L'hem provat en el cas d'acoloriment de grafs, problema equivalent a l'assignació eficient de freqüències en xarxes GSM.

Aquest nou mètode es basa en el comportament social dels borinots: L'evolució en un món virtual d'una població de borinots (cadascun del quals s'associa a una possible solució del problema) porta a les solucions desitjades.

Després d'haver realitzat les proves corresponents i haver analitzat els resultats obtinguts, veiem que és un mètode d'optimització eficaç i eficient. S'ha de tenir en compte que, a mesura que puja el nombre de vèrtexs i/o el nombre de branques, el programa segueix trobant solucions al problema, però no amb tanta velocitat. En comparació amb un algorisme genètic i el mètode d'àngels i mortals, el mètode dels borinots realitza un millor acoloriment i convergeix més ràpidament en la majoria dels casos.

Encara que no ha donat temps a provar l'algorisme en una xarxa GSM real, cal dir que no seria gens complicat fer-ho. Al moment de construir el graf , en lloc de generar-lo aleatòriament, hauríem de considerar la matriu de restriccions (interferències) i enllaçar els vèrtexs que ens indiqués aquesta matriu, associant un pes a cada branca d'acord amb el seus valors.

Finalment cal esmentar que el mètode podria adaptar-se fàcilment a la resolució de qualsevol problema d'optimització combinatòria.

BIBLIOGRAFIA

[1] Abril, J., Comellas, F., Cortés, A., Ozón, J., Vaquer, M., *A multi-agent system for frequency assignment in cellular radio networks*. *IEEE Trans. Vehic. Technology* ,vol. 49 (No. 5) (2000) 1558-1565.

[2] Bonabeau, E., Dorigo, M., & Theraulaz, G., *Inspiration for optimization from social insect behaviour*. *Nature* 406 (2000) 39-42.

[3] Comellas, F., Gallegos R., *Angels & mortals: A new combinatorial optimization algorithm*. *Studies in Fuzziness and Soft Computing*. vol. 166, pp. 397-405, 2004.

[4] Garey, M. R., Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: W.H. Freeman, 1979, ISBN.

[5] Holland., J. H, Genetic algorithms, *Scientific American*, vol. 267, pp. 44-50, 1992.

[6] Local Search in Combinatorial Optimization. Eds. E. Aarts and J.K. Lenstra. Wiley-Interscience Series in Discrete Mathematics and Optimization., 1997. ISBN.

[7] Shnerb, N. M., . Louzoun Y., Bettelheim, E., Solomon, S. *The importance of being discrete: Life always wins on the surface*. *Proc. Natl. Acad. Sci. USA*, vol. 97, pp. 10322-10324, 2000.

[8] *The bumblebee pages*. (http://www.bumblebee.org/lifecycle.htm). Accedit el 21 de Juny de 2005.

[9] Alonso, S., Cordón, A., Fernández, I., Herrera, F., *La Metaheurística de Optimización Basada en Colonias de Hormigas: Modelos y Nuevos Enfoques* pp. 5-6. (http://sci2s.ugr.es/publications/ficheros/OCH%20Modelos%20y%20Nuevos%20Enfoques%20(Chapter).pdf). Accedit el 21 de juny de 2005

[10] *El portal de las telecomunicaciones* (http://telecom.iespana.es/telecom/telef/gsm-info.htm). Accedit el 21 de juny de 2005

ANNEX

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <conio.h>

#define VERTEXS             100
#define MAXFILES    20
#define MAXCOLUMNES     20
#define INSECTES    200
#define MAXPOLEN  40
#define AGOTAT              0    /*valor que controlarà quan s'acaba el pol·len*/
#define BORINOT             -3
#define NIUB            -2
#define BUIT            -1
#define MAXGRAU             300
#define VIDA_MAX     3     /* vida en funcio del fitness de cadascu el valor
                                                    de VIDA_MAX indica la proporcio
que té el millor
                                                    individu amb el total de generacions
que viurà*/
#define MESVIDA                 2
#define MAXGENERACIONS  10000
#define MAXBRANQUES             5000
#define COPS                20
#define SOLUCIONS           5       /*la reina contindrà 5 solucions possibles, les 5
millors del món*/

#define MATAR_VIDA   0   /* 0..0.2< matar*branques es mata*/
#define MATAR_FITNESS  0.4   /*per matar els "pitjors individus*/

/* funcio que genera nombres aleatoris entre 0 i un maxim */
#define RANDOM(c) (int)(rand()/(RAND_MAX*1.0+1)*(c))
#define CENTRAL             10      /*el niu estarà situat a la cel·la central*/
#define RANG        24
typedef int COLOR;
typedef COLOR CROMOSOMA[VERTEXS]; /* cromosoma = llista de colors */

int branques=0,ncolors=0,sortir=0,millor[3],borinots=0,grafix[VERTEXS][MAXGRAU];
char buffer[1000],caracters[1000];

typedef struct      /* cada graf (borinot) ve representat per: */
{
        CROMOSOMA adn;   /* cromosoma que representa una configuracio*/
  int fitness;     /* mesura quant de bo es un cromosoma */
  int vida;                 /* mesura el no de generacions que sobreviura */
  int p;                    /*identificador per saber si porta polen(1) o no (0)*/
  int     ruta_x;      /*rut_x i ruta_y ens serviran per marcar la trajectòria del borinot*/
  int ruta_y;
} INSECTE;
```

```
typedef struct       /*la reina vindrà representada per:*/
{
        CROMOSOMA adn;   /* cromosoma que representa vàries configuracions*/
  int fitness;     /* mesura quant de bo es un cromosoma */
  int vida;                /* mesura el no de generacions que sobreviura */
} REINA;

typedef struct /*El niu central estarà format per 1 reina i 20 borinots*/
{
        REINA reina[SOLUCIONS];
        INSECTE                 borinot[INSECTES];   /*vector dels borinots que hi ha al
niu*/
        int nborinots;/*controla els borinots que hi ha al niu*/

}NIU;


typedef struct  /* cel·la del món */
{
  int estat;                /* pot contenir un borinot,estar buida o contenir el niu */
  INSECTE borinot;   /*en cas que sigui borinot, aquí posem l'insecte */
  NIU niu;                /*en cas que hi hagi niu, és a dir, a la casella central*/
  int    polen; /*en cas que hi hagi polen*/
} cella;

typedef cella MON[MAXFILES][MAXCOLUMNES];  /*matriu de cel·les */
MON mon;


/*****************************************************************************/

void genera_graf(int numBranques)       /* genera un graf aleatori */
{
 int i=0,j=0,k,grauorigen,desti,graudesti;

 for(k=0;k<VERTEXS;k++)   grafix[k][0]= -33;
 while(i<numBranques)
 {

    desti=j+1+RANDOM(VERTEXS-2-j);
    grauorigen=0;
    while ( grafix[j][grauorigen] !=-33 && grauorigen < MAXGRAU)
        grauorigen++;
    if (grafix[j][grauorigen] == -33)
    {                   /* per saber on acaben els vertexs */
        grafix[j][grauorigen]=desti;
            grafix[j][grauorigen+1]= -33;
    }
    graudesti=0;
    while ( grafix[desti][graudesti] !=-33 && graudesti < MAXGRAU)
        graudesti++;
    if (grafix[desti][graudesti] == -33)
    {       /* coloca al vertex de desti el seu corresponent de origen */
        grafix[desti][graudesti]= j;
        grafix[desti][graudesti+1]=-33;
```

```
    }
    i++;
    j++;
    if (grafix[desti][graudesti] == MAXGRAU || grafix[j][grauorigen]== MAXGRAU)
    {  /* en cas que no es pugui crear el graf amb éxit */
        printf("\nel graf no s'ha pogut crear, prova amb pujar MAXGRAU");
    }
    if (j==VERTEXS-2)
    {
        j=0;
    }
  }
}
/*****************************************************************************/
/* funcions auxiliars que ens defineixen la superficie toroidal */

int fi_fila(int fila)
{

        if(fila == MAXFILES)       return 0;
        if(fila<0)             return (MAXFILES+fila);
        return fila;
}
int fi_columna(int columna)
{
if(columna == MAXCOLUMNES)  return 0;
if(columna<0)          return (MAXCOLUMNES+columna);
return columna;
}


/*****************************************************************************/

int assigna_vida(int fitness)
{       /* dona vida al individu en funcio del fitness*/
return((fitness*100/branques)*VIDA_MAX);
}

/*****************************************************************************/

int energia(CROMOSOMA adn)  /* torna l'energia d'un cromosoma */
{
  int bones=0;
  int i,grau;
  for (i=0;i<VERTEXS;i++)
  {
   grau=0;
   while(grafix[i][grau]!= -33 )
   {
        if(adn[i]!=adn[grafix[i][grau]])
    {    /* bones esta per duplicat, ja que compta a origen i desti*/
            /* caldra tornar bones/2 */
        bones++;
    }
    grau++;
```

```
   }
  }
  if(bones<=0)
  {
                printf("\nNO ES POT INICIAR EL PROGRAMA!! (individu sense
energia)");
                exit(1);
  }
  return (bones/2); /* nombre de branques bones , cal que sigui màxim*/
}

/*****************************************************************************************/

void crear_mon(void) /* emplena el mon fisic amb el niu de borinots*/
{
        int i,j,k,l,t,x,y;

        mon[CENTRAL][CENTRAL.estat=NIUB;
        mon[CENTRAL][CENTRAL.niu.nborinots=INSECTES;
        for (l=0;l<INSECTES;l++)      /*aquest bucle crea tots els borinots al niu*/
        {
                for(k=0;k<VERTEXS;k++)
                {

        mon[CENTRAL][CENTRAL.niu.borinot[l].adn[k]=RANDOM(ncolors);
                }
                k = mon[CENTRAL][CENTRAL.niu.borinot[l].fitness =
energia(mon[CENTRAL][CENTRAL.niu.borinot[l].adn);
                mon[CENTRAL][CENTRAL.niu.borinot[l].vida = assigna_vida(k);
                mon[CENTRAL][CENTRAL.niu.borinot[l].ruta_x=0;
                mon[CENTRAL][CENTRAL.niu.borinot[l].ruta_y=0;
                mon[CENTRAL][CENTRAL.niu.borinot[l].p=0;
                borinots++;

        }




        for(t=0;t<SOLUCIONS;t++)    /*La reina guarda inicialment les configuracions
dels 5 primers
                                                borinots, però durant el
programa guardarà les 5 millors dels que
                                                vagin tornant al niu*/
        {
                for(k=0;k<VERTEXS;k++)
                {

        mon[CENTRAL][CENTRAL.niu.reina[t].adn[k]=mon[CENTRAL][CENTRAL.niu.
borinot[t].adn[k];
                }


        mon[CENTRAL][CENTRAL.niu.reina[t].fitness=mon[CENTRAL][CENTRAL.niu.
borinot[t].fitness;
```

```
        }


        for(i=0;i<MAXFILES;i++)   /*posa a BUIT tot el món, exceptuant el niu,
                                                                que es troba a la casella
central*/
  {
        for (j=0;j<MAXCOLUMNES;j++)
        {
                if (mon[i][j].estat!=NIUB)
                {
                        mon[i][j].estat=BUIT;
                        mon[i][j].polen=0;
                }
        }
  }
  i=1;
  while(i<=MAXPOLEN)
  {
                x = RANDOM(MAXFILES);   /*situa el pol·len al món aleatòriament*/
                y = RANDOM(MAXCOLUMNES);
                if (mon[x][y].polen==0)
                {
                        mon[x][y].polen = 5;
                        i++;
                }
        }


}


/*****************************************************************************************/

void mutacio (void) /* si l'energia del nou és pitjor,no es muta */
{
        int x,y,i,j,color,noucolor,vertex,bones_anterior,bones_nou,desti;
        for(x=0;x<MAXFILES;x++)
        {
                for(y=0;y<MAXCOLUMNES;y++)
                {
                        if(mon[x][y].estat == BORINOT)
                        {
                                vertex = RANDOM(VERTEXS);  /* tria el vertex a mutar */
                                color=mon[x][y].borinot.adn[vertex]; /* color inicial abans
de mutar*/
                                do (noucolor= RANDOM(ncolors)); while
(noucolor==color);
                                bones_anterior=bones_nou=i=0;
                 /* s'avalua la contribució de cada color al total d'energia */
                                while(grafix[vertex][i]!=-33)
                                {
                                        desti=grafix[vertex][i]; /*desti*/
```

```
                                        if (color != mon[x][y].borinot.adn[desti])
                                                bones_anterior++;
                                        if (noucolor != mon[x][y].borinot.adn[desti])
                                        bones_nou++;
                                        i++;
                                }
                                if (bones_anterior >= bones_nou) {if (RANDOM(100) > 1)
break;}
                                else  /* si el nou és millor, canvia el color, la energia i vida
*/
                                {
                                mon[x][y].borinot.adn[vertex]= noucolor;
                                mon[x][y].borinot.fitness = mon[x][y].borinot.fitness +
bones_nou - bones_anterior;
                                mon[x][y].borinot.vida =
assigna_vida(mon[x][y].borinot.fitness)-mon[x][y].borinot.vida;
                                }
                        }
                }
        }


        /*Mutació de la reina*/

        for(j=0;j<SOLUCIONS;j++)
        {
                vertex=RANDOM(VERTEXS);
                color=mon[CENTRAL][CENTRAL].niu.reina[j].adn[vertex];
                do (noucolor= RANDOM(ncolors)); while (noucolor==color);
                bones_anterior=bones_nou=i=0;
                while(grafix[vertex][i]!=-33)
                {
                        desti=grafix[vertex][i]; /*desti*/
                        if (color != mon[CENTRAL][CENTRAL].niu.reina[j].adn[desti])
                                bones_anterior++;
                        if (noucolor != mon[CENTRAL][CENTRAL].niu.reina[j].adn[desti])
                        bones_nou++;
                        i++;
                }
                if (bones_anterior >= bones_nou) {if (RANDOM(100) > 1) break;}
                else  /* si el nou és millor, canvia el color, l'energia i la vida */
                {
                        mon[CENTRAL][CENTRAL].niu.reina[j].adn[vertex] = noucolor;
                        mon[CENTRAL][CENTRAL].niu.reina[j].fitness =
mon[CENTRAL][CENTRAL].niu.reina[j].fitness+ bones_nou - bones_anterior;
                        mon[x][y].borinot.vida = assigna_vida(mon[x][y].borinot.fitness)-
mon[x][y].borinot.vida;
                }
        }


}

/*****************************************************************************************/
```

```
void restar_vida(void)  /* restem una vida de cada borinot*/
{
  int x,y,fotut;
  for(x=0;x<MAXFILES;x++)
  {
  for(y=0;y<MAXCOLUMNES;y++)
  {
    if (mon[x][y].estat== BORINOT)
    {
                        mon[x][y].borinot.vida--;
      fotut=mon[x][y].borinot.fitness;
                        if( mon[x][y].borinot.vida <=(MATAR_VIDA*branques) || fotut
<(branques*MATAR_FITNESS))
                        {
                                mon[x][y].estat= BUIT;
                               borinots--;
                        }
        }
  }
  }
}

/*****************************************************************************************/

void buscar_millor(void) /*busca el millor borinot per comprovar si és la solució*/
{
        int x,y;
        for(x=0;x<MAXFILES;x++)
        {
                for(y=0;y<MAXCOLUMNES;y++)
                {
                        if(mon[x][y].estat == BORINOT)
                        {
                                if(mon[x][y].borinot.fitness >= millor[0])
                                {
                                        millor[0]= mon[x][y].borinot.fitness;
                                        millor[1]=x; /*guardem la posició del millor dins el
món*/
                                        millor[2]=y;
                                }
                        }

                }
        }
}

/*****************************************************************************************/

/*La següent funció escriu els resultats a un fitxer*/
void escriu_a_fitxer(int branques,int colors,int generacions)
{
  int i,cont=0;
  char auxiliar[5];
  itoa(branques,auxiliar,10);               /* branques */
  i=0;
```

```
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';
  cont++;
  itoa(colors,auxiliar,10);                    /* colors */
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';
  cont++;
  itoa(generacions,auxiliar,10);               /* generacions */
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';
  cont++;                                      /* millor energia*/
  itoa(mon[millor[1]][millor[2]].borinot.fitness,auxiliar,10);
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';

  cont++;

/*  for (i=0; i<VERTEXS; i++)                  /* solucio */
/*  {
    buffer[cont]= (char)(mon[millor[1]][millor[2]].individu.adn[i]+'A');
    buffer[cont+1]= '\t';
    cont+=2;
  }*/

  buffer[cont]= '\n';

}

/*************************************************************************/

/*La següent funció escriu en un fitxer les mitjanes dels resultats*/
void escriu_fitxer_mitja (int branques,int colors,int cops,int mitja,int nopot)
```

```
{
int i,cont=0;
  char auxiliar[5];
  itoa(branques,auxiliar,10);                /* branques */
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';
  cont++;
  itoa(colors,auxiliar,10);                  /* colors */
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';
  cont++;
  itoa(cops,auxiliar,10);                /* cops */
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';
  cont++;
  itoa(mitja,auxiliar,10);                  /* mitja */
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]= '\t';
  cont++;
  itoa(nopot,auxiliar,10);               /* nopot */
  i=0;
  while(auxiliar[i]!='\0')
  {
    buffer[cont]= auxiliar[i];
    i++;
    cont++;
  }
  buffer[cont]='\n';
  cont++;
}
```

```
/*****************************************************************************************/

void treu_borinot_niu(int i,int j)     /*funció que treu un determinat borinot del niu cap
                                                                        a la casella
(i,j)*/
{
        int c=0,x,y,m;

        mon[i][j].estat=BORINOT;
        mon[i][j].borinot.fitness=mon[CENTRAL][CENTRAL].niu.borinot[0].fitness;
        mon[i][j].borinot.vida=mon[CENTRAL][CENTRAL].niu.borinot[0].vida;
        mon[i][j].borinot.p=0;
        mon[i][i].borinot.ruta_x=mon[CENTRAL][CENTRAL].niu.borinot[0].ruta_x;
        mon[i][j].borinot.ruta_y=mon[CENTRAL][CENTRAL].niu.borinot[0].ruta_y;
        for (m=0;m<VERTEXS;m++)

        mon[i][j].borinot.adn[m]=mon[CENTRAL][CENTRAL].niu.borinot[0].adn[m];


        while (c<mon[CENTRAL][CENTRAL].niu.nborinots-1)/*amb aquest bucle
deplacem la cua al niu

per treure el borinot que ha sortit*/
        {

        mon[CENTRAL][CENTRAL].niu.borinot[c].fitness=mon[CENTRAL][CENTRAL].n
iu.borinot[c+1].fitness;

        mon[CENTRAL][CENTRAL].niu.borinot[c].vida=mon[CENTRAL][CENTRAL].niu.
borinot[c+1].vida;
                mon[CENTRAL][CENTRAL].niu.borinot[c].p=0;

        mon[CENTRAL][CENTRAL].niu.borinot[c].ruta_x=mon[CENTRAL][CENTRAL].n
iu.borinot[c+1].ruta_x;

        mon[CENTRAL][CENTRAL].niu.borinot[c].ruta_y=mon[CENTRAL][CENTRAL].n
iu.borinot[c+1].ruta_y;
                for (m=0;m<VERTEXS;m++)

        mon[CENTRAL][CENTRAL].niu.borinot[c].adn[m]=mon[CENTRAL][CENTRAL].
niu.borinot[c+1].adn[m];
                c++;
        }
        if (mon[i][j].polen!=AGOTAT) /*si el borinot troba pol.len:*/
        {
                mon[i][j].borinot.p=1;
                mon[i][j].polen--;
                if (mon[i][j].polen==AGOTAT) /*si s'acaba el pol·len caldrà generar-lo de
nou*/
                {
                        x = RANDOM(MAXFILES);   /*situa el pol·len al món
aleatòriament*/
                        y = RANDOM(MAXCOLUMNES);
                        mon[i][j].polen+=5;
                }
```

```
        }


        mon[CENTRAL][CENTRAL].niu.nborinots--;
}
/*********************************************************************************************/

void treu_borinot_reina(void)
{
        int x=RANDOM(MAXFILES);
        int y=RANDOM(MAXCOLUMNES);
        int j,k,l;

        j=RANDOM(SOLUCIONS);
        while(mon[x][y].estat!=BUIT)
        {
                x=RANDOM(MAXFILES);
                y=RANDOM(MAXCOLUMNES);
        }
        mon[x][y].estat=BORINOT;
        for(k=0;k<VERTEXS;k++)
        {
                mon[x][y].borinot.adn[k]=mon[CENTRAL][CENTRAL].niu.reina[j].adn[k];
        }
        l=mon[x][y].borinot.fitness=mon[CENTRAL][CENTRAL].niu.reina[j].fitness;
        mon[x][y].borinot.vida=assigna_vida(l);
        mon[x][y].borinot.ruta_x=0;
        mon[x][y].borinot.ruta_y=0;
        mon[CENTRAL][CENTRAL].borinot.p=0;
}


/*********************************************************************************************/


void torna_borinot(int i, int j) /*funció que torna al borinot a la darrera posició del niu*/
{
        int x=0,m,k,t;
        int copia[MAXFILES][MAXCOLUMNES];
        while(x<mon[CENTRAL][CENTRAL].niu.nborinots)
        {
                x++;
        }

        mon[CENTRAL][CENTRAL].niu.nborinots++;
        mon[CENTRAL][CENTRAL].niu.borinot[x].fitness=mon[i][j].borinot.fitness;
        if(mon[i][j].borinot.fitness<MATAR_FITNESS*branques)
                mon[CENTRAL][CENTRAL].niu.borinot[x].vida++;
        else
                mon[CENTRAL][CENTRAL].niu.borinot[x].vida+=5;

        mon[CENTRAL][CENTRAL].niu.borinot[x].vida=mon[i][j].borinot.vida+MESVIDA;
        for (m=0;m<VERTEXS;m++)
```

```
        mon[CENTRAL][CENTRAL].niu.borinot[x].adn[m]=mon[i][j].borinot.adn[m];
        mon[CENTRAL][CENTRAL].niu.borinot[x].p=0;
        mon[CENTRAL][CENTRAL].niu.borinot[x].ruta_x=mon[i][j].borinot.ruta_x;
        mon[CENTRAL][CENTRAL].niu.borinot[x].ruta_y=mon[i][j].borinot.ruta_y;
        mon[i][j].estat=BUIT;
        copia[i][j]=BUIT;

        /*A continuació mirem si la solució que porta el borinot és
                millor que les que porta la reina*/

        for (k=0;k<SOLUCIONS;k++)
        {

        if(mon[CENTRAL][CENTRAL].niu.borinot[x].fitness>mon[CENTRAL][CENTRAL]
.niu.reina[k].fitness)
                {
                        t=k;
                        while(k<SOLUCIONS)
                        {

        mon[CENTRAL][CENTRAL].niu.reina[k+1].fitness=mon[CENTRAL][CENTRAL].
niu.reina[k].fitness;
                                k++;
                        }

        mon[CENTRAL][CENTRAL].niu.reina[t].fitness=mon[CENTRAL][CENTRAL].niu.
borinot[x].fitness;
                }
        }
}


/*********************************************************************************************/

int prob(int x,int y,int j, int k) /*aquesta funció ens serveix per afegir probabilitat
                                                        al moviment del borinot
segons la ruta que tingui assignada*/
{
int r,moviment;
        r=RANDOM(100);

        if (r>=25 && r<=75)
        {
                moviment=x;
        }

        else if (r>=0 && r<25)
        {
                moviment=y;
        }

        else if (r>75 && r<=90)
        {
                moviment=j;
```

```
        }
        else if (r>90 && r<=100)
        {
                moviment=k;
        }

        return (moviment);
}

/*****************************************************************************************/

int resta(int a, int b)
{
        return(a-b);
}

/*****************************************************************************************/

void moure(void)    /* mou borinots pel món */
/*si hi ha algun borinot al niu, el treu cap a on hi hagi rastre, si no està ocupada.
si no, aleatòriament (sempre amb rang 2).
Després, mirem cadascun dels borinots i procurem que segueixin la ruta, tenint en
compte
si porta polen o no, ja que en cas de que porti haurà de tornar al niu*/

{
        int copia[MAXFILES][MAXCOLUMNES];
        int i,j,k,l,m,x,y,fila,r,columna,difx,dify,sortir=0,moviment;
        /*fem la còpia per a saber els borinots que hem mogut i els que no */
        srand(time(NULL));
        for(k=0;k<MAXFILES;k++)
        {
                for(l=0;l<MAXCOLUMNES;l++)
                {
                        copia[k][l]=mon[k][l].estat;
                }
         }
        /*recorrem la còpia i busquem els borinots */
        for(k=0;k<MAXFILES;k++)
        {
                sortir=0;
                for(l=0;l<MAXCOLUMNES;l++)
                {
                        sortir=0;

                        if ((copia[k][l]== BORINOT) && (mon[k][l].borinot.p==0))   /* en
cas que sigui un borinot sense pol·len */
                        {

                                for(i=k-2;i<=k+2;i++) //recorrem els voltants del niu amb
rang 2 per moure
                                {                                       //si hi ha pol·len
disponible

                                        if (sortir) break;
```

```
                                        for(j=l-2;j<=l+2;j++)
                                        {
                                                if(copia[fi_fila(i)][fi_columna(j)]==BUIT &&
mon[fi_fila(i)][fi_columna(j)].estat==BUIT &&
mon[fi_fila(i)][fi_columna(j)].polen>AGOTAT)
                                                {
                                                        fila=i;
                                                        fila=fi_fila(fila);
                                                        columna=j;
                                                        columna=fi_columna(columna);

        mon[fila][columna].estat=BORINOT;/* posem el borinot */
                                                        mon[fila][columna].borinot.p=1;
                                                        mon[fila][columna].polen--;
                                                        for (m=0;m<VERTEXS;m++)

        mon[fila][columna].borinot.adn[m]=mon[k][l].borinot.adn[m];

                                                        if
(mon[fila][columna].polen==AGOTAT)
                                                        {
                                                                x = RANDOM(MAXFILES);
/*situa el pol·len al món aleatòriament*/
                                                                y =
RANDOM(MAXCOLUMNES);
                                                                mon[x][y].polen+=5;

        mon[fila][columna].borinot.ruta_x=0;

        mon[fila][columna].borinot.ruta_y=0;
                                                        }
                                                        else
                                                        {

        mon[fila][columna].borinot.ruta_x=fila;/*guardem la ruta*/

        mon[fila][columna].borinot.ruta_y=columna;
                                                        }

        mon[fila][columna].borinot.vida=mon[k][l].borinot.vida;

        mon[fila][columna].borinot.fitness=mon[k][l].borinot.fitness;
                                                        mon[k][l].estat=BUIT;
/*borrem el borinot i la còpia*/
                                        copia[k][l]= BUIT;

                                                        sortir=1;
                                                }
                                                if (sortir) break;
                                        }
                                }

                                if (copia[k][l]== BORINOT && mon[k][l].borinot.p==0 &&
                                   mon[k][l].borinot.ruta_x==0 &&
mon[k][l].borinot.ruta_y==0)
```

```
                        {  /*moviment totalment aleatori en cas que no hi hagi
ruta*/
                                i=0;
                                while(i<RANG)
                                {
                                        i++;
                                        r=RANDOM(4);
                                        if(r==0) fila=k-2;
                                        if(r==1) fila=k-1;
                                        if(r==2) fila=k+1;
                                        if(r==3) fila=k+2;

                                        r=RANDOM(4);
                                        if(r==0) columna=l-2;
                                        if(r==1) columna=l-1;
                                        if(r==2) columna=l+1;

                                        if(r==3) columna=l+2;
                                        fila=fi_fila(fila);
                                        columna=fi_columna(columna);
                                        if(mon[fila][columna].estat==BUIT &&
copia[fila][columna]==BUIT)
                                        {
                                                mon[fila][columna].estat=BORINOT;

        mon[fila][columna].borinot.fitness=mon[k][l].borinot.fitness;

        mon[fila][columna].borinot.vida=mon[k][l].borinot.vida;

        mon[fila][columna].borinot.p=mon[k][l].borinot.p;/* posem el borinot */
                                                for (m=0;m<VERTEXS;m++)

        mon[fila][columna].borinot.adn[m]=mon[k][l].borinot.adn[m];

        mon[fila][columna].borinot.ruta_x=mon[k][l].borinot.ruta_x;

        mon[fila][columna].borinot.ruta_y=mon[k][l].borinot.ruta_y;
                                                mon[k][l].estat=BUIT;
/*borrem el borinot i la còpia*/
                                                copia[k][l]= BUIT;
                                                sortir=1;
                                        }
                                        if (sortir) break;
                                }
                        }


                        else if ((copia[k][l]==BORINOT) &&
(mon[k][l].borinot.p==0) &&

(mon[k][l].borinot.ruta_x!=0||mon[k][l].borinot.ruta_y!=0))
                        {   /*probabilitat més alta per seguir la ruta*/
                                sortir=0;
                                j=0;
                                while (j<RANG)
```

```
{
        j++;
        difx=resta(k,mon[k][l].borinot.ruta_x);
        dify=resta(l,mon[k][l].borinot.ruta_y);
        if (difx>=2)    //repartim les probabilitats de
moviment en funció de la distància
        {
                moviment=prob(-2,-1,1,2);
        }
        else if (difx<=-2)
        {
                moviment=prob(2,1,-1,-2);
        }
        else if (difx==1)
        {
                moviment=prob(-1,-2,1,2);
        }
        else if (difx==-1)
        {
                moviment=prob(1,2,-1,-2);
        }
        else
                moviment=RANDOM(4);

        fila=k+moviment;
        fila=fi_fila(fila);

        if (dify>=2)      //repartim les probabilitats en
funció de la distància
        {
                moviment=prob(-2,-1,1,2);

        }
        else if (dify<=-2)
        {
                moviment=prob(2,1,-1,-2);
        }
        else if (dify==1)
        {
                moviment=prob(-1,-2,1,2);
        }
        else if (dify==-1)
        {
                moviment=prob(1,2,-1,-2);
        }
        else
                moviment=RANDOM(4);
        columna=l+moviment;
        columna=fi_columna(columna);
        if(mon[fila][columna].estat==BUIT &&
copia[fila][columna]==BUIT)
        {

    mon[fila][columna].estat=BORINOT;/* posem el borinot */
```

```
        mon[fila][columna].borinot.fitness=mon[k][l].borinot.fitness;

        mon[fila][columna].borinot.vida=mon[k][l].borinot.vida;
                                                for (m=0;m<VERTEXS;m++)

        mon[fila][columna].borinot.adn[m]=mon[k][l].borinot.adn[m];

        mon[fila][columna].borinot.p=mon[i][j].borinot.p;

        mon[fila][columna].borinot.ruta_x=mon[k][l].borinot.ruta_x;

        mon[fila][columna].borinot.ruta_y=mon[k][l].borinot.ruta_y;
                                                mon[k][l].estat=BUIT;
/*borrem el borinot i la còpia*/
                                                copia[k][l]= BUIT;
                                                sortir=1;
                                        }
                                        if(sortir) break;
                                }
                        }
                }



                        else if ((copia[k][l]== BORINOT) && (mon[k][l].borinot.p==1)) /*
en cas que sigui un borinot amb pol·len */
                        {
                                /*si el borinot ha trobat pol·len, mirem que torni al niu
directament*/
                                torna_borinot(k,l);
                        }




                        else if (copia[k][l]==NIUB)  /*si som al niu, aquest es buidarà fins
que sigui possible als voltants*/
                        {
                                sortir=0;
                                x=0;

                                for(i=CENTRAL-2;i<=CENTRAL+2;i++) /*recorrem els
voltants del niu amb rang 2 per moure*/
                                {
                                        for(j=CENTRAL-2;j<=CENTRAL+2;j++)
                                        {
                                                if(copia[i][j]==BUIT &&
mon[i][j].estat==BUIT && mon[CENTRAL][CENTRAL].niu.nborinots>AGOTAT)
                                                {
                                                        treu_borinot_niu(i,j);
                                                }
                                                x++;
                                                if
((mon[CENTRAL][CENTRAL].niu.nborinots==AGOTAT)||(x>RANG))
```

```
                                                    sortir=1;
                                            }
                                                if (sortir) break;
                                        }
                                }
                                else continue;
                        }
                }
}


/*************************************************************************/

void mostra_mon(void)   /* ens mostra el mon per pantalla a cada generació*/
{                                                   /* així es veu com es mouen els borinots*/
        int i,j,l;
        printf("\n");
  for(i=0;i<MAXFILES;i++)
  {
         for(j=0;j<MAXCOLUMNES;j++)
         {
                 if(mon[i][j].estat == BUIT)            printf(".");
                 if (mon[i][j].estat==BORINOT && mon[i][j].borinot.p==0 &&
                         mon[i][j].borinot.ruta_x==0 && mon[i][j].borinot.ruta_y==0)
        printf("B");
                 if (mon[i][j].estat==BORINOT && mon[i][j].borinot.p==0 &&
                         (mon[i][j].borinot.ruta_x!=0 || mon[i][j].borinot.ruta_y!=0))
        printf("V");
                 if(mon[i][j].estat==BORINOT && mon[i][j].borinot.p==1)
        printf("*");
     if( mon[i][j].estat == NIUB)
                        printf("N");
   }
         printf("\n");
  }
  printf("\n");
  printf ("%d borinots al niu\n",mon[CENTRAL][CENTRAL].niu.nborinots);

  for(l=0;l<SOLUCIONS;l++)
          printf("\t%d", mon[CENTRAL][CENTRAL].niu.reina[l].fitness);
  printf("\n");
  getche();
}

/*************************************************************************/

void mostra_niu(void)
{
        int x;
        printf("\n");
        for(x=0;x<mon[CENTRAL][CENTRAL].niu.nborinots;x++)
        {
                printf("borinot %d\n", x);
        }
```

```
}


/*************************************************************************/


void main(int argc, char *argv[])

/*El programa vol el nº de branques, i el nº de colors*/
/*si l'usuari no li passa amb la crida, el programa ho demana */
{
  FILE *fitxer;
  time_t t;
  div_t mitja;
  int i,m,j,suma=0,malament=0;
  float suma_t=0,mitja_t,temps[COPS];
  int resultats[COPS];
  clock_t start,end;

  srand((unsigned) time(&t));

  fitxer=fopen("resultat-0212.txt","a");
  if (argc!=3)
  {
          printf("branques = ");     /* no li passem res amb la crida */
          scanf("%i",&branques);
          printf("colors = ");
          scanf("%i",&ncolors);
          }
   else
   {
             sscanf(argv[1], "%i", &branques);   /* li passem el nombre de*/
          sscanf(argv[2], "%i", &ncolors);    /* branques i el de colors*/
   }                        /* quan fem la crida al programa */


   if (branques>MAXBRANQUES)
   {
          fprintf (stderr,"\nEl nombre de branques  ha de ser menor de %i\n",
MAXBRANQUES);
          exit(1);
   }
   if (ncolors<1 || ncolors> VERTEXS)
          {
   fprintf(stderr, "\nEl nombre de colors ha d'estar entre 1 i %i\n", VERTEXS);
   exit(1);
          }

/*Escritura per pantalla dels paràmetres*/

          printf("\nColors=%i, Vertexs=%i, Branques=%i\n", ncolors, VERTEXS,
branques);

   for (m=0;m<COPS;m++)  /* FEM DIFERENTS PROVES */
   {
```

```
        start= clock();
        genera_graf(branques);   /* es genera el graf aleatori */
                crear_mon();            /* es crea el tauler amb els individus */
                sortir=0;
                i=0;
        millor[0]=0;
                while(i<MAXGENERACIONS && sortir ==0) /* comencen les
generacions */
                {
                        moure();     /* es mouen els angels i els individus aleatoriament */
//                      mostra_mon();
                        mutacio();   /* es muten tots els individus, tant els borinots com la
reina */
                        /* mutacio();
                        mutacio();
                        mutacio();
                        mutacio();*/

                        //    salvacio();  /* es dupliquen els que estan prop dels angels*/
                         buscar_millor();  /* es busca quin és el millor individu */
                                                /* i es mira si cal abandonar el programa */
                        if ( branques <= millor[0] || borinots<=0) sortir =1;
                        restar_vida();  /* es resta la vida, passa una generació */
                        if (i%40==0)
                        {
                                treu_borinot_reina();
                        }
                        i++;


                } /* fi while de les generacions */

                printf("\ngeneracions:%d  \tfitness:%d\t",i,millor[0]);
                end = clock();
                end = end - start;
                printf("temps: %d", end/CLOCKS_PER_SEC);
                temps[m]= (float) end/CLOCKS_PER_SEC;
                resultats[m]=i;
                borinots=0;
                if (millor[0]>=branques)
                {
                        suma_t= suma_t + temps[m];
                        suma = suma + resultats[m];
                }

                 else  malament++;

                /* si volem veure totes les solucions en el fitxer resultat o no */
                /*  escriu_a_fitxer(branques,ncolors,i);*/


                /*  activa si volem o no veure el millor individu */
                /*      for (i=0; i<VERTEXS; i++)
                        {
                printf("%c",mon[millor[1]][millor[2]].borinot.adn[i]+'A');
```

```
                        }*/


        } /* fi for del nombre de COPS que executem el programa */
        /* es compten els COPS que s'ha executat, i de les vegades que ho
        ha aconseguit es mira quina és la mitja */
        j=COPS-malament;
  if(j==0) j=1;
  mitja_t = suma_t/j;
  mitja= div(suma,j) ;
  printf("\n MITJA %d VEGADES: %d      MITJA TEMPS:%f
POT:%d\n",COPS,mitja.quot,mitja_t,COPS-malament);

//   for(l=0;l<SOLUCIONS;l++)
//            printf("\n %d", mon[CENTRAL][CENTRAL].niu.reina[l].fitness);

   fprintf(fitxer,"\n MITJA %d VEGADES: %d      MITJA TEMPS:%f
POT:%d",COPS,mitja.quot,mitja_t,COPS-malament);  /*activa si volem veure la mitja
de cops al fitxer resultat */
  escriu_fitxer_mitja(branques,ncolors,COPS,mitja.quot,COPS-malament);
  fwrite(buffer, strlen(buffer),1,fitxer);
  fclose(fitxer);

}
```