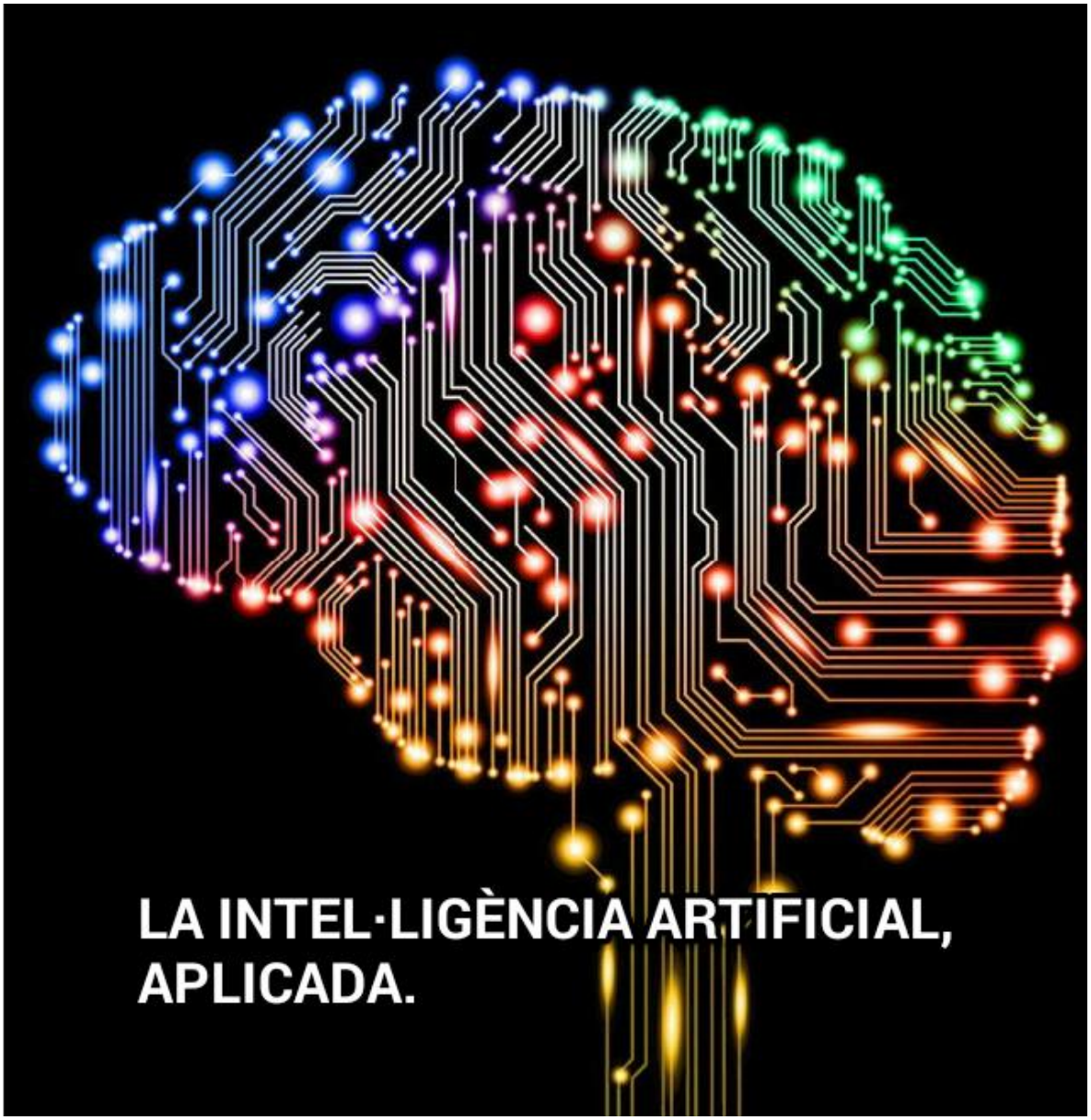

LA INTEL·LIGÈNCIA ARTIFICIAL,
APLICADA.

La intel·ligència artificial, aplicada.

Entenent la intel·ligència artificial i programant-la.

Mario Ramos Montesinos

2n de Batxillerat Tecnològic

INS Miquel Biada

Curs 2014-15

Tutor: Marcos A. Rodríguez

Agraeixo al meu tutor en Marcos l'ajuda proporcionada per realitzar aquest treball, així com a l'Èdgar Llorente i en Jose Escribano per l'ajuda durant la programació i el fonament teòric d'aquest.

ÍNDEX

Segona part - Part pràctica

Tercera part

1.0 Motivacions

Sempre he estat un aficionat dels videojocs, des de ben petit em passava hores mirant com el meu pare jugava amb la PlayStation 1, i als 5 anys vaig començar a jugar i encara continuu, els videojocs sempre han estat un hobbie important i molt agradable per a mi.

No va ser fins que vaig entrar en l'adolescència i començar a interessar per la programació, que em vaig preguntar com funciona internament un videojoc, bàsicament em preguntava dues coses: Com es generen les imatges que veiem, i com els personatges prenen les decisions depenent de molts factors. Fa cosa de dos anys vaig resoldre el meu primer dubte, les imatges es generen amb motors gràfics, models i textures. Però no vaig aconseguir resoldre la segona, la complexitat de la intel·ligència artificial (a partir d'ara IA) és bastant alta i em feia falta alguna cosa més que la Wikipedia.

Per això vaig triar aquest tema, faig un treball de recerca força interessant al meu parer, i acabo de resoldre el dubte que des de sempre, com apassionat del videojoc, he tingut.

D'altra banda, volia programar un exemple pràctic per complementar el treball, per això vaig pensar en programar un Exemple pràctic de vàries IA:

- Un PacMan amb DFS, BFS, UCS i A *
- Un Tic-Tac-Toe amb Minimax

El llenguatge de programació que he pensat utilitzar per el tic-tac-toe i les dames és Java, ja que tinc experiència en ell i està orientat a objectes, cosa que facilita molt les coses.

Mentre que per al Pac-Man faré servir Python per aprofitar la interfície oferta per la Universitat de Berkley, ja que disposa d'un projecte obert a tothom l'objectiu del qual és programar diverses IA plantejant diferents problemes. Té un apartat d'avaluació per determinar fins a quin punt està bé aquesta IA. Aquest projecte es pot trobar en aquest enllaç: http://ai.berkeley.edu/search.html

Aquest treball intenta resoldre les següents qüestions:

- Explicar que és la IA i la seva historia.

- Comparar els diferents algoritmes d'IA d'un joc senzill.

- Programar un tic-tac-toe i un PacMan

Per això dividit el treball en tres parts:

- Una primera part teòrica que inclou els dos primers objectius esmentats.

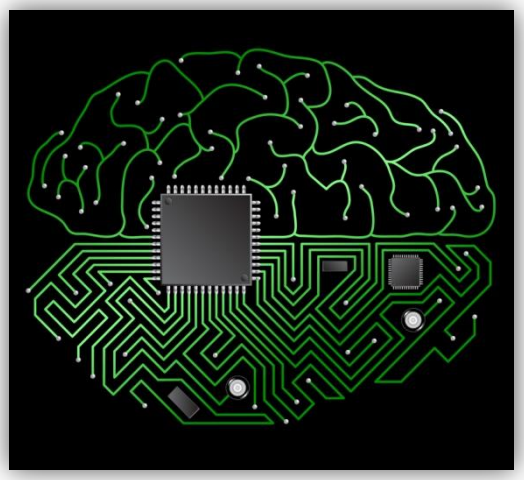
- Una segona part pràctica de fer els jocs i explicar com ho he fet

- Una tercera part amb les conclusions.

Aquest treball tracta sobre la intel·ligència artificial en els jocs de taula senzills, com ara les Dames, Backgammon, tic-tac-toe .. etc. Aquest punt de partida em servirà per fer una recerca sobre la història de la IA i la seva evolució. I també fer una demostració pràctica d'alguns algorismes.

1.1 Introducció a la intel·ligència artificial

1.1.1 Introducció



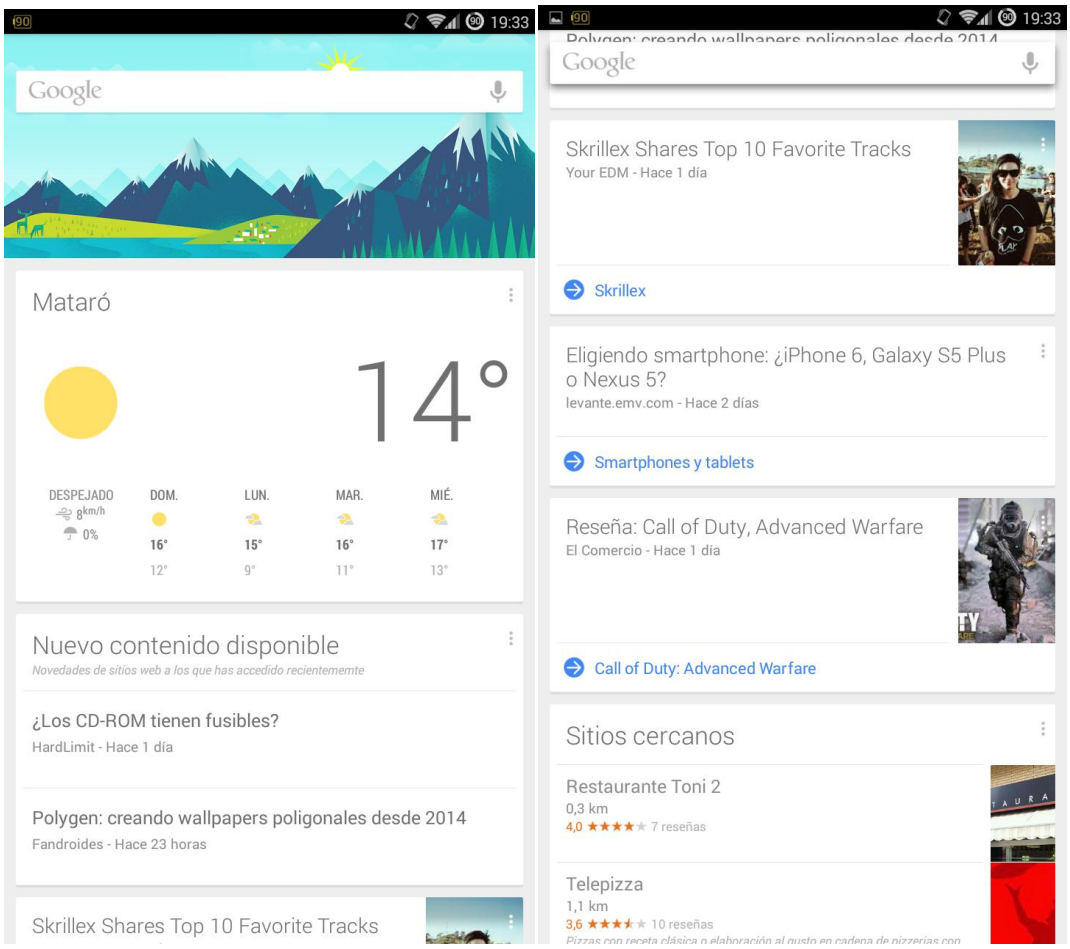
La Intel·ligència Artificial (IA) va començar com a resultat de la investigació en psicologia cognitiva i lògica matemàtica. S'ha enfocat sobre l'explicació del treball mental i construcció d'algorismes de solució a problemes de propòsit general. La idea de construir una màquina que pugui executar tasques que requereixin de la intel·ligència humana, és un atractiu.

Les tasques que han estat estudiades per l'ús de la IA inclouen jocs, traducció d'idiomes, comprensió d'idiomes, diagnòstic de falles, robòtica i assessoria experta en diversos temes, entre moltes altres.

En el nostre dia a dia estem constantment envoltats d'intel·ligència artificial, a menor o major nivell, però per desgràcia molt poca gent es fa la pregunta de "Com ha de funcionar aquest sistema que fa això per mi?". La intel·ligència artificial és una àrea de la ciència de la qual poca gent és conscient de la seva importància.

Un clar exemple d'intel·ligència artificial que (alguns) fem servir diàriament és Google Now.

1.1.2 Exemple d'us diari: Google Now



Google Now, és una joia de la intel·ligència artificial incorporada en tots els smartphones amb Android i descarregable en iOS. Bàsicament és un assistent intel·ligent de Google, que va analitzant els llocs pels quals ens solem moure, els nostres gustos, les webs que freqüentem etc. Després es crea un perfil del propietari i mostra recomanacions personalitzades de manera intel·ligent.

A part de poder mostrar coses que m'interessen personalment com a amant de la música electrònica, tecnologies i videojocs (Sense en cap moment configurar-lo jo per això), el sistema demostra la seva intel·ligència recordant les cites importants del meu calendari, o mostrant-me un accés directe per a la ruta de navegació cap a casa només en cas que estigui fora d'aquesta.

1.1.2 2 Definició: Què és la IA

Per saber bé el que és la intel·ligència artificial no ens podem conformar amb una definició. En el següent quadre veurem les definicions que autors recents han creat per definir-la.

Sistemes que actuen com humans	**Sistemes que pensen com humans**
"L'estudi de com fer ordinadors que facin coses que, de moment, la gent fa millor " (Rich i Knight, 1991)	"L'esforç per fer a les computadores pensar, màquines amb ments en el sentit ampli i literal" (Haugeland, 1985)
"Un camp d'estudi que busca explicar i emular el comportament intel·ligent en termes de processos computacionals" (Schalkoff, 1990)	"L'estudi de les facultats mentals a través de l'estudi de models computacionals" (Charniak i McDermott, 1985)
Sistemes que actuen racionalment	**Sistemes que pensen racionalment**

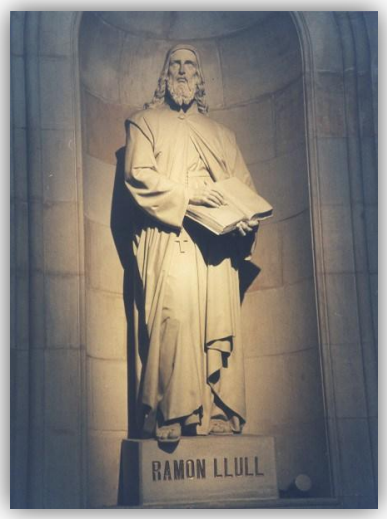
Aquestes definicions varien al voltant de dues dimensions principals: Processos mentals (Part esquerra) i raonament (Part dreta).

D'altra banda les funcions de dalt mesuren la condició desitjable en funció de com ho faria un humà, mentre que les de sota ho fan amb un concepte d'intel·ligència ideal (racionalitat).

Llavors la IA reuneix amplis camps, els quals tenen en comú la creació de màquines capaces de pensar. En ciències de la computació s'anomena intel·ligència artificial a la capacitat de raonar d'un agent no viu.

1.3 Personatges de l'historia de la IA

Ramon Llull



Des de l'antiguitat l'ésser humà ha estat traçant el camí que li permetés desenvolupar "màquines intel·ligents" i "màquines pensants". El primer exemple el trobem en l'Antiga Grècia, amb Aristòtil, que va intentar descriure el funcionament racional de la ment, o Ctesibi d'Alexandria, amb una màquina automàtica que regulava el flux d'aigua, podem trobar els primers passos d'aquesta recerca de les màquines pensants. Però Ramon Llull va anar més enllà.

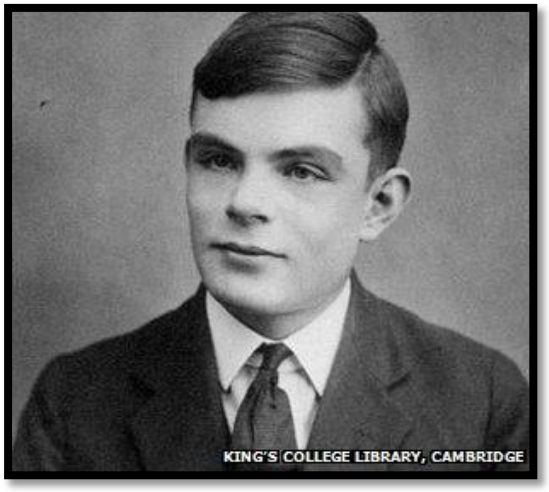
Ramon Lull (1232-1315), és un beat (se li considera patró dels enginyers informàtics) que va néixer a la Ciutat de Mallorca (actual Palma de Mallorca) al segle XIII i va estar vinculat a l'orde franciscà on va exercir de filòsof, poeta, místic, teòleg i missioner. Si bé gran part de la seva obra i accions estan vinculades a la fe cristiana, se'l considera un dels pares del català literari en utilitzar aquesta llengua per escriure textos científics, filosòfics o tècnics entre els quals es troba una feina propera a la Intel·ligència Artificial: l'Ars Magna.

L'any 1.315 (que va ser la seva data de publicació encara que va començar a treballar en aquesta obra sobre 1.275), Ramon Llull va expressar en l'Ars Magna la idea que el raonament podia implementar-se de manera artificial en un artefacte mecànic amb el qual poder mostrar les veritats de la fe cristiana d'una manera tan clara que no hi hagués lloc a discussió.

Ramon Llull va cridar a aquest artefacte Ars Generalis Ultima o Ars Magna (Gran Art) i va suposar un punt d'inflexió en el seu treball, convertint-se en un dels seus treballs més importants.

Tot i que el Ars Magna era un autòmat molt rudimentari, es considera el primer intent d'utilització de mitjans lògics per produir coneixement i una de les primeres implementacions de sistemes d'intel·ligència artificial ja que Ramon Llull volia demostrar, amb aquesta màquina, la veracitat de les doctrines cristianes.

Alan Turing



Alan Mathison Turing (23 juny 1912 - 7 d junh de 1954), va ser un matemàtic, lògic, científic de la computació, criptògraf i filòsof britànic. És considerat un dels pares de la ciència de la computació sent el precursor de la informàtica moderna. Va proporcionar una influent formalització dels conceptes d'algorisme i computació.

Durant la Segona Guerra Mundial, va treballar en desxifrar els codis nazis, particularment els de la màquina Enigma.

Clarament, Alan Turing va ser un home que es va avançar a la seva època. El 1950, en els inicis de la computació, ell ja estava bregant amb un dels grans dilemes de l'àrea de la informàtica: poden pensar les màquines?

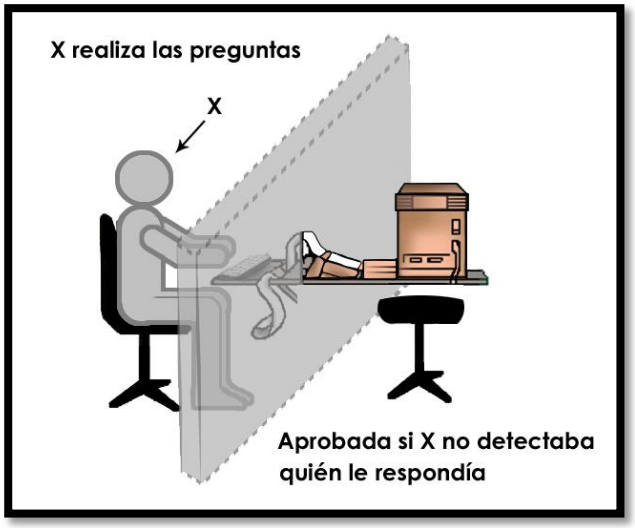
Turing es plantejava aquesta pregunta en moments en que recentment s'estaven desenvolupant les primeres computadores i el terme intel·ligència artificial (IA) encara no havia estat encunyat. Aquest va ser inventat pel científic nord-americà John McCarthy en 1956, dos anys després de la mort de Turing.

Clarament Turing pot ser considerat el pare de la Intel·ligència Artificial, es pot dir que aquesta disciplina va néixer a partir de l'article titulat "Computing Machinery and Inteligence" publicat per ell al 1950.

Turing estava convençut que algun dia, les màquines serien tan intel·ligents com els humans. I per demostrar-ho, va inventar el Test de Turing, el 1950.
El Test de Turing es basa en la idea que si no pots distingir les respostes d'un programa davant les d'un humà, llavors és perquè la intel·ligència artificial és tan intel·ligent com nosaltres.

Test Turing

El test de Turing és una prova proposada per Alan Turing per demostrar l'existència d'intel·ligència en una màquina. Va ser exposat el 1950 i segueix sent un dels millors mètodes per als defensors de la intel·ligència artificial. Es fonamenta en la hipòtesi positivista que, si una màquina es comporta en tots els aspectes com intel·ligent, llavors ha de ser intel·ligent.



La prova consisteix en un desafiament. Se suposa un jutge situat en una habitació, una màquina i un ésser humà en una altra. El jutge ha de descobrir quin és l'ésser humà i quina és la màquina, estant per als dos permès mentir en contestar per escrit les preguntes que el jutge els fes. La tesi de Turing és que si tots dos jugadors eren prou hàbils, el jutge no podria distingir qui era l'ésser humà i qui la màquina.

La primera vegada que un jutge va confondre a una màquina amb un humà va ser l'any 2010, quan el robot Suzette va superar la proba.

Ara per ara el treball de programar un ordinador per passar la prova és considerable. L'ordinador hauria de ser capaç del següent:

- **Processar un llenguatge natural** en qualsevol idioma per així establir una comunicació satisfactòria

- **Representar el coneixement** per així guardar tota la informació que se li hagi donat anteriorment durant l'interrogatori

- **Raonar automàticament**, per tal d'utilitzar informació guardada en respondre preguntes i obtenir noves conclusions

- **Autoaprenentatge de la màquina**, perquè s'adapti a noves circumstàncies i per detectar i extrapolar esquemes determinats.

Hi ha una variant de la prova en la qual la màquina percep objectes i els desplaça de manera intel·ligent. Aquesta prova es diu Prova total de Turing i cal que la màquina estigui dotada de:

- **Vista**, que li permeti percebre objectes

- **Robòtica**, per desplaçar aquests

Al 2014, Eugene Goostman, un bot desenvolupat a Ucraïna que imita el comportament d'un nen de 13 anys, ha superat per primera vegada el Test de Turing, 65 anys després de la seva formulació.

Si el 30% dels jutges no són capaços de distingir qui és l'humà, i qui la màquina, el Test de Turing es considera superat, i el programa es considera que és capaç de pensar com un humà. Això és el que han fet acadèmics de la Universitat de Reading, en un esdeveniment organitzat per la Royal Society de Londres, una de les institucions científiques més prestigioses del món.

Aquest fet es considera una fita en la intel·ligència artificial, i obre les portes a un futur incert on no siguem capaços de distingir si ens comuniquem amb una altra persona, o amb un bot, amb tot el que això implica.

John von Neumann

John von Neumann (Budapest 28 desembre 1903-8 de febrer de 1957) va ser un matemàtic hongarès-nord-americà que va realitzar contribucions fonamentals en física quàntica , anàlisi funcional, teoria de conjunts, teoria de jocs, ciències de la computació, economia, anàlisi numèrica, cibernètica, hidrodinàmica, estadística i molts altres campos. És considerat com un dels més importants matemàtics de la història moderna.

Von Neumann es preguntava si era possible construir una màquina que produís màquines més complexes que si mateixa. La seva recerca estava guiada per les seves increïbles habilitats matemàtiques. Ell va adonar que la clau està dins del concepte d'acte-reproducció.

Considerem que les màquines normals només construeixen màquines més simples que si mateixes. Per exemple, els robots de línies de producció industrial, que són molt més complexos que els artefactes que produeixen en sèrie. Certament nosaltres, éssers humans, encara no hem aconseguit concebre éssers més complexos que nosaltres mateixos. No obstant això, l'evolució suggereix que els organismes vius poden donar naixement a formes de vida més complexes.

Una màquina acte-reproduïble hauria de poder evadir la seva degeneració cap a formes més simples. Ara bé, imaginem aquesta màquina propensa a mutacions, i que eventualment generi una o més màquines més complexes ... Llavors comença la màgia de la vida artificial.

1.4 Definicions i conceptes

1.4.1 Lleis del pensament

Aristoteles va ser un dels primers que van intentar crear la "manera correcta de pensar" creant per a això els seus famosos sil·logismes, esquemes de estructures de argumentació mitjançant les quals sempre s'arriba a conclusions correctes si es parteix de premisses correctes.

Així es va inaugurar el camp de la lògica.

El desenvolupament de la lògica formal a finals del segle XIX va permetre comptar amb una notació precisa per representar tot el que hi ha al món. Ja per 1965 existien programes que amb temps i memòria suficients podien descriure un problema a notació lògica i trobar solució sempre que tingués.

A la IA la tradició logística s'esforça per elaborar programes com l'anterior per crear sistemes intel·ligents.

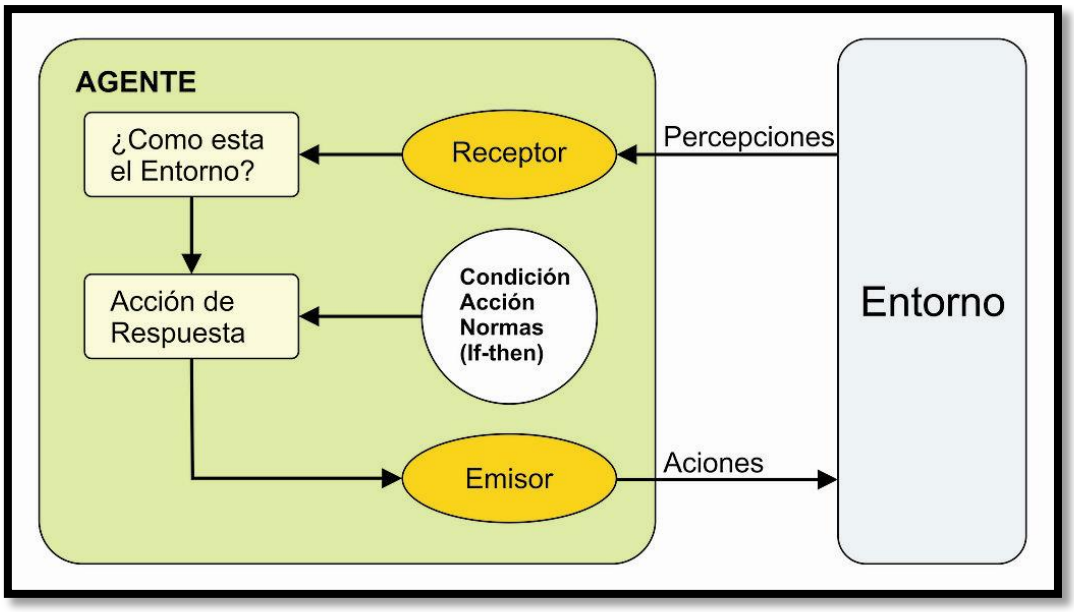
Aquest enfocament presenta dues obstacles:

- En primer lloc no és fàcil rebre un coneixement informal i expressar-ho en termes formals que exigeix la notació lògica, especialment quan aquest coneixement té encara que sigui una mica d'incertesa.

- En segon lloc hi ha una dificultat en resoldre el problema a causa de límits de capacitat de còmput dels ordinadors.

1.4.2 Agents intel·ligents

Un agent és tot allò que pot considerar-se que percep el seu ambient i que respon o actua en aquest ambient de manera racional per mitjà de efectors. En aquest context la racionalitat és la característica que posseeix una elecció de ser correcta, es a dir, de tendir a maximitzar un resultat esperat.



Cal deixar clar que hi ha una diferència entre racionalitat i el que és denominat ominiscencia. Un agent ominiscient és aquell que sap el resultat real que produiran les seves accions, però en la realitat no existeix la omniscencia* ja que hi ha factors que l'agent no pot controlar.

El caràcter de racionalitat del que es fa en el moment donat dependrà de quatre factors:

- De la mesura amb què s'avalua el grau d'èxit aconseguit

- De tot el que fins aquell moment hagi percebut l'agent. A això se li crida la seqüència de percepcions

- Del coneixement que tingui l'agent sobre el medi

- De les accions que l'agent pot emprendre

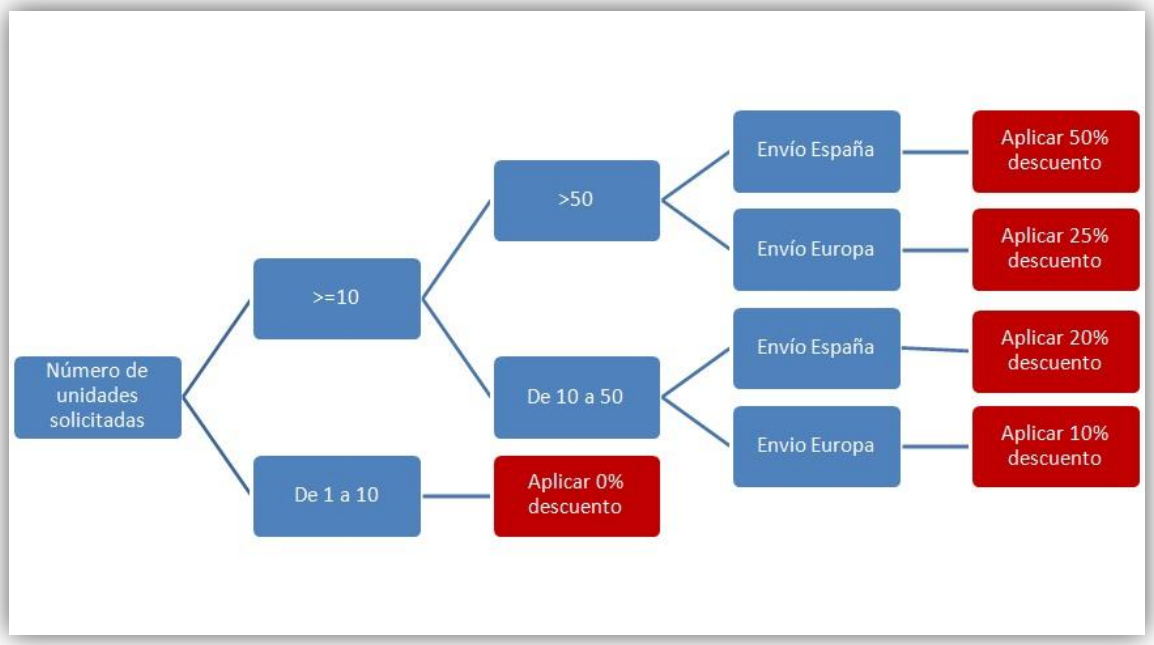
**omniscencia: la capacitat de saber-ho tot*

1.4.3 Arbre de decisió

Els arbres de decisió són models de predicció que s'utilitzen per organitzar gràficament la informació sobre les opcions possibles.

L'arbre de decisió és una bona ajuda per triar entre diversos cursos d'acció. El nom d'arbre de decisió prové de la forma que adopta el model, semblant a un arbre. Està format per múltiples nodes, que representen els punts de decisió, i dels quals sorgeixen branques que representen les diferents alternatives.

Exemple d'arbre de decisió:



Per depèn quin tipus d'arbre de decisió, ens vindrà bé donar un valor a cada acció, és a dir, avaluar la possibilitat per després comparar-la amb altres, i escollir la que millor ens vingui.

O fins i tot podem afegir a cada node un cost, cosa que ajudés a prendre la millor decisió.

Fases en l'elaboració d'un arbre de decisió

1) Identificació del problema

2) Establir l'estratègia inicial

3) Establir les diferents alternatives i successos

4) Representar mitjançant un arbre les diferents alternatives i estratègies.

5) Valoració de cadascuna de les alternatives i successos aleatoris

6) Determinar les decisions òptimes utilitzant el mètode de resolució de marxa enrere.

Avantatges d'aquest tipus d'estructura

- Usa els aventatges de l'estructura consecutiva de les branques de l'arbre de decisió, de tal manera que s'identifiquen de manera immediata les condicions i les accions que s'han de dur a terme.

- Les condicions i les accions de l'arbre de decisió es troben en certes branques però no en altres, a diferència de les taules de decisió, on totes formen part de la mateixa taula.

- Planteja el problema perquè totes les peticions siguin analitzades.

- Analitza les conseqüències de portar a terme una alternativa.

- Facilita la interpretació de la decisió adoptada.

- Mostra un esquema de cost de les diferents alternatives

- Ens Porta a adoptar la millor alternativa amb la informació existent

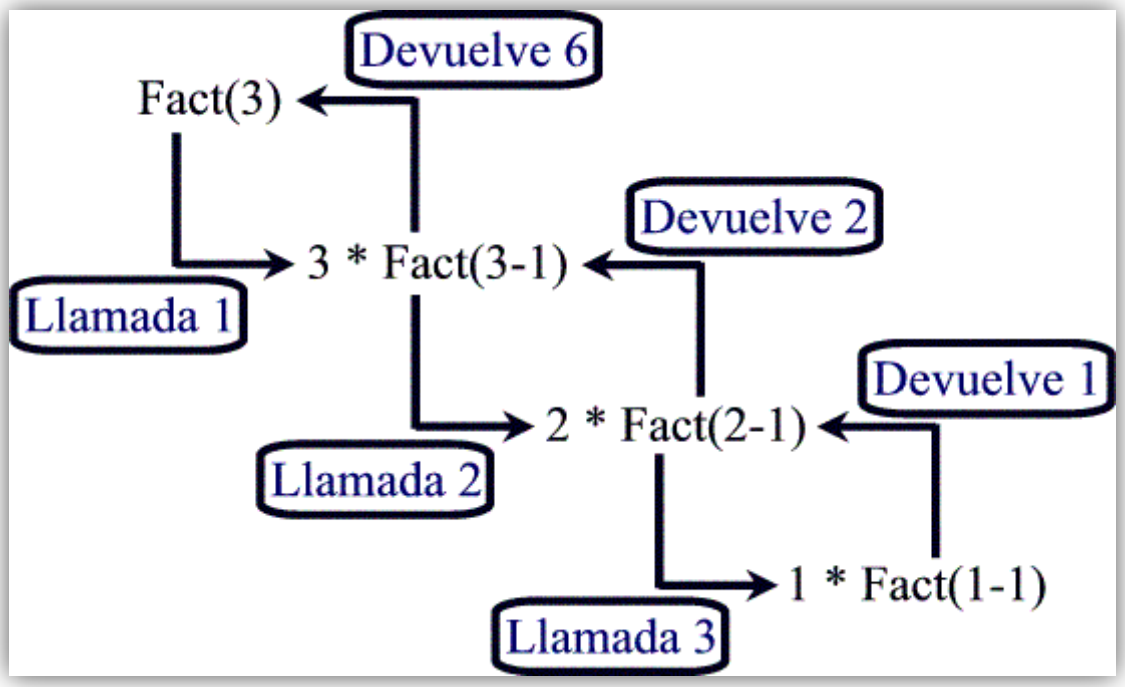
1.4.4 Recursivitat

Recursió és, en ciències de computació, una forma de solucionar problemes. De fet, recursió és una de les idees centrals de ciència de computació. Resoldre un problema mitjançant recursió significa que la solució depèn de les solucions de petites instàncies del mateix problema.

En programació, una funció és recursiva quan es truca a si mateixa, és a dir, quan per acabar una part del programa s'ha de trucar a la mateixa part del programa que alhora truca a la mateixa part del programa ... així fins que alguna d'elles retorna una solució i el programa el sol comença a anar cap enrere.

Quan una funció es truca a si mateix, s'assigna espai a la pila per les noves variables locals i paràmetres. En tornar d'una trucada recursiva, es recuperen de la pila les variables locals i els paràmetres antics i l'execució es reprèn en el punt de la trucada a la funció.

La millor manera de entendre-ho és amb un exemple, aquí tenim com a funcionària la funció recursiva per calcular un Factorial:



La millor definició sens dubte de la recursió, és:

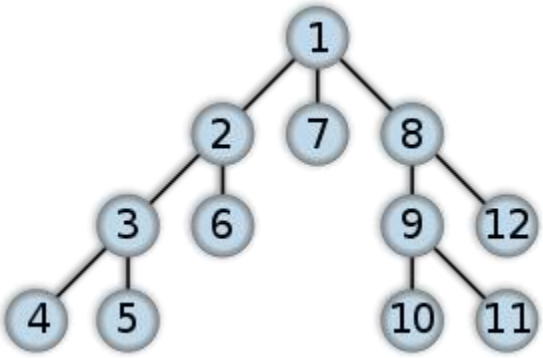
recursió: -veure recursió.

1.5 Algorismes de cerca

1.5.1 DFS

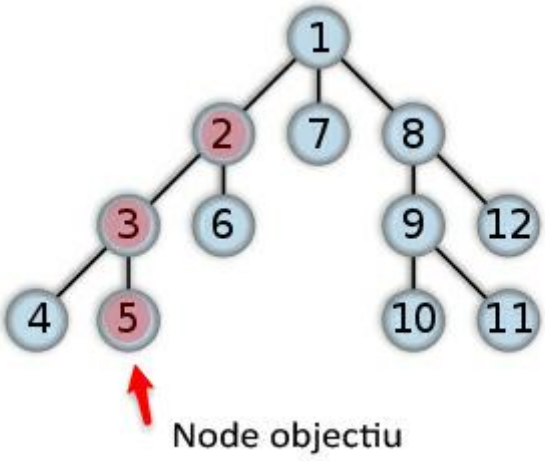
L'algorisme DFS (en anglès Depth First Search) és un algorisme de cerca que crea un arbre de possibilitats, però el va creant en profunditat, és a dir, no canvia a un altre node fins que el node que aquesta examinant no estigui examinat sencer, totes les possibilitats.

La següent imatge indica l'ordre en què els nodes són analitzats:



Aquest efecte s'aconsegueix afegint els "fills" d'un node a una llista de nodes per examinar, i agafar l'últim node afegit d'aquesta llista (el més recent). Després tot és repetir el mateix procés fins que hàgim trobat el node objectiu o en el pitjor dels casos fins que no quedin nodes en aquesta llista, la qual cosa ens indica que no podem arribar a la solució.

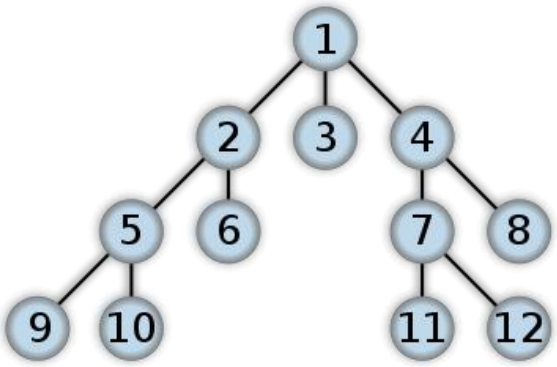
Un cop hàgim trobat el node objectiu, cal trobar el camí, i això consisteix en anar rastrejant els "pares" del node. S'ha de rastrejar el pare del node objectiu, el pare del pare del node objectiu .. etc, fins arribar al primer node expandit.



1.5.2 BFS

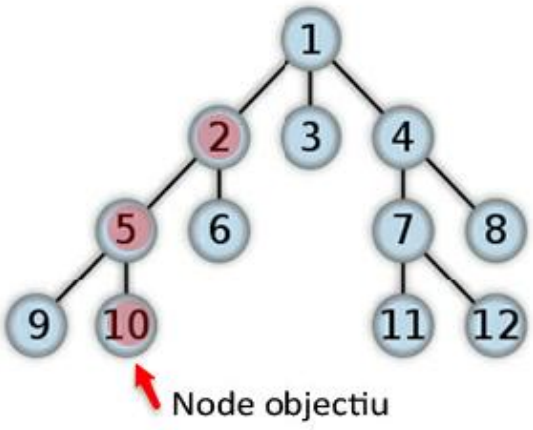
L'algorisme BFS (en anglès *Breadth* First Search), al igual que el DFS, és un algorisme de cerca que crea un arbre de possibilitats, però el va creant en amplada, és a dir, Va analitzant tots els nodes alhora per "capes".

La següent imatge indica l'ordre en què els nodes són analitzats:



De la mateixa manera que el DFS, aquest efecte s'aconsegueix afegint els "fills" d'un node a una llista de nodes per examinar, i agafar el node mes antic de la llista (el primer que va ser afegit). Després tot és repetir el mateix procés fins que hàgim trobat el node objectiu o en el pitjor dels casos fins que no quedin nodes en aquesta llista, la qual cosa ens indica que no podem arribar a la solució.

Un cop hàgim trobat el node objectiu, cal trobar el camí, i això consisteix en anar rastrejant els "pares" del node. S'ha de rastrejar el pare del node objectiu, el pare del pare del node objectiu .. etc, fins arribar al primer node expandit.

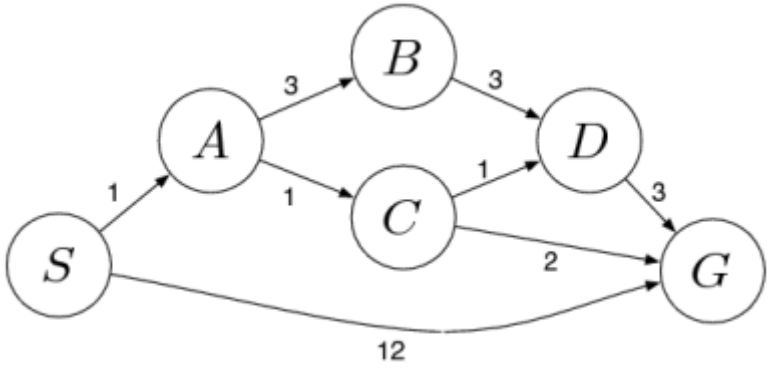


Com podem veure, en aquest cas sent la solució de l'exemple dels dos algorismes (BFS i DFS) la mateixa, és bastant més òptima el DFS, perquè examina menys nodes; Però si la solució hagués estat, per exemple, el tercer node del primer nivell (4rt node examinat per el BFS i 8é node examinat per el DFS), el BFS hagués estat més òptim.

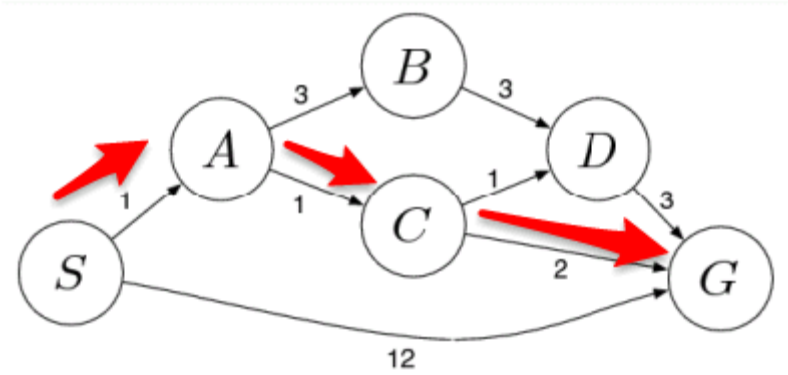
1.5.3 UCS

L'algorisme UCS (en anglès Uniform-cost search) és un algorisme de cerca sobre grafs utilitzat per trobar el camí de cost mínim entre un node origen i un node destí. Hi comença pel node origen i continua visitant el següent node que té menor cost total des de l'origen. Els nodes són visitats d'aquesta manera fins que el node destí és aconseguit.

Imagini que el UCS té aquest arbre de possibilitats, sent S el node d'origen i G el node destí. Els números a les fletxes indiquen el cost que té fer aquest moviment.



Si tenim en compte que a cada cruïlla sempre triarà el camí amb menys cost, sabem que farà aquest camí.



Aquest tipus de cerca no té perquè tornar necessàriament el camí més curt, però si el que té menys cost (Per exemple, creuar una carretera té menys cost de creuar una muntanya. I arribarem abans per la carretera encara que la muntanya ens talli part del camí)

Aquest efecte s'aconsegueix afegint els "fills" d'un node a una llista de nodes per examinar, ordenant la llista de nodes en funció del seu cost, i agafar el primer node (Que serà el de menor cost). Després tot és repetir el mateix procés fins que hàgim

trobat el node objectiu o en el pitjor dels casos fins que no quedin nodes en aquesta llista, la qual cosa ens indica que no podem arribar a la solució.

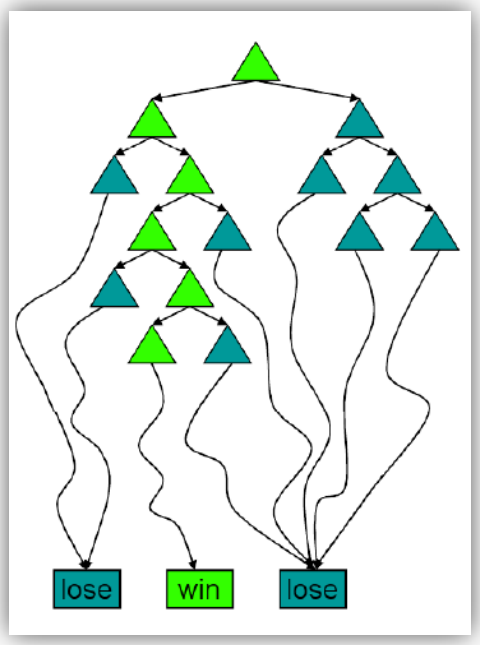
Un cop hàgim trobat el node objectiu, cal trobar el camí, i això consisteix en anar rastrejant els "pares" del node. S'ha de rastrejar el pare del node objectiu, el pare del pare del node objectiu .. etc, fins arribar al primer node expandit.

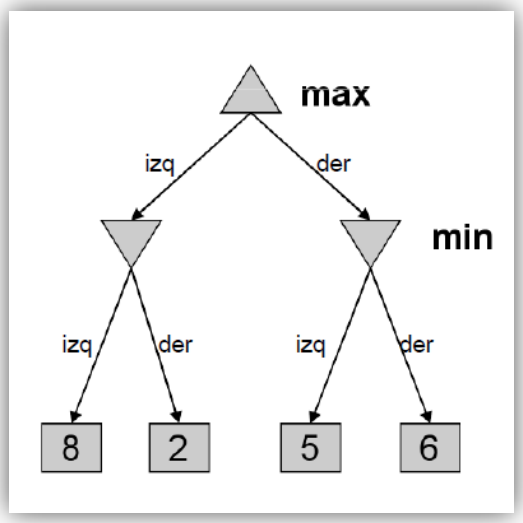
1.5.4 Minimax

El millor algorisme per resoldre jocs de taula simples, és el Minimax. El seu procediment i aplicació al contrari que amb les xarxes neuronals és bastant més fàcil.

En aquest algoritme s'expandeix un arbre amb totes les possibles solucions dels dos jugadors i després es busca el millor moviment de la màquina, sempre pressuposant que l'humà farà el seu millor moviment possible.

Hi han dos jugadors, al jugador que comenci el joc se li digués MAX i l'altre jugador MIN per raons que més endavant es veuran. Aquests dos alternen la seva participació fins que conclou el joc i al final d'aquest s'assignen punts al vencedor (o de vegades càstigs al perdedor).





Amb això tindrem un tipus de problema de cerca format per el següent:

- L'estat inicial, que inclou la posició en el tauler i una indicació de a qui li toca jugar

- Un conjunt d'operadors que defineixen quin tipus de jugades estan permeses pel jugador

- Una prova terminal que defineix el terme del joc. Els estats on acaba el joc se'ls anomena estats terminals

- Una funció d'utilitat (També coneguda com funció de resultat) que assigna un valor numèric al resultat obtingut del joc. En el cas dels escacs o les dames, els resultats possibles són guanyar, perdre o empatar, que es poden representar

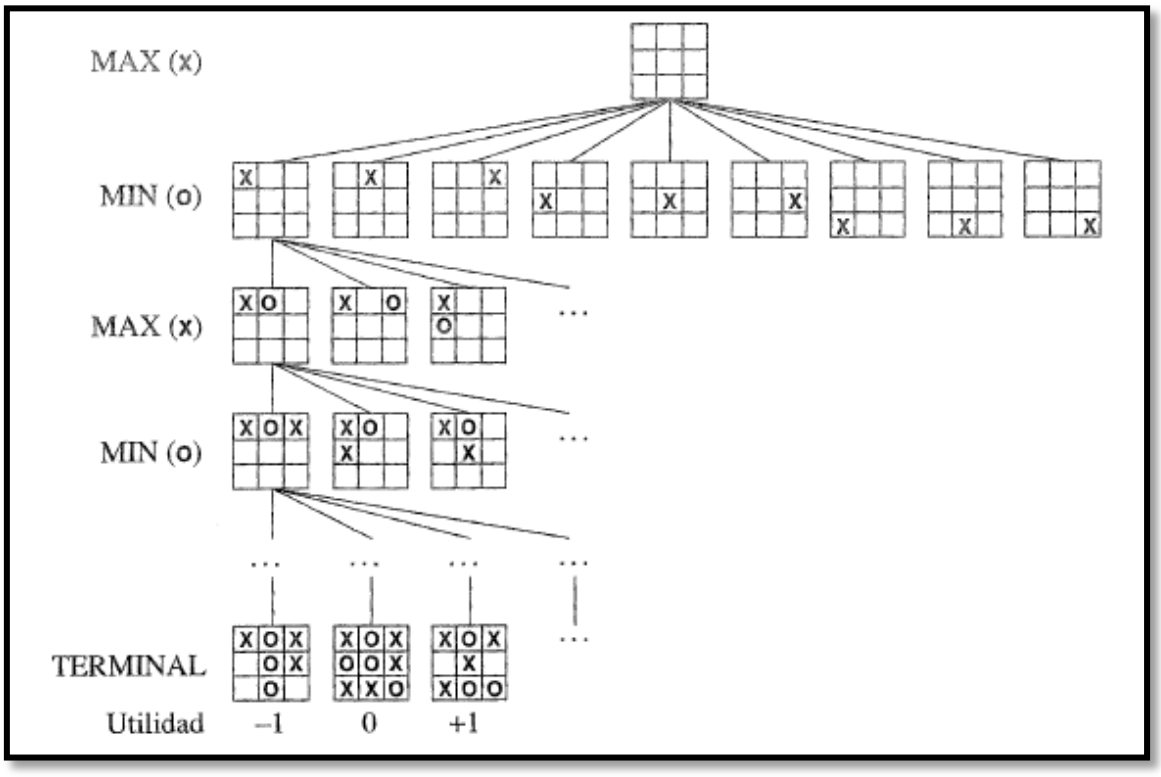
amb +1, 0 o -1. En alguns jocs són més els possibles resultats, en el cas del backgammon aquests varien entre +192 i -192.

Estratègies òptimes

En un problema de recerca normal, la solució òptima seria una seqüència de moviments que condueixen a un estat terminal que és guanyador. Però al ser un joc de dos jugadors, el segon jugador també té els seus moviments.

MAX per tant ha de trobar una estratègia, que especifica el moviment de MAX en l'estat inicial, després els moviments de MAX en els estats que resulten de cada resposta possible d'MIN, després els moviments de MAX en els estats que resulten de cada resposta possible de MIN dels anteriors moviments, etc.

Amb totes les jugades possibles de MIN i MAX es crea un arbre de possibilitats. Aquí en tenim un arbre (parcial) de cerca per al joc tic-tac-toe.



El node de dalt és l'estat inicial, i MAX mou primer, col·locant un X en un quadrat buit. Després és el torn de MIN (O) i així fins que arribem estats terminals en els quals podem assignar valors d'utilitat segons les regles del joc.

En el llenguatge de jocs, diem que aquest arbre és un moviment en profunditat, que consisteix en dos mitjos moviments, cadascun dels quals es diu capa.

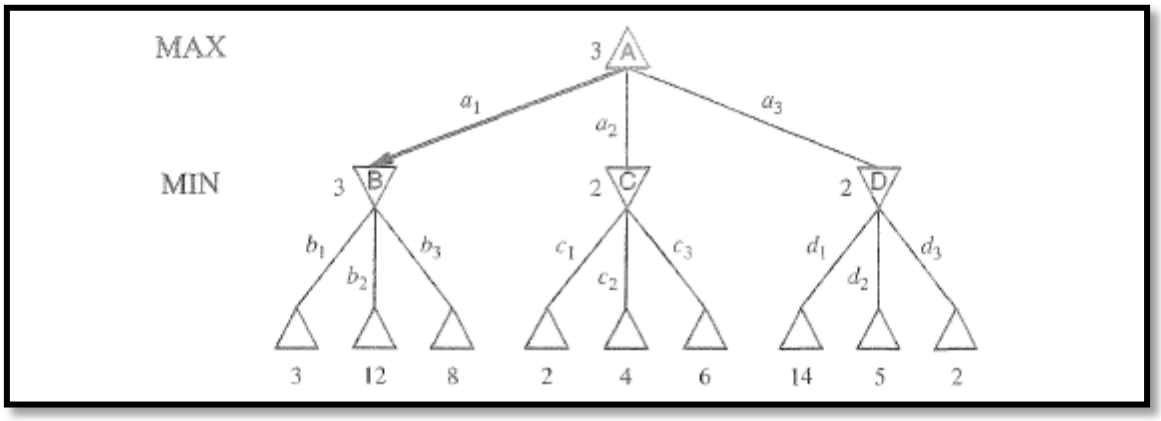
Valor MINIMAX

Considerant un arbre de jocs, l'estratègia òptima pot determinar examinant el valor minimax de cada node.

El valor minimax d'un node és la utilitat (per MAX) i estar en l'estat corresponent, assumint que els dos jugadors juguen òptimament des d'allí al final del joc.

Com ja hem dit, considerant una opció, MAX preferirà moure a un estat de valor màxim, mentre que MIN prefereix un estat de valor mínim.

Per entendre millor, aquí tenim un arbre de joc de dues capes:



Els nodes □ són "nodes MAX", en què li toca moure a MAX, i els nodes □ són "nodes MIN". Els nodes terminals mostren els valors d'utilitat per MAX, mentre que els altres nodes són etiquetats pels seus valors minimax. El millor moviment de Max és a1, perquè el condueix al successor amb el valor de minimax mes alt, i la millor resposta de MIN és b1 perquè condueix al successor amb el valor minimax més baix.

Decisió MINIMAX

Podem identificar també la decisió minimax: l'acció a, és l'opció òptima per MAX perquè condueix al successor amb el valor minimax més alt. Aquesta definició de joc òptim per MAX suposa que MIN també juga òptimament (maximitza els resultats del cas pitjor per MAX).

I si MIN no juga òptimament? llavors és fàcil demostrar que MAX ho farà encara millor. Hi pot haver altres estratègies contra oponents sub-òptims que ho facin millor que

l'estratègia minimax, però aquestes estratègies necessariament ho fan pitjor contra oponents òptims.

1.5.4.1 Com funciona internament

Internament l'algoritme minimax funciona d'aquesta manera, independentment del llenguatge de programació utilitzat:

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Algú una mica familiaritzat amb la programació podrà veure que l'algoritme consta de 3 funcions:

- La funció Minimax, és la funció principal, comença a expandir l'arbre de possibilitats i un cop acabat, s'encarrega d'avaluar la millor possible solució. Generalment comença a expandir l'arbre cridant a la funció Max, ja que si ho fes amb la funció Min, la màquina trobaria la manera més òptima de perdre.

- La funció Max, que donat un paràmetre, fa un moviment en el tauler, i crida a diverses funcions Min, cada trucada amb un moviment possible.

- La funció Min, que donat un paràmetre, fa un moviment en el tauler, i crida a diverses funcions Max, cada trucada amb un moviment possible.

D'aquesta manera, un cop no es pugui (o no vulguem) expandir més l'arbre de possibilitats, l'algoritme comencés a anar cap a enrere donant un valor a cada funció Min i Max que hem trucat, arribant al final a la funció minimax que tornarà el moviment final.

1.5.4.2 Optimització (Poda alfa-beta)

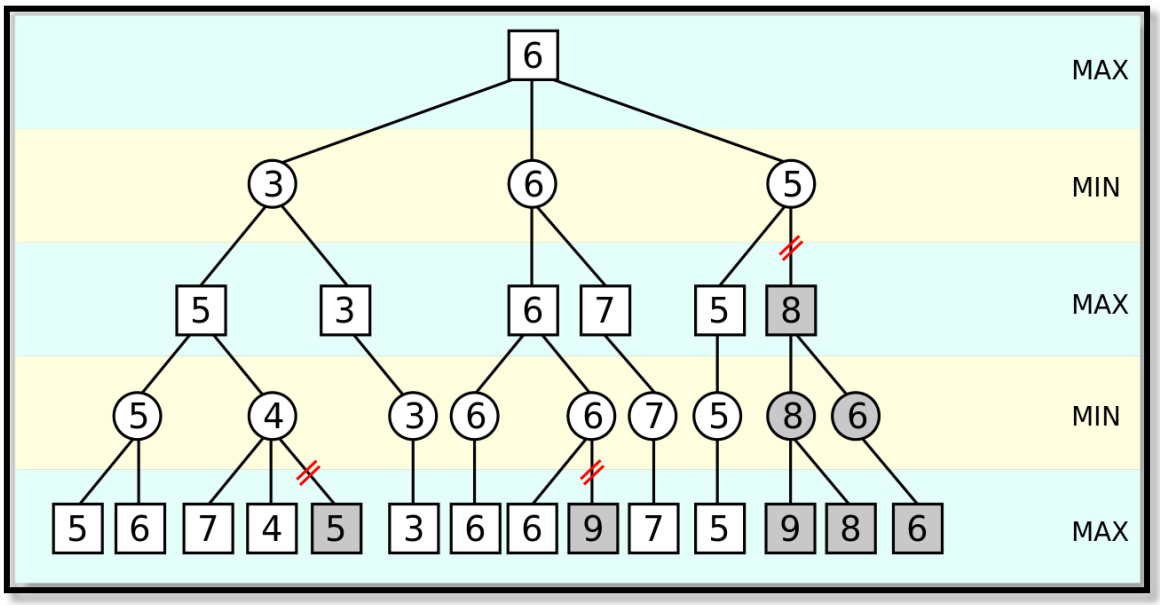
El problema de la recerca minimax és que el nombre d'estats que ha d'examinar és exponencial en el nombre de moviments, la qual cosa requereix molt potencial de la màquina que calcula les possibilitats per escollir una bona jugada dins del temps reglamentari que permet el joc en cada moviment.

Per tant, un ordinador amb no molta capacitat de processament podria explorar menys capes en cada moviment i, tenir menys nivell d'intel·ligència, sent així més fàcil guanyar-li.

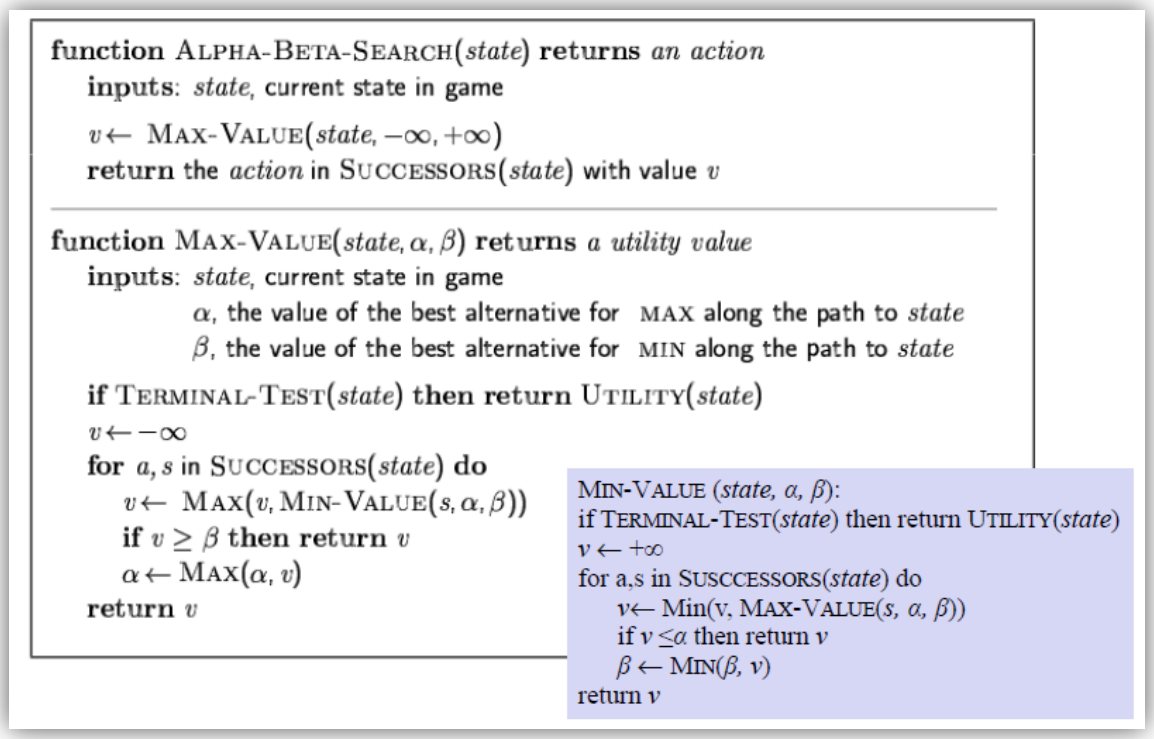
Afortunadament és possible calcular la decisió minimax correcta sense haver de explorar tots els nodes de l'arbre de cerca. Al procés per evitar l'exploració d'una de les branques de l'arbre de cerca se li coneix com Poda de l'arbre de cerca. A la tècnica en particular se li coneix com poda alfa-beta.

Aquesta tècnica aplicada a un arbre minimax estàndard, produeix la mateixa jugada que s'obtindria amb minimax però elimina totes les branques que possiblement no influirien en la decisió final.

Generalment el rendiment augmenta pel fet que l'arbre explorat amb aquest mètode és la meitat. Es a dir, alfa-beta por anticipar el doble que minimax en el mateix temps.



1.5.4.3 Funcionament intern Poda alfa-beta



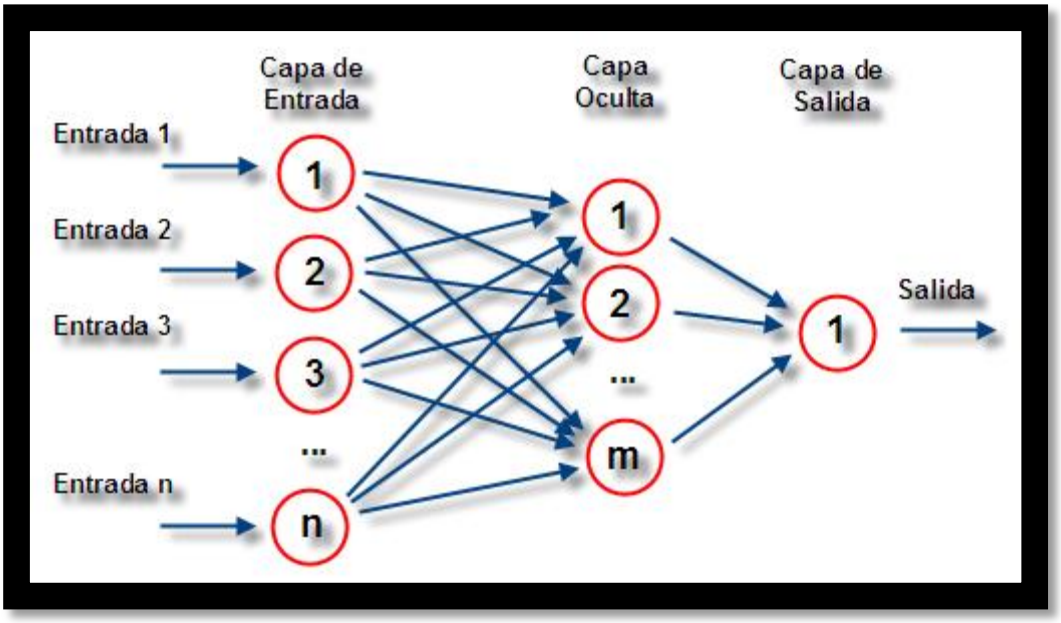
```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

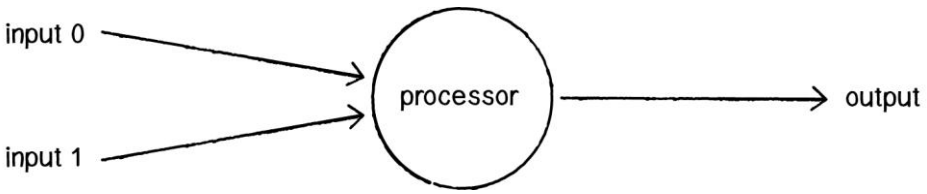
```
MIN-VALUE (state, α, β):
if TERMINAL-TEST(state) then return UTILITY(state)
v ← +∞
for a,s in SUSCCESSORS(state) do
    v← Min(v, MAX-VALUE(s, α, β))
    if v ≤α then return v
    β ← MIN(β, v)
return v
```

1.5.5 Xarxes Neuronals

Les xarxes de neurones artificials són un sistema d'aprenentatge i processament automàtic inspirat en la forma en què funciona el sistema nerviós dels animals. Es tracta d'un sistema d'interconnexió de neurones que col·laboren entre si per produir un estímul de sortida.



La xarxa neuronal es compon d'unitats anomenades Perceptró (neurones). Cada Perceptró rep una sèrie d'entrades a través d'interconnexions i emet una sortida. Tal com ho fa el cervell animal i humà.



- Perceptró

Aplicacions:

Les xarxes neuronals es sonis utilitzar en la robòtica, un dels casos que poden ser més comuns a dia d'avui és en els robots de neteja automàtic.

Segons la informació que rep de les seves múltiples sensors sap cap a on ha de moure o si s'ha deixat alguna zona sense netejar. A poc a poc el robot va aprenent de fer

depenent dels valors que va rebent i el seu comportament va evolucionant a un de millor.

Avantatges:

Les xarxes neuronals artificials (XNA) tenen molts avantatges a causa que estan basades en l'estructura del sistema nerviós, principalment el cervell.

- Aprenentatge: Les XNA tenen l'habilitat d'aprendre mitjançant una etapa que es diu etapa d'aprenentatge. Aquesta consisteix a proporcionar a la RNA dades com a entrada al seu torn que se li indica quina és la sortida (resposta) esperada.

- Organització: Una XNA crea la seva pròpia representació de la informació en el seu interior, traient-li aquesta tasca a l'usuari.

- Tolerància a fallades: A causa que una XNA emmagatzema la informació de forma redundant, aquesta pot seguir responent de manera acceptable fins i tot si es fa malbé parcialment.

- Flexibilitat: Una XNA pot manejar canvis no importants en la informació d'entrada, com a senyals amb soroll o altres canvis en l'entrada (per exemple si la informació d'entrada és la imatge d'un objecte, la resposta corresponent no sofreix canvis si la imatge canvia una mica la seva brillantor o l'objecte canvia lleugerament).

Contra Avantatges:

- Complexitat i dificultat: Les xarxes neuronals són un sistema d'intel·ligència artificial molt avançat i complex.

Segona part - Part pràctica

2.1 Llenguatges de programació utilitzats

2.1.1 Processing

Processing és un programari lliure creat per Casey Reas i Ben Fry cal també assenyalar que sempre està en desenvolupament. El programa està basat en Java, per la qual cosa hereta totes les seves funcionalitats, convertint-se en una eina poderosa a l'hora de crear diferents tipus de projectes.

Per a que serveix?

Perquè qualsevol persona pugui desenvolupar la seva creativitat i plasmar-la en il·lustracions, programes informàtics aplicacions de mòbils, webs interactives, visualitzacions de dades, etc... de manera ràpida i senzilla sense la inconvenient barrera d'un elevat coneixements tècnic.

Com funciona?

1 - Com un senzill software per crear els nostres programes (IDE), que simplifica totes les complexitats de la programació, fent invisibles per a l'usuari.

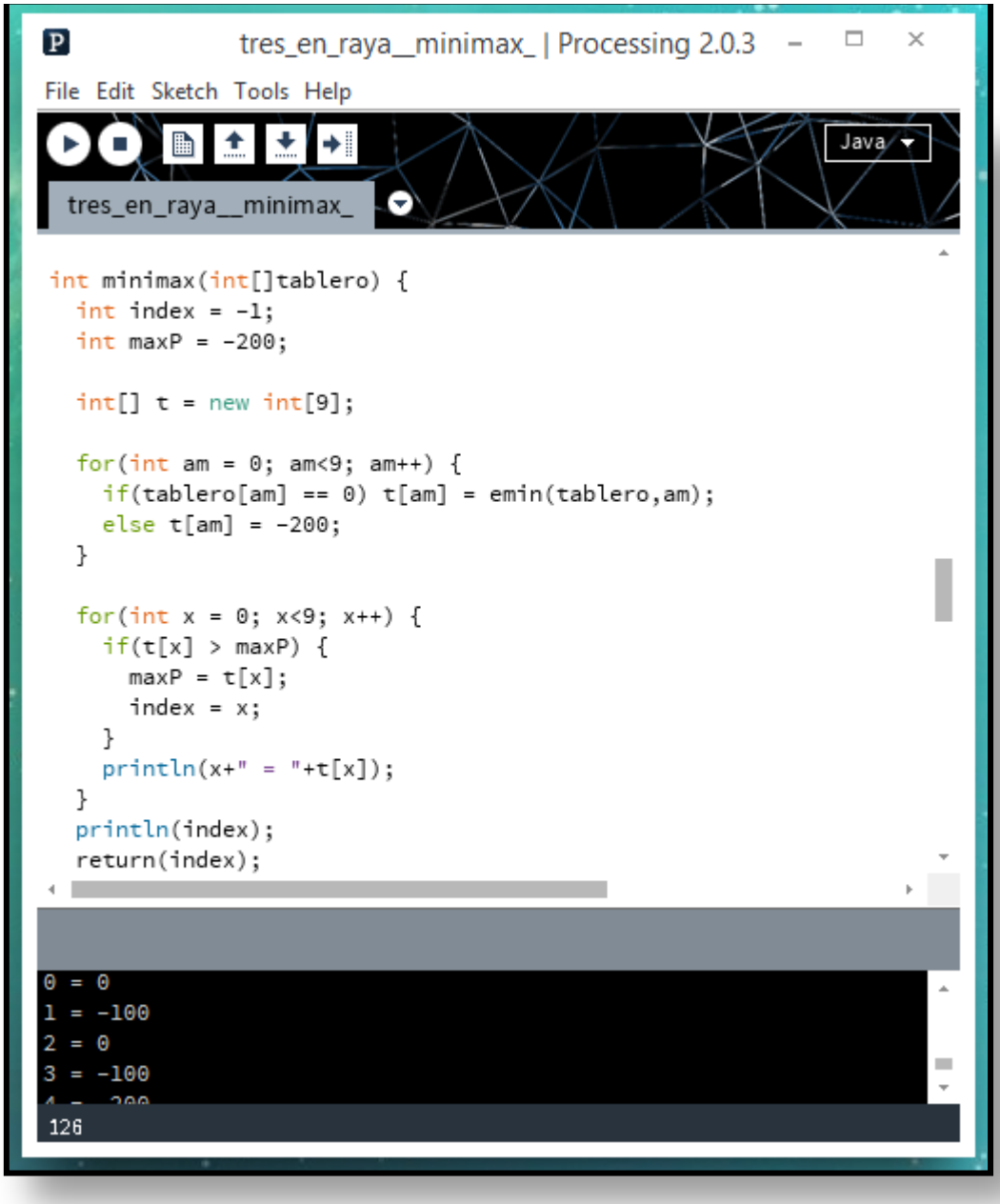
2 - Amb un llenguatge de programació que és una "versió" de JAVA però amb comandaments simplificats.

Quan es va crear?

És un projecte que porta ja uns anys, i que cada vegada té més rellevància en més camps d'aplicació. Actualment, està en un excel·lent estat de salut, cada vegada és més optimitzat, i amb Processing podem crear aplicacions per a totes les plataformes i dispositius amb una eficiència sense rival.

Cal destacar que amb Processing és possible exportar el codi tant a applets (web) com a Android o mòbils amb Java.

Com es pot veure en la següent imatge del programa, la interfície és molt senzilla.



A la part superior tenim els botons per executar, aturar i exportar el codi. Sota això tenim les pestanyes que ens ajuden a separar el codi i organitzar més, després la pantalla per escriure el codi del programa i finalment una consola, on es mostraran coses útils per al programador, com a valors de variables o errors del programa en si.

2.1.2 Python

Python és un llenguatge de programació interpretat la filosofia posa l'accent en una sintaxi que afavoreixi un codi llegible.

Es tracta d'un llenguatge de programació multiparadigma, ja que suporta orientació a objectes, programació imperativa i, en menor mesura, programació funcional. Això vol dir que més que forçar als programadors a adoptar un estil particular de programació, permet diversos estils: programació orientada a objectes, programació imperativa i programació funcional. Altres paradigmes estan suportats mitjançant l'ús d'extensions.

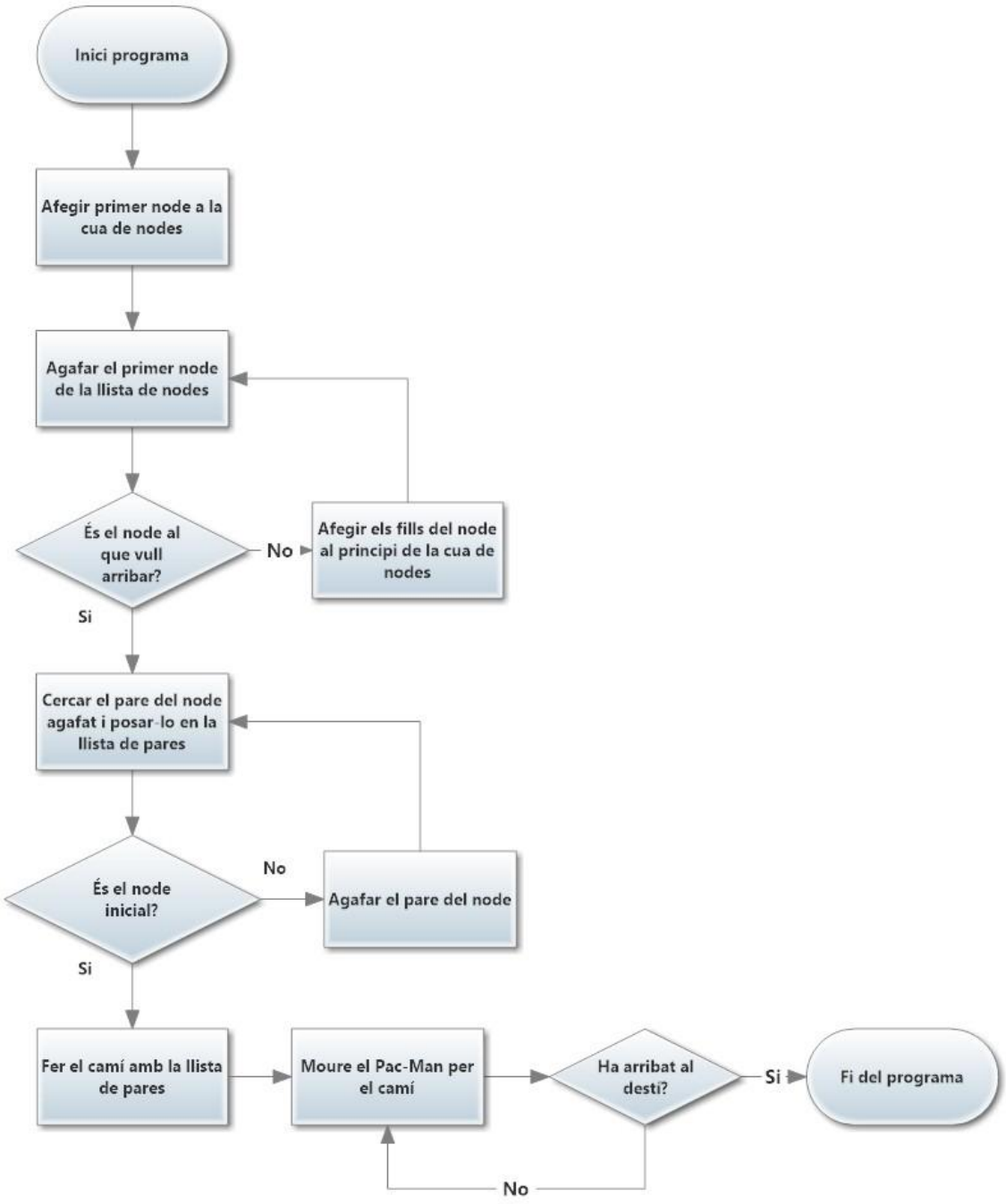
Python fa servir tipat dinàmic i recompte de referències per a l'administració de memòria. Una característica important de Python és la resolució dinàmica de noms; és a dir, el que enllaça un mètode i un nom de variable durant l'execució del programa (també anomenat enllaç dinàmic de mètodes).

Python no té una interfície de programació, ja que es poden utilitzar molts editors per programar el codi.

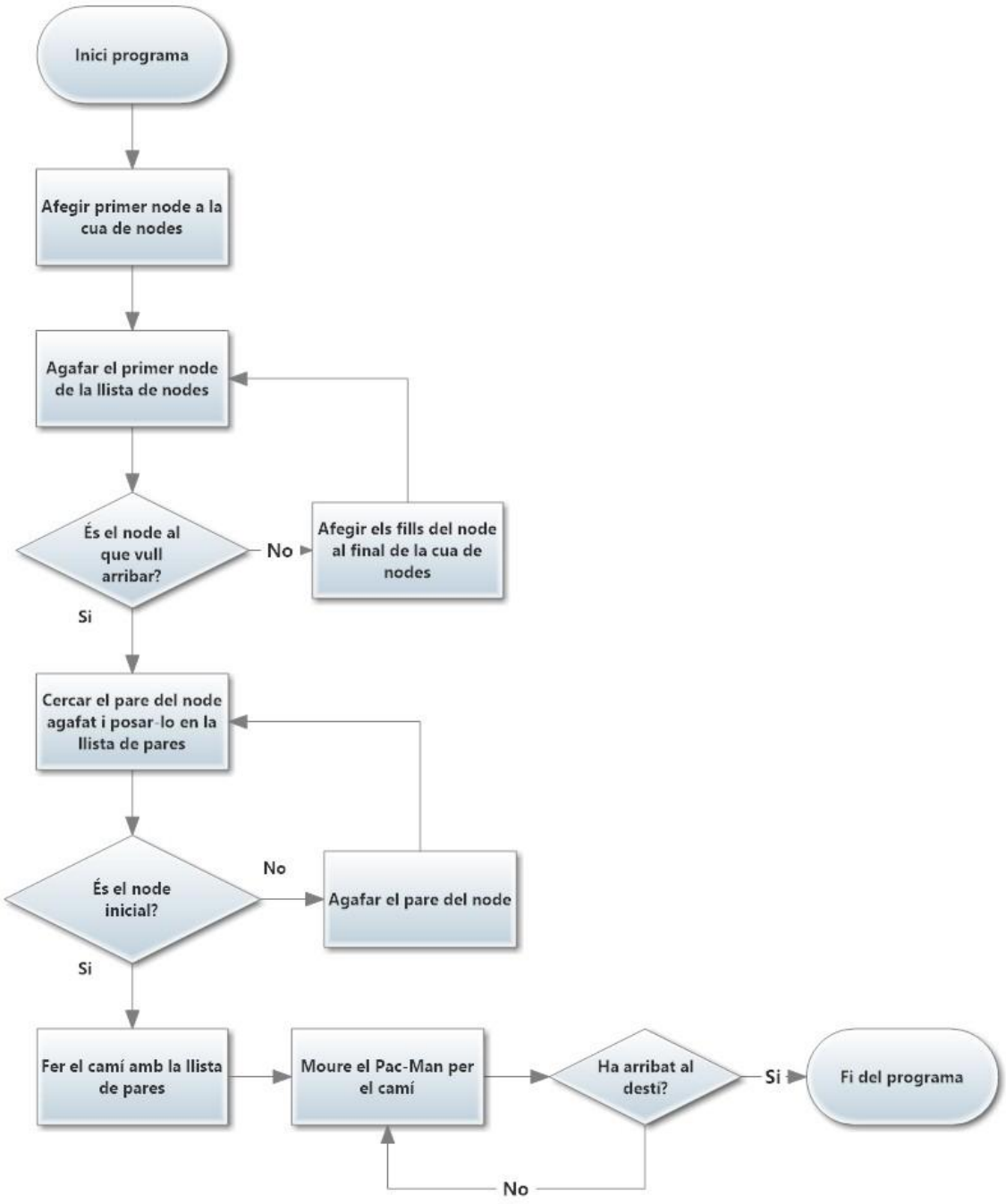
2.2 Jocs programats

2.2.1 Pac-Man

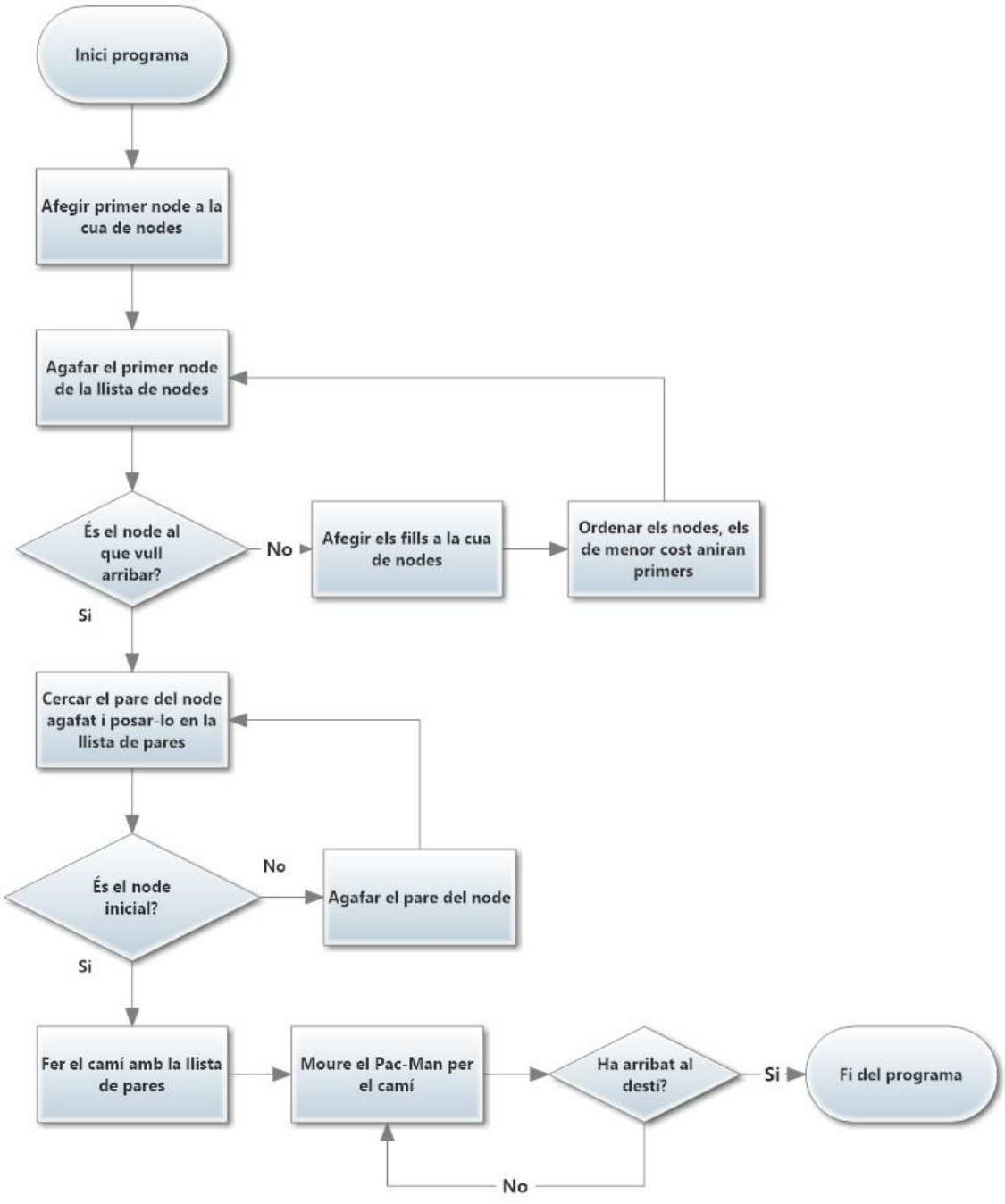
2.2.1.1 DFS



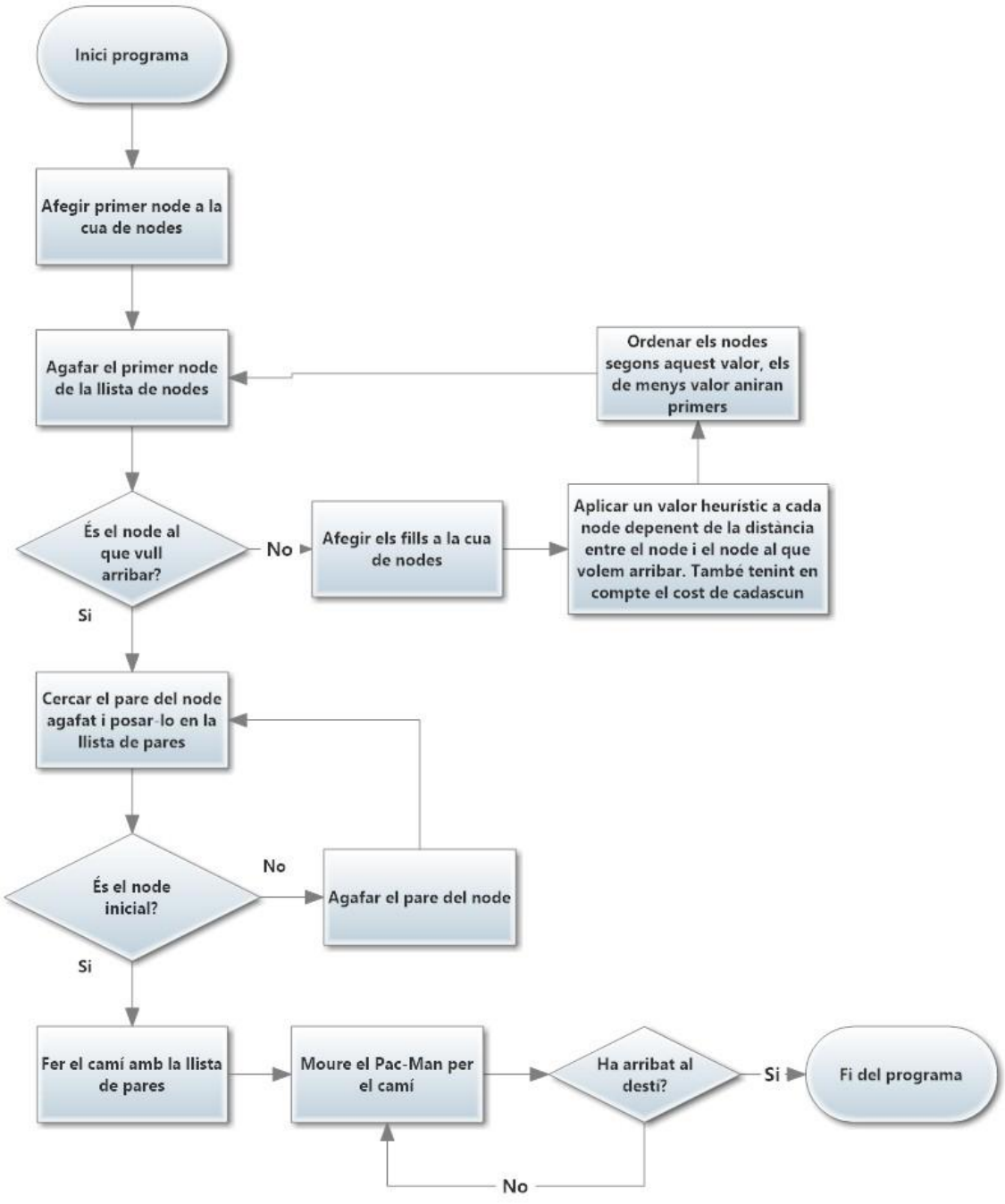
2.2.1.2 BFS


Inici programa
Afegir primer node a la cua de nodes
Agafar el primer node de la llista de nodes
És el node al que vull arribar?
No
Afegir els fills del node al final de la cua de nodes
Si
Cercar el pare del node agafat i posar-lo en la llista de pares
És el node inicial?
No
Agafar el pare del node
Si
Fer el camí amb la llista de pares
Moure el Pac-Man per el camí
Ha arribat al destí?
Si
Fi del programa
No

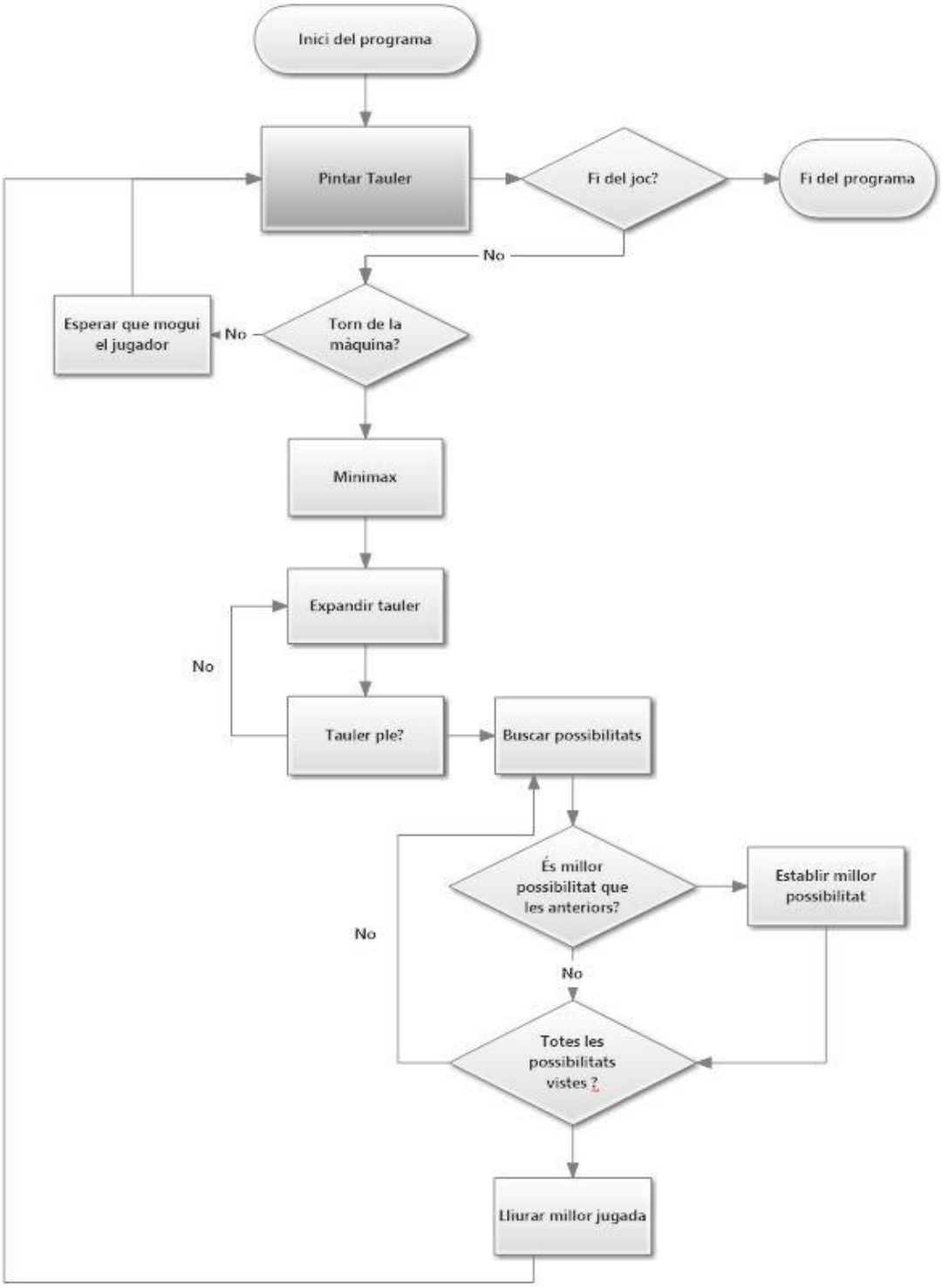
2.2.1.3 UCS


Inici programa
Afegir primer node a la cua de nodes
Agafar el primer node de la llista de nodes
És el node al que vull arribar?
No
Afegir els fills a la cua de nodes
Ordenar els nodes, els de menor cost aniran primers
Si
Cercar el pare del node agafat i posar-lo en la llista de pares
És el node inicial?
No
Agafar el pare del node
Si
Fer el camí amb la llista de pares
Moure el Pac-Man per el camí
Ha arribat al destí?
Si
Fi del programa
No

2.2.1.4 A*


Inici programa
Afegir primer node a la cua de nodes
Agafar el primer node de la llista de nodes
Ordenar els nodes segons aquest valor, els de menys valor aniran primers
És el node al que vull arribar?
No
Afegir els fills a la cua de nodes
Aplicar un valor heurístic a cada node depenent de la distància entre el node i el node al que volem arribar. També tenint en compte el cost de cadascun
Si
Cercar el pare del node agafat i posar-lo en la llista de pares
És el node inicial?
No
Agafar el pare del node
Si
Fer el camí amb la llista de pares
Moure el Pac-Man per el camí
Ha arribat al destí?
Si
Fi del programa
No

2.2.2 Tic Tac Toe



3.1 Conclusions

A principi d'aquest treball em vaig fixar diversos objectius: entendre com funciona una intel·ligència artificial, saber la història d'aquesta, i programar diverses IA simples.

Pel que fa al primer objectiu he aconseguit resoldre el dubte que sempre, des de petit, he tingut sobre com es crea la intel·ligència artificial dels jocs als que jugava. Podent ser més o menys complexa, totes es basen en algorismes d'alta complexitat.

Durant la realització del segon objectiu m'he anat adonant de lo jove que és aquesta àrea de la ciència i tot el que ha aconseguit en tan poc temps. Amb el pas dels anys podrem aconseguir moltíssimes coses gràcies a la intel·ligència artificial i la capacitat de còmput dels ordinadors que, exponencialment, va creixent. Em moro de ganes per saber el que el futur de la Intel·ligència Artificial ens podrà arribar a oferir.

I pel que consta al tercer objectiu, el de programar les IA, m'ha ensenyat que la dificultat que té desenvolupar una, per dos motius: primerament per la dificultat de programar que requereix uns alts nivells de programació, seguit de la dificultat d'entendre en si com funcionen els algoritmes, cosa gens fàcil atesa la seva complexitat.

3.2 Bibliografia

Stuart Russell. *Inteligencia Artificial Un enfoque moderno*. Prentice Hall. 1995.

http://en.wikipedia.org/wiki/DFS

http://es.wikipedia.org/wiki/Alan_Turing

http://www.bbc.co.uk/mundo/noticias/2012/06/120621_turing_inteligencia_artificial_lp.shtml

http://www.imagia.com.mx/hmm/va/Turing.htm

http://computerhoy.com/noticias/software/inteligencia-artificial-supera-65-anos-test-turing-14043

http://hipertextual.com/2012/02/inteligencia-artificial-john-von-neumann-y-el-nacimiento-de-la-vida-artificial

http://es.wikipedia.org/wiki/John_von_Neumann

http://es.wikipedia.org/wiki/B%C3%BAsqueda_de_Costo_Uniforme

http://en.wikipedia.org/wiki/Breadth-first_search

http://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura

http://es.wikipedia.org/wiki/Recursi%C3%B3n_(ciencias_de_computaci%C3%B3n)

http://picandocodigo.net/2008/recursividad-en-programacion/

http://www.javaya.com.ar/detalleconcepto.php?codigo=123&inicio=40

http://descuadrando.com/%C3%81rbol_de_decisi%C3%B3n

http://www.decision-making-confidence.com/ejemplos-de-arboles-de-decision.html

http://es.wikipedia.org/wiki/%C3%81rbol_de_decisi%C3%B3n

http://es.wikipedia.org/wiki/Teor%C3%ADa_de_juegos

http://www.eumed.net/cursecon/juegos/index.htm

http://es.wikipedia.org/wiki/%C3%81rbol_%28inform%C3%A1tica%29

http://es.wikipedia.org/wiki/%C3%81rbol_binario

http://es.wikipedia.org/wiki/Minimax

http://es.wikipedia.org/wiki/Heur%C3%ADstica_(inform%C3%A1tica)

http://es.wikipedia.org/wiki/Poda_alfa-beta

http://www.lsi.upc.edu/~bejar/ia/transpas/teoria/1-IA-introduccion.pdf

http://www.nebrija.es/~cmalagon/ia/transparencias/introduccion_IA.pdf

http://arteconcodigo.com/processing-p5/

http://en.wikipedia.org/wiki/History_of_artificial_intelligence

http://en.wikipedia.org/wiki/Timeline_of_artificial_intelligence

http://en.wikipedia.org/wiki/Artificial_intelligence

http://es.wikipedia.org/wiki/Inteligencia_artificial

http://hipertextual.com/2012/03/inteligencia-artificial-ramon-llull-ars-magna

http://blogs.elpais.com/turing/2012/10/ramon-llull-el-ars-magna-y-la-informatica.html

http://www.elisava.net/es/biblioteca/recursos-de-informacion/como-hacer-una-bibliografia

https://franciscoguemes.wordpress.com/2012/01/18/codigo-fuente-en-word/

3.3 Annexa

3.3.1 Codi de cerca Pac-Man en Python:

```
def busqueda(problem, tipoCola, colaPusher, heuristic):
    tengoSolucion = False
    listaNegra = util.Stack()
    cola = tipoCola
    colaPusher(cola, Nodo(problem.getStartState(), 0, None, None), 1)

    solucion = Nodo([0,0], 0, None, None)
    while cola.isEmpty() is False and tengoSolucion is False:
        nodo = cola.pop()
        while nodo in listaNegra.list:
            if cola.isEmpty():
                break;
            nodo = cola.pop()

        listaNegra.push(nodo)
        if problem.isGoalState(nodo.getState()):
            solucion = nodo
            tengoSolucion = True
            continue;
        hijos = problem.getSuccessors(nodo.getState())
        #print hijos

        for hijo in hijos:
            #print tengoSolucion
            b = hijo
            a = Nodo(b[0], b[1], b[2], nodo)
            if not a in listaNegra.list:
                #colaPusher(cola, a,
problem.getCostOfActions(encontrarCamino(a))+HeuristicaProaso(a, ))
                colaPusher(cola, a,
problem.getCostOfActions(encontrarCamino(a))+heuristic(a.getState(),
problem))
                #colaPusher(cola, a,
problem.getCostOfActions(encontrarCamino(a))+manhattanHeuristic(a.getS
tate(), problem))

    "IR ENCONTRANDO PARENTS"
    return encontrarCamino(solucion)

def encontrarCamino(solucion):
    camino = []
    selectedNodo = solucion
    while not selectedNodo.getParent() == None:
        #print selectedNodo.getParent()
        camino.append(selectedNodo.getDireccion())
        selectedNodo2 = selectedNodo.getParent()
        selectedNodo = selectedNodo2
    #camino.append(selectedNodo.getDireccion())
    return invertir(camino)

def invertir(var):
        return var[::-1]
```

3.3.2 Codi Tic-Tac-Toe:

```
//Version 2.1
//18 Diciembre 2014 -- Por Mario Ramos

int [] tablero = {
  0, 0, 0,
  0, 0, 0,
  0, 0, 0,
};

int xx, yy, pintar, q;

PImage logo, fin;
int ancho = 3;
int alto = 3;
int turno = 1;
int estado = 0;

void setup() {
  size(600, 600);
  imageMode(CENTER);
  logo = loadImage("logo.png");
  fin = loadImage("fin.png");
}

void draw() {
  switch(estado) {
  case 0:
    menu();
    break;
  case 1:
    juego();
    break;
  }
}
```

```
//0 = Empate o aun no se ha ganado
//1 = Gana jugador
//2 = Gana maquina

int Resultado(int[] tablerox) {
  // Iz - Der
  if (tablerox[1]==2 && tablerox[2]==2 && tablerox[0] == 2) return 2;
  if (tablerox[4]==2 && tablerox[5]==2 && tablerox[3] == 2) return 2;
  if (tablerox[7]==2 && tablerox[8]==2 && tablerox[6] == 2) return 2;

  // Arriba - Abajo
  if (tablerox[0]==2 && tablerox[3]==2 && tablerox[6] == 2) return 2;
  if (tablerox[1]==2 && tablerox[4]==2 && tablerox[7] == 2) return 2;
  if (tablerox[2]==2 && tablerox[5]==2 && tablerox[8] == 2) return 2;

  //Diagonales
  if (tablerox[0]==2 && tablerox[4]==2 && tablerox[8] == 2) return 2;
  if (tablerox[6]==2 && tablerox[4]==2 && tablerox[2] == 2) return 2;


  if (tablerox[1]==1 && tablerox[2]==1 && tablerox[0] == 1) return 1;
  if (tablerox[4]==1 && tablerox[5]==1 && tablerox[3] == 1) return 1;
```

```
  if (tablerox[7]==1 && tablerox[8]==1 && tablerox[6] == 1) return 1;

  // Arriba - Abajo
  if (tablerox[0]==1 && tablerox[3]==1 && tablerox[6] == 1) return 1;
  if (tablerox[1]==1 && tablerox[4]==1 && tablerox[7] == 1) return 1;
  if (tablerox[2]==1 && tablerox[5]==1 && tablerox[8] == 1) return 1;

  //Diagonales
  if (tablerox[0]==1 && tablerox[4]==1 && tablerox[8] == 1) return 1;
  if (tablerox[6]==1 && tablerox[4]==1 && tablerox[2] == 1) return 1;

  return(0);
}
```

```
int Minimax(int[] TableroFuncion) {
  int Indice = 0;
  int MaximaPuntuacion = -200; //Se inicializan las variables con
cualquier valor

  int[] TableroRespuestas = new int[9];
  arrayCopy(TableroFuncion, TableroRespuestas);

  //Expande el arbol
  for (int i=0; i<9; i++) {
    if (TableroRespuestas[i] == 0) TableroRespuestas[i] =
fMin(TableroFuncion, i);
    else TableroRespuestas[i] = -200; //Se le da este valor para que
no se seleccione en la busqueda de la mejor opcion, porque ya esta
usado.
  }

  //Busqueda de la mejor opcion
  for (int i=0; i<9; i++) {
    if (TableroRespuestas[i] > MaximaPuntuacion) {
      MaximaPuntuacion = TableroRespuestas[i];
      Indice = i; //Mejor posicion en el tablero
    }
    println(i+" = "+TableroRespuestas[i]);
  }

  println(Indice);
  return Indice;
}

int fMin(int []TableroFuncion, int Posicion) { //Movimiento de la
maquina
  int[] TableroRespuestas = new int[9];
  int[] NuevoTablero = new int[9];
  arrayCopy(TableroFuncion, NuevoTablero);

  NuevoTablero[Posicion] = 2;

  //Compruebo si puedo acabar
  if (Resultado(NuevoTablero) == 2) return 100; //Gana la maquina
  if (Resultado(NuevoTablero) == 1) return -100; //Gana el jugador lel
  if (TableroLleno(NuevoTablero)) return 0; //Empate

  //Sigo expandiendo
  for (int i=0; i<9; i++) {
```

```
    if (NuevoTablero[i] == 0) TableroRespuestas[i] =
fMax(NuevoTablero, i);
    else TableroRespuestas[i] = 200; //Se pone este valor para que no
lo seleccione al Returnear, porque ya esta usado.
  }

  if (min(TableroRespuestas) == 200) return(0); //Empate
  else return min(TableroRespuestas);
}

int fMax(int []TableroFuncion, int Posicion) { //Movimiento simulado
del humano
  int[] TableroRespuestas = new int[9];
  int[] NuevoTablero = new int[9];
  arrayCopy(TableroFuncion, NuevoTablero);

  NuevoTablero[Posicion] = 1;

  //Compruebo si puedo acabar
  if (Resultado(NuevoTablero) == 2) return 100; //Gana la maquina
  if (Resultado(NuevoTablero) == 1) return -100; //Gana el jugador lel
  if (TableroLleno(NuevoTablero)) return 0; //Empate

  //Sigo expandiendo
  for (int i=0; i<9; i++) {
    if (NuevoTablero[i] == 0) TableroRespuestas[i] =
fMin(NuevoTablero, i);
    else TableroRespuestas[i] = -200; //Se pone este valor para que no
lo seleccione al Returnear, porque ya esta usado.
  }

  if (max(TableroRespuestas) == -200) return 0; //Empate
  else return max(TableroRespuestas);
}

boolean TableroLleno(int[] ntablero) {
  if (ntablero[0] != 0 && ntablero[1] != 0 && ntablero[2] != 0 &&
ntablero[3] != 0 && ntablero[4] != 0 && ntablero[5] != 0 &&
ntablero[6] != 0 && ntablero[7] != 0 && ntablero[8] != 0) return true;
  else return false;
}
```

```
float tIA = 225;
void menu(){
  background(0);
  onda();

  tint(255, tIA);
  image(logo, width/2, 100);
  noTint();

  fill(0);
  text("Por Mario Ramos", 10, height-15);
  if (tIA > 0) tIA = tIA - 1;
  else estado=1;
}

float theta = 0.0;
float w=0.01,v=45,a,b,c,aa,bb,cc,tr,trr;
void onda() {
```

```
  if(random(0,20) < 2) b += 0.0001;
  if(random(0,20) < 2) b -= 0.0001;

  v += random(0,0.5); //velocidad +
  v -= random(0,0.5); //velodidad -

  theta += 0.02;
  noStroke();
  fill(200, 0, 0);
  float x = theta;

  for (int i = 0; i <= 130; i++) {
    float y = sin(x)*v; //la subida y bajada de bolas
    ellipse(i*5,y + 500,10,10);
    rect(i*5-5,y + 500-5,20,400);
    x += w; //bote
  }
}
```

```
void juego() {
  background(0);
  onda();
  fill(0);
  text("Por Mario Ramos", 10, height-15);
  PintarTablero();

  if (turno == 2) {
    tablero[Minimax(tablero)] = 2;
    turno = 1;
  } else {
    if (mousePressed == true) {
      xx = (mouseX/200);
      yy = (mouseY/200);
      if (turno == 1 && tablero[yy*ancho+xx] == 0) {
        tablero[yy*ancho+xx] = 1;
        turno = 2;
      }
    }
  }
  //println(Resultado(tablero));
}

boolean lleno = false;
void PintarTablero() {
  lleno = true;
  for (int y=0; y<alto; y++) {
    for (int x=0; x<ancho; x++) {
      pintar = tablero[y*ancho+x];
      if (pintar == 0){
        fill(225, 100);
        lleno = false;
      }
      if (pintar == 1) fill(0, 225, 0);
      if (pintar == 2) fill(0, 0, 225);
      rect(x*200, y*200, 200, 200);
    }
  }

  if(lleno)image(fin, width/2, 450);
}
```

3.3.3 Imatges Pac-Man i Tic Tac Toe