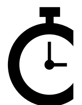
Intel·ligència artificial per a videojocs

PID_00269048

Jordi Duch i Gavaldà
Heliodoro Tejedor Navarro
Samir Kanaan Izquierdo
Gerard Escudero Bakx

Temps mínim de dedicació recomanat: 9 hores



Universitat
Oberta
de Catalunya

Jordi Duch i Gavaldà

Heliodoro Tejedor Navarro

Samir Kanaan Izquierdo

Gerard Escudero Bakx

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: Carles Ventura Royo (2019)

Primera edició: setembre 2019
© Jordi Duch i Gavaldà, Heliodoro Tejedor Navarro, Samir Kanaan Izquierdo, Gerard Escudero Bakx
Tots els drets reservats
© d'aquesta edició, FUOC, 2019
Av. Tibidabo, 39-43, 08035 Barcelona
Realització editorial: FUOC

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com químic, mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit dels titulars dels drets.

Índex

1. Intel·ligència artificial en els videojocs 5
1.1. Història de la intel·ligència artificial en els videojocs 5
1.2. Aplicacions de la intel·ligència artificial en els videojocs 8
1.3. Consideracions sobre la IA en videojocs 9
1.3.1. Ajustament de dificultat 9
1.3.2. Restringir el coneixement del joc 10
1.3.3. Temps real, cost computacional i completesa 11
1.4. Arquitectura de la IA 11
1.4.1. Arquitectura centralitzada 12
1.4.2. Arquitectura distribuïda: agents, sistemes multiagent, IA emergent 12

2. Tècniques de moviment 13
2.1. Moviments cíclics i basats en patrons 15
2.1.1. Moviments predefinits 15
2.1.2. Trajectòries lineals 15
2.1.3. Trajectòries corbes 16
2.2. Cerca de camins 19
2.2.1. Representació dels nivells com a grafs 20
2.2.2. Recorregut en profunditat 23
2.2.3. Recorregut en amplitud 25
2.2.4. Algorisme de Dijkstra 26
2.2.5. Algorisme A* 27
2.2.6. Altres variacions de la cerca de camins 30
2.3. Moviments complexos 31
2.3.1. Moviments de direcció 32
2.3.2. Moviments col·lectius (*flocking*) 34

3. Presa de decisions 37
3.1. El procés de presa de decisions 39
3.2. Sistemes de regles 39
3.3. Màquines d'estats finits 40
3.4. Exploració en arbre 47
3.4.1. Arbres de decisió 47
3.4.2. Arbres de joc: algorisme Minimax 49
3.4.3. Arbres de comportament 52
3.5. Lògica difusa 55
3.6. Mapes d'influència 57

4. Aprenentatge 60
4.1. Estratègies d'aprenentatge en jocs 60
4.1.1. Moments de l'aprenentatge 61

4.1.2. Modes d'aprenentatge 61
4.2. Algorismes d'aprenentatge 65
4.2.1. Accord.NET Framework 69
4.2.2. Naïve Bayes 74
4.2.3. kNN 77
4.2.4. Classificador lineal 80
4.2.5. Arbres de decisió 82
4.2.6. Algorisme ID3 89
4.3. Ajustament de paràmetres 91
4.3.1. Optimització amb algorismes genètics 92

5. Tècniques avançades d'IA.. 101
5.1. Xarxes neuronals 101
5.1.1. Components d'una xarxa neuronal 102
5.1.2. Funcions d'activació 103
5.1.3. Entrenament d'una xarxa neuronal 104
5.1.4. Problemes d'aprenentatge 105
5.1.5. Perspectiva històrica i futura 107
5.2. Aprenentatge profund 107
5.2.1. Arquitectures 108
5.2.2. Llibreries per a aprenentatge profund 109

1. Intel·ligència artificial en els videojocs

Quina és la característica més important perquè un videojoc tingui èxit? Segurament que entretingui els jugadors, de manera que mantingui una àmplia base d'usuaris i així el joc continuï atraient nous jugadors. Com s'aconsegueix que un videojoc entretingui els jugadors? Hi ha molts aspectes importants (idea original, disseny del joc, gràfics i so), però un aspecte fonamental és que ofereixi a l'usuari una experiència excitant, que no resulti avorrit per ser massa fàcil ni frustrant per ser molt difícil.

Però les coses no sempre són tan senzilles com ajustar el nivell de dificultat. En videojocs senzills (del tipus Tetris, Candy Crush i similars) la dificultat es pot ajustar simplement accelerant el joc o restringint el temps disponible. Però això no és aplicable a altres jocs més complexos, en els quals l'usuari espera reptes que posin a prova la seva intel·ligència i habilitat per a aconseguir vèncer en el joc.

I aquí entra l'objectiu principal de la intel·ligència artificial (IA) en els videojocs: dotar les diferents entitats amb les quals interactua l'usuari (soldats enemics, cotxes en una cursa, un altre equip de futbol...) d'un comportament que sembli intel·ligent, de manera que el jugador necessiti al seu torn esforçar-se per superar l'ordinador, però no apujant simplement els punts de vida dels enemics o accelerant els altres cotxes, sinó fent que el jugador tingui la sensació que, si el guanyen, l'han guanyat netament perquè són millors; així, quan sigui el jugador el que guanyi en el joc, la seva satisfacció serà major i, per tant, la seva afició pel joc creixerà.

Agents

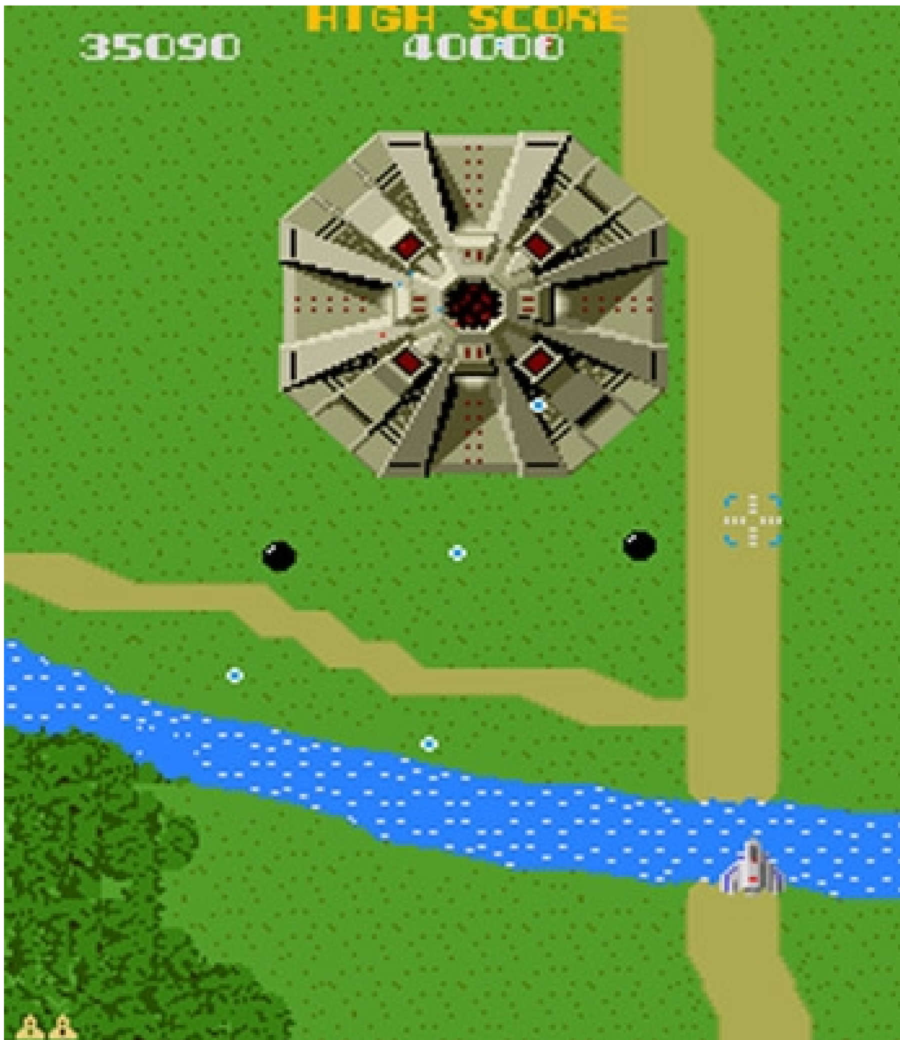
Denominarem **agents** les entitats del joc que hagin de mostrar algun tipus d'autonomia o comportament intel·ligent.

1.1. Història de la intel·ligència artificial en els videojocs

Les necessitats d'IA dels primers videojocs (desenvolupats al llarg dels anys setanta i vuitanta) eren bastant mínimes. Els jocs de plataformes clàssics tenien simplement una sèrie de petits programes o *scripts* (de diferent complexitat depenent del grau de «llestesa» amb què féssim el joc) que definien comportaments simples i repetitius dels enemics.

Moltes vegades, per a aconseguir guanyar un oponent (per exemple, el que es coneixia com «el monstre final de fase»), n'hi havia prou amb fixar-se durant un temps en el patró d'accions que feia de manera repetitiva, i després havíem d'ajustar les nostres accions per a contrarestar el patró.

Xevious



Font: © Namco

Al començament dels anys noranta van aparèixer nous gèneres amb moltes més necessitats d'IA que els jocs de plataformes que hi havia fins aleshores. Un exemple clar d'això és Dune II, primer joc d'estratègia en temps real, el qual requeria una IA capaç de planificar una estratègia en temps real i adaptar-se a les diferents situacions que li anava plantejant el jugador. Per a aquest tipus de jocs, es van començar a utilitzar màquines d'estats finits (que explicarem en l'apartat 3), que es combinaven amb tècniques de cerca de camins i tècniques de planificació d'estratègies. La introducció d'aquestes tècniques va incrementar la popularitat d'aquest tipus de jocs considerablement al llarg dels noranta, amb sagues com Starcraft, Warcraft, Command & Conquer o Age of Empires. L'ús d'aquestes tècniques també es va estendre a altres gèneres en els quals la IA exerceix un paper clau.

Dune II: Battle for Arrakis i Age of Empires II: The Age of Kings

Font: © Westwood Studios i Ensemble Studios respectivament

Jocs posteriors van començar a afegir tècniques més complexes d'IA que s'utilitzen principalment en altres camps, com per exemple l'ús de xarxes neuronals en el joc Battlecruiser 3000AD o l'ús de comportaments emergents (o evolutius) en jocs com Creatures o Black & White. Encara que la utilització d'aquestes tècniques no és molt comuna, són molt útils per a modelar certs comportaments que no podem crear usant tècniques determinístiques com *scripts* o patrons de regles.

Battlecruiser 3000AD i Black & White

Font: © Take Two Programari i Lionehad Studios respectivament

Més recentment, s'han inclòs noves tècniques en la IA de videojocs. Un dels elements que s'ha potenciat és el comportament en grup, és a dir, les interaccions i la coordinació entre agents intel·ligents. Un exemple d'aquest tipus de comportament el podem trobar en jocs com Far Cry, Halo o F.I.A.R, on els enemics es coordinen per envoltar el jugador o preparar-li emboscades, i arriben a utilitzar tàctiques militars que donen molt més realisme al comportament dels enemics.

Un altre dels últims avenços en IA ha consistit a intentar augmentar la reactivitat dels agents intel·ligents amb l'entorn que els envolta, seguint la línia de prendre les decisions segons un patró d'accions predeterminades en *scripts*. La principal gràcia d'aquest tipus de sistemes és que no es repeteix mai dues vegades la mateixa seqüència d'accions. Un dels exemples més destacats d'aquesta tecnologia està en les demostracions del motor Euphoria del joc Star Wars: The Force Unleashed, on els enemics són capaços d'agafar-se a una fusta per no caure o són capaços de salvar un company que és a prop.

Far Cry i Star Wars: The Force Unleashed

© Ubisoft i LucasArts respectivament.

La IA continua millorant dia a dia, principalment perquè la seva aplicació és multidisciplinària, i per tant les teories es poden copiar d'un camp a un altre i, a més, hi ha molta més gent implicada que millora les teories i els mètodes existents. Possiblement no som molt lluny del dia en el qual no podrem diferenciar si juguem contra altres jugadors o contra un personatge controlat per un ordinador.

El test de Turing

El test de Turing és un test que ens permet identificar l'existència d'intel·ligència en una màquina. Si no som capaços de diferenciar si ens comuniquem amb una persona o amb una màquina en usar aquest test, es considera que la màquina és «intel·ligent».

1.2. Aplicacions de la intel·ligència artificial en els videojocs

Si bé és possiblement l'aplicació més visible, la IA en els videojocs no solament s'utilitza per a manejar els agents contra els quals jugaran els usuaris, sinó que té moltes altres aplicacions igualment importants:

- La primera, com ja hem dit, és controlar els agents amb els quals competeix el jugador.
- En molts jocs també controla agents que acompanyen o ajuden el jugador, com els companys que poden triar-se en Diablo o els soldats aliats de videojocs com Call of Duty.
- Generalment, els agents del joc esperen que arribi el jugador per a activar el seu pla d'acció, són reactius; no obstant això, en alguns jocs, com en Alien: Isolation, és l'agent el que surt a la caça del jugador. Això implica un disseny diferent dels objectius de la IA.
- Un aspecte bàsic de molts jocs és aconseguir que les unitats es moguin en el nivell de manera creïble i eficient; és el que es coneix com a cerca de camins.
- En les etapes finals de desenvolupament del joc, és necessari ajustar-ne els paràmetres (vida de les unitats, potència dels cotxes, etc.) per a aconseguir un grau de dificultat apropiat; aquest problema pot ser molt complicat per l'estreta interrelació entre paràmetres, per la qual cosa es necessita aplicar tècniques d'optimització.

- Alguns jocs, com la sèrie Forza (curses de cotxes), aprenen dels jugadors per crear un agent que condueixi com ells, posant-lo la disposició d'altres jugadors per aconseguir adversaris més realistes.

- També veurem que últimament s'utilitzen tècniques innovadores d'IA perquè sigui l'ordinador el que jugui (és a dir, que faci de jugador). De moment, això són experiments i prova de tècniques, però segurament per aquest camí es revolucionarà la idea que tenim d'IA per a videojocs.

- Finalment, les tècniques que veurem en aquest mòdul són molt properes o fins i tot les mateixes que s'apliquen en altres problemes, com en robòtica, visió artificial, conducció automàtica, organització d'empreses i producció, economia, etc.

Agent que acompanya i ajuda el jugador en Diablo III

© Blizzard Entertainment.

1.3. Consideracions sobre la IA en videojocs

La IA que s'utilitza en videojocs té diverses característiques pròpies que no solen donar-se en altres sistemes que utilitzen IA. En aquest apartat en descrivim algunes.

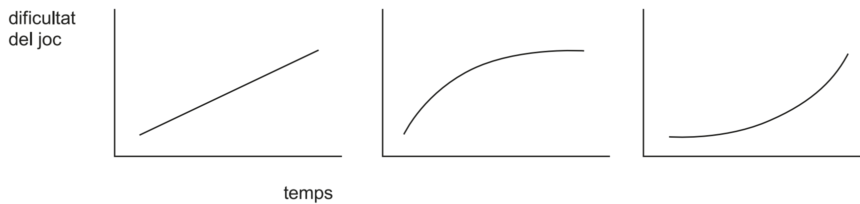
1.3.1. Ajustament de dificultat

En la majoria de les situacions en les quals s'aplica la IA es vol aconseguir els millors resultats possibles (classificar radiografies, predir huracans, conduir un cotxe, decidir quines accions comprar, etc.). En dissenyar la IA per a un videojoc és fàcil caure en la temptació d'intentar aconseguir una IA perfecta. Això és correcte, però l'equivocació és pensar que una IA perfecta per a un videojoc ha de jugar perfectament, ja que si ho fa el joc resultarà impossible per als jugadors humans. Tal com s'ha dit abans, el disseny d'agents contra els

quals jugaran els usuaris ha d'aconseguir un grau de dificultat just, ni molt baix ni molt alt. I així cal dissenyar la IA perquè s'equivoqui alguna vegada, per xocant que sembli al principi.

Això té molta relació amb el que es coneix com a **corba de dificultat** del joc, que representa gràficament com augmenta la dificultat del joc a mesura que s'hi avança. Habitualment, es tracta de corbes lineals, convexes (logarítmiques) o còncaves (exponencials). És important decidir el tipus de corba que es vol i ajustar la IA, juntament amb els altres paràmetres del joc, perquè l'experiència del jugador sigui l'esperada.

Exemples de corbes de dificultat: lineal, logarítmica i exponencial



1.3.2. Restringir el coneixement del joc

En la majoria de les aplicacions d'IA fora de l'àmbit dels videojocs, s'intenta que el sistema d'IA tingui accés a tota la informació que el pugui ajudar a executar les seves funcions i aconseguir els seus objectius. La IA d'un videojoc, però, no ha de tenir accés a tota la informació (per exemple, no ha de veure tot el mapa en un joc d'estratègia, o no ha de sentir el jugador fins que sigui a una certa distància), sinó que ha d'usar la mateixa informació que està disponible per als jugadors; en cas contrari, semblarà que la IA «fa paranys» (això es denomina *cheating AI* en anglès) i els jugadors s'indignaran amb el joc (amb raó).

> En el joc Civilization es van eliminar les aliances entre agents d'IA perquè funcionaven tan bé que els jugadors pensaven que l'ordinador feia paranys. La IA no solament no ha de ser tramposa, sinó que sobretot no ha de semblar-ho.

Per a aconseguir aquesta aparença de legalitat en la IA, s'ha d'evitar que els agents accedeixin a informació privilegiada del motor de joc; en el seu lloc, han d'usar informació «sensorial», que un humà podria veure o sentir en la seva situació.

També es pot considerar parany el fet que la IA s'utilitzi a fons i, per exemple, faci un nombre desorbitat d'accions per minut. La IA ha de semblar humana, no sobrehumana.

Jocs d'estratègia

En jocs d'estratègia com Starcraft, dels quals hi ha competicions internacionals d'esports electrònics (*electronic sports*) amb un gran seguiment i importància, es prenen les accions per minut que pot executar un jugador a manera d'indicador de la seva velocitat en el joc. El rècord mundial el té Park Sung-Joon, amb 818 accions per minut. Nosaltres tampoc no ens ho expliquem.

1.3.3. Temps real, cost computacional i completesa

Un aspecte fonamental de la IA en un videojoc és que, en la majoria dels jocs (excepte en els jocs per torns), ha de reaccionar en temps real a la situació del joc, i té un temps limitat per a donar la seva resposta, i recursos computacionals limitats: mentre que la IA treballa, la resta del joc ha de continuar movent-se, així que no pot ocupar tot el processador. Per tant, cal tenir en compte alguns factors:

- Potser no es podran usar els millors mètodes disponibles, sinó els més senzills i ràpids.

- Cal dissenyar bé on s'executarà la IA. Habitualment, se li dedicarà un fil d'execució (*thread*) a part perquè no bloquegi la resta del joc.

- En qualsevol cas, cal tenir en compte el maquinari existent: nombrosos nuclis, processador gràfic, memòria, etc. En alguns casos caldrà ajustar el consum de la IA en funció dels recursos disponibles en cada màquina.

- En els videojocs el bucle principal del joc és el que dicta els temps, que al seu torn solen dependre del dibuix de cada fotograma (*frame*), i aquest és variable perquè depèn del nombre de triangles en pantalla i d'altres efectes. Per tant, no hi ha un temps fix per a executar una iteració, i així la IA té aquestes opcions:
  – Utilitzar un disseny flexible que la faci capaç de donar un resultat aproximat si no disposa de temps suficient per a acabar.
  – Utilitzar un mètode amb una durada màxima inferior al temps mínim entre iteracions.
  – Executar-se en paral·lel al bucle principal de joc i ser capaç de donar resultats a demanda (execució asíncrona).

Els processadors gràfics

Amb l'extraordinari desenvolupament dels processadors gràfics, els processadors centrals queden relativament descarregats en executar videojocs, especialment si tenen diversos nuclis, així que queden més recursos per a la IA. El problema és que els mètodes més potents d'IA, com l'aprenentatge profund que es veurà en l'apartat 5, també volen executar els seus càlculs en el processador gràfic!

1.4. Arquitectura de la IA

Des d'un punt de vista de la seva organització, com és la IA d'un videojoc? Bàsicament, hi ha dues arquitectures, i sovint el mateix tipus de joc convida a utilitzar-ne una o l'altra.

1.4.1. Arquitectura centralitzada

Es programa una IA a imatge d'un jugador humà, que pot controlar diferents elements (fitxes, unitats, etc.) i coordina el seu ús des d'un pla general. En aquesta arquitectura se segueix una estratègia de dalt a baix (*top-down*), en la qual la IA decideix un pla d'acció general i dona totes les ordres pertinents perquè es dugui a terme aquest pla.

Si la IA controla diversos jugadors, es crearan diverses instàncies de l'arquitectura centralitzada.

Aquesta és la solució preferida per a jocs de tauler o d'estratègia.

1.4.2. Arquitectura distribuïda: agents, sistemes multiagent, IA emergent

En aquesta arquitectura, cada unitat del joc (cada soldat, pilot, esportista, robot, etc.) ha de mostrar un comportament autònom i intel·ligent, així que per a cada agent del joc hi haurà una instància de la IA, és a dir, un mètode amb les seves pròpies dades; es poden usar diferents mètodes per a diferents tipus d'agents. Els agents perceben el seu entorn, hi actuen i es poden comunicar amb altres agents. Per tant, s'acaba obtenint un sistema **multiagent**.

L'estratègia resultant és de baix a dalt (*bottom-up*), ja que comença en cada agent individual i les seves accions se sumen en el conjunt del joc. Si la interacció entre els diferents agents dona lloc a un comportament coordinat que supera la suma de les parts, es parla d'IA **emergent**.

Aquesta és la solució preferida en jocs en els quals es distingeixen les unitats individuals i els jugadors tenen un comportament autònom: jocs en primera persona, d'esports, etc.

Requereix establir mecanismes de comunicació entre agents.

Els sistemes emergents

Els **sistemes emergents** són els sistemes compostos d'entitats relativament senzilles però que en interactuar donen lloc a comportaments complexos. Exemples d'aquests sistemes són les neurones, les formigues.

2. Tècniques de moviment

Probablement, la primera característica dels agents d'un joc que doni la sensació de vida pròpia, i per tant d'intel·ligència, és el moviment. En la majoria dels videojocs apareixen diferents elements (enemics o competidors, objectes que cal atrapar, etc.) que han de moure's d'una manera o altra per a donar sentit al joc. També cal tenir en compte que la IA del joc pot veure's en l'obligació de moure les unitats del jugador, com per exemple en els jocs d'estratègia en temps real, en els quals el jugador selecciona algunes unitats i marca en el mapa a on vol que vagin.

En aquest apartat veurem tècniques que permeten moure els elements del joc de manera adequada, tenint en compte els criteris de disseny del moviment següents:

- **Predictibilitat.** Segons el tipus de joc i el seu disseny, el moviment pot ser predictible, com en un joc de plataformes on els enemics sempre fan el mateix recorregut (ja que l'objectiu del joc és trobar una seqüència de moviments que permeti superar el nivell), o impredictible, com en un joc de trets en primera persona del tipus «arena», en el qual utilitzar agents que es moguessin de manera predictible arruïnaria el joc completament.

- **Realisme.** El tipus de moviment ha d'adequar-se al tipus d'entitat; no es mou igual un camió que un ratolí (en el sentit que no fan el mateix tipus de recorreguts, no en el sentit que l'animació sigui diferent). Els agents del joc poden caminar, córrer, volar, nedar, etc., i en cada cas el traçat de camins serà diferent.

- **Comportament.** Quin tipus de comportament es vol representar? El moviment dels agents ha de reflectir la seva intenció; els tipus de moviment que podem trobar segons aquest criteri són moviment casual (passeig), cerca, persecució, fugida, aguait, moviment grupal, etc.

- **Evitació.** Cada nivell d'un joc té una sèrie de zones per les quals el moviment és permès i unes altres per les quals no ho és; poden ser estàtiques (parets) o dinàmiques (portes que s'obren, barrils que es poden trencar). A més, cada tipus de terreny pot tenir un cost implícit associat que recomani evitar-les (pendents, fang, etc.), o bé pot convenir evitar-les per estar controlades per jugadors o IA enemics.

Per donar resposta als criteris anteriors, veurem diferents tipus de tècniques de moviment:

- **Moviments cíclics i basats en patrons.** En dissenyar el nivell del joc es dissenyen una sèrie de recorreguts per als agents, que seguiran de manera determinista i repetitiva.

- **Cerca de camins.** Quan els agents necessiten trobar un camí de manera dinàmica, la IA ha de fer una cerca en el mapa.

- **Moviments complexos.** De vegades el moviment que es vol no és simplement «anar de A a B», sinó aconseguir objectius dinàmics, com interceptar una altra unitat en moviment (per exemple, una nau dispara un raig làser a una altra), o coordinar-se amb altres unitats que estan en moviment (grups d'enemics que ataquen junts, per exemple). Aquestes situacions comporten uns problemes específics per als quals es veuran algunes solucions.

Abans de començar amb les tècniques de moviment, cal definir alguns conceptes generals que s'usaran en aquest apartat:

- **Zones de mobilitat.** En dissenyar un nivell de joc, és necessari definir per on es poden moure les unitats i per on no; és a dir, la IA ha de saber que hi ha una paret o un riu o qualsevol altre obstacle. La manera habitual de representar aquesta informació és afegint en el mapa de cada nivell una espècie de capa invisible que indica per on es poden moure les unitats. Depenent del disseny del joc i de l'entorn de desenvolupament, les zones de mobilitat s'han de crear manualment o és el mateix entorn el que les calcula. En qualsevol cas, solament s'han de tenir en compte els obstacles permanents, ja que recalcular les zones de mobilitat d'un nivell pot ser molt costós.

Unity

Afortunadament, Unity facilita la tasca de moviment de les unitats mitjançant la **Navigation Mesh** (o simplement **NavMesh**), en la qual Unity calcula per quines zones es poden moure els agents en el nivell, i així aquestes zones de mobilitat ja calculades es poden usar en el joc per a controlar el moviment dels agents. Més endavant es parla de NavMesh.

- **Obstacles.** En alguns jocs pot haver-hi obstacles temporals (barrils, caixes o portes, unitats que es mouen, bloquejos fins a complir-se alguna condició) que per raons d'eficiència no convé incorporar a les zones de mobilitat, sinó tractar com a obstacles dinàmicament en el moment de calcular els moviments de les unitats.

- **Característiques dels agents.** Per a poder calcular el moviment d'un agent per un nivell, cal conèixer-ne la grandària. Generalment, els agents bípedes es representen mitjançant un cilindre pel que fa al moviment, per la qual cosa les seves característiques definitòries són el radi i l'altura. També se sol tenir en compte el màxim que poden saltar i el pendent màxim que poden pujar. Altres tipus d'agent (vehicles, animals) se solen representar mitjançant caixes.

Finalment, situant el control del moviment dels agents en el context general de la IA del joc, en general el moviment és un instrument bàsic perquè els agents puguin dur a terme els seus objectius generals en el joc: atacar el jugador, cooperar amb ell, competir contra ell, entre d'altres. Així, els agents

prendran una sèrie de decisions d'alt nivell, que anomenarem «estratègia» de l'agent, i trobar el camí per a aconseguir els seus objectius és un dels primers elements que s'han de proporcionar a l'agent perquè la IA funcioni en el joc.

2.1. Moviments cíclics i basats en patrons

2.1.1. Moviments predefinits

La manera més senzilla de dotar de moviment els agents del joc és definir una trajectòria i fer que la segueixin. El cas més bàsic dins d'aquesta categoria consisteix a «dibuixar» la trajectòria de l'agent en dissenyar el nivell. Encara que aquesta tècnica tingui moltes limitacions, té la seva utilitat en jocs de tipus plataforma o arcade, en els quals el jugador espera que els agents segueixin sempre el mateix moviment.

Generalment, aquest tipus de moviment es defineix sobre rutes cícliques. També es pot fer que quan l'agent arribi a un extrem del recorregut, doni la volta i faci el recorregut en sentit contrari. Altrament, l'agent es detindria en acabar el recorregut i la dinàmica del joc s'alteraria.

2.1.2. Trajectòries lineals

Una alternativa senzilla que no requereix dibuixar el recorregut complet consisteix a indicar a l'agent quins són els punts d'inici i final del seu recorregut i fer-li seguir la línia recta que porta d'un punt a un altre. L'execució d'aquesta trajectòria consisteix a calcular, per a cada instant de temps en el qual es vulgui actualitzar la posició de l'agent (habitualment per a cada fotograma del joc), la posició de l'agent sobre la recta. Suposant que la velocitat és constant i que t és una variable entre 0 i 1 que indica la proporció de camí recorregut (0 a l'inici, 1 en acabar), l'expressió que calcula la posició de l'agent en el moment t del recorregut és:

La velocitat d'un cos

Estrictament parlant, la velocitat d'un cos és un vector de tres components que indica en quina direcció es mou i amb quina «velocitat» ho fa, que és el mòdul del vector velocitat i que es denomina celeritat per distingir-lo.

$$P(t) = (1-t)\, P_{inici} + tP_{fi}$$

És a dir, es calcula una interpolació lineal entre els dos extrems del recorregut.

Interpolació

S'anomena interpolació l'estimació d'una funció que passi sobre un conjunt de punts donats. Per exemple, la recta que tracem entre dos punts és una interpolació lineal. No obstant això, si hi ha més de dos punts no sempre es pot traçar una recta que passi per tots; per aquest motiu es recorre a funcions d'un altre tipus, en general amb algun tipus de curvatura que permeti ajustar-se millor a tots els punts encara que no estiguin en línia recta.

L'agent estarà orientat en la mateixa direcció del moviment. Generalment, la direcció dels agents s'expressa com un vector unitari (de mòdul 1), de manera que si la velocitat de l'agent és V estarà orientat en la direcció següent:

$$D(t) = \frac{V(t)}{|V(t)|}$$

2.1.3. Trajectòries corbes

Quan es vol definir una trajectòria que passi per més de dos punts, es poden anar encadenant trajectòries lineals, però el resultat serà bastant forçat perquè farà que l'agent es vagi movent en línies rectes i giri en angles bruscament, la qual cosa restarà realisme al joc (tret que l'agent sigui un robot o una cosa per l'estil). Per aquesta raó, quan es defineix una trajectòria amb més de dos punts, s'utilitza algun tipus de corba que recorri els punts sense generar angles.

En informàtica gràfica i videojocs, les interpolacions mitjançant corbes se solen fer amb les corbes anomenades ***spline***, que es defineixen a partir d'un conjunt de **punts de control**.

Hi ha diferents tipus de *splines*, com les corbes Bézier o les corbes Catmull-Rom. En alguns tipus de corbes pot ocórrer que la corba no passi exactament pels punts de control sinó a prop. Això pot ser un inconvenient en un joc, i per aquesta raó usarem les corbes Catmull-Rom, que sempre passen pels punts de control.

Les corbes Catmull-Rom

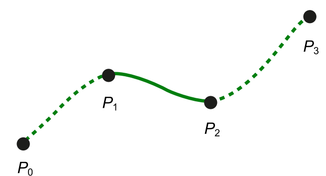
Les corbes Catmull-Rom deuen el nom als seus creadors, Edwin Catmull i Raphael Rom. Edwin Catmull és, entre altres coses, el creador de Pixar Animation Studios; això ja ens hauria de fer convèncer de l'interès que tenen aquestes corbes.

Les corbes Catmull-Rom es defineixen mitjançant quatre punts de control $\{P_0,P_1,P_2,P_3\}$, encara que la corba per si mateixa solament es dibuixa entre els dos punts centrals $\{P_1,P_2\}$. Si es vol dibuixar una corba Catmull-Rom sobre més punts, cal calcular el que es coneix com **una cadena** de corbes desplaçant els punts de control dins de la seqüència de punts, és a dir, calculant la corba per a cada subconjunt de punts $\{P_{n-1},P_n,P_{n+1},P_{n+2}\}$.

Spline

Originàriament, un *spline* era una tira flexible de fusta o metall que s'usava com a guia per a dibuixar corbes.

Un aspecte important del moviment al llarg d'una corba definida per diversos punts de control és que la velocitat de l'agent en recórrer la corba ha de ser constant (tret que el joc requereixi una altra cosa); com que els punts de control no cal que siguin equidistants uns dels altres, no es pot calcular la velocitat (la celeritat) independentment en cada tram, ja que això provocaria que l'agent es mogués més ràpid en els trams més llargs.



Corba Catmull-Rom amb quatre punts de control

Font: Hadunsford - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=28956755

La solució consisteix a calcular la longitud total del camí i calcular un factor de compensació per a cada tram.

> Tenim un *spline* amb 5 punts de control (és a dir, 4 trams) de longituds 2, 4, 3 i 1 metres. La suma total de longituds és, per tant, 10 metres. Si tot l'*spline* s'ha de recórrer en, diguem, 20 segons, la celeritat hauria de ser en tot moment de 10/20 = 0,5 m/s.
>
> El temps invertit a recórrer cada tram haurà de ser diferent, proporcional a la longitud total i a la durada que es vol per al recorregut. Així, el tram de 2 metres s'ha de recórrer en 2 m / (0,5 m/s) = 4 s, el de 4 metres en 4 m / (0,5 m/s) = 8 s, etc. Així es manté una celeritat constant.

El codi següent en C# per Unity calcula la corba Catmull-Rom a partir dels seus quatre punts de control.

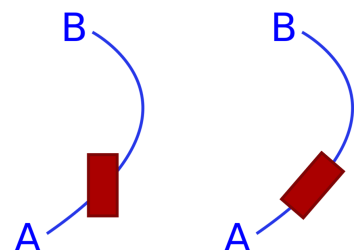
```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Catmull : MonoBehaviour {

//Use GameObject in 3d space as your points or define array with desired points
   public GameObject[] points;

   //Store points on the Catmull curve so we can visualize them
   List<Vector2> newPoints = new List<Vector2>();

   //How many points you want on the curve
   float amountOfPoints = 10.0f;

   //set from 0-1
   public float alpha = 0.5f;

   /////////////////////////////

   void Update()
   {
      CatmullRom();
   }

   void CatmullRom()
   {
      newPoints.Clear();

      Vector2 p0 = new Vector2(points[0].transform.position.x, points[0].transform.position.y);
      Vector2 p1 = new Vector2(points[1].transform.position.x, points[1].transform.position.y);
      Vector2 p2 = new Vector2(points[2].transform.position.x, points[2].transform.position.y);
      Vector2 p3 = new Vector2(points[3].transform.position.x, points[3].transform.position.y);

      float t0 = 0.0f;
      float t1 = GetT(t0, p0, p1);
      float t2 = GetT(t1, p1, p2);
      float t3 = GetT(t2, p2, p3);

      for(float t=t1; t<t2; t+=((t2-t1)/amountOfPoints))
      {
         Vector2 A1 = (t1-t)/(t1-t0)*p0 + (t-t0)/(t1-t0)*p1;
         Vector2 A2 = (t2-t)/(t2-t1)*p1 + (t-t1)/(t2-t1)*p2;
         Vector2 A3 = (t3-t)/(t3-t2)*p2 + (t-t2)/(t3-t2)*p3;
```

```
            Vector2 B1 = (t2-t)/(t2-t0)*A1 + (t-t0)/(t2-t0)*A2;
            Vector2 B2 = (t3-t)/(t3-t1)*A2 + (t-t1)/(t3-t1)*A3;

            Vector2 C = (t2-t)/(t2-t1)*B1 + (t-t1)/(t2-t1)*B2;

            newPoints.Add(C);
         }
      }

      float GetT(float t, Vector2 p0, Vector2 p1)
      {
         float a = Mathf.Pow((p1.x-p0.x), 2.0f) + Mathf.Pow((p1.y-p0.y), 2.0f);
         float b = Mathf.Pow(a, 0.5f);
         float c = Mathf.Pow(b, alpha);

         return (c + t);
      }

      //Visualize the points
      void OnDrawGizmos()
      {
         Gizmos.color = Color.red;
         foreach(Vector2 temp in newPoints)
         {
            Vector3 pos = new Vector3(temp.x, temp.y, 0);
            Gizmos.DrawSphere(pos, 0.3f);
         }
      }
   }
```

Referència

Font: Wikipedia https://en.wikipedia.org/wiki/centripetal_catmull%I2%80%93Rom_spline

D'altra banda, quan un agent es mou al llarg d'una corba, generalment ha d'anar orientant-se en el sentit de la marxa perquè el seu moviment sigui creïble. Calcular l'orientació d'un agent al llarg d'una corba no és tan senzill com al llarg d'una recta. En aquest cas, la seva orientació serà donada pel vector tangent a la corba en el punt en el qual es troba l'agent en aquest moment.



Pel que fa a la trajectòria i orientació, si el cotxe vol anar de A a B seguint la trajectòria, encara que la seva destinació final sigui B no ha de ser orientat cap a B sinó tangent a la trajectòria que seguirà.

Per a indicar a Unity que un agent ha d'ajustar la seva orientació al camí seguit, cal activar simplement la seva propietat *updateRotation*:

```
using UnityEngine;
using System.Collections;

public class MoveTo : MonoBehaviour {

    void Start () {
       NavMeshAgent agente = GetComponent<NavMeshAgent>();
       agente.transform.updateRotation = true
   }
}
```

2.2. Cerca de camins

En la majoria dels videojocs, controlar el moviment dels agents no és tan senzill com predefinir una trajectòria, ja que sovint es trobaran amb obstacles, tant relativament estàtics (caixes, barrils, portes tancades) com dinàmics (altres agents o unitats controlades pel jugador).

A més, no sempre és possible predefinir una trajectòria en dissenyar el nivell, ja que moltes vegades l'objectiu que hem d'aconseguir dependrà del desenvolupament del joc, i en molts casos els nivells són generats dinàmicament o bé els jugadors poden crear nous nivells.

Per totes aquestes raons, sovint és necessari calcular el recorregut dels agents de manera dinàmica, buscant el camí més adequat a cada moment per a aconseguir l'objectiu. En la cerca de camins poden influir molts factors:

- L'estructura mateixa del nivell: obstacles fixos i insalvables (parets, muntanyes, rius, etc.) i zones de mobilitat.

- Els punts d'origen i destinació del moviment: aquest pot canviar durant el mateix recorregut del camí, per exemple perquè es tracta d'atrapar un agent en moviment.

- Els obstacles temporals, com caixes, portes que s'obren i tanquen, barricades, etc., que poden aparèixer i desaparèixer al llarg del desenvolupament del joc.

- Els altres agents del joc, o les unitats controlades pels jugadors.

- Les característiques del terreny: en alguns jocs el cost de passar per uns tipus de terreny pot variar (per exemple, costa més caminar sobre sorra o fang, o costa menys caminar sobre camins o carreteres); també pot ser que es tingui en compte el pendent del terreny per incrementar el cost d'anar costa amunt.

- La situació del joc: en un joc d'estratègia, una unitat no ha de creuar per una zona controlada per l'enemic per a anar a una altra zona.

- Les característiques de l'agent: la seva grandària, els pendents màxims que pot pujar, si pot nedar o volar, la penalització per cada tipus de terreny, etc.

- Possibles dreceres, com teletransportadores o llocs en els quals es pot saltar a un altre nivell.

En aquest apartat estudiarem com generar una representació dels nivells del joc que permeti aplicar diferents algorismes per a trobar camins. A més, aquests camins hauran de tenir en compte les característiques aplicables a cada joc d'entre les llistades anteriorment.

2.2.1. Representació dels nivells com a grafs

Tal com es veurà en els apartats següents, els mètodes d'obtenció de camins són realment mètodes de cerca de camins en grafs. Però què té a veure un graf amb el nivell del meu joc si tinc rius, muntanyes, ponts, habitacions, escales...?

Exemple de mapa de joc amb zones transitables i no transitables. Generat amb l'editor de mapes de Battle for Wesnoth (de programari lliure). Les marques blaves que es veuen a banda i banda del riu a la zona inferior són portals de teletransport.

Graf

Un graf és una estructura de dades que conté dos tipus d'elements: d'una banda, hi ha els nodes o vèrtexs (els cercles en la imatge), i, de l'altra, els enllaços o arestes (les línies). Els grafs permeten representar en l'ordinador conjunts d'entitats (vèrtexs) relacionades entre elles (arestes). Per exemple, els usuaris d'una xarxa social poden ser els vèrtexs, i les seves relacions d'amistat, les arestes. Les arestes d'un graf poden tenir un valor associat, per exemple, en un graf en el qual les ciutats d'un país són els vèrtexs i les arestes són les autopistes que els uneixen, amb la distància quilomètrica associada a cada aresta.

Per a aconseguir que els agents busquin els seus camins en els nivells del joc, cal construir un graf de mobilitat, i, per a crear-lo, cal procedir en dos passos:

1) Trobar totes les zones de mobilitat **convexes** que hi hagi en el nivell. Cada zona serà un vèrtex del graf que representa la mobilitat en el nivell.

2) Determinar quines zones estan connectades (és possible moure's directament d'una a una altra). Cada connexió possible serà una aresta entre els vèrtexs corresponents a aquestes zones.

Què vol dir que una zona de mobilitat és convexa? Intuïtivament, es diu que una superfície és convexa quan el seu límit no té zones que es fiquin «cap a dins» creant entrades o badies; més formalment, una superfície és convexa quan és possible unir dos punts qualssevol del seu interior mitjançant una línia recta sense sortir de la superfície.

Precisament aquesta propietat és la que ens interessa, ja que garanteix que un agent es pugui moure lliurement dins de la zona; per tant, el problema queda reduït a aconseguir que l'agent vagi de zona en zona fins a arribar a la zona que contingui el punt de destinació. I aquí és on ens convé tractar les zones com a vèrtexs d'un graf i aplicar algorismes de cerca de camins en grafs, per a navegar de zona en zona.

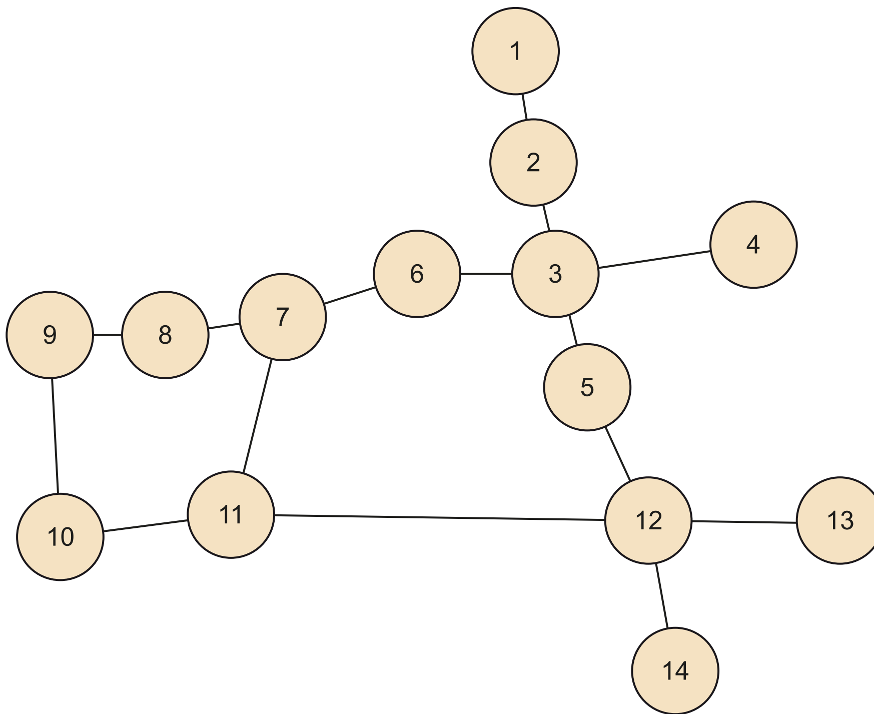
Aquest plantejament permet afegir salts entre zones no contigües, com portals, salts entre diferents nivells, etc., simplement afegint altres arestes al graf de mobilitat.

En la figura següent es poden veure les zones de mobilitat per al mapa anterior; per a no estendre excessivament les explicacions posteriors, no s'han generat les zones per al mapa complet, però la idea seria similar. Noteu que cada zona és convexa, és a dir, no hi ha cap angle obtús en els vèrtexs dels polígons.

Mapa anterior amb les zones de mobilitat superposades. Es fa ressaltar amb una fletxa la connexió entre els portals.

La traducció del mapa de mobilitat anterior a un graf es fa generant un vèrtex per a cada zona i enllaçant zones adjacents mitjançant enllaços entre vèrtexs. Així, el graf resultant seria:

Graf de mobilitat corresponent al mapa d'exemple.



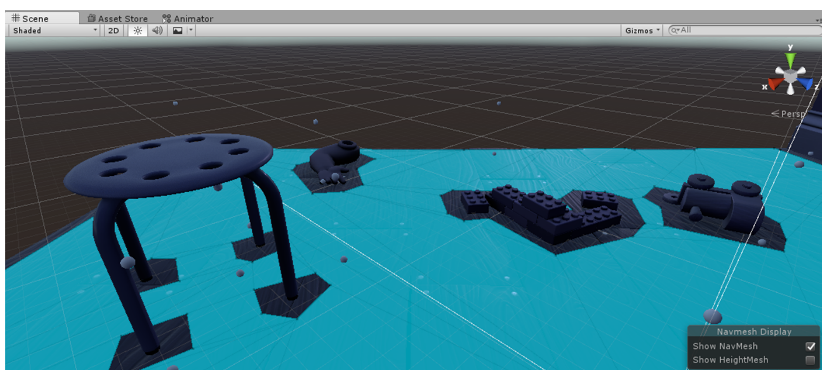
Noteu que la situació dels vèrtexs no cal que coincideixi amb les de les zones en el mapa; l'única cosa que importa són els enllaços entre zones.

Grafs de mobilitat

Unity calcula grafs de mobilitat i els utilitza per a calcular el moviment dels agents. És l'anomenat NavigationMesh, o NavMesh per als amics, en el qual Unity calcula el graf a partir de la geometria del nivell, i a partir d'aquí es pot usar per a moure els agents. Hi ha un apartat del manual que explica com generar el NavMesh:

https://docs.unity3d.com/manual/nav-buildingnavmesh.html.

NavMesh



NavMesh (blau clar) en el projecte *Survival Shooter* d'Unity.

Solament queda per veure com trobar el millor camí entre dos vèrtexs d'un graf per a poder calcular el camí que ha de seguir un agent per a anar de A a B. A continuació veurem diferents algorismes que resolen aquest problema, des dels més senzills (i ineficients) fins als més eficients, i finalment veurem

algunes variants per tenir en compte aspectes addicionals com el cost de caminar sobre diferents tipus de terreny, la necessitat d'evitar obstacles mòbils o d'evitar les zones que controli l'enemic, per exemple.

2.2.2. Recorregut en profunditat

Una manera bastant directa de recórrer completament un graf és fer-ho per **profunditat**: si partim d'un vèrtex (1 en la figura del graf de mobilitat), l'estratègia consisteix a anar al primer vèrtex veí (2), i d'aquest al seu primer veí, i així successivament. Per no entrar en cicles i repetir vèrtexs, es manté una llista en la qual s'anoten tots els vèrtexs que ja s'han visitat.

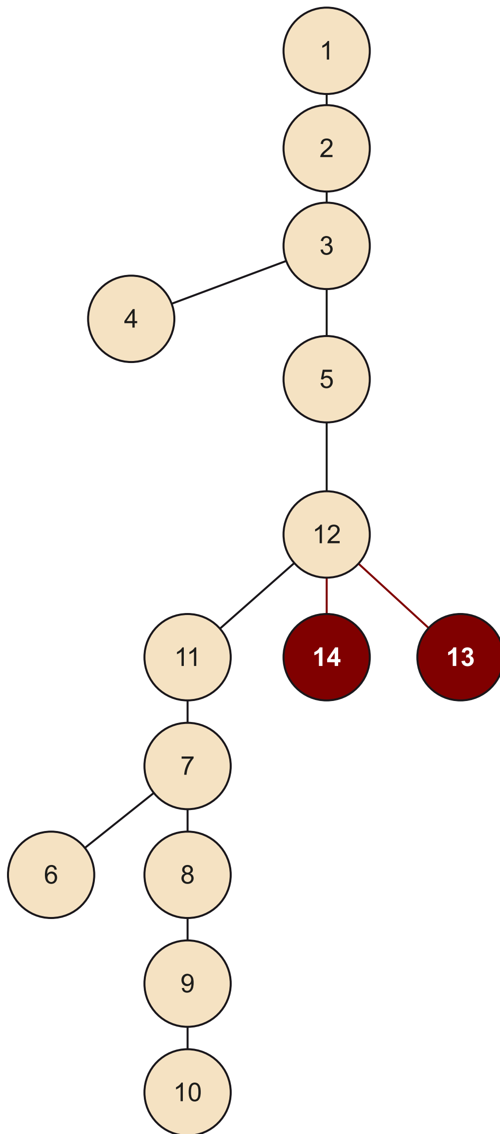
Nota

En aquest exemple i els successius sobre el graf de mobilitat, suposarem que els veïns estan ordenats numèricament, és a dir, el primer veí de cada vèrtex és el veí amb menor valor numèric. Per exemple, els veïns del vèrtex 3 són els vèrtexs 4, 5 i 6, en aquest ordre.

Aplicant-ho a la cerca de camins, per a anar d'un vèrtex A a un altre B, el recorregut en profunditat simplement començarà en el vèrtex A i retornarà el camí que trobés per arribar a B, és a dir, la llista de vèrtexs que ha visitat des que B apareix en el recorregut.

En recórrer el graf evitant passar dues vegades pel mateix vèrtex, es provoca que solament hi hagi un enllaç entre cada parell de vèrtexs; és a dir, el recorregut per si mateix té una estructura d'**arbre**.

Suposem que en el mapa de zones de mobilitat volem anar de la zona 1 a la 10. Què ens proposa l'algorisme de recorregut en profunditat? La figura següent mostra el resultat del recorregut en profunditat, en el qual s'accedeix a cada veí, i d'aquí es busca en els seus veïns fins que s'aconsegueix el vèrtex final o bé s'aconsegueix un vèrtex que no té veïns no visitats.



Recorregut en profunditat del graf de mobilitat per a anar de la zona 1 a la 10. Els nodes en vermell són nodes no visitats per l'algorisme.

Tal com es veu, en arribar a la zona 3 primer explora la 4 (que no té veïns disponibles, així que no segueix) i després la 5; el menor veí de la 5 és la 12, així que creua el pont i d'aquí passa a la 11 pel teletransport. Encara que la zona 11 és al costat de la 10, com que l'ordre dels veïns és numèric ha de visitar primer la 7 i donar tota la volta al petit bosc fins a arribar a la 10. En aquest punt l'algorisme acaba i retorna com a ruta la següent: [1, 2, 3, 5, 12, 11, 7, 8, 9, 10]. Això és bastant millorable, certament.

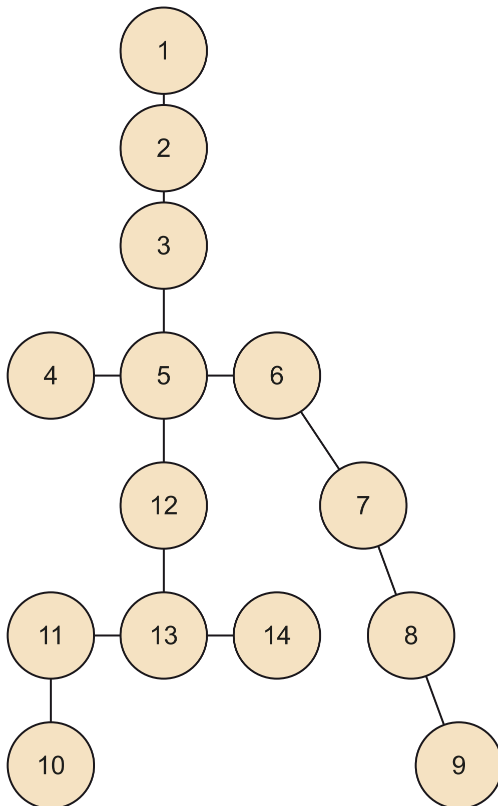
L'avantatge d'aquest algorisme és que és programable fàcilment com una funció recursiva que pren un vèrtex i recorre els seus veïns; solament requereix mantenir la llista dels vèrtexs ja visitats.

El problema d'aquest algorisme és que, si hi ha diversos camins possibles, solament en troba un, que possiblement no és el millor, i això farà que l'agent camini molt més del necessari en moltes ocasions, i, el que és pitjor, el seu moviment serà erràtic a la vista del jugador.

2.2.3. Recorregut en amplitud

Una altra manera de recórrer un graf és en amplitud: partint d'un vèrtex inicial, en primer lloc es visita cadascun dels seus veïns; una vegada fet això, se salta al primer dels veïns i es recorren tots els seus veïns (excepte els ja visitats, com en el recorregut en profunditat); i així successivament. En aquest algorisme, per tant, es descobreixen alhora tots els vèrtexs que són a la mateixa distància (nombre de salts) de l'origen. En principi, en el recorregut en amplitud s'exploren tots els nodes del graf.

En el graf de mobilitat d'exemple, el recorregut en amplitud per a anar de la zona 1 a la 10 donaria l'estructura següent:



Recorregut en amplitud del graf de mobilitat per a anar de la zona 1 a la 10.

Així, el camí resultant és [1, 2, 3, 5, 12, 11, 10]; també creua el pont i després usa el teletransport, però almenys ara no dona la volta al petit bosc. Però per què fa aquesta volta? Si comptem les zones, es visiten menys zones amb aquest camí que anant pel camí més lògic ([1, 2, 3, 6, 7, 8, 9, 10]); i de moment suposem que anar d'una zona a qualsevol altra zona veïna costa el mateix. Així que ha trobat el millor camí possible, almenys tenint en compte el que sap.

La programació d'aquest mètode és més complexa que en el cas del recorregut en profunditat, ja que —entre altres coses— requereix mantenir una llista amb els vèrtexs que s'han visitat però que tenen veïns pendents de visitar-se.

D'altra banda, si el pes associat a cada aresta és constant, és a dir, si costa el mateix moure's entre qualsevol parell de zones veïnes perquè totes tenen la mateixa grandària i no es diferencia el tipus de terreny o similars, llavors aquest algorisme sí que és capaç de trobar el camí mínim entre dos nodes.

No obstant això, aquesta és una limitació important que el restringeix a jocs de tauler o en els quals el nivell sigui estructurat en cel·les homogènies (per exemple, jocs d'estratègia amb mapa hexagonal), ja que en un nivell dissenyat arbitràriament tindrem zones més grans que unes altres i, per tant, hi haurà diferents costos per a anar d'una a una altra, com en l'exemple anterior.

2.2.4. Algorisme de Dijkstra

Precisament, en la majoria dels jocs trobarem gairebé sempre grafs de mobilitat les arestes del qual tenen associat un nombre o pes, que representa el cost de moure's d'un vèrtex (zona) a un altre, és a dir, es tracta d'un graf **ponderat**. Aquest cost pot ser simplement la distància en línia recta entre els centres de les dues zones, o pot tenir en compte altres factors, com el pendent o la dificultat del tipus de terreny.

Graf ponderat

En un graf **ponderat** cada aresta té associat un valor numèric que, per exemple, representa el cost de moure's d'un vèrtex a un altre.

Tal com acabem d'explicar, ni el recorregut en profunditat ni el recorregut en amplitud no poden trobar un camí òptim entre dos vèrtexs en un graf ponderat. En 1959 Edgser Wybe Dijkstra, matemàtic holandès i un dels científics més influents en el desenvolupament de la programació moderna, va desenvolupar un algorisme per a trobar camins òptims entre vèrtexs de grafs ponderats. Aquest algorisme, que descriurem a continuació, rep el nom d'algorisme de Dijkstra.

La descripció de l'algorisme és la següent:

Entrada: un graf de N vèrtexs, dels quals x és el node origen.

1) Crear un vector D amb N elements que emmagatzemarà les distàncies de x a la resta dels vèrtexs. Inicialitzar tots els seus valors a infinit (distància desconeguda), excepte l'entrada del mateix vèrtex x, que valdrà 0.

2) Es pren com a vèrtex actual $a = x$.

3) Recórrer tots els vèrtexs veïns de a, excepte els vèrtexs marcats com a complets.

4) Calcular la distància temptativa des del vèrtex inicial x fins a tots els veïns del vèrtex actual a. És a dir, per a cada veí de a la seva distància temptativa serà la distància des de x fins a a més la distància de a a v_i: $dt(v_i) = D_a + d(a,v_i)$. Si aquesta distancia temptativa és menor que la distància que hi havia fins a aquest moment per al vèrtex en el vector D, es guarda aquesta distància en D: $Dv_i = dt(v_i)$. (Recordem que inicialment les distàncies en D són infinites, així que en trobar el primer camí a un vèrtex es prendrà com el millor camí possible.)

5) Marcar el vèrtex a com a complet.

6) Prendre com a següent vèrtex actual el de menor distància en D i tornar al pas 3 fins que tots els vèrtexs estiguin marcats com a complets.

Sortida: un vector D amb N elements, que són les distàncies mínimes de x a la resta dels vèrtexs del graf (en realitat també es retornaria el camí per si mateix, és a dir, un altre vector amb llistes).

L'algorisme de Dijkstra genera tots els camins possibles entre un vèrtex origen i la resta dels vèrtexs del graf. L'avantatge d'aquest algorisme és, per tant, que sempre troba el camí òptim, encara que si el vèrtex origen canvia és necessari tornar a executar l'algorisme des del principi.

El principal inconvenient és que en explorar tots els camins possibles incorre en un cost computacional bastant gran $O(|V|^2)$, on $|V|$ és el nombre de vèrtexs del graf en la solució bàsica. Això fa que, si el disseny del joc ho permet, aquest algorisme s'utilitzi durant el desenvolupament per a calcular tots els camins possibles, que quedaran emmagatzemats per a usar-se més endavant. No obstant això, aquesta estratègia no es pot aplicar en tots els jocs.

2.2.5. Algorisme A*

Aquest algorisme (es llegeix «A estrella» o «*A star*» en anglès) pretén aprofitar els avantatges del recorregut en profunditat i en amplitud per a aconseguir trobar el camí òptim amb el mínim nombre d'operacions. A més, és aplicable a grafs ponderats, la qual cosa el converteix en l'algorisme de cerca de camins més utilitzat en videojocs.

Com que l'algorisme per si mateix és una mica complex, començarem descrivint de manera més intuïtiva els elements que el componen i com es comporta per a anar buscant el camí òptim. Els elements fonamentals de A* són:

- Un graf (possiblement ponderat).

- Un vèrtex origen i un vèrtex destinació.

- Una funció $g(n)$, que retorna el cost real d'arribar des del vèrtex origen al vèrtex n.

- Una funció $h'(n)$ que retorna un cost estimat d'arribar des del vèrtex n fins al vèrtex final. Aquesta funció rep el nom d'**heurístic**, i en el cas de A* sol ser la distància en línia recta entre el vèrtex n i el vèrtex final; el sentit d'aquesta funció és estimar de manera ràpida un límit inferior a la distància entre n i el vèrtex final, ja que, si en línia recta hi ha una distància d (diguem la distància a vol d'ocell), la distància real serà igual o major però mai menor, tenint en compte els obstacles.

- La funció d'avaluació d'un vèrtex n serà $f(n) = g(n)+ h'(n)$, és a dir, el cost real d'anar de l'origen a n més el cost mínim estimat d'anar de n al final.

- Es manté una cua de prioritat dels vèrtexs *oberts* (encara no visitats), ordenats pel seu valor de $f(n)$; el següent vèrtex que s'avalua sempre és el de menor $f(n)$, i a cada pas de l'algorisme es calcula la $g(n)$ de tots els seus veïns oberts.

- Es manté una estructura de dades amb els vèrtexs *tancats*, aquells que ja s'han avaluat.

- S'arribarà al vèrtex final des del vèrtex veí amb menor $f(n)$. Com que aquest vèrtex és al costat del final, gairebé tot el valor de $f(n)$ es deurà al cost d'arribar des de l'origen, és a dir, la part $g(n)$. Dit d'una altra manera, s'aconseguirà la destinació pel camí amb menor cost des de l'origen, que és el que es buscava.

Un detall important d'aquest algorisme és que els vèrtexs amb una $f(n)$ alta possiblement no s'avaluaran mai, ja que l'algorisme sempre dona prioritat al node amb menor $f(n)$. Aquest és l'avantatge de A*.

La figura següent mostra un exemple d'execució de A* sobre un mapa dividit en zones quadrades i amb un cost de moviment constant per a simplificar la figura.

Funció heurística

Una funció **heurística** és una funció inventada que serveix per a calcular de manera aproximada el valor d'una altra funció que no es coneix o que resulta molt costosa de calcular.

Exemple d'execució de l'algorisme A*, partint de la cel·la verda i arribant a la blava. Se suposa que el cost entre cel·les adjacents és sempre el mateix.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Font: CC Wikimedia Commons

El principal inconvenient de A* és el seu cost en memòria relativament elevat. Hi ha variants subòptimes com IDA* (*iterative deepening A**) o SMA* (*simplified memory-bound A**) que sacrifiquen l'optimitat a canvi de reduir notablement l'ús de memòria.

1) A* en Unity

Unity utilitza A* per a buscar els camins òptims en el graf de mobilitat (el NavMesh). El NavMesh es compon d'un conjunt de polígons convexos que representen les diferents zones de mobilitat. A cada zona se li pot assignar un cost específic, de manera que costi més avançar per sorra que per asfalt, per exemple.

Una vegada definit el NavMesh, cal crear un *script* per a indicar a l'agent que es mogui a la posició d'un objecte determinat. El codi que fa aquesta operació és el següent:

```
// Moveto.cs
   using UnityEngine;
   using System.Collections;

   public class MoveTo : MonoBehaviour {

      public Transform destino;

      void Start () {
         NavMeshAgent agente = GetComponent<NavMeshAgent>();
         agente.destination = destino.position;
```

```
        }
    }
```

2.2.6. Altres variacions de la cerca de camins

Si bé A* s'utilitza habitualment en jocs amb nivells estàndard, hi ha algunes situacions específiques en les quals no resulta pràctic aplicar-lo directament o bé resultaria molt car computacionalment.

1) Cerca de camins jeràrquica

Quan el joc (o aplicació, en general) té un mapa molt extens (pensem en MMORPG com World of Warcraft, amb continents sencers, o en aplicacions com Google Maps), és inviable aplicar A* a tot el mapa de joc alhora. Per això, el que es fa és aplicar una jerarquia als mapes o nivells del joc, per exemple dividint el món en continents, cada continent en regions i cada regió possiblement en diverses zones; a més, dins de cada casa o cada cova es definiran noves zones. D'aquesta manera, per a trobar el camí per a anar d'un punt a un altre del món, en primer lloc es buscaria com anar d'un continent a un altre, en el continent de destinació es buscaria la regió de destinació, etc. En cada nivell de complexitat es pot aplicar A*, ja que el nombre de vèrtexs del graf serà limitat. I, possiblement, el significat de les arestes entre vèrtexs és diferent segons el nivell en la jerarquia, perquè, per exemple, la connexió entre continents pot referir-se a l'existència de vaixells per a moure's entre ells.

MMORPG

Un MMORPG és un joc de rol massiu multijugador en línia (*massive multiplayer online role playing game*). *Massiu* es refereix al fet que hi pot haver una gran quantitat de jugadors (milers per servidor) connectats simultàniament. El programari dels servidors d'aquest tipus de jocs i el protocol de comunicació requereixen un disseny molt acurat per a permetre que els jugadors interactuin en temps real i sense col·lapsar el sistema.

2) Cerca de camins dinàmica

En moltes situacions un agent no pot calcular el seu camí complet quan se li dona l'ordre de moure's. En molts jocs d'estratègia, per exemple, hi ha l'anomenada «boira de guerra» (*fog of war*), que oculta les zones de mapa que encara no s'han explorat. No obstant això, el jugador pot clicar en una zona oculta i ordenar al seu agent que hi vagi (per exemple, per explorar). Se suposa que la IA del joc no ha de fer paranys i ha de respectar la boira de guerra i altres limitacions que també tingui el jugador, així que anirà descobrint camí a mesura que vagi entrant en terreny desconegut.

En altres casos, pot haver-hi obstacles mòbils que impedeixin el moviment que s'havia calculat (una unitat que bloqueja un pont, per exemple) i que obliguin a recalcular la ruta sobre la marxa. Aquest tipus d'algorismes també s'aplica freqüentment en robòtica per planificar els moviments del robot.

En aquestes situacions s'apliquen algorismes de cerca de camins dinàmica, com per exemple el D* (*Dynamic A**). Sense entrar en molts detalls, les característiques d'aquest algorisme, del qual hi ha diferents variants, són les següents:

- S'assumeix que la zona desconeguda no té obstacles i es comença el camí amb aquesta suposició.

- La seva estructura bàsica és similar a l'A*.

- Els vèrtexs, a més d'estar oberts i tancats, poden trobar-se en altres estats: nou (encara no considerat de cap manera), apujat (el seu cost ha pujat des de l'última avaluació pels canvis dinàmics) i abaixat (el seu cost ha baixat). D'aquesta manera, es gestiona la dinàmica del mapa.

- A diferència de A*, D* comença l'exploració des del vèrtex destinació.

Exemple de boira de guerra en el joc d'estratègia per torns Battle for Wesnoth (de programari lliure).

Font: CC Wikimedia Commons

2.3. Moviments complexos

Fins ara hem tractat la gestió de moviments suposant que un agent vol moure's a una posició determinada, però en els jocs els agents s'han de moure sovint uns en relació amb altres. Exemples d'aquestes situacions són: interceptar una pilota en moviment, disparar a un avió calculant el punt en el qual es trobarà, perseguir el jugador per una habitació, moure un escamot de soldats de manera flexible per adaptar-se a l'amplària dels carrers i per canviar la seva formació en arribar i trobar-se amb l'enemic. En aquest apartat estudiarem alguns d'aquests tipus de moviment i com es poden resoldre.

2.3.1. Moviments de direcció

Els moviments de direcció (en anglès *steering behaviours*) són aquells en els quals el moviment ha de permetre interceptar, perseguir o evadir altres agents, o simplement vagabundejar d'una manera que resulti natural.

1) Perseguir i fugir

Una de les situacions més habituals en els jocs consisteix a fer que persegueixin el jugador o un altre agent. És a dir, la destinació del moviment no és un punt fix del nivell de joc sinó la posició d'un altre agent.

«Sòl»

Encara que una gran part dels jocs són en tres dimensions, en la majoria dels casos hi ha un «sòl» de referència que fa que el moviment es calculi en realitat sobre les dues dimensions del sòl. Solament en jocs en els quals hi ha més llibertat de moviments en altura, com en jocs amb avions o naus espacials, s'utilitzen plenament les tres dimensions per a calcular els moviments.

La posició, la velocitat i l'acceleració dels agents en el joc es representen mitjançant vectors de dos o tres components segons les dimensions en les quals es desenvolupi el moviment.

En qualsevol dels dos casos, en posar una fletxa sobre una lletra ($\vec{A}$) volem indicar que es tracta d'un vector.

Suposem un agent A que vol perseguir un agent B. Per tant, la destinació del moviment de A serà la posició de B, $\vec{P}_B$. Per a calcular la direcció en la qual haurà de moure's A (és a dir, el seu vector de velocitat), és necessari tenir en compte la seva posició actual i la seva destinació, és a dir:

$$\vec{V}_A = \vec{P}_B - \vec{P}_A$$

Però com que A tindrà una celeritat (mòdul de la seva velocitat) màxima V_{MAX}, aquesta expressió s'ha d'ajustar de la manera següent:

$$\vec{V}_A = \frac{\vec{P}_B - \vec{P}_A}{|P_B - P_A|} \cdot V_{MAX}$$

En el cas de voler fugir d'un agent, simplement cal canviar el signe del vector $\vec{V}_A$ perquè A es mogui en direcció oposada a B i, per tant, en fugi.

2) Interceptar

En l'exemple de perseguir o fugir, hem suposat que l'agent destinació, B, estava quiet. No obstant això, si B es mou, quan A arribi a la seva posició B ja no estarà allí i haurà de tornar a moure's cap a la nova posició de B, que mentrestant continuarà movent-se, i així successivament. En definitiva, donarà una

sensació molt maldestra als jugadors que ho estiguin veient. A més, és especialment important tenir en compte aquest detall quan es vol llançar un objecte del tipus projectil perquè encerti un objectiu mòbil.

La idea bàsica per a interceptar un objecte en moviment és estimar la seva posició futura a partir de la seva velocitat actual, i calcular en quin punt l'agent pot aconseguir l'objectiu tenint en compte la velocitat màxima a la qual pot arribar.

Sigui $\overrightarrow{P_A}$ la posició de l'agent A, $\overrightarrow{P_B}$ la de l'agent objectiu B i $\vec{V}_B$ la seva velocitat. Volem calcular la velocitat necessària $\overrightarrow{V_A}$ perquè A coincideixi amb B en el punt d'intercepció, és a dir, que els dos agents estiguin en el mateix punt $\vec{P}_X$.

$$\begin{cases} \overrightarrow{P_X} - \overrightarrow{P_A} = t \cdot \overrightarrow{V_A} \\ \overrightarrow{P_X} - \overrightarrow{P_B} = t \cdot \overrightarrow{V_B} \end{cases}$$

Com que també han de coincidir en el temps, *t* ha de ser la mateixa en les dues equacions, així que igualem i obtenim:

$$\overrightarrow{V_A} = \frac{\overrightarrow{P_A} - \overrightarrow{P_B}}{t} + \overrightarrow{V_B}$$

D'aquesta manera, s'obté la velocitat que ha d'adoptar A per a atrapar B.

3) Moviments de direcció en Unity

Quan treballem amb nivells amb obstacles en Unity, que requereixin l'ús del NavMesh, no calcularem directament els vectors de velocitat, sinó que adaptarem els moviments de direcció en cerca de camins de NavMesh. Així, en el cas de la persecució, el que haurem de fer serà actualitzar constantment la destinació del moviment a la posició actual de l'objectiu.

El mètode *Update()* del comportament dels agents de Unity es crida en cada refresc de fotograma; simplement, es pot recalcular el moviment a partir de l'última posició coneguda de l'objectiu:

```
// Moveto.cs
   using UnityEngine;
   using System.Collections;

   public class MoveTo : MonoBehaviour {

      public Transform objetivo;

      void Start () {
// En Start() no fa falta fer res perquè en el primer Update() ja es
```

```
  // calcularà el camí
        void Update () {
          NavMeshAgent agente = GetComponent<NavMeshAgent>();
          agente.destination = objetivo.position;
        }
      }
```

Solament cal tenir en compte el cost computacional que pot implicar aquesta solució, ja que possiblement en cada fotograma es recalcula el camí; si el nombre de zones és elevat i hi ha molts agents que fan això, pot acabar essent una sobrecàrrega notable.

Millores possibles són: no recalcular en cada fotograma, sinó en cada *n* fotogrames o cada *p* mil·lisegons, i també emmagatzemar la posició de la destinació i solament recalcular el camí quan hagi canviat significativament.

Pel que fa a la intercepció, necessitem esbrinar la velocitat de l'objectiu a més de la seva posició. Si per exemple s'està utilitzant el motor de física de Unity, els objectes controlats pel motor són de classe *Rigidbody* i la propietat *velocity* és el vector de velocitat de l'objecte; a partir d'aquí podem determinar la seva trajectòria i la seva posició en el futur. D'altra banda, si s'està utilitzant NavMesh, és la propietat *NavMeshAgent.velocity* la que retorna *el Vector3* amb la velocitat de l'agent al llarg del seu camí.

2.3.2. Moviments col·lectius (*flocking*)

Els agents de joc no sempre funcionen de manera independent, sinó que a vegades diversos agents es mouen en grup: ramats o bandades d'animals, escamots de soldats o munions d'esportistes, formacions d'avions o naus, etc.

En aquest tipus de situacions no és viable calcular un camí independentment per a cada agent del grup per dos motius:

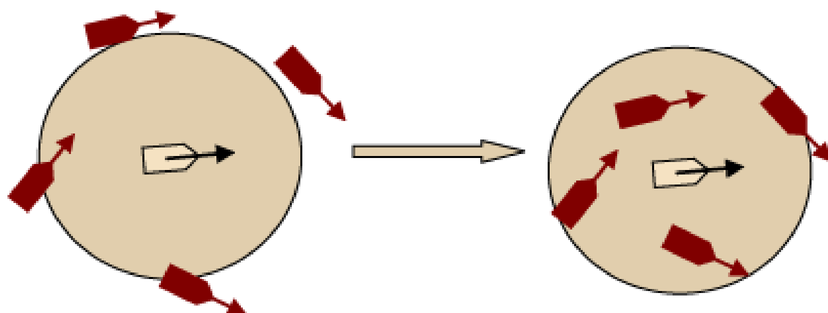
- L'excessiu cost computacional quan es repeteix bàsicament la mateixa operació (mateixos origen i destinació, amb un petit desplaçament).
- Si cada agent seguís el seu camí, sovint xocarien entre ells i s'entorpirien; a més, seria impossible definir un moviment ordenat de tipus formació.

D'altra banda, tampoc no es pot considerar que el grup sigui un únic agent de major grandària, ja que de vegades el grup haurà de passar per zones més estretes i adaptar-se, i en general és desitjable que el moviment dels agents sigui natural i no completament rígid.

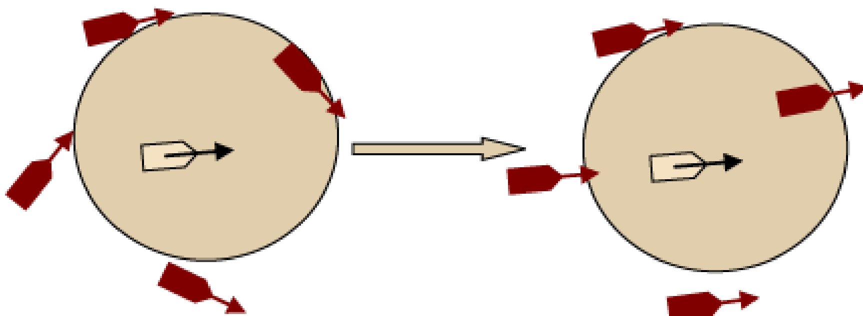
Per aquests motius és necessari utilitzar altres estratègies que permetin modelar el moviment esperat en grups d'agents. En 1987 Craig Reynolds va definir el primer algorisme de moviment col·lectiu (*flocking*), en el qual va batejar com a

boids els agents que formen el grup. L'algorisme de Reynolds dona molt bons resultats i té com a característica que no hi ha un líder del grup. La bellesa d'aquest algorisme prové en gran manera de la seva simplicitat, ja que funciona segons aquests tres principis:

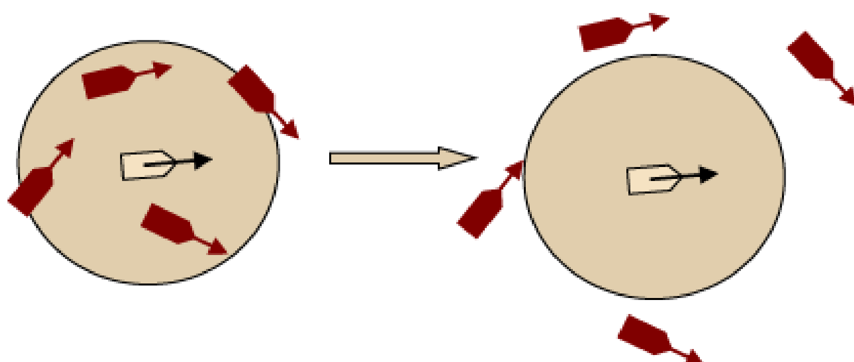
1) Cohesió: cada unitat es dirigeix cap a la posició mitjana dels seus veïns.



2) Alineament: cada unitat s'alinea segons l'alineació mitjana dels seus veïns.



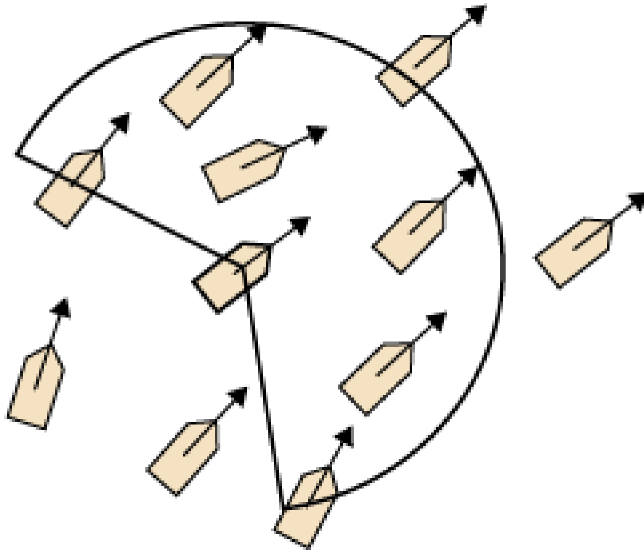
3) Separació: cada unitat ajusta la seva direcció per evitar col·lidir amb els seus veïns.



Nota

S'ha demostrat científicament que els ocells decideixen el seu moviment mirant els seus cinc veïns més propers.

Perquè els agents puguin complir aquestes propietats, necessiten conèixer la posició dels seus veïns en tot moment; per a aconseguir un comportament realista, es defineix un **camp de visió** (*field of view*) que controla quins veïns pot veure l'agent i, per tant, per quins agents es guiarà.



Si el camp de visió és ample, els agents formaran grups amples; per contra, amb camps de visió més estrets els agents tendiran a anar en grups estrets, del tipus fila o caravana.

Web recomanada

L'algorisme de *flocking* per a Unity es pot trobar a http://wiki.unity3d.com/index.php?title=flocking.

3. Presa de decisions

Sens dubte una de les característiques de la IA d'un joc que impressionarà més els jugadors és la seva capacitat de prendre decisions com si es tractés d'una persona, o almenys demostrant intel·ligència. Un exemple molt clar d'això són els jocs d'estratègia: si la IA es mostra maldestra i pren decisions contraproduents tota l'estona, s'arruïnarà la gràcia del joc.

A més, la presa de decisions no solament es produeix en els agents contra els quals juga el jugador humà: moltes vegades hi ha elements controlats per la IA que col·laboren amb el jugador com a acompanyants, i les decisions que prenguin poden ser crítiques per al desenvolupament del joc.

Nota

Convé recordar que, en última instància, l'objectiu de la IA en un joc és aconseguir que el jugador es diverteixi jugant; i, per a aconseguir-ho, en general és necessari que la IA sigui capaç de prendre decisions encertades i que facin l'efecte d'intel·ligència reaccionant adequadament a les accions del jugador.

En pràcticament tots els jocs la IA ha de prendre decisions constantment: quan usar el turbo, activar l'escut de força, atacar o fugir, construir un canó o un tanc, copejar la pilota amb el peu o amb el cap, passar-la a un altre agent o tirar a porteria…

Cal tenir en compte, de totes maneres, que en alguns casos els agents de joc poden seguir un comportament fix, amb una sèrie de passos que es repeteixen exactament, i en els quals la dificultat i l'interès per als jugadors rau a aprendre's bé els passos i perfeccionar l'estratègia. Un exemple clar d'això són els caps (*bosses*) de World of Warcraft, el comportament dels quals és sempre igual i passa per les mateixes fases, ben conegudes i documentades.

Combat amb un cap de *raid* a World of Warcraft

Font: © Blizzard Entertainment

Hi ha alguns aspectes generals que convé considerar en la presa de decisions. En primer lloc, se solen diferenciar segons l'escala temporal en la qual tindrà efecte l'acció:

- **Decisions estratègiques**. Són les de major abast en el temps, i normalment afecten el desenvolupament complet d'una partida. Això implica que són les decisions més complexes de prendre, ja que requereixen tenir en compte més dades d'entrada i han de preveure les seves possibles conseqüències més lluny en el futur. Exemples d'aquestes decisions són: en un partit de futbol, si es jugarà a la defensiva o a l'atac; en un joc d'estratègia, si es fortificarà una posició, s'atacarà ràpidament o s'intentarà conquistar diverses posicions al voltant del jugador.

- **Decisions tàctiques**. Són decisions en un marc de temps intermedi, pensades per a dur a terme l'estratègia decidida prèviament. Normalment es prenen moltes decisions tàctiques en cada partida. Exemples d'aquestes decisions són: construir una muralla o uns barracons, parar en boxes o intentar fer un volt més en una carrera de Fórmula 1, assignar un jugador per a marcar-ne un altre en un partit de bàsquet.

- **Decisions operacionals**. Són decisions molt concretes per a dur a terme les decisions tàctiques; poden prendre-se'n moltes cada segon i depenen d'informació local molt específica del moment. Exemples d'aquestes decisions són: usar un poder especial en un moment donat, tirar la pilota a porteria o passar-la a un company.

Un altre aspecte important que s'ha de tenir en compte és la **informació** que té la IA. Se suposa que ha de «jugar net» i evitar usar informació privilegiada, com per exemple saber amb detall tot el que hi ha en el mapa mentre el jugador no ho sap si surt a explorar. No obstant això, alguns jocs s'aprofiten de tota la informació disponible (què fa el jugador en cada moment) com una manera fàcil de millorar les prestacions de la IA sense necessitat d'utilitzar tècniques molt complexes; és el que es coneix com IA **omniscient**.

També pot haver-hi decisions **proactives**, en les quals l'agent pren la iniciativa i sap que ha de prendre una decisió, i **reactives**, en les quals a causa d'un canvi extern l'agent reacciona i genera una resposta.

Finalment, cal considerar el **determinisme** que volem que ofereixi el joc, és a dir, si la resposta dels agents sempre serà la mateixa en les mateixes condicions o si, per contra, hi ha un cert factor d'**aleatorietat** en la resposta perquè no sigui sempre igual. Aquesta decisió depèn del tipus de joc que es vulgui crear i d'on es volen posar els reptes als jugadors.

Anant un pas més enllà, en el capítol 4 veurem mètodes d'IA que són capaços d'aprendre del que fa el jugador i adaptar-se al seu estil de joc per aconseguir oferir sempre una experiència emocionant. En aquest cas estaríem parlant de comportaments **apresos**.

3.1. El procés de presa de decisions

En línies generals, els passos que ha de seguir la IA per a prendre una decisió són sempre els mateixos, adaptats, això sí, a les característiques pròpies de cada joc, ja que en alguns sistemes alguns passos s'ometran.

1) Recopilar la informació necessària per a identificar la situació actual i saber així quines són les opcions possibles. El tipus i quantitat d'informació que cal recopilar dependrà del nivell de complexitat de la decisió que s'ha de prendre.

2) Identificar els criteris de decisió i assignar-los una ponderació o prioritat. Per exemple, una nau espacial ha d'atendre el manteniment general dels seus sistemes, però en combat ha de donar prioritat a la informació relacionada (naus enemigues detectades, etc.).

3) Determinar les alternatives possibles. En aquest aspecte és on més difereixen els mètodes que es presentaran a continuació. En general, cal calcular quines decisions possibles són viables tenint en compte els dos punts anteriors.

4) Valorar les alternatives trobades en el punt anterior assignant-los una puntuació que permeti decidir entre una altra. En el cas de sistemes deterministes sempre es triarà l'alternativa de major puntuació; en sistemes més aleatoris hi haurà la probabilitat de triar una altra opció.

5) Execució de la decisió duent a terme les accions necessàries (prémer el fre, disparar, passar la pilota…).

6) Avaluar els resultats. Els algorismes d'aprenentatge (vegeu l'apartat 4) valoren els resultats de les seves accions i aprenen d'ells. Els mètodes presentats en aquest apartat en general no fan aquest últim pas.

3.2. Sistemes de regles

Els sistemes de regles són una de les tècniques més senzilles per a programar el comportament dels agents, si bé s'utilitzen amb molta freqüència, ja que en moltes situacions donen una resposta ràpida i efectiva. Els elements que componen aquests sistemes són tres:

- Un conjunt de condicions possibles en el joc.
- La resposta esperada per a cadascuna de les condicions anteriors.
- Un conjunt d'instruccions *if…else* que avalua cadascuna de les condicions possibles i, en cas de complir-se, executa la resposta.

El comportament d'un monstre en un joc de trets en primera persona es podria definir mitjançant les condicions i accions següents:

- Si no hi ha cap jugador a prop, vagabundeja.
- Si sents sorolls, ves a investigar.
- Si veus un jugador, ves cap a ell.
- Si ets al costat d'un jugador, ataca'l.
- Si la teva salut està per sota del 25%, fuig i amaga't.

Pot passar que en un moment donat es compleixin diverses regles (el monstre és al costat d'un jugador i la seva salut està per sota del 25%), la qual cosa requereix que s'apliqui alguna estratègia de resolució de conflictes per a decidir quina regla triar. Hi ha diferents estratègies de resolució de conflictes:

- Executar la primera regla del subconjunt de regles en conflicte.
- Executar aleatòriament qualsevol de les regles del subconjunt.
- Assignar una valoració (fixa o dependent de factors externs) a cada regla i seleccionar la regla amb més valoració.

En qualsevol d'aquests casos, les regles s'utilitzen en el sentit habitual d'una instrucció *if…then*: si es compleix la condició, s'executa la regla. Això és el que es coneix com a **encadenament cap endavant** (*forward chaining*).

No obstant això, el coneixement emmagatzemat en el conjunt de regles també es pot aprofitar en sentit contrari, perquè la IA pugui deduir informació sobre l'estat del joc. En l'exemple anterior, pot saber que si un altre monstre ataca el jugador la seva salut està per sobre del 25%, per exemple. Aquesta tècnica es coneix com a **encadenament cap enrere** (*backward chaining*) i és molt útil perquè la IA dedueixi informació (quan no és omniscient). Per exemple, en els jocs d'estratègia sol haver-hi tecnologies estructurades en forma d'arbre, de manera que per a poder utilitzar una tecnologia és necessari haver desenvolupat altres tecnologies, per exemple la pólvora per a poder fabricar canons. Aplicant *backward chaining*, si la IA veu que el jugador té canons pot deduir que també té pólvora, informació que podrà usar en decisions futures.

Els sistemes basats en regles funcionen bé en situacions relativament senzilles (principalment en decisions tàctiques i operacionals). Un altre avantatge és que són fàcils de programar. Per contra, quan el nombre de regles augmenta pot resultar difícil gestionar-los i evitar contradiccions o situacions imprevistes. A més, si no es troba una condició que es compleixi, el sistema no pot generar una resposta raonable: no hi ha lloc per a improvisar.

3.3. Màquines d'estats finits

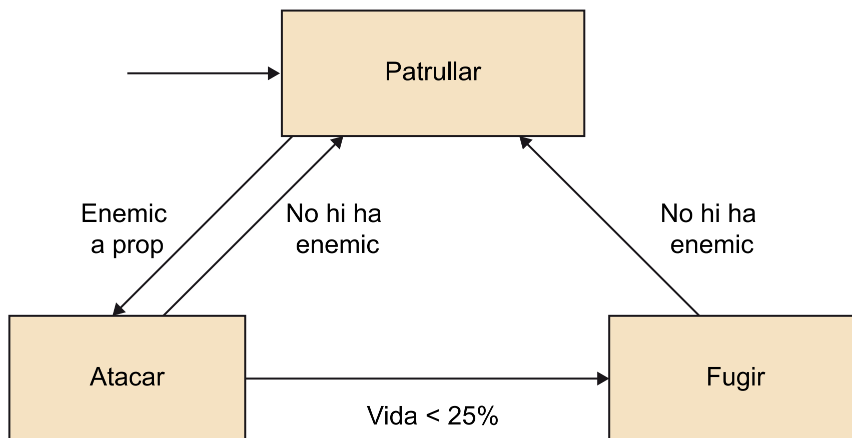
Les màquines d'estats finits (*finite state machines*, FSM) són models de comportament que es componen de dos elements:

- Un conjunt d'estats (finit, no pot haver-hi estats infinits), que són les diferents situacions en les quals pot trobar-se l'agent.

- Un conjunt de transicions, que són condicions que fan que l'agent passi de l'estat A al B. A diferència dels sistemes basats en regles, la transició que es faci depèn de l'estat actual de l'agent.

La figura següent mostra un senzill diagrama d'estats d'un soldat en un joc. La fletxa que no ve de cap part i apunta «Patrullar» indica que aquest és l'estat inicial de l'agent.

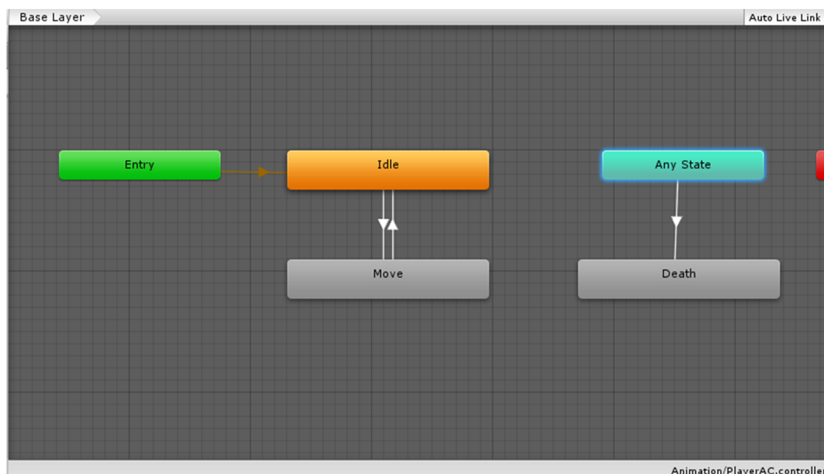
Diagrama d'estats d'un soldat en un joc



Els FSM

Els FSM són un patró de disseny de programari que s'utilitza en innombrables situacions en les quals es vulgui modelar un sistema que té un estat intern que canvia en reacció a diferents condicions.

L'Unity mateix utilitza els FSM en altres llocs, com per exemple els *Animator*, en aquest cas per a modelar les diferents animacions d'un model i les condicions en les quals s'ha de passar d'una a una altra.



És possible (encara que no recomanable des d'un punt de vista d'enginyeria de programari) utilitzar la maquinària d'estats de les animacions per a controlar el comportament de l'agent.

Un altre avantatge dels FSM és que ajuden a dissenyar correctament el comportament d'un agent, ja que obliguen a definir els seus possibles estats i a analitzar en quines circumstàncies es passa d'un estat a un altre. És a dir, són alhora una eina de disseny de programari i una manera de fer aquest disseny en el llenguatge de programació que es vol. Tal com es veurà en el codi de l'exemple següent, l'estructura de classes utilitzada per a programar un FSM organitza les dades i operacions dividint-les en una part comuna a tots els estats i una altra particular de cada estat.

Per la seva especial importància, vegem el codi d'un FSM per al diagrama proposat en la figura anterior (estats Patrullar, Atacar, Fugir).

En primer lloc es defineix una classe, FSMSoldado, que modela el comportament del soldat mitjançant l'FSM de la figura. Dins de la classe es defineixen els tres estats del soldat més un atribut que indica quin és l'estat actual. El mètode *Start()* s'aprofita per a assignar l'estat inicial (*EstadoPatrullar*), i en el mètode *Update()* es demana a l'estat actual que revisi la seva situació per si és necessari canviar a un altre estat.

```
using UnityEngine;
using System.Collections;

public class FSMSoldado: MonoBehaviour
{
   // Aquí aniria la informació del soldat comú en tots els estats



   // Accés als estats
   [HideInInspector] public IEstadoSoldado estadoActual;
   [HideInInspector] public EstadoPatrullar estadoPatrullar;
   [HideInInspector] public EstadoAtacar estadoAtacar;
   [HideInInspector] public EstadoHuir estadoHuir;

   // En aquest mètode es creen els objectes que representen cada un dels tres estats
   private void Awake()
   {
      estadoPatrullar = new EstadoPatrullar (this);
      estadoAtacar = new EstadoAtacar (this);
      estadoHuir = new EstadoHuir (this);

      navMeshAgent = GetComponent<NavMeshAgent> ();
   }

   // Aquí s'indica quin és l'estat inicial
   void Start ()
   {
      estadoActual =  estadoPatrullar;
```

```
    }

    // En cada fotograma es comprova si s'ha de fer el canvi a un altre estat
    void Update ()
    {
        estadoActual.ActualizaEstado();
    }

    private void OnTriggerEnter(Collider other)
    {
        currentState.OnTriggerEnter (other);
    }
}
```

Com que crearem tres classes diferents (per a cadascun dels tres estats) però l'atribut *estadoActual* pot referir-se a qualsevol d'elles, definirem una interfície de C# que inclogui les operacions comunes a tots els estats, que són quatre: una per a la crida d'actualització que es fa des del mètode *Update()* de l'FSM, i tres per a executar la transició a cadascuna dels estats.

```
using UnityEngine;
using System.Collections;

public interface IEstadoSoldado
{
    void ActualizaEstado();

    void AEstadoPatrullar();

    void AEstadoAtacar();

    void AEstadoHuir();
}
```

L'estat Patrullar no canvia fins que no es detecta un enemic durant la patrulla. Tal com s'acaba d'explicar, tots els estats implementen la interfície IEstadoSoldado.

```
using UnityEngine;
using System.Collections;

public class EstadoPatrullar : IEstadoSoldado
{
    // Referència al FSM per poder ordenar-li que canviï d'estat
    private readonly FSMSoldado fsm;

    // Aquí aniria una altra informació útil: el punt següent en el recorregut del soldat, etc.
```

```
    // Anotar l'objectiu FSM per poder demanar-li canvis d'estat
    public EstadoPatrullar (FSMSoldado fsmSoldado)
    {
       fsm = fsmSoldado;
    }

    // Aquí es fa tot el que se suposa que fa el soldat en patrullar: moure's, etc.
    public void ActualizaEstado()
    {
      // Escollir el següent punt de la ruta, moure's, vigilar; si es detecta el jugador, perseguir-lo
       if(es detecta el jugador d'alguna manera) {
          AEstadoAtacar();
       }
    }

    // Mètodes que executen els trancisions a altres estats
    public void AEstadoPatrullar()
    {
       Debug.Log ("No es pot canviar al mateix estat.");
    }

    public void AEstadoAtacar()
    {
       fsm.estadoActual = fsm.estadoAtacar;
    }

    public void AEstadoHuir()
    {
       fsm.estadoActual = fsm.estadoHuir;
    }
 }
```

L'estat Atacar té dues transicions possibles: a Fugir o a Patrullar.

```
 using UnityEngine;
 using System.Collections;

 public class EstadoAtacar : IEstadoSoldado
 {
    private readonly FSMSoldado fsm;

    // Una altra informació útil per a aquest estat: tipus d'atac, etc.

    // Anotar l'objectiu FMS per poder demanar-li canvis d'estat
    public EstadoAtacar (FSMSoldado fsmSoldado)
    {
```

```
        fsm = fsmSoldado;
    }

    // Aquí es fa tot el que se suposa que fa el soldat en atacar
    public void ActualizaEstado()
    {
        // Realitzar l'atac en si
        // Transicions
        if(es perd de vista el jugador) {
            AEstadoPatrullar();
        }
        else if(la salud < 25%) {
            AEstadoHuir();
        }
    }

    // Mètodes que executen les transicions a altres estats
    public void AEstadoPatrullar()
    {
        fsm.estadoActual = fsm.estadoPatrullar;
    }

    public void AEstadoAtacar()
    {
        Debug.Log ("No es pot canviar al mateix estat.");
    }

    public void AEstadoHuir()
    {
        fsm.estadoActual = fsm.estadoHuir;
    }
}
```

Finalment, l'estat Fugir solament té una transició: tornar a Patrullar.

```
using UnityEngine;
using System.Collections;

public class EstadoHuir : IEstadoSoldado
{
    private readonly FSMSoldado fsm;

    // Una altra informació útil per a aquest estat: cap on es fugirà, etc.

    // Anotar l'objectiu FMS per poder demanar-li canvis d'estat
    public EstadoHuir (FSMSoldado fsmSoldado)
    {
```

```
         fsm = fsmSoldado;
      }

      // Aquí es fa tot el que se suposa que fa el soldat en fugir
      public void ActualizaEstado()
      {
         // Determinar on fugir, buscar el camí, córrer pel camí
         // Transició
         if(es perd de vista el jugador) {
            AEstadoPatrullar();
         }
      }

      // Mètodes que executen les transicions a altres estats
      public void AEstadoPatrullar()
      {
         fsm.estadoActual = fsm.estadoPatrullar;
      }

      public void AEstadoAtacar()
      {
         fsm.estadoActual = fsm.estadoAtacar;
      }

      public void AEstadoHuir()
      {
         Debug.Log ("No se puede cambiar al mismo estado.");
      }
   }
```

Nota

Si hi ha molts agents que usen el mateix FSM, es pot estalviar memòria creant cada estat solament quan es necessiti en lloc de crear-los i mantenir-los tots. Encara millor, es poden definir els estats com a *Singleton* i passar-los la referència de l'FSM en cridar el seu *ActualizaEstado()*.

Tal com es pot veure en el codi, el mètode *ActualizaEstado()* de cada estat és el responsable d'anar executant el comportament esperat de l'agent d'aquest estat i de decidir si és necessari canviar a un altre estat. En cada estat solament es programen les sortides possibles (transicions a altres estats): no té importància de quin estat es ve.

Un possible desavantatge dels FSM és que no és possible combinar diferents estats, la qual cosa impedeix modelar alguns comportaments que resulten de combinacions de diversos estats o característiques. Per exemple, en un joc d'estratègia la IA pot construir una serradora i alhora entrenar soldats, però no hi ha cap estat que representi aquestes dues variables, ja que les combinacions

possibles són tantes que serien necessaris moltíssims estats. A més, els estats i les transicions es predefineixen en general en escriure el programa, és a dir, que no es poden afegir estats nous dinàmicament.

Els FSM són deterministes, és a dir, partint d'un estat, si es donen les mateixes condicions, el model sempre canviarà al mateix estat. Això pot provocar que les respostes del joc siguin massa predictibles. Es poden generalitzar els FSM afegint una probabilitat a cada transició, de manera que, si en un moment donat hi ha diverses transicions possibles des de l'estat actual, se'n triï una a l'atzar.

Els FSM es poden aplicar a diferents nivells de decisió (estratègica, tàctica, operacional). De fet, és possible acollir un FSM dins d'un estat d'un altre FSM. Això és el que es coneix com a **arbres de comportament**, que veurem més endavant.

3.4. Exploració en arbre

Els **arbres** són estructures en les quals es parteix d'un element anomenat **node arrel**, del qual parteixen dos **enllaços** (n'hi pot haver més) que condueixen a altres **nodes**, que poden ser **interns** (o branques) si d'ells parteixen nous enllaços, o bé **nodes fulla** o **terminals** si d'ells no parteixen enllaços. A diferència dels grafs, en els arbres no pot haver-hi cicles: els nodes estan organitzats jeràrquicament i no és possible crear un enllaç que torni a un node previ. Habitualment, un node que prové d'un altre es denomina node **fill** del primer, i aquest és lògicament el seu **pare**.

La programació de videojocs

La programació de videojocs ofereix nombroses situacions que es resolen de manera natural mitjançant estructures en forma d'arbre. Un exemple fora de la IA són les escenes i els seus objectes: una escena en Unity és l'arrel de l'arbre, cada objecte de l'escena és un node que prové de l'escena, els objectes associats a altres objectes (les rodes d'un cotxe, l'espasa d'un guerrer) són al seu torn nodes fill dels objectes als quals van associats.

En aquest apartat veurem diferents aplicacions dels arbres per a dotar d'intel·ligència els nostres jocs.

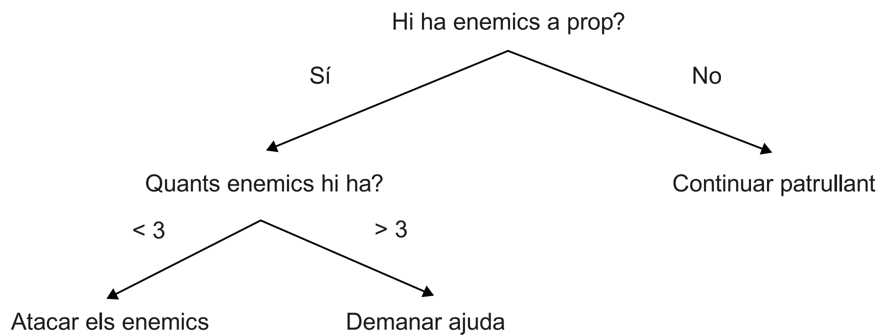
3.4.1. Arbres de decisió

Els **arbres de decisió** són un mètode de decisió o classificació que permet encadenar un conjunt de condicions (com les dels sistemes basats en regles) de manera estructurada i jerarquitzada. Generalment, es comença plantejant la condició més important (la que permet distingir el major nombre de casos), i dins de cada alternativa es plantegen altres condicions, en les quals hi haurà al seu torn altres condicions, i així successivament.

Des del punt de vista de la programació, un arbre de decisió es pot programar mitjançant un conjunt d'*if…else* aniuats o jerarquitzats. La seqüència de decisions, representada gràficament, dona lloc a un arbre; d'aquí el nom del mètode.

Un exemple d'arbre de decisió seria el següent:

Exemple senzill d'arbre de decisió.



Els arbres de decisió tenen dos tipus d'elements:

- Els nodes **interns**, que són les condicions que ha d'avaluar el mètode. Els nodes interns poden ser **deterministes** —si en les mateixes situacions sempre prenen la mateixa decisió—, o **probabilístics** —si la seva decisió depèn parcialment o totalment de l'atzar.

- Els nodes **finals** o fulles, que són les decisions finals que prendrà l'arbre, és a dir, el resultat de l'avaluació de l'arbre.

Si volem utilitzar un arbre de decisió en el nostre joc, hem de tenir en compte que es distingeixen dues etapes:

1) Disseny de l'arbre, etapa en la qual s'ha de decidir quines són les decisions més importants, que s'han de posar a l'inici de l'arbre, i deixar les decisions menys influents per al final. Això es fa en programar el joc.

2) Execució de l'arbre durant el joc, per aplicar la cadena de decisions durant el desenvolupament de la partida.

Els avantatges dels arbres de decisió són les següents:

- Permeten estructurar la presa d'una gran quantitat de decisions.

- Poden ser més eficients que les seqüències de condicions dels sistemes de regles, ja que en cada execució solament s'avalua una línia de condicions, no totes.

Mètodes de classificació

Els arbres de decisió, i altres mètodes que anirem estudiant, són **mètodes de classificació**, que són capaces de classificar una mostra de dades a partir de les seves característiques. Per exemple, classificar un cotxe en esportiu, familiar, tot terreny, etc., en funció de les seves característiques: tipus de motor, grandària de la roda, etc. Aquesta capacitat es pot aplicar al problema de la presa de decisions fent que els mètodes prenguin com a entrada la situació actual del joc i donin com resultat la classe d'acció que s'ha d'executar.

Nota

En aquest apartat hem vist arbres de decisió **predissenyats**, en els quals el programador fixa les condicions i l'ordre en el qual s'executaran; el programa les executa tal qual. En l'apartat 4 veurem que els arbres de decisió poden entrenar-se perquè aprenguin per ells mateixos quines decisions els convé prendre en cada moment.

- Controlant la profunditat de l'arbre, es pot controlar el temps d'execució del mètode d'una manera senzilla.

- És possible seguir la seqüència de decisions que porta a l'arbre a prendre una decisió final.

3.4.2. Arbres de joc: algorisme Minimax

Els **arbres de joc** (arbres amb **totes les jugades possibles** d'un joc) són molt populars per als jocs en els quals participen dos jugadors, sobretot en els anomenats jocs de taula. Els elements d'un arbre de joc són aquests:

- Cada node representa un estat possible del joc. El node arrel és l'inici del joc. Els nodes fulla són els diferents finals del joc.

- Cada enllaç representa una jugada d'un jugador, que lògicament fa que el joc canviï d'un estat (node) a un altre.

- Per tant, tots els enllaços que són al mateix nivell (mateixa distància del node arrel) són les jugades possibles en un moment donat, i tots els nodes del mateix nivell són els estats possibles del joc després de la jugada corresponent. Per exemple, els nodes de nivell 3 (suposant que l'arrel és nivell 0) són els estats possibles després de tres jugades.

- En ser jocs de dos jugadors, cada nivell d'enllaços correspon a les jugades de cadascun dels dos jugadors alternativament (la primera fila d'enllaços són les jugades possibles del jugador 1; la segona, les del jugador 2; la tercera, les del jugador 1, etc.)

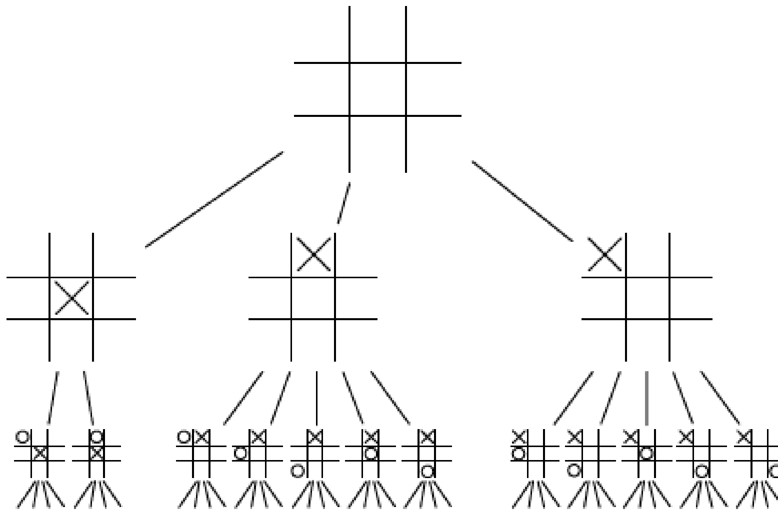
- Així mateix, els nivells de nodes corresponen a l'estat del joc després de la jugada de cada jugador alternativament (nivell 1, estat després de jugada del jugador 1, etc.).

Normalment, la generació d'arbres de joc és limitada pels recursos del sistema, per la qual cosa, en general, no podrem arribar fins a la situació final del joc (guanyar, perdre o empatar, per exemple), excepte en jocs molt senzills.

El tres en ratlla és un exemple perfecte per a utilitzar un arbre de joc. Partint del tauler buit (que serà el node arrel), i suposant que comenci el jugador amb les fitxes marcades com a «X», es genera un node fill per a cada jugada possible del jugador, en aquest cas col·locar la «X» en cadascuna de les nou caselles del tauler (recordem que és la primera jugada).

Al seu torn, en resposta a cadascuna de les nou jugades possibles, el contrincant pot col·locar la seva fitxa «O» en alguna de les vuit caselles que quedaran lliures, per la qual cosa cada node generarà al seu torn vuit nodes fill amb cada resposta possible del jugador «O»; cadascun d'aquests nodes, al seu torn, permetrà set posicions possibles al jugador «X», i així successivament fins a finalitzar el joc (per ocupar totes les caselles o perquè un dels dos jugadors fa tres en ratlla) en els nodes fulla.

Exemple d'arbre de joc per al tres en ratlla.



Tres en ratlla

Tal com es veu, el nombre de nodes creix ràpidament; en el cas del tres en ratlla, que és un joc molt senzill, les jugades possibles són de l'ordre de 9! = 362880; en realitat, són menys, ja que algunes branques acaben abans de completar totes les possibilitats, quan s'aconsegueix fer tres en ratlla.

En altres jocs més complexos el màxim nombre de nivells de l'arbre normalment és bastant petit. Per exemple, el supercomputador Deep Blue que va guanyar Gary Kasparov solament era capaç d'introduir en l'arbre els dotze moviments següents per a aplicar-hi després una heurística a sobre.

L'objectiu d'utilitzar arbres de joc és que la IA triï la jugada que li proporcioni més possibilitats de guanyar. Una senzilla manera de fer-ho seria explicar quantes branques derivades de cadascuna de les opcions actuals porten a la victòria (o quina proporció d'aquestes hi porten: per exemple, 100 em donen la victòria i 20 em fan perdre). No obstant això, aquest algorisme tan senzill té dues limitacions importants:

- Requereix que es pugui generar l'arbre de joc complet (o que es pugui avaluar si una branca conduirà a la victòria amb una certa probabilitat).

- Suposa que el contrincant juga triant a l'atzar, ja que solament mira el nombre de branques que donen la victòria, però perquè aquest senzill càlcul de probabilitats funcioni és necessari que el contrincant no mostri al seu torn cap intel·ligència.

Per això, és necessari utilitzar algorismes més avançats de gestió dels arbres de joc. Un dels més utilitzats és l'algorisme **Minimax**, que intenta buscar la millor estratègia dins del coneixement de l'arbre, assumint que tots dos jugadors jugaran sempre el seu millor moviment, és a dir, assumint que el contrincant és intel·ligent. El jugador A (el que usa l'algorisme) ha d'intentar aconseguir el valor màxim (el que li dona el major benefici), mentre que el jugador B (el seu adversari) jugarà perquè A obtingui un valor mínim (menor benefici). En altres paraules, el funcionament de Minimax pot resumir-se en triar el millor moviment per a un mateix suposant que el contrincant seleccionarà el pitjor per a nosaltres.

La primera versió formal de Minimax

La primera versió formal de Minimax va ser publicada per John Von Neumann (el mateix que va definir l'arquitectura moderna dels ordinadors, que va permetre que fossin programables) en 1928, i assumeix que s'aplica en els anomenats jocs de **suma zero**, en els quals la pèrdua de punts o equivalent d'un jugador comporta el mateix guany per a l'altre jugador, i viceversa. Els escacs i el pòquer són exemples de jocs de suma zero.

El funcionament de l'algorisme és bastant simple. En primer lloc, s'ha de definir una **funció d'utilitat** que assigni un valor numèric a un node de l'arbre. Per exemple, en jocs com el tres en ratlla hi ha tres valors possibles: +1 (guanyar), –1 (perdre), 0 (empatar). Altres jocs poden tenir un major rang de valors (per exemple, els punts aconseguits).

Una vegada generat l'arbre de joc, s'aplica la funció d'utilitat a cada node fulla (que, recordem, implica la fi de la partida).

Suposant un joc de dos jugadors, cada nivell de nodes de joc correspon alternativament a cadascun dels jugadors (és com queda el joc després de la jugada d'un dels jugadors). Per aquest motiu, des del punt de vista del jugador que executa l'algorisme (i que se suposa que vol guanyar el joc), hi haurà dos tipus de nivells:

- **Nivells «max»:** nivells que corresponen al jugador, en els quals s'ha de **maximitzar el guany**, ja que és el jugador el que tria la jugada.
- **Nivells «min»:** nivells que corresponen a l'adversari, en els quals s'ha de **minimitzar la pèrdua**, ja que és l'adversari el que tria la jugada i l'única cosa que podem fer és protegir-nos per no perdre molt faci el que faci l'adversari.

En els escacs, un nivell «max» ben aprofitat podria implicar menjar una peça a l'adversari (diguem una torre), però si a canvi ens hem arriscat molt, en el nivell següent (que serà «min») pot ser que hi perdem més (que ens mengi la reina, que val més).

Resumint: s'han de maximitzar els guanys i minimitzar les pèrdues.

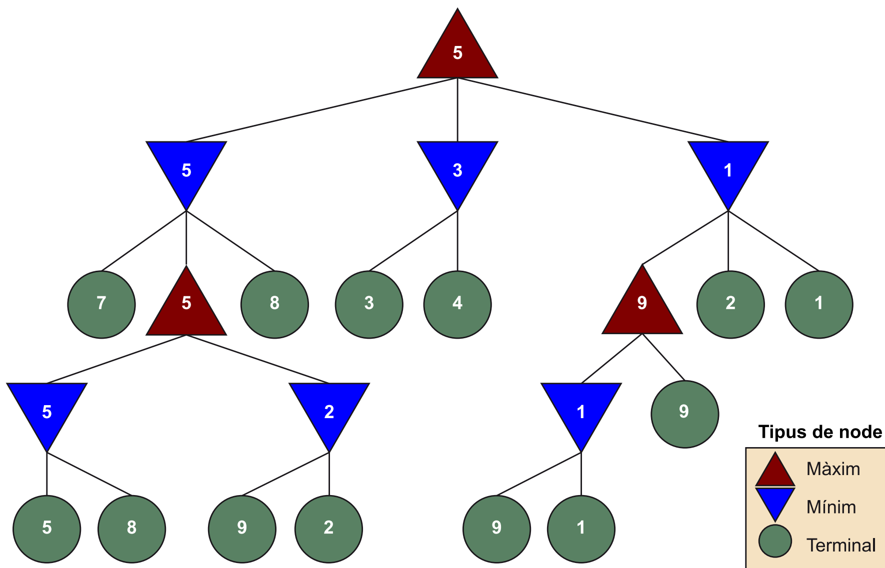
En l'exemple següent es pot veure un exemple d'aplicació de Minimax, en el qual tenim un joc amb un resultat possible (funció d'utilitat) entre 1 i 9.

- En primer lloc, es calculen aquests valors (cercles verds), que són els possibles resultats finals del joc.
- A continuació es va emplenant l'arbre cap amunt tenint en compte si estem en un nivell «max» o «min», és a dir, si tria el jugador o l'adversari.
- Si tria l'adversari («min», triangles blaus que apunten cap avall), triarà l'opció amb menys punts d'entre els nodes fill (per exemple, el 5 de la penúltima fila, esquerra, ja

que pot triar entre fer una jugada que ens doni 5 punts i una que ens en doni 8, i ell vol que aconseguim la menor quantitat de punts possible).

- Si triem nosaltres («max», triangles vermells que apunten cap amunt), seleccionarem la jugada que ens doni la màxima puntuació. Per exemple, el triangle de la tercera fila, segon per l'esquerra, que tria 5, ja que pot anar a un estat amb 5 punts i a un altre amb 2).

- Una vegada completat l'arbre, es comença el joc i es van prenent les millors decisions segons el que s'ha vist en l'arbre.



Font: CC Wikimedia Commons, per ArthaSphinx

1) Millores

El principal problema dels algorismes d'anàlisis d'aquests arbres és que són computacionalment molt costosos. En un joc real normalment no s'avaluen totes les solucions perquè en alguns casos podem veure des del principi que ens portaran a un fracàs. En aquest cas, es diu que «podem» la branca.

Una de les tècniques més utilitzades és la **poda alfa-beta**, que consisteix en una simple modificació del Minimax per a descartar les branques que no poden millorar el resultat actual.

Seguint amb l'exemple anterior, en el nivell 0 (arrel) estem buscant el màxim. Analitzem les branques des de l'esquerra i veiem que en la primera obtenim un 5. En la segona branca obtenim un 3. No és necessari analitzar en profunditat la tercera branca (la de l'1), ja que encara que té una branca que ens pot donar 9, mai no aconseguirem aquest 9 perquè és un nivell «min» i l'adversari ens donarà l'1; així que la branca del 9 i els seus descendents es poden eliminar de l'exploració perquè l'adversari mai no ens les donarà, ja que té l'alternativa de l'1 però tenim quelcom millor en un altre node «max» (el 5).

3.4.3. Arbres de comportament

Les màquines d'estats finits vistes anteriorment permeten modelar el comportament d'un agent però tenen alguns inconvenients:

- Si el comportament que es vol modelar és complex, dissenyar una màquina d'estats finits pot convertir-se en una tasca molt complexa i propensa a errors.

- Per situacions inesperades o fallades de disseny, sol passar que els agents controlats per màquines d'estats finits es quedin embussats en un estat i siguin incapaços de sortir-ne.

Com sempre que un sistema es torna molt complicat per a tenir un nombre de components massa elevat per a ser manejable, cal organitzar i estructurar els components per a aconseguir que funcioni com esperem. Els **arbres de comportament** (AC) són una alternativa a les màquines d'estats finits que permeten crear models de comportament més complexos. Es van començar a utilitzar en jocs com Halo 2 i Bioshock, que destacaven per la IA dels seus NPC.

Els AC tenen els elements següents:

1) Els nodes fulla reben el nom d'**executors** i representen accions que l'agent pot dur a terme (disparar, caminar, obrir una porta, accelerar...). Quan reben l'ordre per a executar-se poden retornar un d'aquests tres resultats possibles:

- **Executant.** L'acció corresponent s'està executant però no ha acabat. Per exemple, l'agent està caminant.

- **Èxit.** L'acció corresponent s'ha executat i ha acabat amb èxit. Per exemple, l'agent estava caminant i ha arribat a la seva destinació.

- **Fallada.** L'acció corresponent no s'ha pogut iniciar o en intentar-ho s'ha fallat. Per exemple, no ha pogut obrir la porta.

2) Uns nodes **controladors**, que sempre tenen fills (de tipus executor o controlador). La seva funció és organitzar l'execució dels nodes executors segons diferents tipus de comportament. Bàsicament, representen una tasca més o menys complexa que ha de dur a terme l'agent. Quan reben l'ordre per a executar-se reexpedeixen l'ordre d'execució als seus fills segons algun dels comportaments següents:

- **Seqüència.** El node controlador envia l'ordre d'execució als seus nodes fills d'esquerra a dreta per fer una operació que requereix seguir una sèrie de passos. Quan un fill retorna «èxit», el controlador passa al següent; mentre el fill retorna «executant», hi roman; si un fill falla, el controlador falla.

- **Selector.** El node controlador envia l'ordre d'execució als seus nodes fills d'esquerra a dreta. Quan un fill retorna «èxit», el controlador retorna «èxit» i acaba. Si el fill retorna «executant», el controlador retorna «executant». Si un fill falla, el controlador passa al següent; si tots fallen, retorna «fallada».

3) Un node **arrel**. La seva funció és enviar, cada cert temps, un senyal (*tick*) al seu node fill, que és una ordre perquè s'executi. El node arrel solament té un fill, de tipus controlador (vegeu a continuació).

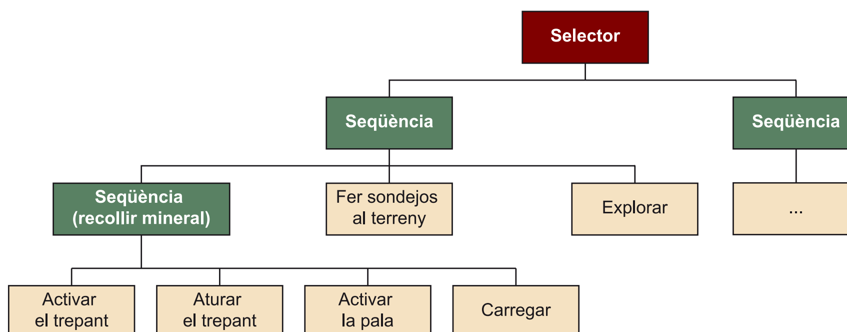
Per tant, una IA controlada per AC ha de generar un esdeveniment cada cert temps (poc, inferior al segon) perquè es vagin executant les tasques. La mateixa estructura de l'AC permet modelar un comportament mitjançant un controlador general: la IA ha de seguir uns passos o simplement fer una tasca que triarà entre diverses. A continuació, en un altre nivell de detall, es defineixen les tasques fonamentals que s'han de dur a terme; aquestes tasques es poden dividir en altres tasques més concretes, i així fins a aconseguir accions del joc, que seran els nodes d'execució.

AND i OR i un altre tipus de controladors

Podem veure els controladors de seqüència com AND per a tasques en les quals és necessari seguir tots els passos (recollir troncs, moure el fuster a la posició que es vol, construir la casa). Els controladors selectors, per la seva banda, són una OR en la qual qualsevol de les accions és suficient per a resoldre la tasca (obrir una porta amb la maneta, amb una clau, o esbotzar-la). Hi ha altres tipus de controladors, com els inversors, repetidors, selectors aleatoris, etc.

Un robot que recorre un planeta a la recerca de minerals valuosos tindrà al principi un node selector per a triar entre dues tasques: recollir minerals i mantenir-se en funcionament. La tasca principal és recollir minerals, així que serà la primera, i sempre que sigui possible s'executarà aquesta. Al seu torn, la tasca selectiva de recollir minerals es compondrà de diverses subtasques: prendre mineral, fer sondejos al terreny, explorar. Noteu que l'ordre és el contrari de l'esperat perquè s'especifica la prioritat de les tasques: si el robot té minerals identificats al seu abast, els ha de recollir. Si no és així, ha de sondejar el terreny per a veure si hi ha alguna cosa que valgui la pena. I, si no, ha de continuar explorant. La tasca de prendre el mineral és representada per un controlador seqüencial: activar el trepant, aturar el trepant, activar la pala, carregar al contenidor, i així successivament amb la resta de les tasques.

Exemple d'arbre de comportament d'un robot miner. Els nodes granat són els selectors; els verds, els de seqüència; i els rosa, els d'accions.



Propietats dels AC:

- Com que un node controlador pot tenir altres nodes controladors com a fills, l'execució de l'AC pot ser recursiva.

- Per a l'execució correcta, els nodes de l'AC han de recordar el seu estat entre una invocació (*tick*) i la següent; si un agent camina, ha de recordar-ho.

- Permeten representar comportaments molt complexos amb un gran nivell de detall però amb una visió global que evita embussos en estats dissenyats pobrament.

- És senzill prendre una part d'un AC per reutilitzar-la en un altre: són modulars.

Behavior Designer

L'*asset* de Unity anomenat Behavior Designer permet dissenyar i utilitzar arbres de comportament fàcilment.

Les principals limitacions dels AC són aquestes:

- Si els arbres són molt extensos, pot resultar costós computacionalment executar-los.

- La qualitat final del comportament depèn de la qualitat de les condicions programades en cada node, la qual cosa continua essent en molts casos un problema sense resoldre.

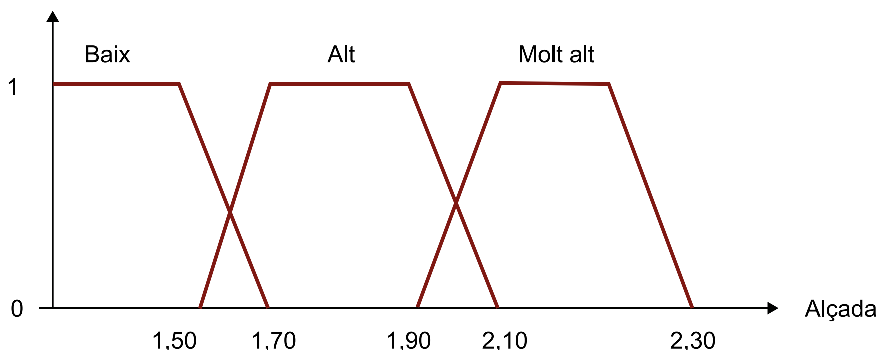
3.5. Lògica difusa

Una proposició en la lògica clàssica solament admet dos valors possibles: veritable o fals. No obstant això, en la vida real les observacions no són tan clares per a poder afirmar que una proposició és veritable o falsa, sinó que pot haver-hi diferents punts de vista que siguin relatius a l'observador.

La lògica difusa és una alternativa que ens permet quantificar aquesta incertesa, ens dona l'opció d'assignar certa probabilitat a cadascun dels possibles valors i, per tant, afegir la relativitat de l'observador. Segons el seu creador, la idea original que hi ha darrere de la lògica difusa és imitar el funcionament del raonament humà, el qual normalment no treballa amb valors exactes sinó amb valors relatius.

> Imagineu que volem jugar un partit de bàsquet i hem de col·locar cada jugador en la seva posició. Per a aquesta tasca, classifiquem la gent com a alta, baixa, lenta o ràpida, però mai no utilitzem mesures exactes, com «fa 1,80 metres» o «corre els 100 metres en 15 segons». La lògica difusa ens permet introduir aquests conceptes, més relatius als que estem acostumats dins d'un sistema d'IA.

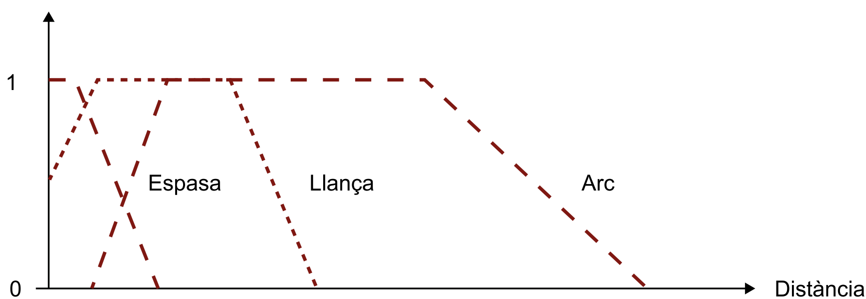
Un valor en la lògica difusa pot estar entre veritable i fals, i pot tenir valors de molt, una mica... Aquesta conversió d'un valor quantitatiu a un valor qualitatiu es fa mitjançant el que es coneix com a subconjunt difús (*fuzzy set*), i representa la probabilitat que algú assigni un cert valor qualitatiu a un de quantitatiu.



En la figura anterior tenim una manera de convertir els valors. Per a algú que faci menys d'1,50, la probabilitat que l'anomenem baix és 1, però si en canvi fa 1,60 hi haurà un cert percentatge de gent que el podrà classificar com a alt i un altre com a baix.

La lògica difusa no és una tècnica com les que hem presentat fins ara, que ens permeten prendre una decisió a partir de l'observació del sistema; la podem considerar més aviat com un sistema complementari dels vistos, que provoca que les decisions que es prenguin tinguin un vessant més relativista i per tant més «humà».

1) Si combinem la lògica difusa en un sistema de regles *if…then*, podem fer que les regles semblin més realistes. Imaginem un cas en el qual hem de triar l'arma amb la qual atacar (espasa, llança o arc) depenent de la distància de l'objectiu. En les regles bàsiques podem definir tres rangs on decidirem l'arma que volem portar (per exemple, per a menys de cinc metres usarem una espasa, entre cinc i vint metres, una llança, i a partir d'aquí, l'arc). En canvi, si volem que sigui més realista, podem introduir un subconjunt difús com el següent i després triar l'arma segons la probabilitat de cadascuna.



2) Si combinem la lògica difusa amb les màquines d'estat, obtenim el que es coneix en IA com a màquines d'estat difuses. Aquest tipus de màquines permeten que el sistema es trobi en més d'un estat alhora, és a dir, tenim una sèrie d'estats actius amb una certa probabilitat i a cada moment decidim quin és el que ens interessa utilitzar.

Mètode IA d'utilitat

La lògica difusa ha guanyat actualitat amb el mètode anomenat **IA d'utilitat** (Utility AI), en el qual es combinen diverses mesures difuses amb les diferents opcions que té un agent i la utilitat que tenen per a ell en un moment donat per a triar així l'opció que li sigui més útil. Aquest mètode té l'avantatge d'aconseguir un comportament flexible i resulta fàcil de dissenyar, ja que no és necessari preveure totes les situacions (estats) possibles, sinó que dona sempre una resposta.

Utility AI està disponible per a Unity mitjançant l'*asset* Apex Utility AI (de pagament).

3.6. Mapes d'influència

Els mapes d'influència són una representació discreta del coneixement que té el jugador sobre el món. Són un element necessari per a processos de decisió estratègics i tàctics.

Popularitat dels mapes d'influència

Són molt populars en tot tipus de jocs d'estratègia, des de jocs de tauler fins a jocs d'estratègia en temps real, ja que permeten estudiar la situació i influència de les peces de l'enemic.

Els mapes d'influència es poden utilitzar per a diferents objectius:

- detectar les regions que ens interessen, les regions que hem d'evitar i el lloc on hi ha la frontera entre aquestes regions;
- buscar punts febles de l'adversari analitzant la localització de les seves defenses;
- guiar el moviment d'un element;
- quantificar si el nostre oponent té més forces que nosaltres (si els valors negatius indiquen la força del contrari i els positius les nostres, la suma de tots els valors ens indicarà qui és més fort).

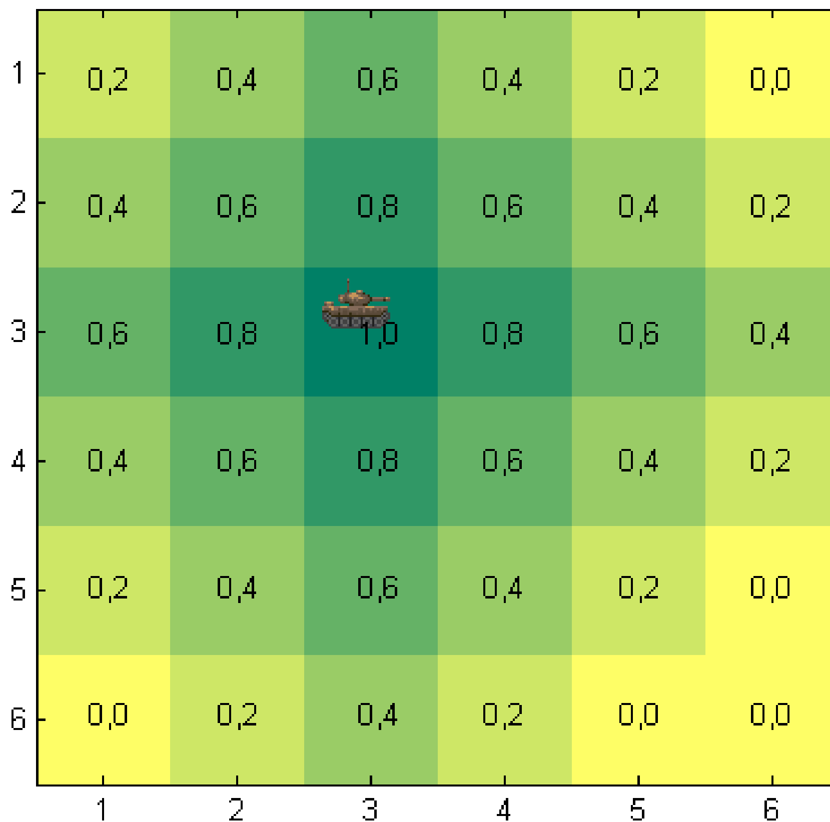
Per a crear un mapa d'influència, el primer que hem de fer és segmentar el mapa del món en regions discretes. Podem utilitzar tant els *grids* com les malles poligonals vistes en el mòdul anterior, i no cal que coincideixin amb la geografia real del món.

Després hem d'assignar un valor numèric a cadascuna de les zones del mapa d'influència que té relació amb la partida que es desenvolupi (força estratègica de la posició, nombre d'unitats enemigues a la zona, contingut en recursos d'aquesta zona, etc.).

El càlcul del valor d'aquesta posició és un element clau per a triar posteriorment les decisions de manera correcta, així que serà necessari el disseny d'una bona equació que tingui en compte dos grups d'elements:

- Tots els elements que siguin a la zona i que influeixin en el seu valor puntual.
- La influència dels elements que siguin a les zones limítrofes. Com més lluny siguin, menor serà la seva influència.

Exemple d'un mapa d'influència en un *grid* quadrat



Una manera bastant utilitzada de calcular mapes d'influència és calcular un mapa per cada zona que contingui algun element interessant (per exemple, una unitat o un edifici) i afegir la influència d'aquest element a les altres zones. Posteriorment, agreguem tots aquests mapes per obtenir el mapa final.

Els mapes d'influència són elements dinàmics. Al principi del joc podem calcular el valor inicial de les zones, però cada vegada que hi hagi algun moviment haurem de recalcular un nou mapa amb la nova situació. Cal tenir en compte que no és necessari recalcular totes les caselles: solament les afectades pel canvi, i així el procés és menys costós del que sembla.

Podem definir un comportament senzill a manera d'exemple. Suposem que el valor de cada cel·la del mapa d'influència és un valor entre –1 i 1 que defineix la «força» d'un oponent. Si és un valor positiu, indica que tenim més força que el nostre oponent en aquest punt, mentre que en cas contrari l'oponent en tindrà més que nosaltres.

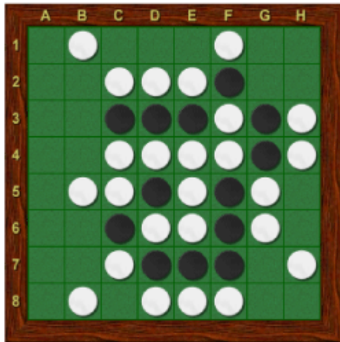
D'altra banda, cada element (soldat, tanc, etc.) té un valor de força entre –1 i 1, però el seu significat canvia una mica. Si el valor és negatiu, indica que té problemes i vol fugir de l'enemic (podríem definir un altre comportament, com anar a la recerca d'ajuda, aliments, etc.). Si el valor és positiu, ens assenyala que està en condicions d'atacar.

El mapa d'influència l'hem definit a base de quadrats. Així, l'element solament pot moure's a una cel·la de les seves vuit adjacents. D'aquestes vuit, triarà la que tingui un valor menor que el seu si està en condicions d'atacar, o un nombre major si té problemes.

Un exemple més simple de l'ús de mapes d'influència el trobem en la planificació d'estratègies en jocs de tauler.

Per exemple, en el joc Othello (també conegut com Reversi), que consisteix a col·locar les fitxes en el tauler per poder capturar les del contrincant, a cada posició del tauler se li pot assignar un pes que ens indica quin benefici obtenim si posem una peça en cada casella.

Exemple d'una partida, juntament amb el seu mapa d'influència



34		4	6	6		-1	34
-2	-9					-8	-1
6	2						
3	1						
3							5
6	2						4
-2	-9					-7	
34		8				-5	34

Aquesta informació és crucial per a poder desenvolupar una bona estratègia, per a intentar col·locar les peces en les posicions més favorables i evitar les posicions menys avantatjoses.

4. Aprenentatge

Els mètodes vistos fins ara ofereixen solucions per a un ampli ventall de situacions de joc i són molt utilitzats per la seva efectivitat a l'hora de resoldre els problemes per als quals han estat dissenyats. Però precisament aquí rau també la seva limitació principal: són tècniques que han de dissenyar i ajustar els desenvolupadors del joc, sovint mitjançant un costós procés d'assaig i error que no sempre dona bons resultats. I, a més, el seu resultat, com que és prefixat en el disseny del joc, és bastant predictible. Imaginem, per exemple, un arbre de decisió en el qual la cadena de decisions es repeteix milers de vegades i en el qual els jugadors poden d'alguna manera acabar reconeixent-la i aprendre les respostes dels agents de la IA. Una solució simple per a evitar aquesta resposta predictible consisteix a introduir elements aleatoris, però això provoca, per la seva pròpia naturalesa aleatòria, que els agents facin coses sense molt sentit de tant en tant. El jugador percebrà una certa intel·ligència en els agents, però tampoc no molta.

L'ideal seria que la IA reaccionés de manera flexible davant les diferents situacions del joc, i que fins i tot innovés en funció de l'estil de joc del jugador, perquè aquest realment se sorprengués en veure la IA aprenent com juga i trobant-li punts febles.

Però per a això és necessari que la IA aprengui durant l'execució del joc: no es poden predissenyar totes les respostes possibles. Per a aconseguir dotar els jocs d'aquesta intel·ligència avançada, en aquest apartat veurem mètodes que aprenen a partir de les diferents situacions del joc i són capaços de corregir el seu comportament per a millorar els seus resultats i, el que és més important, continuar entretenint el jugador.

4.1. Estratègies d'aprenentatge en jocs

En general, els mètodes d'IA que aprenen (també coneguts com a mètodes d'**aprenentatge automàtic** o *machine learning*) distingeixen dues fases. En primer lloc, hi ha una fase d'**entrenament**, en la qual el mètode rep un conjunt de dades (sovint juntament amb la resposta correcta) i l'utilitza per a aprendre ajustant els seus paràmetres interns i creant un model del problema al qual s'enfronta. En el nostre cas, les dades seran les diferents situacions de joc, i el problema que haurem de resoldre consisteix a decidir les accions correctes perquè l'agent aconsegueixi la màxima puntuació en el joc o simplement ofereixi un repte més interessant als jugadors. Normalment, són desenvolupadors els que juguen contra la IA per entrenar-la.

Entrenament de la IA

Depenent de l'organització de l'empresa que desenvolupa el videojoc, aquesta labor la poden dur a terme els mateixos desenvolupadors de la IA, els *internal testers* (que proven versions intermèdies del joc) o els *quality assurance testers* (que proven el joc acabat). Fins i tot moltes vegades s'obre al públic una versió beta del joc per aprofitar que jugadors reals usin el joc i així usar aquest temps de joc per a acabar d'ajustar la IA.

Una vegada entrenat el mètode, es passa a la fase de **prova**, en la qual s'utilitza el model après per a reaccionar de la millor manera possible en cada situació de joc que es presenti. En el cas dels videojocs, la prova és l'ús dels mètodes en el joc definitiu contra jugadors reals (no contra desenvolupadors que es dediquen a entrenar la IA).

Un agent per a un joc de tennis pot entrenar jugant contra persones o contra altres agents i anar aprenent dels seus èxits i errors per trobar estratègies òptimes en cada situació (bola baixa, bola alta, bola a l'altra part de la pista...). Aquesta seria la fase d'entrenament. La fase de prova del mètode seria la posada en ús de l'agent en el joc real utilitzant el model après prèviament per a prendre les decisions necessàries a cada moment.

4.1.1. Moments de l'aprenentatge

La fase d'entrenament de la majoria dels mètodes d'IA és relativament costosa en termes de consum de memòria i temps d'execució, per la qual cosa en la majoria dels casos no és pràctic dur-la a terme durant l'execució del joc (a més, un jugador vol un joc que ja sàpiga jugar, no que encara hagi d'aprendre i comenci jugant molt malament). Per aquest motiu, la fase d'entrenament es fa durant el desenvolupament i ajustament del joc; en aquest cas es parla d'aprenentatge **fora de línia**, ja que no es fa en temps d'execució del joc final.

D'altra banda, alguns mètodes més lleugers poden aprendre durant el joc per si mateixos; en aquest cas parlarem d'aprenentatge **en línia**.

Finalment, hi ha la possibilitat de fer un entrenament previ en mode fora de línia però deixar els ajustaments finals (per exemple, per a reaccionar a l'estil de joc de cada jugador) per a fer una mica d'entrenament en línia.

4.1.2. Modes d'aprenentatge

Un altre aspecte fonamental dels mètodes d'aprenentatge automàtic és com es fa l'aprenentatge, és a dir, de quina informació disposa el mètode per a anar aprenent. Segons aquest criteri, es distingeixen quatre modes fonamentals:

- **Aprenentatge no supervisat.** El mètode rep una sèrie d'atributs o característiques dels exemples que ha de processar. Tasques típiques d'aquest mode són l'agrupament (*clustering*) i la reducció de la dimensionalitat. Per exemple, podem rebre les dades d'ús d'un joc (temps mitjà jugat, modes de joc utilitzats, puntuació aconseguida) i en funció d'això formar grups de perfils d'usuari típics.

- **Aprenentatge supervisat.** A més dels atributs dels exemples d'entrenament, es rep també una **etiqueta** que indica de quina classe o tipus és aquest exemple. Tasques típiques d'aquest mode són la detecció i classificació d'anomalies. Per exemple, podem rebre les característiques d'un avió i la seva classe perquè un pilot automàtic d'un simulador de vol aprengui si es tracta d'un avió de combat, de passatgers, etc.

- **Aprenentatge semisupervisat.** Alguns exemples d'entrenament estan etiquetats i uns altres no; es comença aprenent dels etiquetatges i després se suposa una etiqueta als no etiquetats en funció de la seva similitud amb els etiquetatges. Això és útil perquè sovint és costós etiquetar exemples (ha de fer-ho un humà), i així s'aprofita un conjunt d'etiquetatge petit per a entrenar amb un conjunt d'exemples molt més gran. Per exemple, si volem que un sistema de conducció automàtica detecti si hi ha vianants en una imatge, podem etiquetar algunes imatges (centenars o milers) i a més proporcionar-li milions d'imatges sense etiquetar per a millorar l'entrenament.

- **Aprenentatge per reforç (*reinforcement learning*).** En molts problemes no és natural parlar d'exemples i etiquetes, ja que es tracta d'entrenar un sistema perquè interactuï amb un entorn (real o virtual) i vagi millorant amb la pràctica. Donat l'especial interès d'aquest tipus d'aprenentatge per als videojocs, li dedicarem un apartat complet.

Aprenentatge per reforç

L'aprenentatge per reforç aplicat a la IA en videojocs ha adquirit una fama gran i merescuda pels seus recents èxits, com el sistema DQN (de Deep Q-Network, que ara s'explicarà), que és capaç de jugar a jocs de la consola Atari i guanyar jugadors humans experts, o el sistema AlphaGo, que ha guanyat el campió mundial de go (un joc de tauler d'origen xinès que és molt més complex en nombre de jugades possibles que els escacs). En tots dos sistemes s'ha demostrat el potencial d'aquesta tècnica.

També s'estan aplicant amb èxit sistemes de conducció automàtica o robòtica a simuladors de diferents tipus; en tots aquests àmbits, l'agent pot fer accions i aprendre de les seves conseqüències.

Observació

Cal destacar que el sistema DQN aprèn a jugar sense cap coneixement previ del joc: solament rep la imatge de vídeo (el mateix que veuria un jugador humà) i va provant diferents accions per anar aprenent a poc a poc amb quines accions aconsegueix una major puntuació. A més, no se li ha programat cap tipus de coneixement sobre cada joc en particular, i malgrat això ha estat capaç de dominar els jocs i fins i tot trobar «trucs», com en el joc Breakout, en el qual aprèn a fer un forat en la fila de blocs perquè la bola pugi a la part superior i així destrueixi molts blocs ràpidament.

Captura de pantalla del joc Breakout de la consola Atari 2600

Font: CC McLoaf per a Wikimedia Commons a partir del joc Breakout propietat d'Atari Inc.

Vegem en què consisteix el *reinforcement learning* (RL abreujat). Tenim un problema o situació per la qual volem entrenar un agent de manera que aprengui a fer accions adequades en cada moment. Perquè l'agent sàpiga que ho està fent bé, ha de rebre una «recompensa» quan la seva acció encerta en el sentit del comportament que es vol: jugar bé a un videojoc, conduir bé un cotxe per una autopista, moure un robot entre dues posicions esquivant obstacles, etc. Més detalladament, els elements que componen el sistema RL són aquests:

- Un conjunt S d'**estats** possibles del problema (joc, etc.). En l'exemple del Breakout, els estats possibles són totes les possibles posicions de la barra que mou el jugador, per totes les possibles posicions i velocitats de la bola, per totes les combinacions possibles de maons sencers o destruïts. Tal com es veu, fins i tot per a un joc tan senzill, el nombre de possibles estats és immens.

- Un conjunt A d'**accions** possibles. En el joc Breakout són solament tres: moure's a esquerra, a la dreta o quedar-se quiet. En un joc d'estratègia, per exemple, seran moltíssimes més.

- Una funció R de **recompensa** , que retorna un nombre real en recompensa per l'acció presa per l'agent en un estat determinat, $R: S \times A \rightarrow \mathbb{R}$. És molt important notar que la recompensa depèn de l'acció presa i de l'estat en el qual es trobava el joc, ja que de vegades moure's a l'esquerra serà bo però altres vegades no.

- En principi, l'agent ha d'aprendre's una **taula de recompenses** $Q = S \times A$ per a saber així què fer en cada possible situació del joc. Lògicament, això és inviable en gairebé qualsevol joc (que sigui més complicat que el tres en ratlla), i per aquest motiu s'han proposat diferents mètodes per a aprendre una aproximació de Q acceptablement bona i que sigui pràctica. Aquests mètodes reben el nom de *Q-Learning*.

Observació

La Q que veiem és la del mètode DQN esmentat anteriorment; *deep* i *network* es refereixen al fet que el mètode no aprèn la matriu Q tal qual (seria impossible), sinó que aprèn a aproximar-la mitjançant una xarxa neuronal (d'aquí el *network*) profunda (d'aquí el *deep*). En l'apartat 5 s'explicarà què són aquestes dues tècniques i com poden usar-se amb RL.

El funcionament de l'RL amb tots aquests elements es produeix en moments de temps discret, {...,t-2, t-1, t, t+1, t+2,...}, per exemple en cada iteració del bucle de joc (cada *frame*). Quan l'agent actua en el temps t, passa el següent:

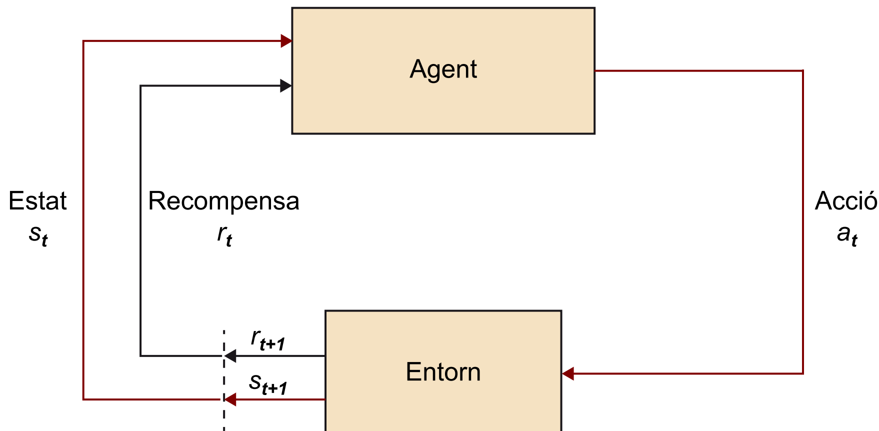
1) Rep la recompensa de la seva acció anterior, R_t.

2) Ha d'identificar quin és l'estat actual S_t a partir de la informació que li proporcioni el joc.

3) En funció de l'estat actual, l'agent ha de decidir quina és l'acció A_t que li proporcionarà una major recompensa en la propera iteració, R_{t+1}.

4) En aplicar l'acció, l'entorn canvia i s'inicia una nova iteració (tornada al pas 1). La recompensa rebuda s'associa a l'ús de A_t en S_t, és a dir, si la recompensa és bona s'associarà aquesta resposta a aquest estat, i si és dolenta es deixarà de costat aquesta acció com a resposta a aquest estat.

Diagrama de control d'un sistema RL



Font: adaptat d'https://deeplearning4j.org/reinforcementlearning

Una manera senzilla de programar un agent perquè prengui una decisió és mitjançant l'algorisme ϵ-*greedy* (voraç-èpsilon), que consisteix a triar l'acció que doni la major recompensa en la iteració següent amb probabilitat $1-\epsilon$, o una acció a l'atzar en un altre cas. En el fons és una manera simple de tractar un problema de RL: les recompenses no sempre són immediates, sinó que poden arribar al cap de moltes iteracions.

En el joc Breakout, la pilota baixa, l'agent mou la barra cap a la pilota, la copeja, la pilota rebota i al cap d'uns segons trenca un bloc i dona 10 punts a l'agent. Però mentrestant han passat, per exemple, 3 segons, que poden ser 90 *frames*, és a dir, potser 90 iteracions. Com ho fa, el mètode RL, per a relacionar l'acció anterior amb la recompensa futura?

Una estratègia més avançada per a afrontar aquest problema és la consideració de les recompenses futures en l'algorisme d'aprenentatge. Si en l'instant t el joc es troba en l'estat S_t i s'ha fet l'acció A_t, s'han d'associar les recompenses rebudes en les iteracions següents, $\sum_{i=t}^{t+N} R_i$, no solament la recompensa següent; d'aquesta manera, es premia una acció encara que la recompensa arribi tard. En aquesta estratègia se suposa que la recompensa s'aconsegueix després de N iteracions.

Si no és així i no hi ha manera de conèixer quan s'aconseguirà la recompensa, se sol aplicar la fórmula següent, que inclou un descompte $0 < \gamma < 1$ que va reduint el pes de la recompensa a mesura que s'allunya en el temps: $\sum_{i=t}^{\infty} \gamma^i R_i$.

En qualsevol dels dos casos, encara que es complica el desenvolupament d'aprenentatge, es tenen en compte recompenses futures, la qual cosa millora els resultats de l'algorisme en la majoria de les situacions.

Bootstrapping

Anteriorment hem vist que l'entrenament d'un agent d'IA per a un joc se sol fer en la seva major part fora de línia, d'una banda per no ocupar recursos computacionals quan s'executa el joc, i de l'altra per no lliurar un producte incomplet als clients. Però com es duu a terme aquest entrenament quan es vol entrenar una IA que jugui contra humans? Es contracten equips de jugadors professionals perquè juguin contra la IA i li ensenyin a jugar? A vegades es fa això precisament, però en molts casos les hores d'entrenament que necessita la IA són massa i no resulta viable utilitzar aquest recurs, almenys no exclusivament.

En aquest àmbit, la solució se sol denominar *bootstrapping* i consisteix en una cosa molt senzilla: que una IA jugui contra una altra. Les dues poden usar el mateix algorisme (es recomana que amb ajustaments diferents) o algorismes diferents. Fins i tot és interessant posar a competir la IA amb diversos algorismes diferents perquè vagi aprenent de cadascun. I a mesura que els dos agents milloren, han d'esforçar-se més per a guanyar l'altre i encara han de millorar més.

> Per exemple, utilitzant aquesta estratègia s'ha aconseguit una IA que juga als escacs com un expert entrenant solament unes hores amb un ordinador personal, a diferència del famós Deep Blue que va guanyar Kasparov en 1997 i que era un supercomputador que calculava milions de jugades possibles. En teniu més informació a https://arxiv.org/pdf/1509.01549v2.pdf.

Bootstrapping

En l'àmbit més general de l'aprenentatge automàtic, *bootstrapping* és una tècnica d'aprenentatge semisupervisat que consisteix a atribuir etiquetes a exemples no etiquetats i usar-les com a entrenament, o fins i tot a «inventar-se» exemples semblats als existents per ampliar el conjunt d'entrenament i millorar els resultats. En l'àmbit de l'RL, s'utilitza el mateix terme per a l'estratègia exposada aquí per la similitud d'usar un agent d'IA per a generar exemples d'entrenament per a un altre agent d'IA.

4.2. Algorismes d'aprenentatge

L'objectiu principal d'aquest apartat és donar una visió dels algorismes d'aprenentatge automàtic i la seva possible aplicació. Abans de veure els diferents algorismes vegem quins problemes aborden, la seva nomenclatura i com els podem aplicar als videojocs.

Tots els algorismes d'aprenentatge es basen en l'extracció de coneixement a partir de conjunts de dades. El problema més important en aprenentatge és la classificació. En classificació partim d'un conjunt de dades etiquetades que representen objectes. La classificació consisteix a etiquetar objectes nous amb les classes pertinents partint del coneixement extret de les dades prèvies.

A manera d'exemple, imaginem que disposem d'una col·lecció de correus electrònics amb la informació afegida de si són correus brossa o no. La tasca de la classificació correspon a crear un model a partir d'aquestes dades i poder

aplicar aquest model per a construir un classificador de correu brossa amb l'objectiu de poder etiquetar automàticament si un conjunt nou de correus són brossa o no.

Aquests objectes (en el cas anterior, correus) se solen anomenar exemples i tenen associada una etiqueta a la qual anomenem classe (en el cas anterior si correspon a brossa o no). El vector que representa cada objecte sol rebre el nom de conjunt d'atributs, atributs i prou o entrades del sistema[1]. La classe pot rebre també el nom de sortida[2]. Si agrupem tots els vectors d'atributs en forma de matriu, rep el nom d'entrades o X; i el vector de totes les classes, el de sortida o vector Y.

(1) En anglès, *inputs*.

(2) En anglès, *output*.

El problema més important a l'hora de construir un classificador és la representació de la informació. És a dir, com guardem la informació dels diferents objectes en forma d'atributs i definim el conjunt de classes. Imaginem ara que tenim a la nostra disposició els informes mèdics d'un grup de pacients d'un hospital i volem desenvolupar un classificador per a diagnosticar de manera automàtica una certa malaltia a partir d'aquests informes. El primer pas que se'ns planteja és definir el conjunt de classes. Aquestes podrien ser simplement si el pacient pateix la malaltia o no. Però també podríem pensar a construir un conjunt de classes que ens permetessin destriar el grau de la malaltia: si està completament sa, si la té però en un estat inicial, etc.

El segon pas serà decidir com representem el pacient en forma d'atributs. Quines informacions són rellevants per a detectar la malaltia? La febre, la tensió arterial, l'edat, si és home o dona…

Una vegada decidit que volem guardar la febre, això pot ser si el pacient té febre o no, o directament la temperatura corporal. Alternativament, podríem pensar a guardar la temperatura cada 8 hores durant l'última setmana.

Quan trobem un atribut representat en forma numèrica, es diu que l'atribut és numèric. Una altra alternativa és que sigui nominal. Un atribut nominal correspon a un en el qual els valors pertanyen a un conjunt de categories. L'exemple de si un pacient té febre o no o el color d'un objecte serien exemples d'atributs nominals. Això és especialment important perquè hi ha molts algorismes que funcionen solament amb atributs nominals o solament amb atributs numèrics. Es pot donar el cas en el qual el nostre problema contingui els dos tipus d'atributs. Hi ha tècniques per a convertir un tipus d'atribut en l'altre, com la creació d'intervals per a passar un atribut numèric a nominal o l'assignació d'un nombre a cada valor nominal en el cas contrari. Molt sovint també és útil la normalització dels valors numèrics quan els valors dels diferents atributs són de rangs molt dispars[3].

(3) Les tècniques més usuals són l'estandardització i l'escalat (en anglès *standardization* i *scaling*).

Tots aquests temes els haurem de pensar en el moment de dissenyar el problema. Això és especialment rellevant en el cas de la programació de videojocs, ja que no hi ha molta documentació sobre aquest tema i els resultats depenen en gran manera d'aquest disseny.

En un joc de lluita, podríem decidir implementar un classificador que guiï un cert personatge no jugador predint d'alguna manera els moviments del jugador. Això obligaria el jugador a anar canviant el seu sistema d'atac. Per a dissenyar el classificador, haurem de dissenyar primer el conjunt de classes. Aquestes podrien ser el moviment que ha de fer el personatge no jugador: un cert atac, una certa defensa, una seqüencia tipus de defensa més atac… però també podríem pensar que predigui quin moviment farà el jugador per a, per exemple, poder triar després aleatòriament el moviment a fer d'entre uns quants en funció de la predicció. Això faria que fos menys predictible. Les decisions següents que haurem de prendre tindran a veure amb la representació de les situacions. És a dir, amb la representació dels exemples. Podríem pensar en els últims moviments que ha fet el jugador en una certa finestra de temps. Normalment, discretitzem el temps i establim una certa finestra (x unitats de temps). Però també podríem pensar a fer el mateix amb els moviments tant del jugador com de l'oponent no jugador. Podríem afegir informació de quin dels dos va guanyar el joc o pensar a ponderar els exemples en funció de si va guanyar o va perdre amb molta diferència. Tot això genera un grup nombrós de possibilitats de disseny en el qual hi haurà molts dissenys que no funcionin i uns altres que sí. Una vegada dissenyat tot, haurem de triar l'algorisme d'aprenentatge a utilitzar tenint en compte el tipus d'atributs que tenim, qüestions d'eficiència computacional, etc.

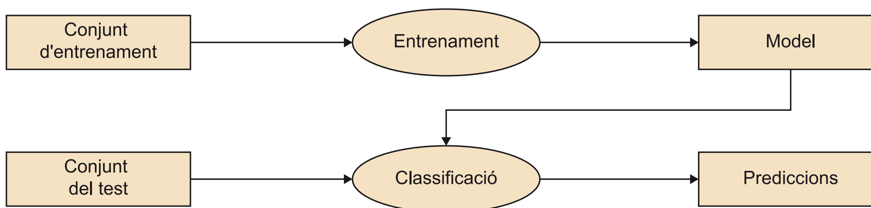
En un joc de rol en el qual el jugador controla una petita patrulla d'herois, podríem dissenyar classificadors a diferents nivells. D'una banda, podríem pensar en un classificador per a guiar els companys de l'heroi que no controli ell. En aquest cas podríem donar a l'usuari l'opció de definir les característiques generals de cadascun. D'altra banda, podríem dissenyar classificadors per a controlar les patrulles enemigues a dos nivells. Podríem pensar a dissenyar un classificador que guiï l'estratègia general de la patrulla. Aquesta prendria com a entrada els components de la patrulla del jugador i els de la seva pròpia, i establiria la tàctica general de la patrulla no jugadora. En aquest cas hauríem de definir *a priori* quines són aquestes tàctiques i proposar-les com a classes. A manera d'exemple, una d'aquestes tàctiques podria ser concentrar el foc dels arquers sobre els mags i evadir en tot moment els guerrers contraris, de manera que els mags confonguessin els guerrers i els guerrers ataquessin els arquers. En un segon nivell, podríem definir classificadors per a controlar els personatges individuals. A manera d'exemple, podríem tenir el classificador per a un arquer. En aquest cas les classes podrien ser fugir, posar-se en actitud defensiva amb escut i espasa, atacar el mag més feble o amb menys vida, atacar l'enemic que ataqui el seu company més feble… i els seus atributs podrien ser l'estratègia general que defineix el classificador superior, el seu nivell de vida, el dels seus companys, el dels membres de la patrulla enemiga, el nivell

de mannà o característica que s'utilitzi per a atacs especials... També podríem pensar a incorporar una mica d'història, com la que s'ha explicat anteriorment per al joc de lluita.

En un joc d'estratègia o simulador d'algun esport de grup podem pensar en una arquitectura de classificadors a dos nivells, tal com s'explica per al cas del joc de rol anterior.

Per a acabar la introducció d'aquest apartat, es fa indispensable parlar de les diferents fases involucrades en el procés de classificació i dels temes d'eficiència computacional. La figura següent mostra en forma de rectangle els conjunts de dades i en forma d'ovals els processos involucrats.

Fases i dades d'un procés de classificació



Partim sempre del conjunt d'entrenament[4] que correspon a les dades que representen els exemples i les seves classes corresponents, recopilades prèviament. Aquestes dades són l'entrada per al procés d'entrenament, aprenentatge o construcció del model[5]. La sortida d'aquest procés serà el model de classificació.

(4) En anglès, *training set*.

(5) En anglès, *learning* o *training*.

El conjunt del test correspon a la representació dels exemples nous. Aquests poden ser, a manera d'exemple, els objectes nous en temps de joc. El procés de classificació pren com a entrada aquest conjunt i el model generat en temps d'entrenament, i dona com a sortida les prediccions dels exemples del conjunt de test.

En el camp de la programació de videojocs cal tenir molt en compte l'eficiència computacional. És molt important saber on posar els dos processos i quins algorismes utilitzar tenint en compte això. Donat un problema concret, podríem decidir incloure el procés d'entrenament en temps d'implementació del joc i el de classificació en temps de joc. Això és possible perquè són dos processos independents i sempre podem guardar d'alguna manera el model de classificació.

En les seccions següents veurem com utilitzar una llibreria d'aprenentatge automàtic que es pot utilitzar des de Unity i una sèrie d'algorismes de classificació inclosos en aquesta. En l'apartat «Anàlisi del mètode» de cadascun dels algorismes adaptarem a cada cas concret els temes computacionals esmentats anteriorment.

4.2.1. Accord.NET Framework

L'accord.NET és un sistema de llibreries per a C# que conté la implementació de molts algorismes d'aprenentatge automàtic i de gestió de les dades associades a aquests. Utilitzarem aquesta llibreria durant tot aquest apartat.

Veurem dues maneres de treballar: una directament en Visual Studio per a aprendre en temps de disseny i una altra amb Unity 3D per a aprendre en temps de joc. A continuació hi ha dos subapartats que indiquen com invocar i/o instal·lar la llibreria en aquests dos entorns i com donarem les dades de sortida en cadascun.

Accord.NET i Visual Studio

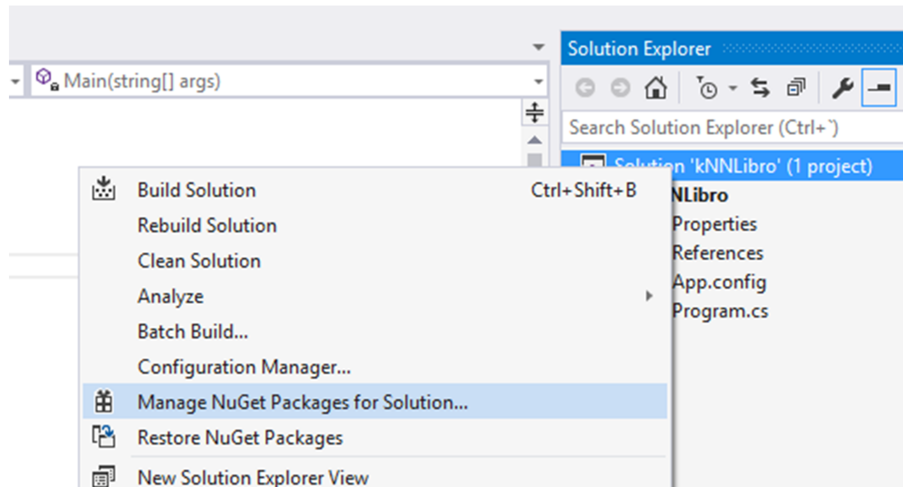
El primer que hem de tenir en compte és que aquesta llibreria s'instal·la en el pla del projecte. Així, en aquest apartat veurem com es crea un projecte nou en Visual Studio i com li instal·lem l'Accord utilitzant una eina de gestió de paquets anomenada NuGet.

En treballar directament en Visual Studio ho farem amb aplicacions de consola, que són les que s'assemblen més als *scripts* de Unity.

Tenint en compte tot això, començarem clicant l'opció File del menú, New i Project. Ens apareixerà un quadre de diàleg en el qual seleccionarem Console Aplication dins de C# i triarem la ubicació i el nom del projecte que ens interessi.

Una vegada confirmat, ens apareixerà la plantilla per a crear el programa a l'esquerra i una finestra a la dreta anomenada Solution Explorer. Per a invocar el gestor de paquets NuGet, hem de clicar amb el botó dret del ratolí el text superior de l'arbre que mostra aquesta finestra (sol portar el nom de *Solution '<nom del projecte>'*). Ens apareixerà un menú de context en el qual hem de seleccionar l'opció Manage NuGet Packages for Solution… tal com mostra la figura següent.

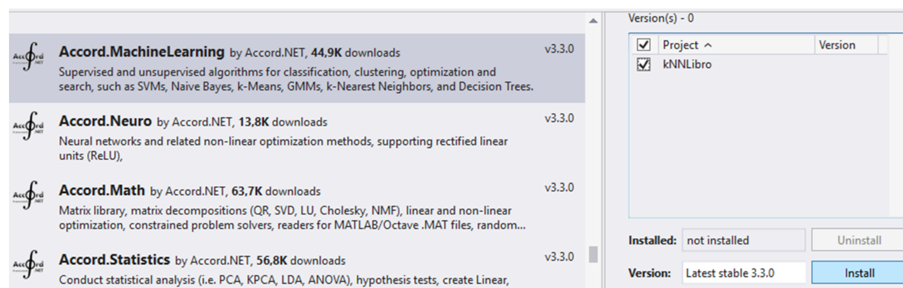
Accés al NuGet



En la finestra de l'esquerra ens apareixerà el gestor de paquets. En aquesta finestra introduïm «Accord» en el quadre de text i cliquem Browse. Per a la majoria dels exemples (si no tots) de l'apartat en tenim prou amb instal·lar el mòdul Accord.MachineLearning[6]. Seleccionem el mòdul, activem el projecte a la dreta i cliquem el botó Install tal com mostra la figura següent.

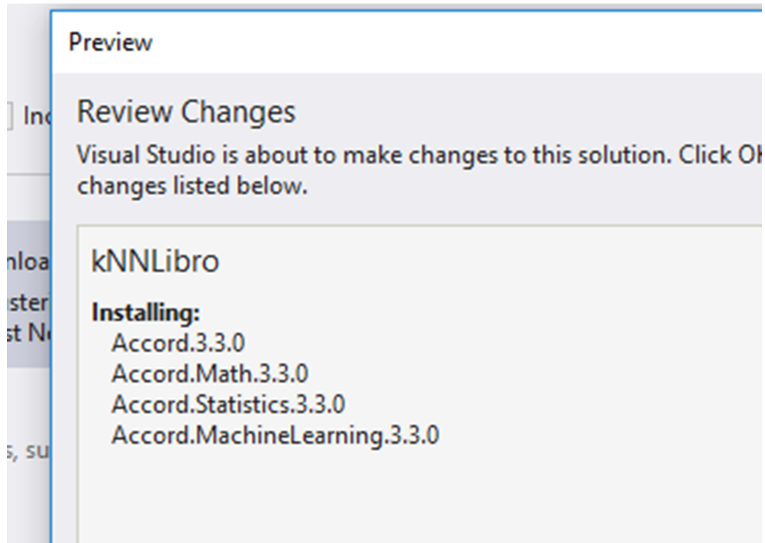
[6] A més, hi ha un mòdul anomenat Accord.MachineLearning.GPL, que no és el que necessitem.

Instal·lació d'Accord.MachineLearning



Veurem que se'ns demana confirmació per a instal·lar les dependències (vegeu la figura següent).

Dependències de l'Accord.MachineLearning



Cliquem *Ok* i acceptem la llicència que ens mostra a continuació. Una vegada fet això, podem tornar al programa i utilitzar la llibreria.

En els exemples que veurem en aquesta secció s'utilitzarà directament Visual Studio començant pel que s'acaba de descriure i es donarà la sortida dels programes amb Console.Writeln. En cas de voler executar-los en Unity, s'haurà de canviar aquesta línia per la que es descriu en l'apartat següent.

Accord.NET i Unity 3D

En aquest apartat veurem com instal·lar i invocar la llibreria Accord en un projecte de Unity i com adaptar els exemples d'aquest cas que apareixen al llarg d'aquesta secció.

Abans de començar, hem de parlar d'una característica de Unity que afecta en profunditat l'ús de llibreries externes C# o .NET. Unity està basat en una versió de C# de programari obert una mica antiga.

Observació

Fa uns anys C# no era de programari obert i disposava d'una versió que sí ho era. Unity es basa en aquesta versió antiga. Ja fa alguns anys que Microsoft va obrir el C# i per tant ara sí que ho és.

Això fa que solament es puguin utilitzar llibreries que són compilades per a la versió 3.5 de .NET. Des de la comunitat de desenvolupadors es demana que s'actualitzi la versió de C# inclosa en Unity. En teoria, quan això passi, la instal·lació de l'Accord funcionarà tal com es detalla en l'apartat anterior, invocant Visual Studio des d'un *script* de Unity i utilitzant el gestor de paquets NuGet.

A dia del desenvolupament d'aquest material això encara no és una realitat[7]. A partir d'aquest moment detallem com crear una aplicació molt simple en Unity 3D que invoqui un dels exemples (el del kNN) de les seccions següents.

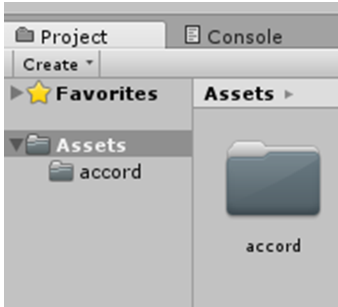
(7) Unity 3D versió 5.4.1f1 Personal.

Començarem per crear un projecte en 2D nou en Unity. La manera més simple de treballar amb Accord en Unity és copiar les llibreries de l'Accord en una carpeta situada en Assets (vegeu la figura següent). Baixarem la lli-

(8) URL: http://accord-framework.net/index.html

breria de la Web[8], la descomprimirem i copiarem tots els arxius de la carpeta Release/net35 relacionats amb Accord, Accord.Math, Accord.Statistics i Accord.MachineLearning en una carpeta de l'Assets del projecte de Unity.

Carpeta dins d'Assets



Ara hem de canviar l'API Compatibility Level. Per a fer-ho, hem de mostrar els Player Settings a l'Inspector clicant Edit - Proyect Settings - Player. Seleccionem l'opció .NET 2.0 de l'API Compatibility Level de l'apartat Optimization tal com mostra la figura següent.

API Compatibility Level

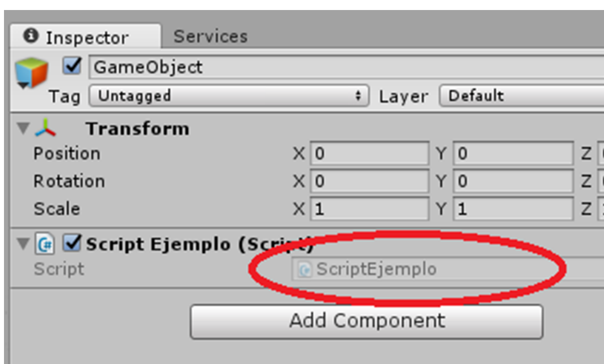
Una vegada fet això, Unity ja reconeix la llibreria i la podem invocar.

A continuació afegirem un objecte buit[9] al projecte. Una vegada fet això, li afegirem un *script*[10]. Ara podem accedir a l'*script* amb un doble clic en el seu nom, tal com indica la figura següent. Això invocarà el Visual Studio amb el codi de l'*script* obert.

(9) Finestra Hierarchy – Create – Create Empty.

(10) En la finestra de l'Inspector, amb l'objecte seleccionat, afegim l'*script* Add component.

Accés a un *script*



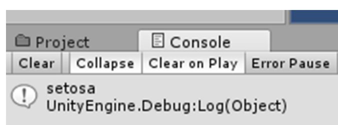
En aquest punt, cal copiar el codi que apareix a continuació:

```
using Accord.MachineLearning;
using UnityEngine;
```

```
public class ScriptEjemplo : MonoBehaviour {
void Start () {
      double[][] inputs =
      {
            new double[] {5.1, 3.5, 1.4, 0.2},
            new double[] {4.9, 3.0, 1.4, 0.2},
            new double[] {6.1, 2.9, 4.7, 1.4},
            new double[] {5.6, 2.9, 3.6, 1.3},
            new double[] {7.6, 3.0, 6.6, 2.1},
            new double[] {4.9, 2.5, 4.5, 1.7},
         };
      int[] outputs = { 0, 0, 1, 1, 2, 2 };
      // Codificació per a la sortida: 0->"setosa", 1->"versicolor", 2->"virginica"
      string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };

      // Entrenament: k=3
      int k = 3;
      var knn = new KNearestNeighbors(k, inputs, outputs);

      // Exemple de test
      double[] instance = { 4.9, 3.1, 1.5, 0.1 };

      // Classificació: resultat int
      int pred = knn.Compute(instance);

      // Escriptura del resultat en Visual Studio
      //      Console.WriteLine(codificacionOutputs[pred]);
      // Escriptura del resultat en la consola de Unity 3D
      Debug.Log(codificacionOutputs[pred]);
   }
}
```

Després de guardar, compilar i tancar el Visual Studio, estem preparats per a executar el codi de l'exemple tal com mostra la figura següent.

Execució de l'exemple Unity 3D



En el codi anterior hem d'esmentar la diferència en donar els resultats de sortida en les versions de Visual Studio. En aquest cas utilitzarem el Debug.Log, i en els de Visual Studio el Console.WriteLine, i eliminarem el Console.ReadKey() per a l'espera d'una tecla.

4.2.2. Naïve Bayes

Suposem que volem fer una aplicació per a telèfons mòbils que ajudi els afeccionats a recol·lectar bolets a destriar els bolets verinosos dels comestibles a partir de les seves propietats: forma i color del barret, i grossor i color del tronc. La taula 1 mostra exemples d'aquest tipus de dades:

Taula 1. Conjunt d'entrenament

cap-shape	***cap-color***	***gill-size***	***gill-color***	***class***
convex	*brown*	*narrow*	*black*	*poisonous*
convex	*yellow*	*broad*	*black*	*edible*
bell	*white*	*broad*	*brown*	*edible*
convex	*white*	*narrow*	*brown*	*poisonous*
convex	*yellow*	*broad*	*brown*	*edible*
bell	*white*	*broad*	*brown*	*edible*
convex	*white*	*narrow*	*pink*	*poisonous*

Font: problema «mushroom» del repositori UCI (Frank i Asunción, 2010)

El Naïve Bayes[11] és el representant més simple dels algorismes basats en probabilitats; està basat en el teorema de Bayes.

(11) L'algorisme de Naïve Bayes el descriuen Duda i Hart en la seva forma més clàssica en 1973.

L'algorisme de Naïve Bayes classifica nous exemples $x = (x_1,\ldots,x_m)$ assignant-los la classe *k*, que maximitza la probabilitat condicional de la classe donada la seqüència d'entrades de l'exemple observada. És a dir,

$$argmax_k \quad P(k|x_1, \cdots, x_m) = argmax_k \frac{P(x_1, \cdots, x_m|k)P(k)}{P(x_1, \cdots, x_m)} \approx argmax_k P(k) \prod_{i=1}^{m} P(x_i|k)$$

On $P(k)$ i $P(x_i|k)$ s'estimen a partir del conjunt d'entrenament utilitzant les freqüències relatives (estimació de la màxima versemblança[12]).

(12) En anglès, *maximum likelihood estimation.*

Exemple d'aplicació

La taula 1 mostra un exemple de conjunt d'entrenament en el qual hem de detectar si un bolet és verinós o comestible en funció de les seves propietats.

Durant el procés d'entrenament començarem per calcular $P(k)$ per a cadascuna de les classes. Així, aplicant una estimació de la màxima versemblança obtenim *P*(*poisonous*) = 3/7 = 0,43 i *P*(edible) = 4/7 = 0,57. El segon pas consisteix a calcular $P(x_i|k)$ per a cada parella atribut-valor i per a cada classe, de la mateixa manera que en el cas anterior. La taula 2 mostra els resultats d'aquest procés. L'1 de la cel·la corresponent a la fila *cap-shape: convex* i columna *poisonous* ve dels tres exemples etiquetats com a *poisonous*, que tenen el valor *convex* per a l'atribut *cap-shape*.

Taula 2. Valors de $P(x_i|k)$

atribut-valor	*poisonous*	*edible*
cap-shape: convex *cap-shape: bell*	1 0	0,5 0,5
cap-color: brown *cap-color: yellow* *cap-color: white*	0,33 0 0,67	0 0,5 0,5
gill-size: narrow *gill-size: broad*	1 0	0 1
gill-color: black *gill-color: brown* *gill-color: pink*	0,33 0,33 0,33	0,25 0,75 0

Amb això haurem acabat el procés d'entrenament. A partir d'aquí, si ens arriba un exemple nou com el que mostra la taula 3, haurem d'aplicar la fórmula $argmax_k P(k)\prod_{i=1}^{m} P(x_i|k)$ per a classificar-lo.

Taula 3. Exemple de test

cap-shape	*cap-color*	*gill-size*	*gill-color*	*class*
convex	*brown*	*narrow*	*black*	*poisonous*

Font: problema «mushroom» del repositori UCI (Frank i Asunción, 2010)

Comencem per calcular la part del valor *poisonous*. Per a això, haurem de multiplicar entre elles les probabilitats: $P(poisonous)$, $P(cap - shape{:}convex|poisonous)$, $P(cap - color{:}brown|poisonous)$, $P(gill - size{:}narrow|poisonous)$ i $P(gill - color{:}black|poisonous)$, que equival a un valor de 0,05. Fent el mateix procés per a la classe *edible*, obtenim un valor de 0. Per tant, la classe que maximitza la fórmula de les probabilitats és *poisonous*, i així el mètode classifica correctament l'exemple de test, donat aquest conjunt d'entrenament.

Ús en Accord.NET

A continuació teniu a disposició un *script* que implementa l'exemple anterior utilitzant la llibreria Accord:

```
using Accord;
using Accord.MachineLearning.Bayes;
using Accord.Math;
using Accord.Statistics.Filters;
using System;
using System.Data;

namespace NaiveBayesLibro
{
   class Program
   {
      static void Main(string[] args)
      {
         // Creació del conjunt de dades
         // Càrrega de dades en la variable data
```

```
            DataTable data = new DataTable("Mushroom Data");
            data.Columns.Add("cap-shape", "cap-color", "gill-size", "gill-color", "class");
            data.Rows.Add("convex", "brown", "narrow", "black", "poisonous");
            data.Rows.Add("convex", "yellow", "broad", "black", "edible");
            data.Rows.Add("bell", "white", "broad", "brown", "edible");
            data.Rows.Add("convex", "white", "narrow", "brown", "poisonous");
            data.Rows.Add("convex", "yellow", "broad", "brown", "edible");
            data.Rows.Add("bell", "white", "broad", "brown", "edible");
            data.Rows.Add("convex", "white", "narrow", "pink", "poisonous");

            // Codificació del conjunt de dades
            // Generació de les variables inputs i outputs a partir de data
            Codification codebook = new Codification(data, "cap-shape", "cap-color", "gill-size",
            "gill-color", "class");
            DataTable symbols = codebook.Apply(data);
           int[][] inputs = symbols.ToArray<int>("cap-shape", "cap-color", "gill-size", "gill-color");
            int[] outputs = symbols.ToArray<int>("class");

            // Entrenament
            // Construcció del model en la variable nb
            // Es poden consultar les probabilitats del model
            var learner = new NaiveBayesLearning();
            NaiveBayes nb = learner.Learn(inputs, outputs);

            // Creació de l'exemple de test
            // Generació i codificació de l'exemple de test en instance
            int[] instance = codebook.Translate("convex", "brown", "narrow", "black");
            // Classificació
            // El resultat és numèric
            int pred = nb.Decide(instance);

            // Decodificació de la predicció
            string result = codebook.Translate("class", pred);
            // Escriptura del resultat
            Console.WriteLine("Predicció: {0}", result);

            // Espera d'una tecla
            Console.ReadKey();
        }
    }
}
```

Anàlisi del mètode

Un dels problemes del Naïve Bayes, que ha estat esmentat freqüentment en la literatura, és que l'algorisme assumeix la independència dels diferents atributs que representen un exemple. Així, és poc probable que el color del tronc d'un bolet no estigui relacionat amb el color del seu barret.

Una altra característica destacable és que quan el conjunt d'entrenament no està equilibrat[13], tendeix a classificar els exemples cap a la classe que té més exemples dins del conjunt d'entrenament.

(13) Un conjunt d'entrenament està equilibrat quan té el mateix nombre d'exemples de cadascuna de les classes que conté.

Dos dels avantatges que presenta el mètode són la seva simplicitat i l'eficiència computacional. Les dues fases de l'aprenentatge són ràpides. La fase d'entrenament consisteix a carregar i/o preparar les dades i a estimar les probabilitats $P(k)$ i $P(x_i|k)$. La fase de test o classificació consisteix a preparar els exemples de test i a multiplicar les probabilitats pertinents. Cal tenir en compte on fem cadascuna de les fases, sobretot la d'entrenament. Les dues fases poden estar separades, ja que disposem del model.

No obstant això, aquest mètode ha estat molt utilitzat històricament malgrat els seus inconvenients i s'han obtingut bons resultats per a molts conjunts de dades. Això passa quan el conjunt d'entrenament representa bé les distribucions de probabilitat del problema.

Fins ara hem parlat exclusivament d'atributs nominals. En cas de tenir un problema representat amb algun atribut continu, com els numèrics, és necessari algun tipus de procés. Hi ha diverses maneres de fer-ho: una consisteix a categoritzar-los dividint el continu en intervals; una altra consisteix a assumir que els valors de cada classe segueixen una distribució gaussiana i aplicar la fórmula

$$P(x = v|k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{(v-\mu_k)^2}{2\sigma_k^2}\right)$$

on x correspon a l'atribut, v al seu valor, k a la classe, μ_k a la mitjana de valors de la classe k i σ_k a la seva desviació estàndard. Aquesta modificació de l'algorisme rep el nom de Gaussian Naïve Bayes i és d'ús molt comú. Lamentablement, no tenim aquesta implementació en la llibreria Accord.net.

4.2.3. kNN

Suposem que volem fer una aplicació per a un magatzem de flors, on arriben diàriament milers de productes. Disposem d'un sistema làser que ens subministra una sèrie de mesures sobre les flors i ens demanen que el sistema les classifiqui automàticament per transportar-les mitjançant un robot a les dife-

rents prestatgeries del magatzem. Les mesures que envia el sistema làser són la longitud i amplària del sèpal i el pètal de cada flor. La taula 4 mostra exemples d'aquest tipus de dades.

Taula 4. Conjunt d'entrenament

sepal-length	***sepal-width***	***petal-length***	***petal-width***	***class***
5,1	3,5	1,4	0,2	*setosa*
4,9	3,0	1,4	0,2	*setosa*
6,1	2,9	4,7	1,4	*versicolor*
5,6	2,9	3,6	1,3	*versicolor*
7,6	3,0	6,6	2,1	*de Virgínia*
4,9	2,5	4,5	1,7	*de Virgínia*

Font: problema «iris» del repositori UCI (Frank i Asunción, 2010)

En kNN (*k* veïns més propers[14]) la classificació de nous exemples es fa buscant el conjunt dels *k* exemples més propers d'entre un conjunt d'exemples etiquetats i guardats prèviament, i seleccionant la classe més freqüent d'entre les seves etiquetes. La generalització es posposa fins al moment de classificar nous exemples[15].

(14) En anglès, *k nearest neighbours*.

(15) Aquesta raó és per la qual a vegades és anomenat aprenentatge mandrós (en anglès, *lazy learning*).

Una part molt important d'aquest mètode és la definició de la mesura de distància (o similitud) apropiada per al problema a tractar. Aquesta hauria de tenir en compte la importància relativa de cada atribut i ser eficient computacionalment. El tipus de combinació per a triar el resultat d'entre els *k* exemples més propers i el valor de la mateixa *k* també són qüestions per decidir d'entre diverses alternatives.

Aquest algorisme guarda en memòria, en la seva forma més simple, tots els exemples durant el procés d'entrenament, i la classificació de nous exemples es basa en les classes dels *k* exemples més propers[16]. Per a obtenir el conjunt dels *k* veïns més propers, es calcula la distància entre l'exemple per classificar $x = (x_1, \cdots, x_m)$ i tots els exemples guardats $x_i = (x_{i_1}, \cdots, x_{i_m})$. Una de les distàncies més utilitzades és l'euclidiana:

(16) Per aquesta raó es coneix també com a basat en memòria, en exemples, en instàncies o en casos.

$$de(x, x_i) = \sqrt{\sum_{j=1}^{m} \left(x_j - x_{i_j}\right)^2}$$

Exemple d'aplicació

La taula 4 mostra un conjunt d'entrenament en el qual hem de classificar flors a partir de les seves propietats. En aquest exemple, aplicarem el kNN per a valors de *k* d'1 i 3 utilitzant com a mesura de distància l'euclidiana.

El procés d'entrenament consisteix a guardar les dades; no hem de fer res. A partir d'aquí, si ens arriba un exemple nou com el que mostra la taula 5, hem de calcular les distàncies

entre el nou exemple i tots els del conjunt d'entrenament. La taula 6 mostra aquestes distàncies.

Taula 5. Exemple de test

sepal-length	***sepal-width***	***petal-length***	***petal-width***	***class***
4,9	3,1	1,5	0,1	*setosa*

Font: problema «iris» del repositori UCI (Frank i Asunción, 2010)

Taula 6. Distàncies

0,5	0,2	3,7	2,5	6,1	3,5

Per a l'1NN triem la classe del més proper que coincideix amb el segon exemple (distància 0,2) que té per classe *setosa*. Per al 3NN triem els tres exemples més propers: primer, segon i quart; amb distàncies respectives 0,5, 0,2 i 2,5. Les seves classes corresponen a *setosa*, *setosa* i *versicolor*. En aquest cas assignarem també a *setosa* per ser la classe més freqüent. En els dos casos el resultat és correcte.

Ús en Accord.NET

A continuació teniu a disposició un *script* que implementa l'exemple anterior utilitzant la llibreria Accord:

```
using System;
using Accord.MachineLearning;

namespace kNNLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Conjunt d'entrenament: inputs i outputs
      double[][] inputs =
      {
      new double[] {5.1, 3.5, 1.4, 0.2},
      new double[] {4.9, 3.0, 1.4, 0.2},
      new double[] {6.1, 2.9, 4.7, 1.4},
      new double[] {5.6, 2.9, 3.6, 1.3},
      new double[] {7.6, 3.0, 6.6, 2.1},
      new double[] {4.9, 2.5, 4.5, 1.7},
      };
      int[] outputs = { 0, 0, 1, 1, 2, 2 };
      // Codificació per a la sortida: 0->"setosa", 1->"versicolor", 2->"virginica"
      string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };

      // Entrenament: k=3
      int k = 3;
      var knn = new KNearestNeighbors(k, inputs, outputs);

      // Exemple de test
```

```
            double[] instance = { 4.9, 3.1, 1.5, 0.1 };

            // Classificació: resultat int
            int pred = knn.Compute(instance);

            // Escriptura del resultat
            Console.WriteLine(codificacionOutputs[pred]);

            // Espera d'una tecla
            Console.ReadKey();
        }
        }
    }
```

Anàlisi del mètode

Un aspecte que cal tenir en compte és l'eficiència computacional: l'algorisme fa tots els càlculs en el procés de classificació. Així, tot i ser un mètode ràpid globalment, hem de tenir en compte que el procés de classificació no ho és. Això pot arribar a ser crític per a jocs que necessitin una resposta ràpida.

En el moment en què vulguem aplicar aquest algorisme a un problema, la primera decisió que hem de prendre és la mesura de distància, i la segona, el valor de *k*. Se sol triar un nombre imparell o primer per a minimitzar la possibilitat d'empats en les votacions. La qüestió següent que cal decidir és el tractament dels empats quan la *k* és superior a 1. Alguns heurístics possibles són: no donar predicció en cas d'empat, donar la classe més freqüent en el conjunt d'entrenament d'entre les classes seleccionades per a votar, etc. Se sol triar l'heurístic en funció del problema que hàgim de tractar.

Les distàncies més utilitzades són l'euclidiana i la de Hamming, en funció del tipus d'atributs que tinguem. La primera s'utilitza majoritàriament per a atributs numèrics, i la segona, per a atributs nominals o binaris. La distància de Hamming és inclosa en la llibreria Accord.NET.

Un dels grans avantatges d'aquest mètode és la conservació d'excepcions en el procés de generalització.

4.2.4. Classificador lineal

Molt semblant a la de l'algorisme anterior, una altra manera d'abordar el problema de classificar flors es basa en l'ús de centres de massa o centroides. El model de classificació d'aquest algorisme consta d'un centroide, que representa cadascuna de les classes que apareixen en el conjunt d'entrenament[17]. El valor de cada atribut del centroide es calcula com la mitjana del valor del mateix atribut de tots els exemples del conjunt d'entrenament que pertanyen a

(17) Aquest mètode també rep el nom de mètode basat en centroides o en centres de masses.

la seva classe. La fase de classificació consisteix a aplicar l'1NN amb distància euclidiana, seleccionant com a conjunt d'entrenament els centroides calculats prèviament.

Exemple d'aplicació

Com a exemple, aplicarem l'algorisme en els exemples de la taula 4. El procés d'entrenament consisteix a calcular els centroides. La taula 7 mostra el resultat. A manera d'exemple, el 5 de l'atribut *sepal-length* del centroide *setosa* s'obté de la mitjana entre 4.9 i 5.1 de la taula 4.

Taula 7. Centroides

sepal-length	***sepal-width***	***petal-length***	***petal-width***	***class***
5	3,25	1,4	0,2	*setosa*
5,85	2,9	4,15	1,35	*versicolor*
6,25	2,75	5,55	1,9	*de Virgínia*

A partir d'aquí, si ens arriba un exemple nou com el que mostra la taula 5, hem de calcular les distàncies entre el nou exemple i tots els centroides per a l'1NN. La taula 8 mostra aquestes distàncies. En triar el més proper (distància 0,2), que té per classe *setosa*, veiem que l'exemple queda ben classificat.

Taula 8. Distàncies

0,2	3,1	4,6

Ús en Accord.NET

A continuació teniu a disposició un *script* que implementa l'exemple anterior utilitzant la llibreria Accord:

```
using System;
using Accord.MachineLearning;

namespace CentroidesLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Conjunt d'entrenament: inputs i outputs
      double[][] inputs =
      {
      new double[] {5.1, 3.5, 1.4, 0.2},
      new double[] {4.9, 3.0, 1.4, 0.2},
      new double[] {6.1, 2.9, 4.7, 1.4},
      new double[] {5.6, 2.9, 3.6, 1.3},
      new double[] {7.6, 3.0, 6.6, 2.1},
      new double[] {4.9, 2.5, 4.5, 1.7},
      };
```

```
        int[] outputs = { 0, 0, 1, 1, 2, 2 };
        // Codificació per a la sortida: 0->"setosa", 1->"versicolor", 2->"virginica"
        string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };

        // Entrenament
        var learner = new MinimumMeanDistanceClassifier(inputs, outputs);

        // Exemple de test
        double[] instance = { 4.9, 3.1, 1.5, 0.1 };

        // Classificació: resultat int
        int pred = learner.Decide(instance);

        // Espera d'una tecla
        Console.WriteLine(codificacionOutputs[pred]);

        // Espera d'una tecla
        Console.ReadKey();
      }
    }
  }
```

Anàlisi del mètode

Els principals avantatges d'aquest mètode són la simplicitat i eficiència computacional tant en temps com en espai. Els inconvenients són que solament serveix per a atributs numèrics i que no serveix per a abordar problemes molt complexos.

4.2.5. Arbres de decisió

Abans de començar amb aquest algorisme, cal fer una petita ressenya explicant la diferència entre l'arbre de decisió explicat en l'apartat de presa de decisions (apartat 3) i l'actual. Els dos arbres són bàsicament el mateix: els dos guarden un model per a la presa de decisions. Es diferencien en qui ha generat l'estructura. Els arbres dels quals es parla en l'apartat de presa de decisions els han generat generalment els implementadors o dissenyadors del joc. En el cas de l'aprenentatge automàtic, l'arbre es genera automàticament a partir de les dades i constitueix el model per a un classificador.

Una vegada feta aquesta anotació, suposem que volem fer una aplicació per a telèfons mòbils que ajudi els afeccionats a la recollida de bolets a destriar els bolets verinosos dels comestibles a partir de les seves propietats: forma i color del barret, i color del tronc.

Un arbre de decisió és una manera de representar regles de classificació inherents a les dades, amb una estructura en arbre *n*-ari que particiona les dades de manera recursiva. Cada branca d'un arbre de decisió representa una regla que decideix entre una conjunció de valors d'un atribut bàsic (nodes interns) o fa una predicció de la classe (nodes terminals).

L'algorisme dels arbres de decisió bàsic és pensat per a treballar amb atributs nominals. El conjunt d'entrenament queda definit per $S = \{(x_1,y_1),\ldots,(x_n,y_n)\}$, on cada exemple x correspon a $x_i = \left(x_{i_1}, \cdots, x_{i_m}\right)$, on m guarda el nombre d'atributs; i $A = \{a_1,\ldots,a_m\}$ el conjunt d'atributs, on $dom(a_j)$ correspon al conjunt de tots els possibles valors de l'atribut a_j, i per a qualsevol valor d'un exemple d'entrenament $x_{i_j} \in dom\left(a_j\right)$.

El procés de construcció de l'arbre és un procés iteratiu, i, en cada iteració, s'hi selecciona l'atribut que particiona millor el conjunt d'entrenament. Per a fer aquest procés, hem de mirar la bondat de les particions que genera cadascun dels atributs i, en un segon pas, seleccionar el millor. La partició de l'atribut a_j genera $|dom(a_j)|$ conjunts, que corresponen al nombre d'elements del conjunt. Hi ha diverses mesures per a mirar la bondat de la partició. Una mesura bàsica consisteix a assignar a cada conjunt de la partició la classe majoritària d'aquest, comptar quants queden ben classificats i dividir-ho pel nombre d'exemples. Una vegada calculades les bondats de tots els atributs, triem el millor.

Cada conjunt de la millor partició passarà a ser un nou node de l'arbre. A aquest node s'arribarà mitjançant una regla del tipus atribut = valor. Si tots els exemples del conjunt han quedat ben classificats, el convertim en node terminal amb la classe dels exemples. En cas contrari, el convertirem en node intern i apliquem una nova iteració al conjunt («reduït») eliminant l'atribut que ha generat la partició. En cas de no quedar atributs, el convertiríem en node terminal assignant la classe majoritària.

Per a fer el test, explorem l'arbre en funció dels valors dels atributs de l'exemple de test i les regles de l'arbre fins a arribar al node terminal, i donem com a predicció la classe del node terminal al qual arribem.

Exemple d'aplicació

La taula 9 és una simplificació de la taula 4. Per a construir un arbre de decisió a partir d'aquest conjunt, hem de calcular la bondat de les particions dels tres atributs: *cap-shape*, *cap-color* i *gill-color*. L'atribut *cap-shape* ens genera una partició amb dos conjunts: un per al valor *convex* i un altre per a *bell*. La classe majoritària per al conjunt de *convex* és *poisonous* i la de *bell* és *edible*; la seva bondat és *bondat*(*cap-shape*) = (3+2)/7 = 0,71.

El 7 correspon al nombre total d'exemples. Si fixem el valor *convex* ens queden les classes *{poisonous, edible, poisonous, edible, poisonous}* per a *cap-shape*; la majoritària correspon a *poisonous*, que hi és tres vegades. Les dues classes de *bell* són *edible*, que hi és dues vegades.

Si fem el mateix procés per a la resta dels atributs, obtenim: *bondat*(*cap-color*) = (1+2+2)/7 = 0,71 i *bondat*(*cap-color*) = (1+3+1)/7 = 0,71.

Taula 9. Conjunt d'entrenament

cap-shape	***cap-color***	***gill-color***	***class***
convex	*brown*	*black*	*poisonous*
convex	*yellow*	*black*	*edible*
bell	*white*	*brown*	*edible*
convex	*white*	*brown*	*poisonous*
convex	*yellow*	*brown*	*edible*
bell	*white*	*brown*	*edible*
convex	*white*	*pink*	*poisonous*

Font: problema «mushroom» del repositori UCI (Frank i Asunción, 2010)

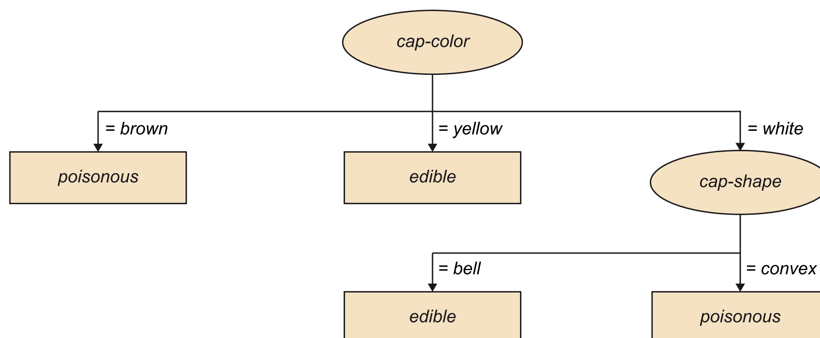
El pas següent consisteix a seleccionar el millor atribut. Hem obtingut un empat entre els tres atributs, així que en podem triar qualsevol; escollim *cap-color*. Els nodes generats per al conjunt de *brown* i *yellow* són terminals i els assignem les classes *poisonous* i *edible* respectivament. Això es deu al fet que els dos conjunts obtenen una bondat d'1. El node de *white* el tornem a fer iterar a partir del conjunt que mostra la taula 10. Aquest conjunt l'obtenim d'eliminar l'atribut *cap-color* dels exemples que tenen el valor *white* per a l'atribut *cap-color*. La bondat de les noves particions és *bondat*(*cap-shape*) = (2+2)/4 = 1 i *bondat*(*gill-color*) = (2+1)/4 = 0,75. El millor atribut és *cap-shape*, que genera dos nodes terminals amb les classes *edible* per a *bell* i *poisonous per a convex*.

Taula 10. Conjunt en la segona iteració

cap-shape	***gill-color***	***class***
bell	*brown*	*edible*
convex	*brown*	*poisonous*
bell	*brown*	*edible*
convex	*pink*	*poisonous*

La figura següent mostra l'arbre construït en aquest procés.

Arbre de decisió



Per a etiquetar un exemple de test com el que mostra la taula 11, hem de recórrer l'arbre, partint de l'arrel, triant les branques corresponents als valors dels atributs dels exemples de test. Per a aquest cas, mirem el valor de l'atribut *cap-color* i baixem per la branca que correspon al valor *brown*. Arribem a un node terminal amb classe *poisonous*, i així l'assignarem a l'exemple de test com a predicció. Aquesta predicció és correcta.

Taula 11. Exemple de test

cap-shape	*cap-color*	*gill-color*	*class*
convex	*brown*	*black*	*poisonous*

Ús en Accord.NET

A continuació teniu a disposició un *script* que implementa l'exemple anterior utilitzant la llibreria Accord:

```
using System;
using System.Data;
using Accord;
using Accord.Math;
using Accord.Statistics.Filters;
using Accord.MachineLearning.DecisionTrees;
using Accord.MachineLearning.DecisionTrees.Learning;

namespace DTsLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Creació del conjunt de dades
      // Càrrega de dades en la variable data
      DataTable data = new DataTable("Mushroom Data");
      data.Columns.Add("cap-shape", "cap-color", "gill-color", "class");
      data.Rows.Add("convex", "brown", "black", "poisonous");
      data.Rows.Add("convex", "yellow", "black", "edible");
      data.Rows.Add("bell", "white", "brown", "edible");
      data.Rows.Add("convex", "white", "brown", "poisonous");
      data.Rows.Add("convex", "yellow", "brown", "edible");
      data.Rows.Add("bell", "white", "brown", "edible");
      data.Rows.Add("convex", "white", "pink", "poisonous");

      // Codificació del conjunt de dades
      // Generació de les variables inputs i outputs a partir de data
     Codification codebook = new Codification(data, "cap-shape", "cap-color", "gill-color", "class");
      DataTable symbols = codebook.Apply(data);
      double[][] inputs = symbols.ToArray<double>("cap-shape", "cap-color", "gill-color");
      int[] outputs = symbols.ToArray<int>("class");

      // Entrenament
      string[] inputColumns = { "cap-shape", "cap-color", "gill-color" };
      var attributes = DecisionVariable.FromCodebook(codebook, inputColumns);
      DecisionTree tree = new DecisionTree(attributes, 2);
```

```
        C45Learning c45 = new C45Learning(tree);
        c45.Learn(inputs, outputs);

        // Creació de l'exemple de test
        // Generació i codificació de l'exemple de test en instance
        int[] instance = codebook.Translate("convex", "brown", "black");
        // Classificació
        // El resultat és numèric
        int pred = tree.Decide(instance);
        // Decodificació de la predicció
        string result = codebook.Translate("class", pred);
        // Escriptura del resultat
        Console.WriteLine("Predicció: {0}", result);

        // Espera d'una tecla
        Console.ReadKey();
    }
    }
}
```

Tractament d'atributs numèrics

Ara suposem que volem fer una aplicació per a un magatzem de flors on arriben milers de productes diàriament. Disposem d'un sistema làser que ens subministra la longitud del sèpal de les flors, i ens demanen que el sistema les classifiqui automàticament per transportar-les mitjançant un robot a les diferents prestatgeries del magatzem.

L'algorisme dels arbres de decisió va ser dissenyat per tractar amb atributs nominals, però hi ha alternatives per a tractar atributs numèrics. El més comú es basa en la utilització de punts de tall, que són un punt que divideix el conjunt de valors d'un atribut en dos (els menors i els majors del punt de tall).

Per a calcular el millor punt de tall d'un atribut numèric, s'ordenen els valors i s'eliminen els elements repetits. Es calculen els possibles punts de tall com la mitjana de cada dos valors consecutius. Com a últim pas, es calcula el millor com el que té millor bondat.

Hem de tenir en compte que el tractament d'atributs numèrics genera arbres (i decisions) binaris. Això implica que en els següents nivells de l'arbre haurem de tornar a processar l'atribut numèric que acabem de seleccionar, a diferència dels atributs nominals.

Exemple del càlcul del millor punt de tall d'un atribut numèric

A continuació s'exposa el càlcul del millor punt de tall per a l'atribut *sepal-length* del conjunt de dades que mostra la taula 4. La taula 12 mostra aquests càlculs: les columnes 1 i 2 mostren la classe i l'atribut ordenats per l'atribut, la 3 mostra els valors sense els

repetits; la 4, els punts de tall com la mitjana de cada dos valors consecutius, i l'última, les bondats dels punts de tall. El millor punt de tal és 5,35, amb una bondat de 66,7%.

Taula 12. Càlculs per al millor punt de tall

class	***se-pal-length***	**sense repetits**	**punts de tall**	**bondat**
setosa	4,9			
de Virgínia	4,9	4,9	5	(1 + 2) / 6 = 0,5
setosa	5,1	5,1	5,35	(2 + 2) / 6 = 0,67
versicolor	5,6	5,6	5,85	(2 + 1) / 6 = 0,5
versicolor	6,1	6,1	6,85	(2 + 1) / 6 = 0,5
de Virgínia	7,6	7,6		

Ús en Accord.NET

A continuació teniu a disposició un *script* que implementa l'exemple anterior utilitzant la llibreria Accord:

```
using Accord.MachineLearning.DecisionTrees;
using Accord.MachineLearning.DecisionTrees.Learning;
using System;

namespace DTsIrisLibro
{
   class Program
   {
   static void Main(string[] args)
   {
      // Conjunt d'entrenament: inputs i outputs
      double[][] inputs =
      {
      new double[] {5.1, 3.5, 1.4, 0.2},
      new double[] {4.9, 3.0, 1.4, 0.2},
      new double[] {6.1, 2.9, 4.7, 1.4},
      new double[] {5.6, 2.9, 3.6, 1.3},
      new double[] {7.6, 3.0, 6.6, 2.1},
      new double[] {4.9, 2.5, 4.5, 1.7},
      };
      int[] outputs = { 0, 0, 1, 1, 2, 2 };
      // Codificació per a la sortida: 0->"setosa", 1->"versicolor", 2->"virginica"
      string[] codificacionOutputs = { "setosa", "versicolor", "virginica" };

      // Entrenament
      // nom dels atributs
      DecisionVariable[] features =
         {
```

```
        new DecisionVariable("sepal length", DecisionVariableKind.Continuous),
        new DecisionVariable("sepal width", DecisionVariableKind.Continuous),
        new DecisionVariable("petal length", DecisionVariableKind.Continuous),
        new DecisionVariable("petal width", DecisionVariableKind.Continuous),
        };

        // Creem un arbre per a 3 classes
        var tree = new DecisionTree(inputs: features, classes: 3);

        // Fem servir el C4.5 per a l'entrenament
        var teacher = new C45Learning(tree);

        // Induïm l'arbre
        teacher.Learn(inputs, outputs);

        // Exemple de test
        double[] instance = { 4.9, 3.1, 1.5, 0.1 };

        // Classificació
        int pred = tree.Decide(instance);
        Console.WriteLine(codificacionOutputs[pred]);

        Console.ReadKey();
    }
    }
}
```

Anàlisi del mètode

Aquest mètode és el menys eficient computacionalment dels que hem vist en aquesta secció, però té el gran avantatge de facilitar la interpretació del model d'aprenentatge. Un arbre de decisió ens dona la informació clara de la presa de decisions i de la importància dels diferents atributs involucrats. L'arbre de decisió generat es pot convertir en un conjunt de regles que es poden implementar amb estructures alternatives.

Dos dels grans inconvenients que planteja són l'elevada fragmentació de les dades en presència d'atributs amb molts valors i l'elevat cost computacional que això implica. Això provoca que no sigui un mètode molt adequat per a problemes amb grans espais d'atributs.

Un altre inconvenient que hem de tenir en compte és que els nodes terminals corresponents a regles que donen cobertura a pocs exemples d'entrenament no produeixen estimacions fiables de les classes. Tendeixen a sobreentrenar el conjunt d'entrenament[18]. Per a suavitzar aquest efecte, es pot utilitzar alguna tècnica de poda[19].

(18) Aquest efecte es coneix en anglès com a *overfitting*.

(19) En anglès, *prunning*.

4.2.6. Algorisme ID3

L'algorisme ID3 és una modificació dels arbres de decisió que utilitza el guany d'informació[20] com a mesura de bondat per a analitzar els atributs. El guany d'informació és un concepte que està basat en l'entropia. L'ús de l'entropia tendeix a penalitzar més els conjunts barrejats.

(20) En anglès, *information gain*.

L'entropia és una mesura de la teoria de la informació que quantifica el desordre. Així, donat un conjunt S, la seva fórmula ve donada per

$$H(S) = -\sum_{y \in I} p(y) log_2(p(y))$$

on $p(y)$ és la proporció d'exemples de S que pertany a la classe y. L'entropia serà mínima (0) quan en S hi ha una sola classe, i serà màxima (1) quan el conjunt és totalment aleatori.

A manera d'exemple, l'entropia del conjunt que mostra la taula 9 és donada per

$$H(S) = -\frac{3}{7} log_2 \frac{3}{7} - \frac{4}{7} log_2 \frac{4}{7} = 0{,}985$$

La fórmula del guany d'informació per al conjunt S i l'atribut queda així:

$$G(S, a_j) = H(S) - \sum_{v \in a_j} p(v) H(S_v)$$

on $p(v)$ és la proporció d'exemples de S que tenen el valor v per a l'atribut a_j, i S_v és el subconjunt de S dels exemples que prenen el valor v per a l'atribut a_j.

A manera d'exemple, el guany d'informació del conjunt que mostra la taula 9 i l'atribut *cap-shape* és donat per

$$G(S, cs) = H(S) - \frac{5}{7} H(cs = convex) - \frac{2}{7} H(cs = bell) = 0{,}292$$

on *cs* és *cap-shape*,

$$H(cs = convex) = -\frac{3}{5} log_2 \frac{3}{5} - \frac{2}{5} log_2 \frac{2}{5} = 0{,}971$$

i

$$H(cs=bell)=-\frac{0}{2}log_2\frac{0}{2}-\frac{2}{2}log_2\frac{2}{2}=0$$

Poda dels arbres de decisió

L'objectiu de la poda dels arbres de decisió és obtenir arbres que en les fulles no tinguin regles que afectin pocs exemples del conjunt d'entrenament. És desitjable no forçar la construcció de l'arbre per a evitar el sobreentrenament[21].

(21) En anglès, *overfitting*.

La primera aproximació per a abordar la solució d'aquest problema consisteix a establir un llindar de reducció de l'entropia. És a dir, pararem una branca quan la disminució d'entropia no superi aquest llindar. Aquesta aproximació planteja el problema que pot ser que no es redueixi en un cert pas però sí en el següent.

Una altra aproximació, que soluciona el problema de l'anterior i que se sol utilitzar, consisteix a construir tot l'arbre i, en una segona fase, eliminar els nodes superflus (poda de l'arbre[22]). Per a fer aquest procés de poda, recorrem l'arbre de manera ascendent (de les fulles a l'arrel) i anem mirant si cada parell de nodes incrementaria l'entropia per sobre d'un cert llindar si els ajuntéssim. Si això passa, desfem la partició.

(22) En anglès, *prunning*.

Ús en Accord.NET

A continuació teniu a la vostra disposició un *script* que implementa l'exemple anterior utilitzant la llibreria Accord:

```
using Accord;
using Accord.MachineLearning.DecisionTrees;
using Accord.MachineLearning.DecisionTrees.Learning;
using Accord.Math;
using Accord.Statistics.Filters;
using System;
using System.Data;

namespace ID3Libro
{
   class Program
   {
      static void Main(string[] args)
      {
         // Creació del conjunt de dades
         // Càrrega de dades en la variable data
         DataTable data = new DataTable("Mushroom Data");
         data.Columns.Add("cap-shape", "cap-color", "gill-color", "class");
```

```
            data.Rows.Add("convex", "brown", "black", "poisonous");
            data.Rows.Add("convex", "yellow", "black", "edible");
            data.Rows.Add("bell", "white", "brown", "edible");
            data.Rows.Add("convex", "white", "brown", "poisonous");
            data.Rows.Add("convex", "yellow", "brown", "edible");
            data.Rows.Add("bell", "white", "brown", "edible");
            data.Rows.Add("convex", "white", "pink", "poisonous");

            // Codificació del conjunt de dades
            // Generació de les variables inputs i outputs a partir de data
            Codification codebook = new Codification(data, "cap-shape", "cap-color", "gill-color",
            "class");
            DataTable symbols = codebook.Apply(data);
            int[][] inputs = symbols.ToArray<int>("cap-shape", "cap-color", "gill-color");
            int[] outputs = symbols.ToArray<int>("class");

            // Entrenament
            string[] inputColumns = { "cap-shape", "cap-color", "gill-color" };
            var attributes = DecisionVariable.FromCodebook(codebook, inputColumns);
            DecisionTree tree = new DecisionTree(attributes, 2);
            ID3Learning id3learning = new ID3Learning(tree);
            id3learning.Learn(inputs, outputs);

            // Creació de l'exemple de test
            // Generació i codificació de l'exemple de test en instance
            int[] instance = codebook.Translate("convex", "brown", "black");
            // Classificació
            // El resultat és numèric
            int pred = tree.Decide(instance);
            // Decodificació de la predicció
            string result = codebook.Translate("class", pred);
            // Escriptura del resultat
            Console.WriteLine("Predicció: {0}", result);

            // Espera d'una tecla
            Console.ReadKey();
        }
    }
}
```

4.3. Ajustament de paràmetres

Imaginem que hem creat un magnífic joc d'estratègia, o de tipus *tower defense*, o qualsevol altre en el qual calgui equilibrar diferents atributs de les unitats (poder d'atac, punts de vida, etc.). Abans de comercialitzar-lo, és fonamental ajustar molt bé aquests paràmetres per a aconseguir que el joc tingui el nivell de dificultat just per a resultar entretingut, i també per a aconseguir que tots els

tipus d'unitats resultin útils, és a dir, que no hi hagi un tipus d'unitat clarament millor que les altres, ja que això acabarà fent que ningú no usi les altres unitats i el joc sigui molt més avorrit del que preteníem.

Ens trobem davant un problema d'**ajustament de paràmetres**, en el qual hi ha un conjunt de paràmetres interrelacionats i cal trobar una combinació adequada per als objectius desitjats, en aquest cas ajustar la dificultat i varietat del joc. El problema és que, en general, no es poden ajustar d'un en un, ja que com que un modifica el valor de tots els altres s'ha de reajustar perquè el conjunt funcioni bé. Per exemple, si a un tanc li augmentem l'atac pot ser que sigui necessari abaixar-li la defensa, ja que en cas contrari serà una unitat massa poderosa i desequilibrarà completament el joc.

> Per a aconseguir aquest bon ajustament de paràmetres, podríem pensar a provar totes les combinacions possibles i quedar-nos amb la millor, però de seguida veurem que això és impossible: imaginem un joc amb deu tipus d'unitats, amb tres atributs cadascuna (atac, defensa i vida) i en la qual aquests atributs poden prendre un valor entre 1 i 20. Hem d'ajustar 3 x 10 = 30 paràmetres, i com que cadascun té 20 valors possibles, en total hi ha 20^{30} combinacions de paràmetres. Evidentment, no és molt pràctic provar-les totes.

Maledicció de la dimensionalitat

Es diu maledicció de la dimensionalitat (*curse of dimensionality*) el fet que cada paràmetre que s'afegeix a un problema augmenta l'espai de solucions possibles en una dimensió, i així el creixement del nombre de solucions possibles és exponencial respecte al nombre de paràmetres.

Per a afrontar aquest tipus de situacions, hi ha els algorismes d'**optimització**, en els quals s'utilitza un mètode que permet trobar una solució «raonablement» bona a un problema en un temps delimitat. La clau és que, com que no és possible provar totes les solucions, s'intenta trobar la millor solució possible en el temps disponible per a dur a terme l'optimització. D'aquí ve la denominació «raonablement», ja que en general no es pot assegurar que la solució trobada sigui la millor possible (caldria provar-les totes), però s'espera que almenys sigui millor que una solució qualsevol generada a l'atzar.

Observació

L'ajustament dels paràmetres del joc és una activitat que es duu a terme durant el seu desenvolupament, no s'executa en l'ordinador dels usuaris mentre juguen; diem que és una execució fora de línia. Encara que s'usin algorismes d'optimització molt eficients, el seu temps d'execució sol ser elevat, excepte per als problemes més senzills.

4.3.1. Optimització amb algorismes genètics

Un mètode d'optimització molt adequat per a resoldre problemes d'ajustament de paràmetres, en els quals hi ha una gran quantitat de paràmetres interdependents, són els **algorismes genètics**.

Teoria de l'evolució

La teoria de l'evolució de Charles Darwin afirma que els individus millor adaptats a un entorn són els que tenen més possibilitats de tenir descendència, de manera que les seves característiques s'imposen a les d'altres individus pitjor adaptats. Els algorismes genètics s'inspiren en aquesta idea, canviant «entorn» per «problema», «individu» per «solució», i «adaptació» per «qualitat d'una solució».

Amb més detall, donat un problema *P* amb un espai de solucions *S*, els components d'un algorisme genètic són:

- Un **individu**, que definim com una solució concreta. Per exemple, en l'ajustament d'atributs de les unitats del joc, un individu serà una combinació concreta d'atributs de totes les unitats. Seguint la metàfora evolutiva, cada atribut d'un individu és com un gen, i tots els seus atributs (generalment en un vector) són el seu genoma complet.

- Una **població**, és a dir, un conjunt d'individus que competeixen entre ells, ja que els millor adaptats (les millors solucions) seran els que es reproduiran. En augmentar la grandària de la població es millora l'exploració de *S*, però s'incrementa el cost computacional. A manera d'orientació, es recomana que la grandària de la població (nombre d'individus) sigui igual o superior al nombre d'atributs.

- Una **funció objectiu** (o funció d'**error** si el que volem és reduir un error). Exemples d'aquesta funció són: durada d'una partida, diferència entre els punts aconseguits per cada tipus d'unitat, temps invertit a recórrer un circuit, distància del punt d'impacte d'un projectil a un objectiu. La funció objectiu ha de ser contínua i s'ha d'evitar que tingui canvis bruscos i zones constants.

- Un **sentit d'operació**. Depenent de com l'hàgim definit i quin sentit tingui, voldrem minimitzar o maximitzar la funció objectiu.

L'esquema genèric dels algorismes genètics és el següent:

1) Es crea una població inicial aleatòriament.

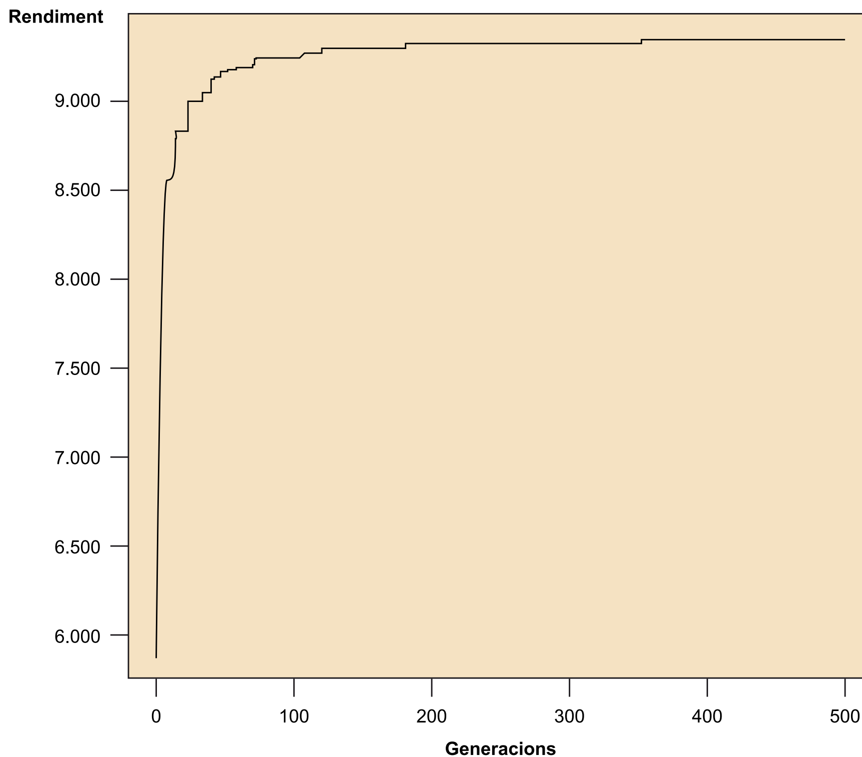
2) S'aplica la funció objectiu a cada individu. Els que obtinguin millors resultats seran els més ben adaptats.

3) Els individus més ben adaptats se **seleccionen** per reproduir-se mitjançant **creuament**, és a dir, combinació dels seus gens. Això dona lloc a una nova **generació** d'individus.

4) Per a augmentar la diversitat de la població i explorar així noves possibilitats, els nous individus poden patir una **mutació**, això és, un canvi aleatori en alguns dels seus gens.

5) Tornar a 2 fins que s'aconsegueixi el nombre de generacions (equivalent a iteracions) que es vol.

En principi, com més generacions es provin millors solucions s'obtindran, però a canvi d'un cost computacional major. Per a saber quan convé detenir-se, interessa observar la funció objectiu i veure a partir de quin moment s'estanca; és a dir, no hi ha millora encara que es continuïn generant nous individus.



Funció objectiu (rendiment d'un sistema) en funció del nombre de generacions. Observeu el ràpid creixement inicial i l'estancament relatiu a partir de dues-centes generacions.

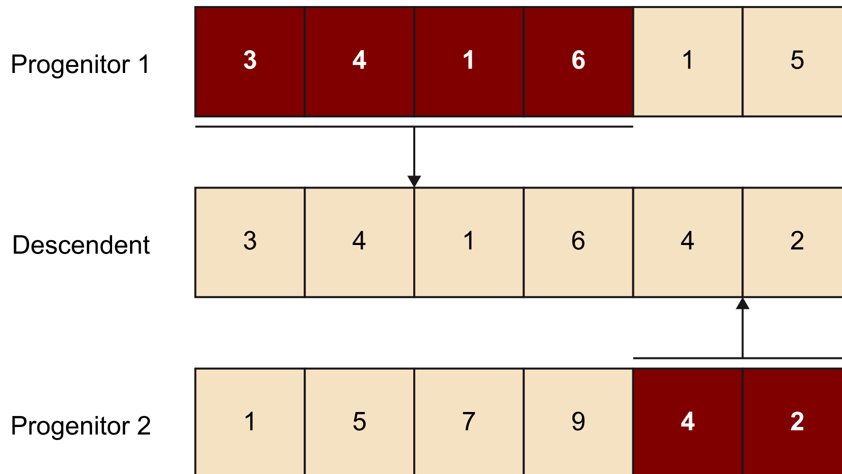
Algunes de les operacions esmentades en l'esquema necessiten ser concretades. La **selecció** d'individus fa referència a quins individus es trien perquè es reprodueixin i donin lloc a una nova generació. En principi, es podrien agafar els millors individus, però això faria que en unes quantes iteracions tots els individus fossin descendents d'uns pocs progenitors i es perdria la diversitat de solucions, i així s'empobriria el rendiment de l'algorisme (aquí també és important la biodiversitat!).

Una alternativa molt utilitzada és la **selecció per torneig**: se seleccionen dos individus a l'atzar i es tria el més ben adaptat (millor valor en la funció objectiu); a continuació se selecciona un altre individu a l'atzar i es compara amb el vencedor del pas anterior, quedant el millor adaptat dels dos; aquest procés es repeteix un nombre de vegades (unes tres) i finalment queda seleccionat l'individu que venç en l'última ronda. Tot aquest procés es repeteix tantes vegades com sigui necessari (en general, si necessitem generar n nous individus, seran necessaris $2n$ progenitors).

Precisament, el **creuament** d'individus es refereix a com es combinen els gens de dos individus (poden ser més però això no és habitual) per donar lloc a un descendent que formarà part de la població de la generació següent. En general, es persegueix barrejar els atributs (gens) dels dos individus per obtenir un nou individu que, amb una mica de sort, reuneixi el millor dels dos progenitors. Hi ha diverses estratègies de creuament, encara que les més habituals són les següents:

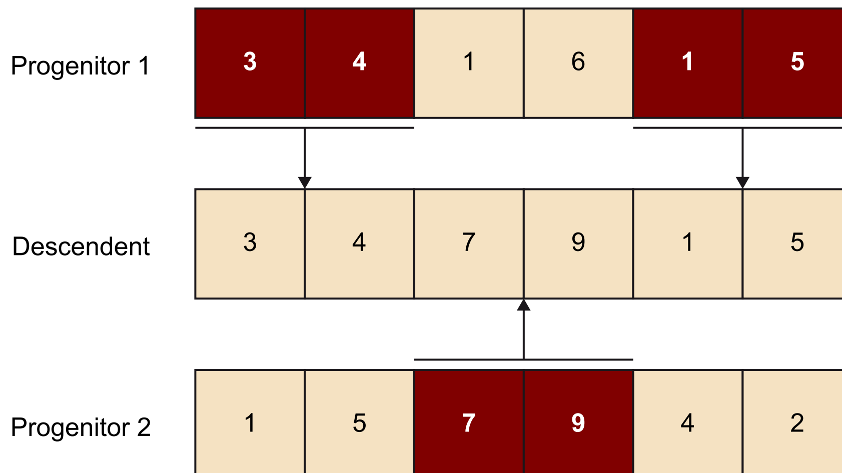
a) En el creuament **simple** es tria un punt del genoma a l'atzar i es tallen els genomes dels progenitors per aquest punt, prenent d'un progenitor els atributs fins al punt de tall i de l'altre els atributs a partir del punt de tall. És útil quan no hi ha un gran nombre d'atributs.

Exemple de creuament simple



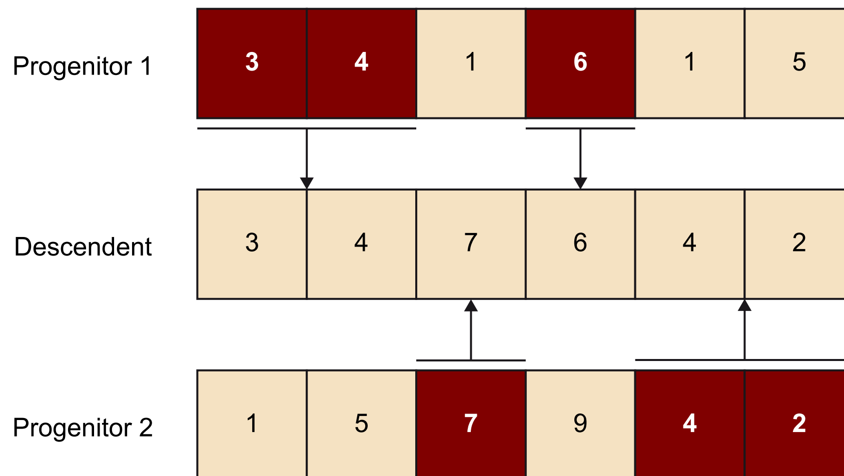
b) El creuament **doble** funciona de manera semblant al simple, però en aquest cas es trien dos punts de tall i es prenen els atributs entre els dos punts de tall d'un progenitor i els restants de l'altre. És útil si hi ha un gran nombre d'atributs per a aconseguir més varietat.

Exemple de creuament doble



c) Finalment, en el creuament **uniforme** es pren cadascun dels atributs aleatòriament d'un o altre progenitor. És útil quan no hi ha *localitat* en els atributs, és a dir, no és important mantenir fragments seguits de genoma perquè no hi ha cap estructura en el vector d'atributs.

Exemple de creuament uniforme



Una limitació d'utilitzar solament el creuament per a generar nous individus és que en tot moment es recombinen valors ja existents en el patrimoni genètic de la població, però no s'introdueixen valors nous que permetin explorar altres possibilitats. Per a superar aquesta limitació, es defineix el tercer tipus d'estratègia, anomenada **mutació**, en la qual alguns atributs es modifiquen aleatòriament per provar valors nous i explorar noves zones de l'espai de solucions.

Les mutacions solen ser alteracions aleatòries la forma concreta de les quals depèn del tipus de dades dels atributs. Els més habituals són aquests:

- Atributs binaris: la mutació canvia a l'altre valor possible (de 0 a 1 i viceversa). Per exemple, volar o no volar.

- Atributs categòrics: es tria un nou valor aleatòriament d'entre els valors possibles. Per exemple, el color d'una unitat d'entre una paleta.

- Atributs numèrics: se suma una quantitat aleatòria al valor actual. Si per exemple la massa d'un objecte és 1.500 kg, s'afegeix o resta una quantitat a l'atzar.

En dissenyar l'algorisme genètic se sol ajustar la mutació d'acord amb dos paràmetres:

- Probabilitat de mutació individual: probabilitat que un individu tingui mutacions. Sol ser baixa (<10 %), ja que en cas contrari la lògica de l'algorisme genètic es perd.

- Probabilitat de mutació d'atributs: donat un individu que tindrà mutacions, probabilitat de cadascun dels seus atributs de ser mutat. També hauria de ser baixa (<10 %), o en cas contrari l'algorisme genètic s'acabaria convertint en una exploració aleatòria.

Algorismes genètics

Es denominen «algorismes genètics», en plural perquè hi ha moltes maneres de concretar-los per a resoldre un problema (codificació dels individus, disseny de la funció objectiu, grandària de la població, nombre de generacions, estratègies de selecció, creuament i mutació, etc.), així que es consideren una família d'algorismes, no un únic algorisme que sempre és igual.

Utilitzarem els algorismes genètics per a equilibrar les característiques de deu cotxes en un joc de carreres del tipus Fórmula 1. Cada cotxe té quatre atributs (velocitat màxima, acceleració, frenat, adherència), per la qual cosa una solució del problema és un vector amb valors per als quatre atributs dels deu cotxes, és a dir, un vector de quaranta elements de tipus real.

Encara que no desenvoluparem l'exemple complet (falta afegir el joc per si mateix i la IA que condueix els cotxes), sí que veurem la part d'ajustament de paràmetres. Els elements que definiran l'algorisme genètic per a aquest problema són:

- Individu: vector de quaranta atributs reals.
- Població: almenys quaranta individus (millor si poden ser cent).
- Nombre de generacions: convé provar amb unes cent i anar controlant què passa amb la funció objectiu.
- Selecció: per torneig en tres passos.
- Creuament: pot ser simple o doble. D'entrada, no interessa l'uniforme perquè cada bloc de quatre atributs té un sentit (pertany al mateix cotxe).
- Mutació: sumar o restar una quantitat aleatòria a l'atribut controlant que es mantingui en el rang permès (hi haurà un mínim i un màxim per a cada tipus d'atribut).
- Funció objectiu: hi ha moltes maneres de valorar quina combinació de cotxes és millor. Aquí hi ha dues propostes:
  - Analítica: dissenyar una funció que assigni una puntuació a un cotxe en funció dels seus atributs. La funció objectiu pot ser la desviació estàndard d'aquesta funció; si és zero, vol dir que tots els cotxes puntuen igual. Un problema és que pot ser difícil dissenyar una funció així i que sigui realista.
  - Experimental: si ja es disposa del joc i de la IA de conducció, es pot executar el joc amb cada configuració (individu) i analitzar fins a quin punt arriben igualats els cotxes. És millor si es pot executar diverses vegades per a cada individu i es comprova que queden de mitjana bastant igualats. La funció objectiu seria la desviació estàndard en les posicionis o temps obtinguts pels cotxes.

Biblioteca AForge.NET

És possible programar l'algorisme genètic complet, ja que des del punt de vista de la programació no són molt complexos. No obstant això, hi ha biblioteques que els contenen (i ja estan provats i ofereixen diferents opcions); en C# els podem trobar a la biblioteca AForge.NET.

En qualsevol cas, cal tenir en compte que els components d'algorismes genètics d'AForge.NET no estan tan desenvolupats com en altres biblioteques, especialment pel que es refereix a la documentació i a la flexibilitat per a personalitzar-ne el funcionament. Si es necessita un millor control sobre l'algorisme, es recomana, per exemple, la biblioteca DEAP (Distributed Evolutionary Algorithms in Python), que és evidentment a Python.

Webs recomanades

La pàgina oficial de la biblioteca AForge.NET és http://www.aforgenet.com/.

I la seva pàgina sobre algorismes genètics és http://www.aforgenet.com/framework/features/genetic_algorithms.html.

AI4unity es pot obtenir a https://github.com/davidgutierrezpalma/ai4unity.

La instal·lació d'AForge.NET en Unity no és senzilla; es recomana utilitzar l'adaptació AI4unity i instal·lar els *assets* Core, Genetic i Math de la manera que s'explica en la secció 4.2.1. A més, cal afegir un fitxer anomenat «gmcs.rsp» a la carpeta Assets que contingui simplement el text «-unsafe» (sense les cometes). Algunes característiques d'AForge.NET no es poden utilitzar en Unity perquè utilitza una versió antiga de C# (2.0).

Observació

La traducció d'un paràmetre d'un rang 0...1 a un altre rang és senzilla: sigui *min* el valor mínim del rang que es vol i *max* el màxim, si l'algorisme ens dona un valor *x*, aquest valor traduït al rang *min…max* ve donat per

$$y = x\,(max - min) + min$$

Podeu comprovar que si $x = 0$ $y = min$, i que si $x = 1$ $y = max$.

Tornant a l'exemple de l'ajustament de característiques dels cotxes de Fórmula 1, el codi següent utilitza un algorisme genètic per a trobar un ajustament equilibrat. Per a poder donar un exemple complet, prendrem una funció objectiu senzilla. En primer lloc, la puntuació d'un cotxe serà la mitjana dels seus quatre paràmetres; la funció objectiu serà la desviació estàndard entre les puntuacions dels cotxes d'una solució donada, i l'objectiu serà reduir-la a zero.

Les quatre característiques tenen rangs diferents: la velocitat màxima 200-300, l'acceleració 5-10, la frenada 5-20 i l'adherència 0-1. No obstant això, per a simplificar l'algorisme es manejaran quatre valors reals qualssevol per a aquestes característiques, i després s'adaptaran als rangs reals en el joc.

Evidentment, aquesta funció objectiu no és molt realista i la solució obtinguda no mostrarà probablement una gran jugabilitat; tal com s'ha comentat abans, el millor seria utilitzar un joc real i fer competir els cotxes per mesurar-ne el rendiment en la carrera.

El codi següent s'ha d'associar a un objecte buit (*empty*) de Unity creant un *script* tal com s'ha descrit en l'apartat 4.2.1:

```
using AForge.Genetic;

using AForge.Math;
using System;         // Para Math.Pow
using UnityEngine;


// Classe per definir la funció objectiu (calcular la desviació
// estàndard de les mitjanes de cada cotxe)
public class DiferenciasCoches : IFitnessFunction
{
   // Constructor (podria rebre dades necessàries per calcular
   // la qualitat de les solucions)
   public DiferenciasCoches()
   {
   }

   // Per visualitzar els resultats
   public object Translate(IChromosome cromosoma)
   {
      return cromosoma.ToString();
   }

   // Calcular la funció objectiu
   public double Evaluate(IChromosome cromosoma)
   {
      // Com AForge considera que un individu amb major funció
      // objectiu és millor (és a dir, solament maximitza) hem
      // d'invertir el resultat de la desv. estàndard.
      // Sumem 1 al denominador per evitar divisions per zero.
      return 1.0 / (1.0 + evalua(cromosoma));
   }
```

```
    // Funció que calcula la desviació estàndard d'un cromosoma
    // interpretant que hi ha n cotxes amb 4 atributs cadascun
    public double evalua(IChromosome cromosoma)
    {
       // Per accedir als valors propis del array de reals
       // Es prenen els valors de 4 en 4 (cada cotxe) i es calcula la
       // mitjana de cada grup
       DoubleArrayChromosome miCrom = (DoubleArrayChromosome)cromosoma;
       int numCoches = miCrom.Length / 4;
       double[] medias = new double[numCoches];
       for (int i = 0; i < numCoches; i++)
       {
          // Calcula la mitjana de les 4 components de cada cotxe
          medias[i] = media(miCrom.Value[i * 4],
                       miCrom.Value[i * 4 + 1],
                       miCrom.Value[i * 4 + 2],
                       miCrom.Value[i * 4 + 3]);
       }
       return desvEstandar(medias);
    }

    // Calcula la desviació estàndard de les valoracions
    // dels cotxes
    public double desvEstandar(double [] valores)
    {
       double suma = 0.0;
       foreach (double x in valores)
          suma += x;
       double media = suma / valores.Length;

       double suma2 = 0.0;
       foreach (double x in valores)
          suma2 += Math.Pow(x - media, 2);
       return Math.Sqrt(suma2 / (valores.Length-1)) ;
    }

    // Calcula mitjana dels quatre valors
    public double media(double veloc, double acel, double freno,
    double agarre)
    {
       return (veloc + acel + freno + agarre) / 4.0;
    }
}



public class AlgoritmoGenetico : MonoBehaviour {

// Utilitza aquesta funció per a la inicialització
void Start () {
      // Funció objectiu (calcula la qualitat d'una solució)
      DiferenciasCoches difCoches = new DiferenciasCoches();

      // Generador de nombres aleatoris per crear la població,
      // mutacions, etc.
      AForge.Math.Random.UniformOneGenerator generador = new
      AForge.Math.Random.UniformOneGenerator();

      // Nombre de paràmetres en el problema (10 cotxes x 4
      paràmetres/cotxe)
      int numParams = 40;

      // S'ha de generar un vector base de 40 reals
      double[] valores = new double[numParams];
      Debug.Log("A==============================");
      // Paràmetres del constructor de Population
      // -mida de la població (nombre d'individus)
      // -mida d'un individu (nombre d'atributs, 4 x 10)
      // -difCoches -> objecte que conté la funció objectiu
      // -RouletteWheelSelection -> funció de selecció (a l'atzar
```

```
      // proporcional a la qualitat de cada solució)
      Population population = new Population(50,
            new DoubleArrayChromosome(generador, generador,
            generador, valors),
            difCoches,
            new RouletteWheelSelection());

      // Fer evolucionar a la població durant un nombre
      // de generacions; mostrar resultats cada 10
      for(int i=0; i<1000; i++)
      {
         population.RunEpoch();
         if(i % 10 == 0)
         {
            Debug.Log(i);
            Debug.Log(population.BestChromosome.ToString());
            Debug.Log(difCoches.evalua(population.BestChromosome));
         }
      }

      Debug.Log("Valor final");
      Debug.Log(population.BestChromosome.ToString());
      Debug.Log(difCoches.evalua(population.BestChromosome));

   }

   // Update es crida un cop per frame
   void Update () {

   }
}
```

En la sortida per la consola de depuració es pot observar que la desviació estàndard descendeix a mesura que avancen les generacions, la qual cosa indica que els diferents cotxes estan equilibrats entre ells i, per tant, un jugador en pot triar qualsevol i guanyar en el joc.

Resumint, els algorismes genètics són una eina molt útil per a resoldre problemes d'optimització com per exemple els d'ajustament de paràmetres, però tenen els desavantatges de requerir bastanta memòria i temps de computació, i per això s'utilitzen en general en processos fora de línia o en moments en els quals no interfereixin amb l'execució del joc (mentre es mostren puntuacions, etc.)

5. Tècniques avançades d'IA

En aquest apartat estudiarem les últimes tècniques d'IA que s'investiguen ara mateix i que prometen revolucionar el que coneixem per IA, tant en jocs com en altres àmbits. A partir del 2015 hi ha hagut assoliments impensables anteriorment, entre els quals destaquem:

- AlphaGo: un sistema que ha derrotat l'expert en go (joc mig escacs mig dames xineses, més complex que els escacs) Lee Se-dol, considerat el segon millor jugador del món.

- DQN: un sistema que, veient simplement les imatges d'un simulador de consola Atari 2600, és capaç d'aprendre a jugar sense cap instrucció prèvia. Fins i tot juga millor que humans experts, i és capaç de trobar trucs per a millorar el joc.

- Un sistema que juga al joc de trets en primera persona Doom a partir de les imatges i és capaç de guanyar jugadors humans; de fet, està especialitzat a caçar jugadors humans (i això ha suscitat una certa polèmica).

- Un pilot automàtic de caces de combat que ha derrotat pilots experts en un simulador de vol militar.

- Fora de l'estricte àmbit dels videojocs, sistemes de conducció automàtica, reconeixement de veu en mòbils, reconeixement de rostres en xarxes socials, millores en la traducció automàtica, etc.

Tots aquests avenços tenen en comú el mètode en el qual estan basats, conegut com a aprenentatge profund (*deep learning*), que descriurem en aquest apartat. En primer lloc, veurem els seus elements constituents, les xarxes neuronals, i a continuació quina és l'arquitectura dels sistemes d'aprenentatge profund i quines opcions hi ha.

5.1. Xarxes neuronals

El cervell humà (i de la resta dels animals) és format per diferents tipus de cèl·lules. Les neurones són les cèl·lules que, establint connexions entre elles i enviant missatges per aquestes connexions, donen lloc al comportament emergent que coneixem com a intel·ligència.

La meta de les **xarxes neuronals** és crear una mena de neurones computacionals per intentar reproduir la intel·ligència humana en un ordinador, si bé l'analogia es queda en el fet que hi ha uns elements anomenats **unitats** —

equivalents a les neurones— que es connecten unes amb altres i es passen missatges mitjançant **connexions**; més enllà d'això, no cal pensar que les xarxes neuronals s'assemblin a les neurones del cervell.

5.1.1. Components d'una xarxa neuronal

L'arquitectura bàsica d'una xarxa neuronal es compon dels elements següents:

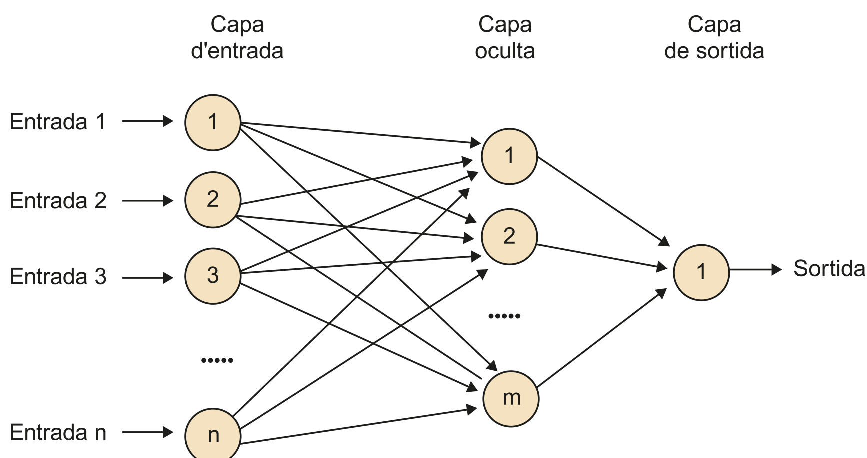
- Una capa d'unitats (equivalents a les neurones) d'**entrada**, que reben un conjunt de valors de l'exterior (píxels d'una imatge, posició dels jugadors d'un partit de bàsquet, etc.)

- Una capa d'unitats de **sortida**, composta per una o més unitats que produeixen una sortida a l'exterior, que seria el «resultat» de l'execució de la xarxa, per exemple, un valor que indica si un cotxe ha de frenar, si el ninot d'un joc s'ha de desplaçar a esquerra o a dreta, etc.

- Una (o més) capes d'unitats **ocultes**, que reben connexions de la capa d'entrada i es connecten a les unitats de sortida.

- El conjunt de **connexions** entre capes. En l'arquitectura més bàsica, anomenada xarxa neuronal prealimentada (*feed forward neural network*) o perceptró multicapa (*multilayer perceptron*), les connexions sempre van de les unitats d'una capa a la següent, és a dir, de la capa d'entrada a l'oculta i d'aquesta a la capa de sortida.

Arquitectura bàsica d'una xarxa neuronal del tipus «prealimentada»



El funcionament general de les xarxes neuronals és, sense entrar en els detalls tècnics, la **propagació** de senyals d'una capa a la següent. Vegem pas a pas el que ocorre davant un determinat conjunt d'entrades en el qual cada entrada a la xarxa tindrà un valor diferent:

1) Les unitats de la capa d'entrada reaccionaran **activant** la seva sortida o no en funció de l'entrada rebuda.

2) Com que la sortida de les unitats de la capa d'entrada està connectada a les entrades de les unitats de la capa oculta, aquestes reben un conjunt de senyals d'entrada (per a elles) enfront de les quals també han de reaccionar. Per a això, cada unitat de la capa oculta té un **vector de pesos**, un valor per cada connexió que li arriba, que es combina amb l'entrada rebuda i fa que al seu torn s'activi o no la sortida de cada unitat oculta.

3) Les unitats de la capa de sortida reben senyals de les unitats ocultes i apliquen el seu propi vector de pesos a aquests senyals per decidir si activen la seva sortida o no, i això serà el resultat final de l'execució de la xarxa per a l'exemple rebut.

Xarxes neuronals

Les xarxes neuronals són un formalisme que permet aprendre una funció multivariant (amb més d'una variable d'entrada) i multivaluada (amb més d'una variable de sortida). Amb suficients unitats i capes ocultes, és possible aprendre qualsevol funció.

Les unitats d'una capa poden tenir connexions cap a totes les unitats de la següent (llavors es parla de capes totalment connectades o denses) o, per contra, cada unitat pot estar connectada solament a algunes de les unitats de la capa següent.

Nota

En aquest apartat s'explicaran les opcions més utilitzades, encara que hi ha moltes alternatives que ometrem, ja que un curs complet sobre xarxes neuronals requeriria un llibre complet (o més).

5.1.2. Funcions d'activació

Hi ha nombroses alternatives per a cadascun dels elements que componen una xarxa neuronal. Per exemple, els senyals d'activació generats per les unitats poden ser **binaris** (plantejament original) o **reals** (ús actual).

El **vector de pesos** de cada unitat sol ser un vector de nombres reals W, tants com entrades tingui la unitat, i això és realment la «memòria» de la xarxa; a més, se sol incloure un valor escalar b (per *bias*, és a dir, un biaix o valor de desplaçament). Així, si el vector d'entrades a una unitat és X, es fa aquesta operació:

$$z = W^T X + b \qquad (1)$$

(és a dir, es multiplica cada entrada pel seu pes associat, se sumen aquests productes i al final se suma b). El valor z obtingut així (un nombre real) s'utilitza llavors com a entrada per a una funció d'**activació**, que és la que decideix quina és la sortida.

La funció d'activació més utilitzada en unitats ocultes és l'anomenada *ReLU* (de *rectified linear unit*), que a partir del valor z calculat anteriorment aplica:

$$g(z) = \max\{0,z\}$$

És a dir, la sortida és 0 si l'entrada és negativa, i és igual a z si z és positiva.

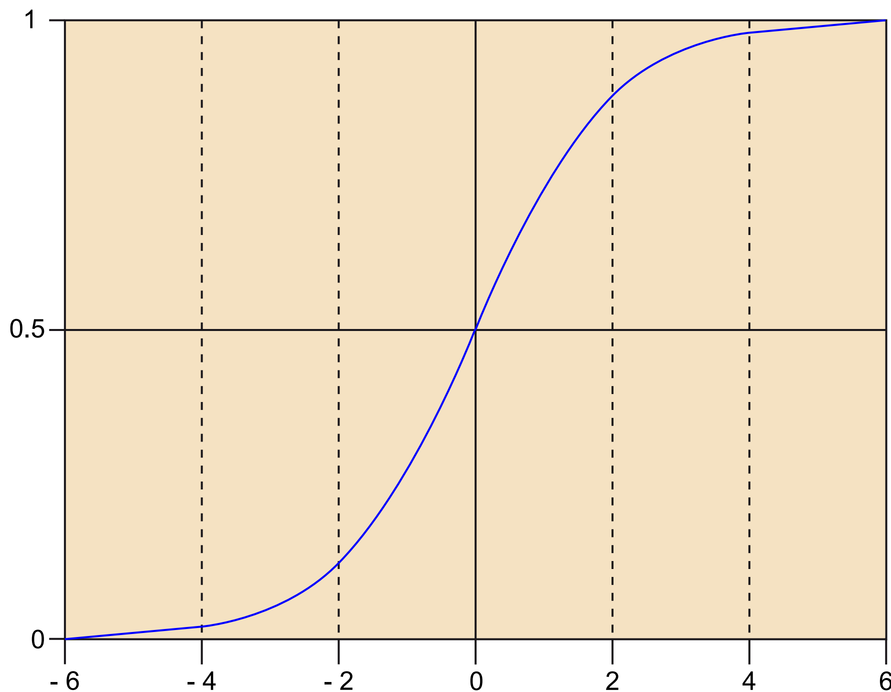
En el cas d'unitats de sortida, s'utilitzen altres funcions d'activació, que depenen del tipus de sortida que es vol:

- Valor **real** (per exemple, quant trepitjar l'accelerador): unitats lineals, és a dir el valor z calculat segons l'expressió (1).

- Valors **binaris** (per exemple, decidir si saltar o no): unitats **sigmoides**, que usen la funció logística sigmoide, la qual té aquest aspecte:

Observació

És millor que la sortida de les unitats *softmax* no sigui exactament 1 en una i 0 en les altres per a facilitar l'entrenament.



- Valors **multiclasse** (per exemple, en decidir quin tipus d'unitat produir en un joc d'estratègia): funció **softmax**, en la qual la sortida és un vector amb tants valors com classes possibles es puguin triar (per exemple, un valor per a triar obrer, un altre per a soldat, un altre per a arquer, etc.); la sortida que correspon al valor triat tindrà un valor proper a 1 i les altres tindran un valor proper a 0.

5.1.3. Entrenament d'una xarxa neuronal

Acabem de veure quines són les operacions que duu a terme una xarxa neuronal en resposta a un conjunt d'entrades, però hem donat per suposats alguns valors que apareixien com per art de màgia en les fórmules, concretament els vectors de pesos W de les unitats internes i de sortida i els valors de biaix b. Lògicament, aquests valors no apareixen sols, sinó que primer la xarxa els ha d'aprendre, ja que les xarxes neuronals passen per dues fases: una primera d'**entrenament** i una altra d'**execució**; hem vist com operen en la fase

d'execució per donar la sortida corresponent a una entrada, però falta veure com entrenar la xarxa perquè aprengui els pesos i biaixos adequats si volem que reaccioni correctament.

En la fase d'entrenament, a la xarxa se li dona un conjunt d'exemples d'entrenament que inclouen la sortida esperada per a cadascun.

> Desenvolupem un joc per a mòbil o tauleta i volem que reconegui gestos de l'usuari en la pantalla. El conjunt d'entrenament seria un conjunt de gestos (la captura dels píxels activats en el gest) i el codi del gest corresponent. Perquè la xarxa pugui aprendre, serà necessari proporcionar-li molts exemples de cada classe de gest.

En iniciar l'entrenament, la xarxa s'inicialitza amb valors aleatoris (es recomanen valors petits, entorn de 0,1). A continuació, s'hi van donant els exemples i s'analitza la diferència entre la sortida generada per la xarxa i la sortida esperada; llavors es torna cap enrere (de la capa de sortida a la capa d'entrada) ajustant els pesos i els biaixos per intentar minimitzar la diferència entre la sortida obtinguda i l'esperada. Aquest procés es repeteix per a tots els exemples d'entrenament fins que s'aconsegueix minimitzar la diferència. Aquesta tornada enrere per a anar ajustant els pesos es denomina **retropropagació** (*backpropagation*).

L'algorisme d'optimització que se sol utilitzar per a fer aquest ajustament de pesos i biaixos és el **descens de gradients estocàstic** (*stochastic gradient descent*), que consisteix, simplificant, a anar provant valors aleatoris (de pesos i biaixos) al voltant del valor actual i prendre el que mostri un error menor.

Al final de la fase d'entrenament la xarxa serà capaç de produir una sortida d'acord amb l'entrada que rebi segons els exemples que ha rebut.

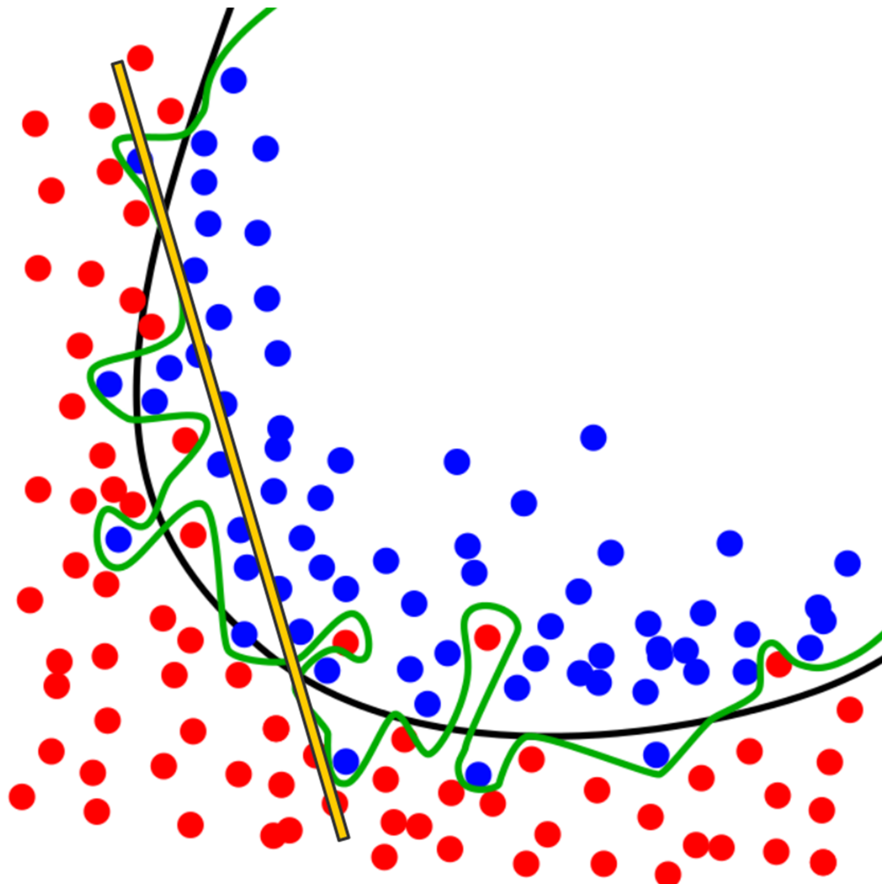
5.1.4. Problemes d'aprenentatge

Tant en les xarxes neuronals com en altres mètodes d'aprenentatge, ens podem trobar amb aquests dos problemes:

- **Infraajustament** (*underfitting*): si la funció que volem aprendre és molt complexa i la nostra xarxa neuronal no té suficients unitats o capes ocultes, no podrà aprendre bé la funció (no tindrà suficient «memòria», que es denomina **capacitat** del model).

- **Sobreajustament** (*overfitting*): per contra, si la nostra xarxa té massa capacitat (unitats i capes ocultes), aprendrà perfectament tots els exemples d'entrenament però no serà capaç de generalitzar exemples lleugerament diferents quan es vulgui posar en ús.

Problemes d'aprenentatge



Problemes d'aprenentatge: infraajustament (línia groga), sobreajustament (línia verda), aprenentatge correcte (línia negra).
Font: adaptat de Chabacano, CC Wikimedia Commons,(GFDL). https://commons.wikimedia.org/w/index.php?curid=3610704

Com a conclusió, la capacitat de la xarxa ha de ser adequada a la funció que es vol aprendre. La capacitat adequada es coneix utilitzant un protocol d'**entrenament** més **validació**.

1) Les dades d'entrenament es divideixen en dos grups: entrenament i validació.

2) La xarxa s'entrena amb les dades d'entrenament i es monitora l'error d'entrenament (diferència entre la sortida esperada i l'obtinguda). Mentre aquest error es redueix, la xarxa millora l'aprenentatge (evitem l'infraajustament). Convé augmentar la capacitat de la xarxa.

3) En paral·lel, es monitora l'error que produiria la xarxa si es provés amb les dades de validació, que equival a provar les dades en una situació real amb exemples que no havia vist abans.

4) Si l'error de validació comença a créixer, es produeix *sobreajustament* perquè la xarxa aprèn amb tant detall els exemples d'entrenament que és incapaç de generalitzar. És el moment de deixar d'ampliar la xarxa.

Hi ha altres estratègies per a reduir el sobreajustament, que es coneixen com a **regularització** del model per a fer-lo més general. En xarxes neuronals, l'estratègia de regularització més utilitzada s'anomena ***dropout*** i, sorprenentment, consisteix a apagar aleatòriament algunes unitats ocultes durant l'entrenament. La idea és que així la xarxa serà capaç d'adaptar-se amb més facilitat a canvis en els valors d'entrada en lloc de seguir al peu de la lletra els exemples d'entrenament, i en el fons el sobreajustament consisteix en això.

5.1.5. Perspectiva històrica i futura

Les xarxes neuronals són un dels primers mètodes d'IA; de fet, el perceptró simple (sense capes ocultes) va ser proposat en 1958 per Frank Rosenblatt. Durant la seva història han sofert molts alts i baixos: quan es van proposar semblava que anaven a aconseguir emular ràpidament la intel·ligència humana; al cap d'uns anys van caure en desús; als anys vuitanta del segle XX van tornar a ressorgir amb el desenvolupament de l'algorisme de retropropagació; van tornar a decaure, ja que no es van produir més avenços; i especialment des de l'any 2010 han tornat a sorgir amb molta força gràcies a diferents factors que permeten crear xarxes amb més capes ocultes, més neurones, major estabilitat numèrica i que poden entrenar-se i executar-se a gran velocitat gràcies als avenços en el maquinari, especialment els processadors gràfics. En l'apartat següent veurem les característiques d'aquest renaixement de les xarxes neuronals.

5.2. Aprenentatge profund

Es denominen **aprenentatge profund** (*deep learning*, DL) les diferents arquitectures de xarxes neuronals la característica principal de les quals és que contenen un gran nombre de capes ocultes; la profunditat es refereix al nombre de capes de la xarxa.

El DL ha estat possible per diferents factors científics i tecnològics que han sorgit des del 2004 aproximadament, i entre els quals destaquen:

- L'ús de funcions d'activació com ReLU, que faciliten l'optimització de les xarxes neuronals, en contrast amb funcions que s'utilitzaven anteriorment i provocaven problemes numèrics o donaven una resposta pobra.

- L'aplicació del descens de gradients estocàstic com a algorisme d'optimització en l'entrenament de les xarxes.

- El gran desenvolupament de maquinari capaç d'operar amb vectors, concretament els processadors gràfics, que poden dividir per deu o més el temps necessari per a dur a terme un entrenament.

- L'abundància de grans conjunts de dades que són adequats per a aquest tipus de mètodes, al contrari del que ocorre amb altres mètodes

d'aprenentatge automàtic, que requereixen crear matrius de *kernel* o similars, i això no és viable quan hi ha milions d'exemples.

L'avantatge de tenir una xarxa profunda rau en el fet que cada capa extreu característiques interessants de l'anterior, i així la xarxa obté a poc a poc característiques cada vegada més sofisticades sense necessitat de programar-la expressament per a això. Per exemple, d'una imatge es pot començar extraient els contorns de les figures, en un altre nivell es poden identificar formes bàsiques, després unir diverses d'aquestes formes per formar cares, etc. D'aquesta manera, la xarxa és útil per a resoldre tasques amb dades noves.

Per contra, tenir xarxes amb moltes unitats però poques capes (és a dir, capes molt amples) solament serveix perquè la xarxa aprengui diferents combinacions de les dades d'entrada però sense capacitat de generalitzar; és a dir, tendeixen més al sobreajustament.

5.2.1. Arquitectures

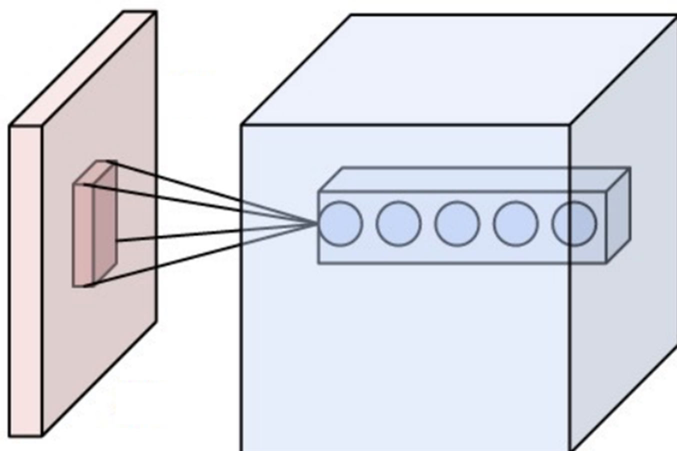
Dins del DL hi ha diferents tipus d'arquitectures, cadascuna orientada a resoldre determinats tipus de problemes:

- Xarxes **prealimentades** (*feed-forward neural networks*, FNN): es tracta de l'arquitectura bàsica, descrita en l'apartat anterior. La seva característica és que les connexions sempre van de l'entrada a la sortida, i no hi ha connexions entre unitats de la mateixa capa fins a capes anteriors. S'utilitzen en problemes de classificació de dades, de generació de valors reals, etc.

- Xarxes neuronals **recurrents** (*recurrent neural networks*, RNN): en aquestes xarxes sí que hi ha algunes connexions cap a capes anteriors (cap a l'entrada) amb l'objectiu que la xarxa recordi exemples anteriors i els tingui en compte en l'aprenentatge. S'utilitzen per a dades que tenen un sentit temporal, per exemple, en processament de senyals o de llenguatge escrit i parlat.

- Xarxes neuronals **convolucionals** (*convolutional neural networks*, CNN): en aquestes xarxes les unitats de la capa d'entrada no estan connectades a totes les unitats de la capa següent, sinó solament a unes quantes. La idea és aprofitar la localitat de les entrades, per exemple, dels píxels d'una imatge: solament els píxels adjacents tenen relació entre ells, i per això les entrades corresponents a aquests píxels es connecten solament a les unitats que els corresponen. Això té l'avantatge de reduir el cost computacional (els vectors de pesos són molt més petits) i permetre que la xarxa detecti característiques de detall (per exemple, contorns en una imatge) sense necessitat de processar la imatge sencera. Aquest tipus de xarxes s'utilitzen per a processar imatges o qualsevol tipus de dada en la qual hi hagi localitat en els atributs.

Xarxa neuronal convolucional



Xarxa neuronal convolucional: cada zona de la xarxa solament processa una part de la imatge d'entrada. Font: Aphex34, own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45659236

D'altra banda, ens trobem amb els **autocodificadors** (*autoencoders*), que es poden aplicar a qualsevol de les tres arquitectures anteriors i que consisteixen a crear una xarxa que tingui el mateix nombre de sortides que d'entrades, i obligar-la perquè les sortides siguin iguals a les entrades (per exemple, si entra la foto d'un gat, que surti la mateixa foto). Això té el sentit que algunes capes ocultes tenen menys unitats que les entrades, la qual cosa obliga la xarxa a codificar la informació amb menys dades, de manera que després es pugui reconstruir l'original. D'aquesta manera, s'aconsegueix generar una reducció de la dimensionalitat en la imatge.

Una aplicació del DL de gran importància per als videojocs és l'anomenat Q-Learning, en el qual s'utilitza una xarxa DL per a aprendre la matriu d'estats i accions (*Q*) de problemes d'aprenentatge per reforç (vegeu l'apartat 4). Tal com vam veure, en gairebé qualsevol videojoc el nombre d'estats possible és immens, i si es multiplica pel nombre d'accions possibles resulta una matriu de possibles accions per a un agent que no es pot emmagatzemar. Però és possible usar una xarxa DL perquè aprengui una aproximació de la matriu *Q* fent que associï cada entrada (estat del joc) amb una acció i una recompensa. Això és el que fa, per exemple, l'algorisme DQN (Deep Q-Network), el que va aprendre a jugar als videojocs d'Atari.

5.2.2. Llibreries per a aprenentatge profund

És possible utilitzar DL en videojocs, si bé la fase d'entrenament sol ser molt costosa computacionalment (i a més es recomana, per velocitat, executar-la en el processador gràfic, cosa complicada durant el joc). Per aquest motiu, en general s'entrenaran les xarxes fora de línia i després s'usaran dins del joc.

Com que hi ha bastantes matemàtiques i programació vectorial i matricial darrere dels algorismes de DL, és convenient utilitzar alguna de les llibreries existents.

Lamentablement, en Unity i C# no hi ha llibreries molt desenvolupades:

- En **Accord.NET** hi ha algunes funcions per a fer operacions bàsiques, però en un estat bastant primitiu.

- En **AForge.NET** hi ha més desenvolupament fet, si bé contiua essent bastant limitat en opcions.

Les llibreries de DL més desenvolupades actualment són les següents:

- **Theano.** És una llibreria en Python que ofereix tot el suport matemàtic necessari per a DL, amb tot tipus d'objectes i amb codi optimitzat especialment per a CUDA (llenguatge de programació del processador gràfic creat per NVidia). No és recomanable usar-la directament si no es té un coneixement profund de les matemàtiques i els algorismes de DL.

- **Lasagne.** És una llibreria en Python que facilita enormement la creació de xarxes DL sobre Theano; en Lasagne creem capes d'unitats i les connectem, oblidant-nos una mica de vectors i matrius, i gradients. És una de les millors opcions actualment.

- **Keras.** Com Lasagne, es tracta d'una altra llibreria d'alt nivell en Python, i en aquest cas pot funcionar tant sobre Theano com sobre Tensorflow. Facilita molt la creació de xarxes neuronals.

- **nolearn.** És una llibreria en Python que treballa sobre Lasagne per facilitar encara més les coses, encara que no compensa utilitzar-la excepte per a exemples bastant senzills, ja que es perd fàcilment el control sobre el que s'està fent.

- **Tensorflow.** És l'alternativa de Google a Theano, també en Python. Té alguns problemes d'eficiència amb matrius grans.

- **Caffe.** És una llibreria de visió artificial en Python que solament suporta xarxes convolucionals.

- **Torch.** És una llibreria en Lua. A favor seu es pot dir que la utilitzen Facebook i Twitter, entre altres companyies.

- **DL4J.** És una llibreria de DL per a Java. Els seus promotors addueixen que és l'única llibreria preparada per a l'entorn empresarial (es considera que Python pertany a l'entorn científic).

Com veieu, hi ha una gran varietat d'opcions, i amb tota seguretat continuaran apareixent noves llibreries tenint en compte la importància que està adquirint el DL.