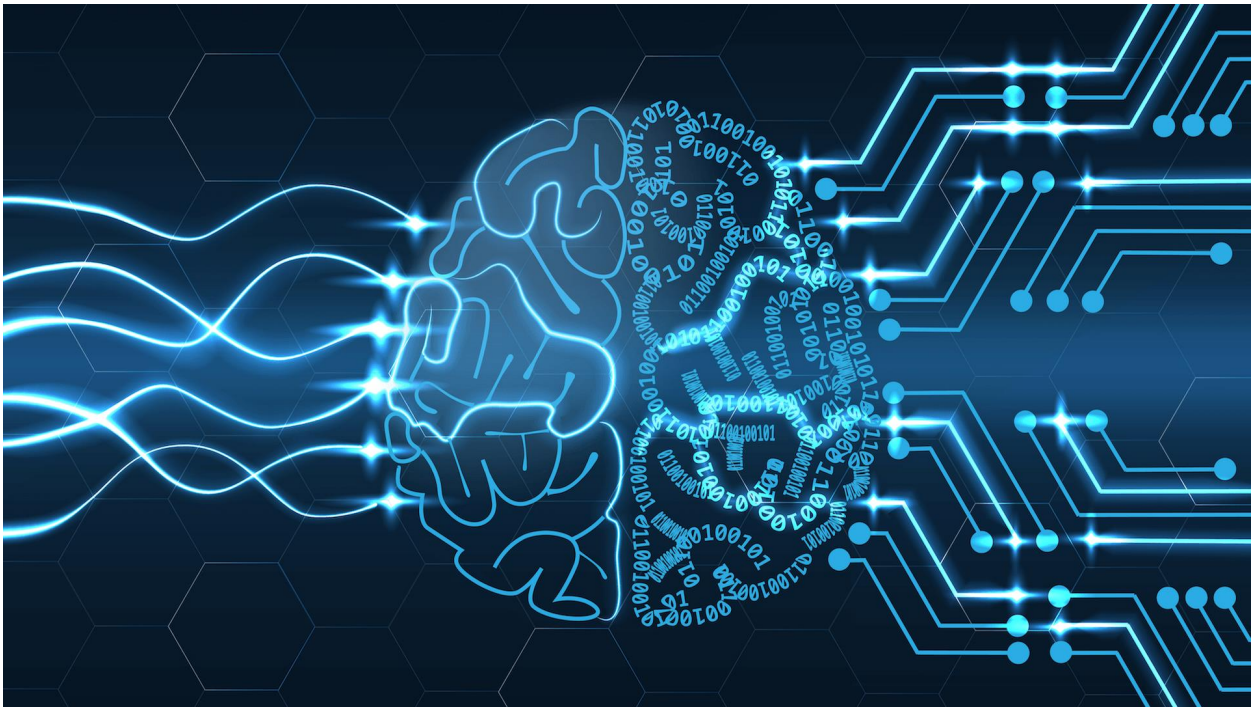
INTEL·LIGÈNCIA ARTIFICIAL

Disseny d'algorismes intel·ligents pel joc de les dames i el problema del viatjant



Pseudònim: DeepBlue

Curs acadèmic 2020-2021

24/10/2020

ÍNDEX

OBJECTIUS I INTRODUCCIÓ

El principal objectiu d'aquest treball de recerca és cercar informació sobre la intel·ligència artificial per així poder aplicar els coneixements en dos casos pràctics:

- Comparar un programa el codi del qual contingui algorismes d'intel·ligència artificial amb un que no en tingui i que faci calcular a l'ordinador totes les possibilitats.
- Fer un programa que jugui a les dames amb algorismes d'intel·ligència artificial. Programar el codi i una interfície que faci possible jugar amb comoditat.

Finalment, analitzar el resultat de les pràctiques:

- Fer una comprovació del funcionament dels programes. També es podria estudiar, encara que queda fora d'aquest treball, la seva eficiència i eficàcia.

La primera part del treball és el marc teòric. En aquest apartat, en primer lloc, es fa una petita introducció del significat d'Intel·ligència Artificial i la seva evolució. A continuació es fa una explicació més tècnica dels fonaments del treball i els algorismes utilitzats. S'ha intentat fer una explicació el màxim d'intuïtiva, però suficientment específica per després entendre la part pràctica.

La part pràctica del treball es centra en dos casos: el problema del viatjant o TSP i el joc de les dames. Al primer cas es comprova la millora que suposa la Intel·ligència Artificial en la resolució de problemes. Amb aquest objectiu, es comparen algorismes intel·ligents amb un que calcula totes les possibilitats.

En la segona pràctica, per veure la presència de la Intel·ligència Artificial en un problema més complicat, s'elabora un programa capaç de jugar al joc de les dames. Aquesta part és la més important ja que s'ha programat tot el codi implementant sistemes intel·ligents.

Finalment, es fa una valoració dels resultats obtinguts a les pràctiques a l'apartat de conclusions. S'avaluen els resultats dels programes del problema del viatjant i s'avalua l'eficàcia i el funcionament del joc de les dames.

METODOLOGIA DEL TREBALL

1. METODOLOGIA

En aquest capítol s'expliquen les eines utilitzades per elaborar la totalitat del treball. S'ha dividit l'explicació de la metodologia en tres apartats diferents. Cada apartat es correspon amb una part important del treball i es troben explicats els diferents recursos que s'han fet servir.

1.1. Fonaments teòrics

Abans de desenvolupar la part pràctica s'ha hagut de fer una cerca bibliogràfica per tal d'assimilar alguns conceptes teòrics. Amb aquest objectiu, s'ha fet servir Internet i un manual d'Intel·ligència Artificial en paper com a fonts principals d'informació.

Les idees més bàsiques com la definició d'Intel·ligència Artificial o la seva evolució durant la història s'han buscat a pàgines d'Internet i s'ha fet un recull de la informació més important. Aquest recull de conceptes essencials es troba a l'apartat: *Introducció a la Intel·ligència Artificial*. Ara bé, els fonaments més tècnics del treball, com el tractament de problemes o els tipus d'algorismes, s'han trobat al manual i s'expliquen als apartats: *IA i resolució de problemes* i *Classificació d'algorismes*

1.2. Metodologia d'aprenentatge

Amb l'objectiu d'elaborar i entendre els programes utilitzats a la part pràctica, s'ha hagut de passar per un procés d'aprenentatge de programació. Aquesta part no es veu reflectida al treball, ja que no és un treball de programació, sinó d'Intel·ligència Artificial. Tot i això, l'aprenentatge ha estat fonamental i se n'ha d'explicar la metodologia.

En primer lloc, es va seguir un tutorial d'iniciació a la programació de la pàgina web w3schools[1]. En aquesta fase es van començar a escriure els primers programes, de moment molt bàsics, per aprendre les diferents estructures de variables, condicionals, classes i objectes... A continuació, es va fer servir el manual esmentat anteriorment per

[1] Link al tutorial: w3schools.com/python

aprendre algorismes i sistemes més orientats a la IA. Tot això ha format part del procés de formació per, finalment, aplicar tots aquests coneixements en l'elaboració del programa del joc de les dames.

1.3. Programació: el llenguatge Python

Ara bé, quines eines s'han utilitzat per programar? Bàsicament s'ha fet servir el llenguatge de programació Python i un intèrpret interactiu. Es podria fer servir qualsevol llenguatge de programació per fer aquest treball, ja que no hi ha cap especialitzat en IA. Tot i això hi ha característiques del Python que fan que ressalti respecte als altres.

Il·lustració 1. Emoticona de Python

Principalment s'ha triat aquest llenguatge perquè és molt intuïtiu i per començar a programar és dels millors. L'objectiu fonamental de Guido van Rossum, creador de Python, era fer que el seu llenguatge fos fàcil de comprendre. La sintaxi concisa i senzilla permet que el codi ocupi el mínim de línies. Tot i que avui en dia tots ja ho són, cal aclarir que és un llenguatge de programació orientat a objectes[2], el que permet un codi molt més encapsulat[3]. A més a més, és molt potent pel que fa a l'emmagatzematge i manipulació de llistes de dades. Això serà molt útil per la part pràctica, on s'hauran d'enregistrar grans quantitats de dades, prediccions, possibilitats...

[2] Paradigma de programació basat en el disseny de classes que permeten crear objectes. Els objectes contenen paquets de propietats i mètodes (funcions executables) descrits a la seva classe. Això permet fer un codi molt més encapsulat.

[3] Un codi és encapsulat quan està dividit en porcions (càpsules) independents el màxim d'autònomes possibles. D'aquesta manera el codi es pot reutilitzar sense repetir-lo, facilitant el manteniment i fent-lo més estructurat i senzill. Gràcies a tot això es disminueix el risc de cometre errors.

Tot i que té moltes avantatges, es tracta d'un llenguatge interpretat. Això vol dir que cada vegada que s'executa un programa l'intèrpret el compila al moment (el tradueix a un llenguatge comprensible per l'ordinador). Per aquest motiu és menys eficient, però els ordinadors ara ja són prou potents per processar els programes que s'utilitzaran en aquest treball. Malgrat això, aquesta característica té un avantatge significatiu: no cal preocupar-se de compilar el programa cada vegada que volem fer proves.

Per poder escriure el codi molt més còmodament necessitarem un intèrpret interactiu. Amb aquest programa es poden fer proves (debugger[4]), ajudar a escriure sense cometre errors de sagnat o sintaxi, estructurar el codi i en general és molt visual. Per poder programar s'haurà d'instal·lar el programa Python a l'ordinador a través de Python.org i seguidament l'intèrpret interactiu que més agradi. Tots els programes en aquest treball s'han escrit a través d'*Spider,* un intèrpret interactiu molt còmode i visual per Python.

[4] El debugger és una eina de l'intèrpret interactiu que permet executar el codi per parts.

MARC TEÒRIC

Durant aquest capítol s'exposarà tota la informació necessària sobre la IA i programació per comprendre la part pràctica. Primer es farà una introducció a la Intel·ligència Artificial. A continuació es descriurà el procediment de resolució de problemes i les seves característiques i finalment s'explicarà com es classifiquen els diferents algorismes.

2. INTRODUCCIÓ A LA INTEL·LIGÈNCIA ARTIFICIAL

2.1. Què és la Intel·ligència Artificial?

Si busquem una descripció concreta de què és la Intel·ligència Artificial, es troba que és un terme bastant més complex del que sembla. No hi ha un consens, i molt menys una descripció clara i concisa que ens permeti saber quins programes són intel·ligents. No s'ha aconseguit cap mètode acceptat per tothom per a comprovar si una màquina és intel·ligent o no. Es podria dir que un programa o una màquina amb intel·ligència és aquella que respon de manera intel·ligent, com ho faria un humà. Però que vol dir que un humà és intel·ligent? El problema és que no sabem ni descriure el concepte d'intel·ligència amb certesa. Considerem que un humà és intel·ligent perquè és capaç de raonar, aprendre i prendre decisions tenint en compte diversos factors. Però, en què es diferencia la manera de pensar i actuar d'un animal que considerem que no és intel·ligent?

A més a més, la intel·ligència artificial ha evolucionat molt des de que va aparèixer fins a l'actualitat. Això fa que quedi encara més borrosa una possible descripció. A mesura que les màquines són cada vegada més capaces, sistemes considerats intel·ligents anteriorment, ja no ho són. Per exemple, el reconeixement de caràcters ja no es fa servir com a exemple d'un sistema intel·ligent. Això passa perquè els cànons actuals ja són molt més complexos en comparació amb la simplicitat d'aquest tipus d'algorismes.

Al llarg de la història, diferents personatges han buscat una explicació del que és la intel·ligència artificial. John McCarthy el 1995 afirmà que "és fer que una màquina es comporti d'una manera que seria considerada intel·ligent en un ésser humà". Andreas Kaplan i Michael Haenlein consideren que és "la capacitat que té un sistema per interpretar dades externes correctament, aprendre d'aquestes dades, i fer servir els

coneixements adquirits per completar tasques i assolir objectius específics mitjançant una adaptació flexible". Aquestes definicions s'ajusten bastant bé a la idea que tenim d'una computadora intel·ligent, ja que són bastant actuals.

Encara que una computadora actuï com un humà davant d'alguns problemes, això no vol dir que dins seu es reprodueixin els mateixos processos mentals. El que passa és que a través de tècniques d'IA es fa que la computadora tingui certes qualitats humanes. Fins al moment és impossible fer que una computadora pensi o tingui sentiments com un humà, perquè això implicaria saber com pensa realment un humà. Els coneixements que tenim fins al moment són suficients per seguir investigant com crear i millorar algorismes que responguin davant dels estímuls d'una manera cada vegada més semblant a un humà.

Per últim, el model racional és aquell que es centra en que una màquina actuï racionalment i basa les seves tècniques en la lògica i els agents intel·ligents. Aquests són entitats capaces de rebre percepcions a través de sensors i processar-les a través d'una sèrie de normes i condicions que es basen en la racionalitat. L'agent donarà una resposta depenent de si les dades són més o menys completes i del temps de què es disposa. Últimament s'estan investigant els *sistemes multiagent*, agents intel·ligents amb diferents capacitats que cooperen per solucionar un problema.

2.2. Test de Turing

La descripció més interessant de la IA és la d'Alan Turing, qui és considerat el pare de la informàtica. El matemàtic i informàtic anglès va ser el primer en intentar descriure-la. En un article publicat el 1950, proposà que si una màquina actua com un humà, podrem concloure que és intel·ligent. Això es basa en una premissa que hem de donar per vàlida: que l'humà és intel·ligent i si no fos així seria impossible que es fes una màquina intel·ligent.

A partir d'aquest punt, Turing no es va posar a argumentar què vol dir que siguem intel·ligents. En comptes d'això, va proposar un test que comprovaria si una computadora és intel·ligent: el Test de Turing. Aquesta prova consisteix en que una persona es comuniqui a través d'un terminal informàtic amb una entitat situada a una sala contigua que pot ser una persona o una computadora intel·ligent. Si després d'una conversa, la

persona no és capaç de dir amb certesa si està parlant amb un humà o amb una màquina, la podrem considerar intel·ligent en el cas de que sigui una màquina.

El Test de Turing, encara que va ser formulat fa 70 anys, és important en l'actualitat. Aquest exigeix a la màquina una sèrie de qualitats que encara avui conformen la majoria de requisits que ha de tenir una computadora per a ser intel·ligent. Per poder passar el test, la màquina ha de tenir les següents qualitats: reconeixement del llenguatge natural, raonament, aprenentatge i representació del coneixement.

El **reconeixement del llenguatge natural** és una qualitat imprescindible per a la computadora. Aquesta ha de ser capaç de reconèixer la informació que li transmetem, analitzar-la i trobar una resposta. En principi una computadora podria processar qualsevol llenguatge natural, però en els programes que es fan servir (siri, google assistant...) només les llengües més utilitzades es processen. El llenguatge es pot processar en forma de cadenes de text, àudio o fins i tot signes, però el més avançat és el de cadenes de text. L'àudio i els signes només afegeixen la complicació d'haver de passar un so o signe a text, el que moltes vegades porta a cometre errors.

Els humans són capaços de tenir unes percepcions i, a través d'uns processos mentals, treure conclusions o respostes. Així doncs, a la informàtica, a aquests processos mentals els anomenem **raonaments automàtics.** Aquesta part és la més important ja que és on la computadora ha d'aplicar certs mecanismes d'inferència, mecanismes pels quals aquesta obté una resposta a partir d'unes premisses donades. Aquests mecanismes poden ser implícits al codi o que a través de l'experiència la pròpia computadora els hagi après. Les primeres vegades que es va voler implementar aquesta qualitat a una màquina va ser a través dels sistemes experts. Aquests arriben a conclusions lògiques a partir de premisses introduïdes a priori al sistema. Actualment es fan servir tècniques més eficaces, com les *xarxes probabilístiques*, que ajuden a trobar respostes tot i que hi hagi incertesa. Com el seu nom indica, aquesta tècnica busca la solució que és més probable que sigui vàlida.

Com s'ha dit anteriorment, la computadora pot tenir implícits uns sistemes de raonament, però si només es queda amb aquests la llista de situacions que pot solucionar i la validesa de les respostes és molt limitada. És per això que una màquina intel·ligent ha de tenir **aprenentatge automàtic**: ha de ser capaç d'adaptar-se i ser flexible davant les possibles

situacions. Per exemple, un nen aprèn des de molt petit que si es cau es fa mal, però per arribar a aquesta generalització s'ha hagut de caure varies vegades. Seria reproduir aquest funcionament, però amb una computadora, per així augmentar el rang de problemes que és capaç de solucionar i maximitzar la precisió de les respostes. Actualment es fan servir xarxes i mètodes probabilístics, així com sistemes que reprodueixen el funcionament de les *xarxes neuronals*.

Per tenir capacitat de raonament i aprenentatge, la computadora ha de tenir sistemes per emmagatzemar i recuperar les dades. Això s'anomena **representació del coneixement**. Aquesta branca de la IA estudia maneres de guardar les dades per poder accedir a la informació fàcilment i eficientment. A part, també s'ha de trobar la forma de passar les dades que ens donen a on les volem emmagatzemar. La computadora funcionarà bé quan pugui accedir a la informació de manera eficaç per treure conclusions.

Actualment s'ha desenvolupat encara més el Test de Turing i hi ha variants en que l'ordinador es comunica a través d'una videoconferència. Per això també es necessita que la computadora tingui visió i robòtica. Per passar el Test de Turing Total, la computadora ha de ser capaç de manipular objectes i tenir visió que en la IA s'anomena robòtica. En aquest treball no s'aprofundirà en aquesta branca de la IA perquè ens centrarem més en com programar computadores estàtiques.

2.3. Història de la IA

Els primers pensadors grecs ja parlaven d'autòmats capaços de pensar com humans. Però això només eren suposicions teòriques. Com ja s'ha explicat anteriorment, les idees de Turing van ser fonamentals per crear aquesta disciplina. El 1936 ja es va introduir el concepte d'algorisme, fet que va posar les bases per la computació moderna. En un article que va publicar el 1950, *Computing machinery and intelligence,* ja començava a parlar de la possibilitat de dotar d'intel·ligència a una computadora.

El 1956, després de la mort de Turing, es va fer servir per primera vegada el terme "Intel·ligència Artificial". Durant una conferència convocada per John McCarthy es van fer prediccions de futur sobre aquesta nova i prometedora disciplina, les quals mai es van complir. El 1957, Frank Rosenblat va dissenyar la primera xarxa neuronal artificial. En el transcurs dels següents 15 anys no es van fer pràcticament avenços. Hi havia

màquines que feien raonaments senzills, però no es va invertir temps ni diners durant més d'una dècada.

A la dècada dels 80, amb estudis sobre Sistemes Experts, semblava que es reprendria l'estudi de la IA, però tot i les inversions milionàries, les expectatives no es van complir. A meitat dels 80, es van fer noves investigacions sobre les xarxes neuronals. Al 1987 es van descriure per primera vegada els atributs d'un agent intel·ligent. Un any després es van establir els llenguatges de programació orientats a objectes. La computació va fer un salt gràcies a aquests nous llenguatges, que eren molt més avançats en encapsulació i molts altres aspectes. Com més encapsulat està un programa, més fàcil és fer canvis en aquest i mantenir-lo perquè cada part funciona per si sola i no ens hem de preocupar de la resta del codi a l'hora de fer un canvi.

A la dècada dels 90, es van seguir varies línies d'investigació: *xarxes probabilístiques, xarxes bayesianes (teoria de grafs), agents intel·ligents...* Aquestes i altres investigacions han portat a la IA, ja com a ciència i disciplina acceptada, a un altre nivell en que les seves aplicacions són cada vegada més comunes a les nostres vides.

Finalment, al 1997, la companyia IBM va programar una computadora intel·ligent, Deep Blue, que jugava als escacs. Aquesta computadora va ser capaç de derrotar al campió mundial d'escacs, Garri Kaspárov.

2.4. Present i futur

Tal com s'ha explicat a l'apartat anterior, la IA ha passat per una evolució des del seu començament. En un primer moment només era una idea teòrica, però amb el temps es va convertir en un repte pels investigadors. Amb els descobriments de Turing es va donar la primera empenta, però als últims 20 anys la IA ha passat a un altre nivell i no hi ha dubte que seguirà prenent més i més importància a la societat. Dotar d'intel·ligència una màquina ha sigut i segueix sent un dels majors reptes per a la humanitat.

Al 2000 es van començar a desenvolupar els *chatbots*, programes que simulen a persones amb qui es pot parlar. Tot i això, aquests *chatbots* no superaven el Test de Turing. Al 2006 es va celebrar un congrés en espanyol pels 50 anys de la IA. Al 2007 arriben els primers telèfons mòbils intel·ligents, un camp revolucionari on la IA té moltes implicacions. En general durant els últims vint anys han millorat molt els programes que

intenten superar les capacitats humanes. Per exemple: el 2009 es van començar a desenvolupar computadors capaços de reconèixer sentiments. El 2016 una computadora va guanyar a l'actual campió mundial del joc Go.

Malgrat tots els avenços en la IA, el Test de Turing segueix resistint a ser solucionat. El 2014 es va fer una prova amb 150 jutges que parlaven amb un ordinador. Aquests havien d'esbrinar si estaven parlant amb un nen ucraïnès de 13 anys o amb una computadora. Aquesta prova passava parcialment el Test de Turing, ja que 30 dels 150 jutges no sabien si era una computadora o un nen humà. Tot i que es van fent proves, encara queda bastant per assolir el repte del Test de Turing.

Encara que costa adonar-se'n, la IA cada vegada té més implicacions en la presa de decisions de la gent. Es veuen influenciades per les recomanacions que fa *Netflix* segons el que s'ha vist, o els anuncis que surten a internet segons el que s'ha buscat anteriorment. Empreses com Google o Microsoft ja han optat per ser "AI First"[5] i estan invertint molts diners en desenvolupar aquest camp. A l'actualitat, l'anàlisi de dades està esdevenint molt important (*Big Data*[6]). Si totes les dades a les que es pot accedir s'analitzen a través de sistemes d'IA, que gràcies al poder computacional dels ordinadors són molt més potents, la informació pot ser molt profitosa. Aquesta és la revolució que s'està produint a les grans empreses informàtiques i per això s'està invertint tant.

Com s'ha vist al llarg de la història, és molt difícil fer previsions en aquest camp. La majoria d'expectatives mai s'han complert. No es pot concretar amb claredat com evolucionarà la IA, però hi ha alguns camps molt prometedors. És molt possible que es desenvolupin molt més els sistemes multiagent, les tècniques probabilístiques, els assistents personals (google assistant, siri...), cotxes autònoms, drons o avions no tripulats. Totes aquestes empreses investiguen com desenvolupar sistemes intel·ligents, però també n'hi ha d'altres que van més enllà i exploren nous models intel·ligents molt més revolucionaris. En concret l'empresa Numenta està investigant un nou model de màquina intel·ligent basat en el funcionament del neocòrtex humà. Si aquest projecte i molts altres triomfaran no ho podem saber.

[5] Les empreses "AI First" es centren en fer investigacions i fer servir nous sistemes d'IA.

[6] Les dades massives o "Big Data" són el gran conjunt de dades, processos informàtics i aplicacions que per la seva varietat i velocitat de processament sobrepassen la capacitat del sistemes informàtics habituals.

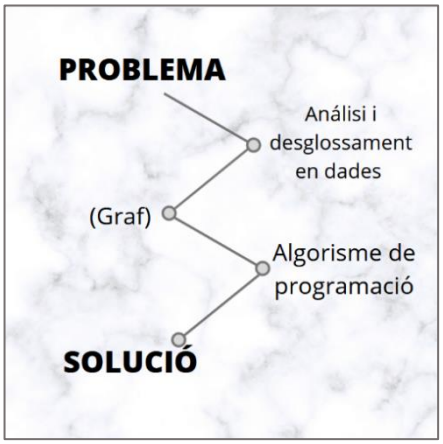
3. IA I RESOLUCIÓ DE PROBLEMES

En aquest apartat i el següent s'expliquen els fonaments més tècnics del treball. Tota aquesta informació servirà per entendre la part pràctica i poder analitzar-la amb més exactitud. A partir d'aquí comencen a aparèixer alguns conceptes específics de programació i teoria de grafs. La primera vegada que s'introdueixin aquests termes es donarà la descripció i els que apareixen més estan explicats a l'apartat: *Glossari de conceptes específics*. A continuació s'exposa com tractar amb problemes per trobar una solució des de la computació i la intel·ligència artificial.

3.1. Resolució de problemes

La IA és un camp molt extens on s'estudien sistemes per aconseguir que una màquina o computadora tingui qualitats intel·ligents (semblants a les humanes) o actuï d'una manera que s'assembli al màxim a la d'un humà. És per això que hi ha molts fronts d'investigació oberts. Per exemple: s'investiga com fer que una màquina actuï com si tingués sentiments, també com fer que una computadora contesti com ho faria un humà o com fer que una màquina solucioni problemes de manera intel·ligent.

Per acotar el tema, aquest treball es centra en aquesta última capacitat. Per tant, es dirigirà la recerca en aquest sentit. A través d'algorismes[7] d'IA i fent ús de la programació es pot fer una part pràctica interessant sobre resolució de problemes. Per aquest motiu aquest treball es centra en aquest camp de la IA.



Il·lustració 2. Procés de resolució d'un problema

En aquest apartat s'explica com tractar alguns tipus de problemes. Amb aquest objectiu, es reduiran els problemes a un grup de dades principals. També es representaran juntament amb el procés de resolució a través sistemes de representació del coneixement.

[7] Un algorisme és un conjunt d'instruccions, passos o processos que tenen l'objectiu de trobar una solució a un problema. Als algorismes informàtics aquestes instruccions es troben escrites en un llenguatge de programació al llarg d'un codi.

Una vegada analitzat, es podrà trobar una solució al problema amb algorismes de programació.

3.2. Característiques dels problemes

La resolució de problemes és una capacitat que es considera intel·ligent. Els humans són capaços de resoldre'n de molts tipus: mots encreuats, trobar el camí més ràpid en un mapa, decidir la feina més convenient… En aquest camp, la IA té l'objectiu d'aconseguir que una computadora resolgui algunes d'aquestes qüestions. És obvi que existeixen problemes de molts tipus amb molts factors i característiques. Per poder buscar una solució s'haurà de definir qualsevol qüestió de manera que es pugui solucionar automàticament. Es trobaran certes característiques que es generalitzen en la majoria dels problemes.

S'ha de trobar:

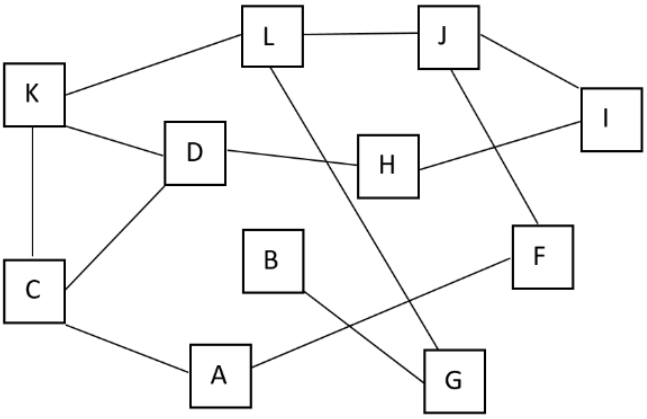
- Un punt de partida
- Un objectiu
- Accions que podem realitzar per trobar diferents solucions al problema (operadors).
- Una funció[8] d'avaluació, que avalua la qualitat de la solució trobada o si és la millor.
- Altres restriccions rellevants definides pel domini del problema[9]

[8] Una funció en programació és un conjunt de línies de codi que executen un seguit d'accions i normalment retornen alguna cosa. Aquesta és la manera de dividir el codi per tal que sigui més ordenat i encapsulat.

[9] El domini d'un problema és un conjunt de conceptes interrelacionats que enuncia el problema i que són necessaris per proposar una solució adequada.

PROBLEMA 1:

Trobar la ruta amb menys desplaçaments per passar per tots els punts sortint del punt F i tornant-hi al final. Els desplaçaments es poden fer en qualsevol sentit i només pels camins descrits al graf[10].



Il·lustració 3. Graf del Problema1

Per aquest problema el punt de partida és punt F i l'objectiu és la combinació de lletres que descriu el camí pel que s'ha d'anar per passar per tots els punts i tornar al F. Els operadors que es poden aplicar són anar d'un punt a un altre. La funció d'avaluació determinaria si la solució és la combinació amb la que es facin menys desplaçaments estrictament. Per últim, hi ha una altra restricció: només es pot anar d'un punt a un altre seguint els camins descrits.

PROBLEMA 2:

Ordenar una llista que conté els valors 1-4 desordenats per tal que quedi [1, 2, 3, 4]. Per ordenar la llista, només podem canviar de lloc els nombres que estan de costat. La solució ha de ser la combinació de moviments per ordenar la llista donada, però s'han de fer el mínim de moviments.

Per aquest problema el punt de partida és la combinació donada (per exemple [4, 3, 1, 2]). L'objectiu és la combinació de moviments per ordenar la llista. Hi ha tres operadors que es poden anomenar A, B i C, on A intercanvia els dos primers termes de la llista, B els del mig i C els dos últims. La funció d'avaluació determinaria si la solució trobada és la més curta[11] possible per tal d'ordenar la llista. Observant el domini del problema es pot deduir que la llista donada ha de ser una combinació dels nombres 1-4 i que els hem d'ordenar de forma ascendent.

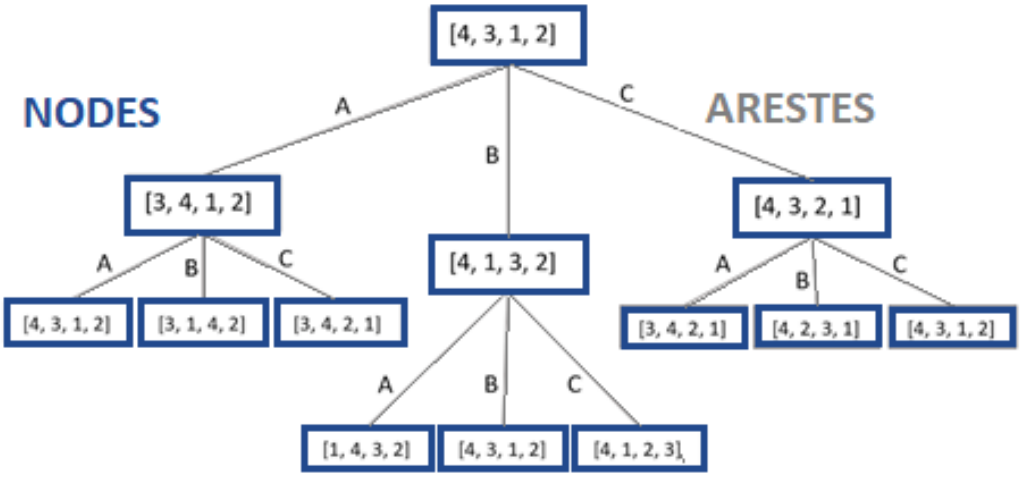
[10] Un graf és un gràfic d'un conjunt de vèrtexs o nodes i arestes que els uneixen. Serveix per modelitzar relacions entre parelles d'objectes. Això s'explica amb més profunditat al següent apartat.

[11] La solució serà la més curta quan es passi pel mínim de nodes.

3.3. Representació de problemes: grafs

Moltes vegades és necessari fer una representació gràfica del problema per tal de tenir la seva estructura ordenada i clara. Normalment, amb aquest propòsit, es fan servir els grafs. Hi ha molts tipus de representacions gràfiques, però els grafs tenen unes característiques determinades: es componen per nodes i arestes que els uneixen. Els nodes representen objectes, dades, estats[12], i les arestes modelitzen relacions binàries entre aquests. Les arestes poden representar operadors o tràmits que transformen un node en un altre o simplement els "camins" per arribar d'un a l'altre.

A vegades el mateix problema ja ens el dona el graf, perquè és necessari per explicar-lo (*problema 1*). Però la majoria de vegades s'haurà de construir aquesta representació a partir de l'enunciat. Per fer-ho, es descartarà tota la informació innecessària de l'enunciat, es desglossarà en les característiques principals i es formalitzarà el problema en un conjunt de nodes i arestes. Per exemple, al *problema 2* la combinació inicial serà de la que sortiran tots els altres nodes. Però com es pot passar d'un node a un altre? A través dels tres operadors. Sabent totes aquestes dades ja es pot construir el graf.



Il·lustració 4. Graf de tipus arbre del Problema 2

[12] Un estat és un conjunt de conceptes o dades que descriuen el problema en un moment determinat. Per exemple, si el problema fos un joc de taula, un estat és una posició de les peces al tauler.

Com es pot veure, aquest graf no és igual que el del *problema 1.* No tots els grafs són del mateix tipus; aquest és de tipus arbre[13]. De cada node surt una branca diferent. Per exemple, del primer node surten les tres branques principals. En aquest tipus de grafs no cal fer la representació fins a trobar la solució ja que amb el que es veu a la imatge ja és suficient. Els grafs serveixen per fer-se una idea de l'estructura del problema i així facilitar el procés de programar l'algorisme que el solucionarà.

La teoria de grafs en sí ja és una branca de les matemàtiques i la computació. Hi ha molts tipus de grafs i es poden classificar segons molts criteris. En aquest treball no s'aprofundirà en aquest àmbit ja que els coneixements bàsics són suficients per afrontar els problemes que es tracten. Ara bé, en el que sí s'incidirà més és en els algorismes per tractar grafs, en concret per recórrer arbres. D'algorismes n'hi ha molts (de cerca en amplitud, profunditat, de cerca local, heurística…). Però només s'explicaran els que es facin servir per solucionar els problemes.

3.4. Estratègies de cerca per problemes

Les tècniques per tractar els problemes des de la computació s'anomenen de forma genèrica estratègies de cerca. A través d'aquestes estratègies s'intenta trobar una solució acceptable dins de l'espai d'estats[14].

Normalment quan es presenta un problema s'analitzen les accions que es poden fer per arribar a l'estat desitjat. Depenent de les accions que s'executin s'arribarà a un estat millor o pitjor, però sempre hi haurà moltes combinacions d'accions que donen com a resultat molts possibles estats. A través de les estratègies de cerca s'exploraran aquestes combinacions per trobar la dona el millor estat possible.

[13] Els grafs "arbres" s'anomenen així per la seva estructura: d'un node en surten un número determinat més i d'aquests també en surten i així successivament. És com les branques d'un arbre que es van dividint.

[14] L'espai d'estats és el conjunt de tots els estats possibles per un problema.

4. CLASSIFICACIÓ D'ALGORISMES

Els algorismes que es faran servir més tard a la part pràctica són de cerca informada, no informada, cerca local o per jocs. Degut a això, seguidament s'explicarà aquesta classificació d'algorismes segons les seves característiques i objectius. D'aquesta manera quan s'aprofundeixi més en cada un dels algorismes es tindrà un marc teòric previ sobre el tipus de sistema del qual s'està parlant.

4.1. Cerca no informada

Els algorismes de cerca no informada són aquells que no depenen de la informació i característiques pròpies del problema que s'està tractant. Això vol dir que no es disposa d'aquests criteris que podrien ajudar a saber com descartar solucions o trobar la millor solució fàcilment. Degut a això, la cerca no informada proporciona mètodes generals per recórrer arbres d'espai d'estats aplicables a moltes circumstàncies.

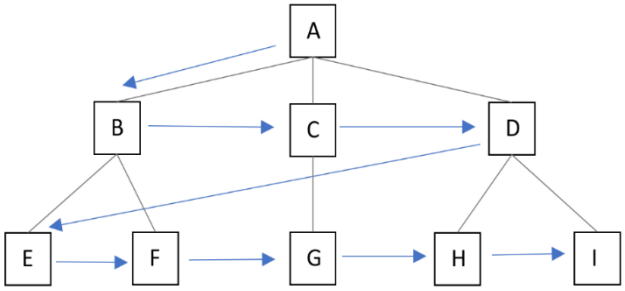
El problema d'aquests algorismes és que en general són exhaustius: si cal s'explora tot l'espai d'estats per trobar la solució al problema. Això fa que el seu cost sigui molt elevat si s'intenten implementar a problemes reals on aquest espai d'estats acostuma a ser immens. D'altra banda, tenen l'avantatge que com que no és necessari conèixer informació addicional, sempre són aplicables.

Com ja s'ha explicat anteriorment, hi ha moltes formes de representar els problemes. Normalment, la cerca no està enfocada a l'anàlisi d'arbres. Els dos algorismes principals de cerca no informada són la cerca en amplitud i la cerca en profunditat. Per aquestes cerques es començarà amb un node arrel[15], que és una solució al problema que acostuma a ser aleatòria. A partir d'aquest node haurem de descriure uns operadors per fer canvis en aquesta solució. Quan aquests operadors es vagin aplicant, es creen nous nodes i l'arbre es va ramificant.

[15] El node arrel és el primer node d'un arbre, del qual deriven la resta de nodes.

4.1.1. Cerca en amplitud

Si l'algorisme visita els nodes per capes parlarem de cerca en amplitud. Aquest algorisme visita el node arrel, tot seguit l'expandeix[16] i visita tots els seus fills[17], els expandeix i visita els nodes nous i així successivament. D'aquesta manera, es va explorant l'arbre nivell per nivell.



Il·lustració 5. Representació cerca en amplitud

Aquest recorregut s'implementa a través d'una cua *First In First Out* (FIFO) per la llista de nodes pendents per visitar o llista de nodes frontera. Això vol dir que els nodes s'extreuen de la cua en l'ordre que van ser afegits; el primer node afegit serà el primer en ser extret. El pseudocodi de cerca en amplitud es veu a la *Il·lustració 6*.

```
node_arrel = estat_inicial
nodes_frontera = llista tipus FIFO
nodes_visitats = llista
emmagatzemar node_arrel a nodes_frontera
mentre nodes frontera no buit:
    node_actual = extreure primer node de nodes_frontera
    si node actual és == solució:
        sortir amb la solució
    introduir node_actual a nodes_visitats
    per cada operador:
        node_fill = operador(node_actual)
        si node_fill no a nodes_visitats ni a nodes_frontera:
            afegir node_fill a nodes_frontera
```

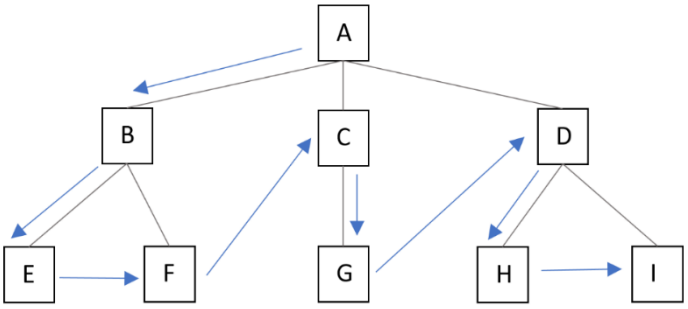
Il·lustració 6. Pseudocodi cerca en amplitud

[16] En programació, expandir un node vol dir generar els seus nodes fills.

[17] Un node fill és un node que prové d'un altre node, que serà el node pare d'aquest.

4.1.2. Cerca en profunditat

En canvi, si l'algorisme baixa per una branca de l'arbre fins arribar a l'últim nivell, es tracta de cerca en profunditat. Aquest algorisme visita el node arrel, l'expandeix i visita el primer fill, expandeix aquest fill i visita el primer dels seus fills i el torna a expandir. Fent aquest procés successivament, s'arribarà fins el fons de l'arbre abans d'explorar altres branques.



Il·lustració 7. Representació cerca en profunditat

La cua de nodes frontera en aquest cas és *Last In First Out* (LIFO). Això vol dir que l'últim node que s'afegeix és el primer que s'extreu de la cua. Fent servir aquesta llista, encara que hi hagi un node més antic, primer s'analitza l'últim que s'afegeixi. D'aquesta manera, es baixarà en profunditat com ja s'ha explicat. El pseudocodi de cerca en profunditat amb cua LIFO es veu a la *Il·lustració 8*:

```
node_arrel = estat_inicial
nodes_frontera = llista tipus LIFO
nodes_visitats = llista
emmagatzemar node_arrel a nodes_frontera
mentre nodes frontera no buit:
    node_actual = extraure l'últim node afegit a nodes_frontera
    si node actual és == solució:
        sortir amb la solució
    introduir node_actual a nodes_visitats
    per cada operador:
        node_fill = operador(node_actual)
        si node_fill no a nodes_visitats ni a nodes_frontera:
            afegir node_fill a nodes_frontera
```

Il·lustració 8. Pseudocodi cerca en profunditat

4.2. Cerca informada

Hi ha problemes que per la forma en què estan plantejats o per la seva naturalesa ens aporten informació que es pot utilitzar per optimitzar la cerca. En aquests casos es faran servir algorismes de cerca informada. Aquests sistemes aprofiten aquesta informació per depreciar branques[18] de l'arbre poc prometedores i així centrar-se en altres que ho són més.

La cerca informada s'implementa a través d'una funció heurística. El que fa aquesta funció és avaluar si un node fill s'apropa més a la solució que el node pare. D'aquesta manera si s'ha empitjorat la solució es pot ignorar aquesta branca. És molt important trobar una bona funció heurística perquè aquesta determinarà la rapidesa de l'algorisme.

Posem com a exemple el TSP. L'objectiu és trobar la ruta més curta per passar per totes les ciutats partint d'una ruta aleatòria. Una possible funció heurística podria determinar si la nova ruta generada recorre més km que l'anterior i si és així deixar d'explorar la branca.

El problema d'aquest procediment és que hi ha la possibilitat de que, encara que un nou estat sigui pitjor que el seu pare, en els pròxims nivells millori considerablement. Per evitar això es pot fer servir la funció heurística juntament amb un sistema de tornada enrere o *backtracking* en anglès. Aquest sistema detecta quan una branca ja no és útil gràcies a la funció heurística, la descarta i torna enrere per explorar altres branques (d'aquí ve el seu nom).

Altres sistemes heurístics de cerca informada són l'algorisme A* i la cerca local.

4.3. Cerca local

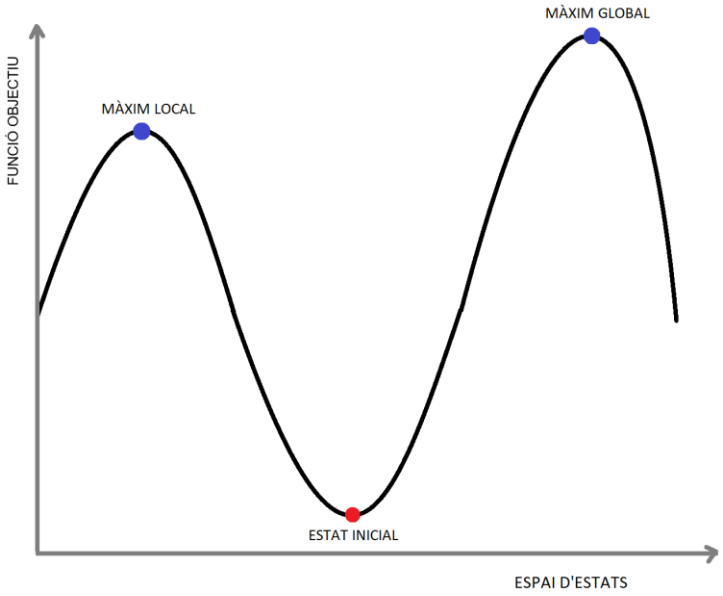
La cerca local és un tipus de cerca informada. Existeixen molts algorismes de cerca local que s'adapten de manera diferent als problemes, per exemple el *hill climbing*, *simulated annealing*, algorismes genètics, la cerca tabú, entre d'altres. Tot i això, parlant del seu funcionament, tots tenen algunes característiques comunes.

[18] Una branca d'un arbre d'estats en programació és tota la ruta que es genera a partir d'un node fins al fons de l'arbre.

Els algorismes de cerca local són la base de molts mètodes d'optimització. Això vol dir que la majoria de vegades es faran servir per trobar una solució bona[19] en un temps raonable. Molts problemes tenen un espai d'estats immens, que augmenta exponencialment, així que al tractar-los s'aspira a trobar una solució òptima, ja que arribar a la millor és pràcticament impossible. La cerca local es centra en explorar l'espai d'estats, sense donar importància al camí fins a la solució.

La cerca local es pot veure com a un procés iteratiu[20] que comença amb un possible estat i el va millorant fent canvis locals. En concret l'algorisme busca els "veïns" (estats semblants de l'estat actual) a través d'uns operadors definits que respectin les restriccions del problema i així va millorant la solució. Aquests operadors són funcions que canvien d'alguna manera un estat i serveixen per moure's entre solucions veïnes. Per poder definir quan una solució és millor que una altra es necessita una funció d'avaluació que determini la qualitat d'aquesta, sense estar relacionada amb cap altre cost[21] necessàriament.

La *Il·lustració 9* mostra un esquema simplificat del funcionament de la cerca local. L'eix d'ordenades representa la funció objectiu[22] i el d'abscisses l'espai d'estats. Comencem des de l'estat inicial (punt vermell) i tenim l'objectiu de trobar el màxim valor de la funció d'avaluació. L'algorisme de cerca local generarà els dos veïns de l'estat inicial, punts consecutius a aquest. En aquesta situació hi ha



Il·lustració 9. Gràfica de l'espai d'estats en un problema

[19] La funció d'avaluació del problema determina com és d'òptim cada un dels possibles estats (solucions).

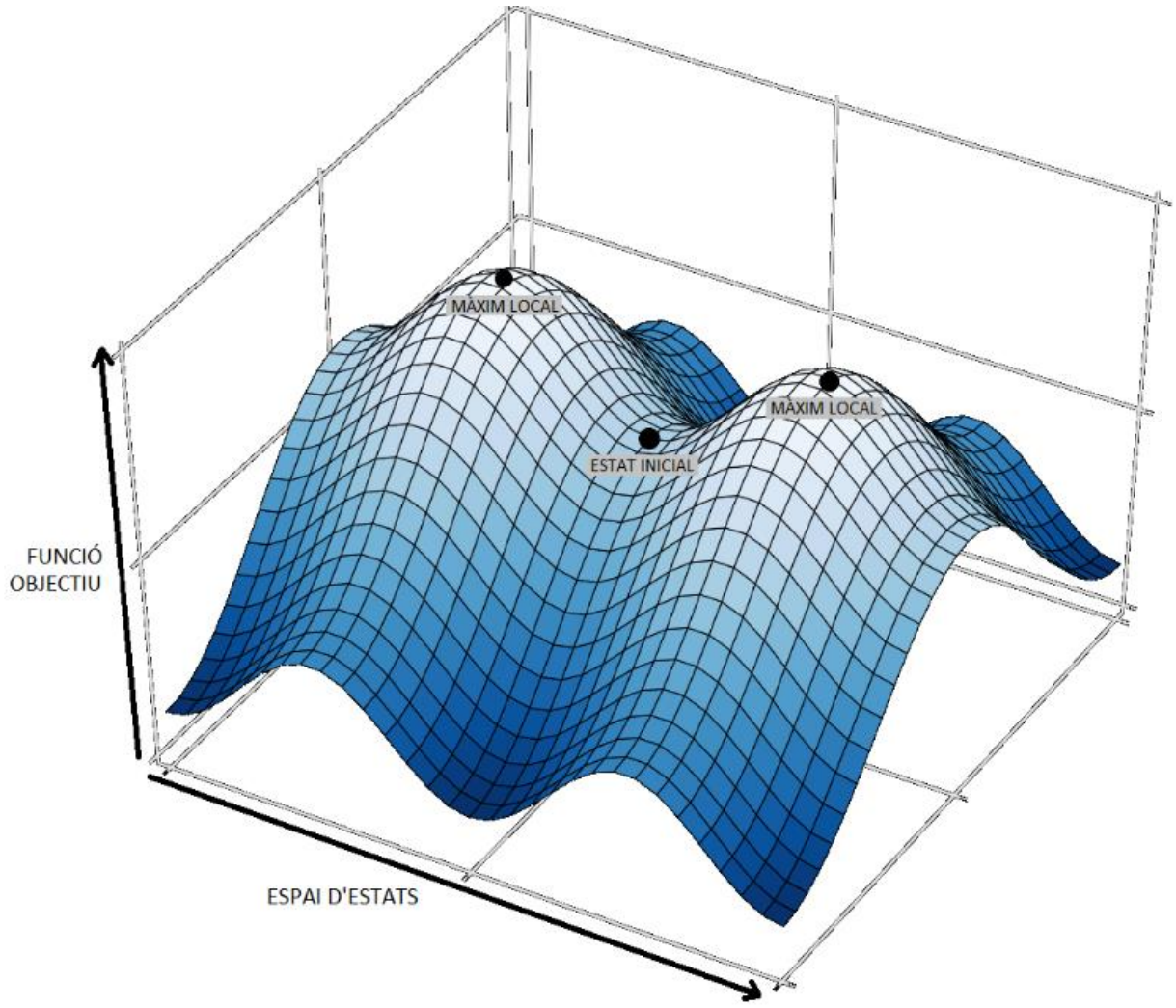
[20] Un procés és iteratiu quan es repeteix varies vegades amb l'objectiu de trobar una solució d'un problema.

[21] Es parla d'altres costos, com el cost espacial (quantitat de memòria consumida) o el cost temporal (quantitat de temps consumit).

[22] La funció objectiu és aquella el retorn de la qual va augmentant segons les solucions s'apropen a l'objectiu.

dues opcions: que esculli el veí dret o l'esquerre. Si escull l'esquerre arribarà a un pic i detectarà que ha arribat al màxim de la funció, però en realitat es tracta del que anomenem màxim local[23]. En canvi, si escull el dret arribarà al màxim global[24] (màxim de tota la funció). El problema d'aquests algorismes és que moltes vegades cauen en màxims locals sense arribar al global.

Normalment els problemes no són tant senzills ja que cada estat té més d'un veí. Conseqüentment, la funció és més complicada: té 3 dimensions. Un exemple podria ser la següent representació.



Il·lustració 10. Gràfica de l'espai d'estats d'un problema en 3D

[23] Un màxim local és el punt de la representació d'una funció on s'arriba al valor màxim per un interval determinat de la funció.

[24] El màxim global és el punt de la representació d'una funció on s'arriba al valor màxim de l'eix d'ordenades (altura màxima).

4.3.1. Algorismes per jocs

Els jocs són un camp molt interessant per investigar i millorar els algorismes de cerca. A més a més, suposen un gran repte ja que la majoria requereixen molta intel·ligència. Molts investigadors han intentat crear i optimitzar al màxim algorismes que juguen a jocs d'estratègia com els escacs. Aquest treball es centra en algorismes per jocs de les següents característiques:

- Dos jugadors s'enfronten amb objectius contraris.
- Els moviments són alterns: primer fa un moviment un jugador i tot seguit el seu contrincant.
- Jocs perfectament informats: es sap en tot moment si un moviment és perjudicial o beneficiós.

Les dames, els escacs, el 3 en ratlla o el connecta 4 tenen aquestes característiques.

L'algorisme més famós i òptim per aquest tipus de jocs és el Minimax. Es tracta d'una mena de cerca en profunditat que va recollint el valor final d'un conjunt de moviments per tal de decidir quin moviment és millor. A la part pràctica s'aprofundeix més en l'algorisme Minimax i en com optimitzar-lo.

PART PRÀCTICA

5. TSP: IA CONTRA PODER COMPUTACIONAL

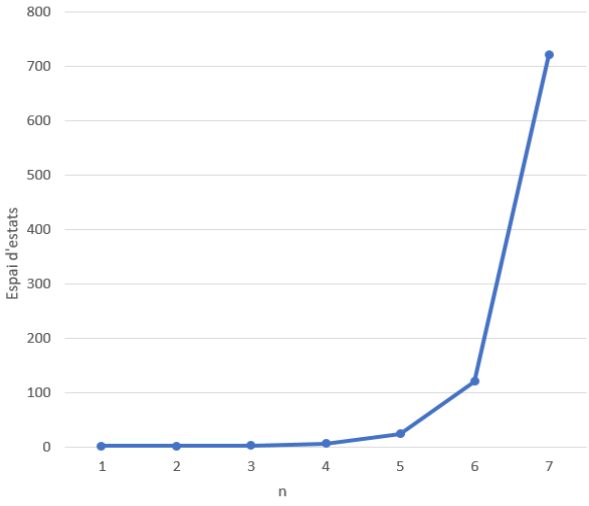
En aquest apartat s'estudiaran diferents enfocaments envers a la resolució del problema del viatjant (TSP). Aquesta part servirà com a experiment per comprovar el poder que té la IA. Es compararà fins a quin punt un algorisme amb intel·ligència artificial optimitza la cerca en comparació amb un algorisme que només fa servir el poder computacional[25]. Finalment es farà una valoració dels resultats obtinguts dels diferents algorismes.

5.1. El problema del viatjant: característiques

El *Problema del Viatjant o TSP*[26] és un problema clàssic estudiat per molts informàtics i matemàtics al llarg de la història. L'enunciat del problema és el següent:

Donat un nombre de ciutats n i la distància que les separa, quina és la ruta més curta que passa per totes les ciutats una vegada i torna a la ciutat d'origen?

Tot i que sembla un problema senzill, degut a la facilitat per entendre'l, trobar la millor solució pot ser molt complex. Es tracta d'un problema no tractable per la computació tradicional, ja que l'espai d'estats augmenta exponencialment a mesura que augmentem la quantitat de ciutats. Encara que es facin servir els ordinadors més potents, és impossible calcular tots els estats per a un nombre de ciutats moderat en un temps raonable.



Il·lustració 11. Gràfic d'estats possibles

[25] El poder computacional és la capacitat que té un ordinador per executar processos, tractar amb dades...

[26] Sigles de Travelling Salesman Problem. A partir d'ara es faran servir aquestes sigles per anomenar el problema del viatjant.

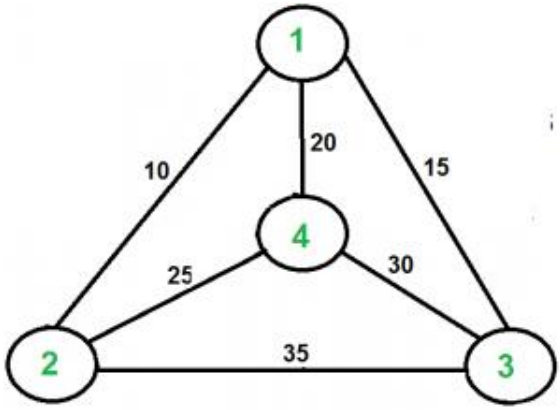
Això passa perquè la funció que calcula l'espai d'estats és el factorial del nombre de ciutats: n![27], la qual augmenta de manera exponencial, com es veu a la *Il·lustració 11*.

Ara bé, fent servir algorismes d'IA sí es poden trobar aproximacions bastant bones. Aquests sistemes aprofiten algunes característiques del problema per reduir l'espai d'estats.

El TSP és un problema molt general i fent-li algunes adaptacions pot tenir diverses aplicacions. Canviant els conceptes de distància i ciutat, es pot fer servir en l'optimització, logística i fabricació de circuits elèctrics o les cadenes d'ADN. Les ciutats serien els punts de soldadura, porcions de cadena d'ADN o clients, i les distàncies serien el cost, el temps de soldar o alguna mesura de similitud entre els fragments d'ADN. Aplicant el TSP per solucionar altres problemes s'afegeixen altres condicions i factors a tenir en compte, el que augmenta molt la complexitat del problema.

5.2. Anàlisi del problema

En primer lloc haurem de trobar una manera de representar el problema per a que l'ordinador pugui treballar. Depenent del problema serà més eficient un tipus de representació del coneixement o un altre. Per exemple, en aquest cas es pot construir un graf ponderat[28]. El graf per al TSP amb 6 ciutats podria ser el de la *Il·lustració 13*. També es pot representar com una taula de valors, com a la *Il·lustració 12*.



Il·lustració 12. Possible graf pel TSP

	1	2	3	4
1	0	10	15	20
2	10	0	35	25
3	15	35	0	30
4	20	25	30	0

Il·lustració 13. Representació del graf pel TSP en forma de taula

[27] El factorial d'un nombre s'expressa amb el signe ! i equival a multiplicar n per tots els nombres naturals que estan entre 0 i n, excloent 0.

[28] Un graf ponderat és aquell que té pes o cost associat a cada aresta.

Però en tots els algorismes de resolució, per tal de simplificar el problema, introduirem una llista de les ciutats i les seves coordenades i calcularem la distància en línia recta d'una ciutat a una altra. La funció d'avaluació i els operadors utilitzats són:

- La funció d'avaluació consisteix en una funció que va calculant les distàncies en línia recta entre les ciutats i les suma. També tindrà en compte la curvatura de la terra a l'hora de fer el càlcul. Com més petit sigui el valor que retorna la funció d'avaluació, millor és la solució ja que busquem la ruta més curta que compleix els requisits.
- Els operadors per trobar els nodes fills d'un node canvien de posició dues ciutats de la ruta.

Les rutes es representaran com a una llista de les ciutats per les que es passa fins a tornar a la ciutat d'origen. Per exemple, una possible solució pel TSP amb 8 ciutats seria:

[València, Sevilla, Santiago, Santander, Granada, Màlaga, Madrid, Salamanca]

A la majoria de les proves es faran servir aquestes ciutats. S'ha triat fer-ho amb 8 perquè és el límit de ciutats que fa que tots els algorismes solucionin el problema en un temps raonable. Tot i això, també es faran proves amb més ciutats.

5.3. Algorisme de poder computacional

La manera més intuïtiva de trobar la millor solució al TSP seria calcular totes les solucions i quedar-nos la millor. Aquest algorisme s'anomena *Naive Alorithm*, que literalment vol dir "algorisme ingenu", i segueix el següent procediment:

1. Generar totes les permutacions possibles (n! on *n* es el nombre de ciutats).
2. Calcular el cost de totes les possibles solucions i guardar la més curta.
3. Retornar la permutació que fa menys recorregut.

S'ha de fer servir algun sistema per trobar totes les permutacions possibles. Amb aquest propòsit s'aplicarà una cerca en profunditat que a partir del node inicial busqui totes les combinacions tal i com s'explica en l'apartat *4.1.2*, on es tracta la cerca en profunditat. La cerca s'atura en el moment que s'hagin visitat totes les permutacions.

Tot i que la cerca en profunditat és un tipus de cerca no informada, és un algorisme exhaustiu. Això vol dir que ha de calcular totes les possibilitats per trobar la solució. Així doncs, no hi intervé cap tipus de sistema d'intel·ligència artificial, només es fa servir el poder computacional de l'ordinador per trobar la solució.

Com ja s'ha explicat, el nombre de nodes possibles es correspon amb el factorial del nombre de ciutats de la ruta: n!. Això fa que la quantitat de nodes que ha d'explorar l'algorisme ingenu augmenti exponencialment quan anem afegint ciutats. En conseqüència, aquest algorisme només troba una solució en un temps factible si el nombre de ciutats és molt petit. Per exemple, amb 8 ciutats, el retorn de l'algorisme es mostra a la *Il·lustració 14*.

```
In [1]: runfile('C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Algoritmos tdr/
TSP_poder_computacional.py', wdir='C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Algoritmos
tdr')
['Valencia', 'Granada', 'Malaga', 'Sevilla', 'Salamanca', 'Santiago', 'Santander', 'Madrid']
Distància total: 2376.3731713936277 km
Temps d'execució': 274.1745159626 segons
```

Il·lustració 14. Retorn de l'algorisme de poder computacional

L'algorisme ha trigat uns 4 minuts i 34 segons en trobar la ruta més curta possible. Amb 8 ciutats encara és factible fer servir aquest algorisme però a partir de 10 el temps que tarda no és factible.

5.4. Hill climbing

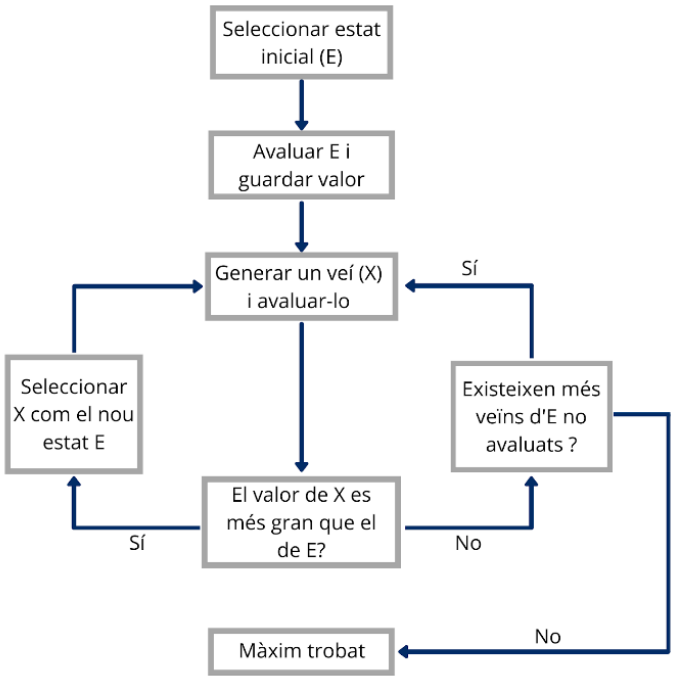
L'algorisme *hill climbing* és un dels més senzills de cerca local en termes de funcionament. També s'anomena "algorisme voraç" (*greedy* en anglès) perquè constantment busca un veí millor sense anar més enllà. Això té un avantatge important: troba un màxim en molt poc temps.

L'algorisme va analitzant estats veïns de l'estat actual i escull el primer que troba amb un valor millor a l'actual, segons la funció d'avaluació utilitzada. Aquest nou estat passa a ser l'estat actual i es torna a repetir l'operació. El procés acaba en el moment que s'arriba a un estat que no es pot millorar, és a dir, que no té cap veí amb un valor millor. Això vol dir que ha arribat a un màxim.

L'algorisme es diu literalment "escalada del turó". Si imaginem que l'espai d'estats és una funció semblant a la de la *Il·lustració 9* mostrada anteriorment, l'algorisme el que fa és

anar pujant una pendent (escalant) fins arribar al màxim global en el millor dels casos o un dels màxims locals.

La *Il·lustració 15* mostra el funcionament d'aquest algorisme.



Il·lustració 15. Funcionament del hill climbing

Ara bé, a aquest algorisme se li pot fer una millora molt important. Si només s'executa una vegada el procediment, agafant un node inicial aleatori, el més probable és que es trobi un màxim local. Ara bé, si es programa l'algorisme per tal que el procediment s'executi més vegades, augmenten molt les possibilitats de trobar el màxim global o si més no un màxim local millor. A més a més, com és un procediment molt senzill, repetir-lo no augmenta molt el temps d'execució. Aquest algorisme s'anomena *hill climbing iteratiu*, ja que fa el procés iterativament.

S'ha implementat l'algorisme *hill climbing iteratiu* amb la funció d'avaluació i operadors descrits anteriorment i el resultat obtingut es mostra a la *Il·lustració 16*.

```
In [36]: runfile('C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Búsqueda informada/
TSP_iter_hill_climbing.py', wdir='C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Búsqueda
informada')
['Salamanca', 'Sevilla', 'Malaga', 'Granada', 'Valencia', 'Madrid', 'Santander', 'Santiago']
Distància total: 2376.373171393628
Temps d'execució': 0.0109999180 segons
```

Il·lustració 16. Retorn de l'algorisme hill climbing

El procediment és pràcticament instantani perquè amb només 8 ciutats fent 2 iteracions ja troba la millor solució possible a totes les execucions.

5.5. Simulated annealing

El major perill en la cerca de solucions pel TSP, com ja s'ha comentat a l'apartat anterior, és quedar-se atrapat als màxims locals. Per escapar d'aquests màxims, hi ha alguns algorismes, que s'explicaran a continuació, que utilitzen sistemes per fer més variable l'elecció del node nou. Així doncs, no sempre s'escollirà el primer node que millori a l'anterior. A vegades és possible que s'esculli un node que empitjora la solució anterior. Això es fa perquè hi ha la possibilitat que un node el qual a priori empitjori a l'anterior finalment resulti en un màxim millor.

L'algorisme *simulated annealing* o temperat simulat es basa en un procés físic. Per tal de canviar les propietats d'un material com el ferro o el vidre, s'escalfa molt i es refreda ràpidament.

Aquest algorisme té un comportament semblant al del *hill climbing.* Si el nou node millora a l'anterior es triarà però si no el millora potser el triarà o potser no. Que s'esculli un node o no depèn d'una funció de probabilitat que depèn de dos factors. Primer es comprovarà si el nou node empitjora molt a l'anterior o no i finalment es tindrà en compte el factor de la temperatura. La funció és la següent:

$$p = e^{\frac{eval(N_a) - eval(N_n)}{T}}$$

on p és la probabilitat entre 0 i 1, T és la temperatura, Na és el node actual, Nn és el node nou i eval() és la funció d'avaluació. $eval(N_a) - eval(N_n)$ serveix per disminuir les possibilitats si el nou node empitjora molt, però en què influeix la T? Si es prenen valors de prova, es pot comprovar que com més gran és la temperatura més probabilitat hi ha de que es seleccioni un node pitjor.

L'algorisme va disminuint la temperatura progressivament mentre va avançant en la resolució. Això el que comporta és que al principi tingui més en compte tots els nodes, encara que siguin pitjors, i que es vagi estabilitzant a un màxim. Quan la temperatura és molt baixa ja no es seleccionen nodes pitjors i l'algorisme es centra en trobar el màxim pel qual s'ha decantat.

El pseudocodi del *simulated annealing* es veu a la *Il·lustració 17*.

```
Simulated_annealing(estat_actual):
      T = temperatura inicial
      T_min = temperatura mínima
      V_refredament = velocitat de refredament

      Mentre T > T_min:
            Fer de 0 a V_refredament vegades:
                  estat_nou = escollir un estat veí

                  Si eval(estat_nou) > eval(estat_actual):
                        estat_actual = estat_nou
                  Sinó:
                        Si random(0,1) < e^((eval(estat_actual)-eval(estat_nou))/T):
                              estat_actual = estat_nou
            Refredar(T)
```

Il·lustració 17. Pseudocodi simulated annealing

A la temperatura se li ha de donar un valor suficientment gran per que l'algorisme tingui temps de trobar un estat acceptable, però no massa gran perquè tardaria molt. La T_min és el límit de temperatura al que arriba abans d'aturar el procés de cerca; és la condició d'aturada.

La V_refredament determina quants veïns es visiten abans de refredar la temperatura. Ha de ser prou alta per permetre que s'arribi a un estat estable abans de tornar a baixar la temperatura. També hi ha diferents formes de refredar la T, però es farà servir un refredament lineal, restant-li un número fixe cada vegada.

L'elecció d'aquests factors dependrà del problema i de l'objectiu de la cerca, però sempre es pot experimentar amb ells. Aquest algorisme s'ha implementat pel TSP i el resultat que dona es veu a la *Il·lustració 18.*

```
In [24]: runfile('C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Búsqueda informada/
TSP_simulated_annealing.py', wdir='C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Búsqueda
informada')
['Sevilla', 'Salamanca', 'Santiago', 'Santander', 'Madrid', 'Valencia', 'Granada', 'Malaga']
Distància total: 2376.3731713936286
Temps d'execució': 0.0468733311 segons
```

Il·lustració 18. Retorn de l'algorisme simulated annealing

5.6. Cerca tabú

Un altre sistema per evitar els òptims locals és la cerca tabú. Amb l'objectiu de diversificar la cerca, la cerca tabú té una memòria adaptativa que recorda els últims canvis fets. L'algorisme comprova si ha fet el mateix canvi anteriorment i si l'ha fet, encara que sigui millor que l'actual, descarta el node i en busca un nou. Però si el nou node millora el valor del millor node trobat es fa una excepció i se'l selecciona. Als camins que s'han pres recentment se'ls anomenarà "tabú".

Aquest sistema el que fa és augmentar la possibilitat de que s'escullin altres nodes que no s'haguessin triat, ja que empitjorarien el node actual. Hi haurà un nivell de persistència que indica quantes iteracions han de passar per tal que es desbloquegi un node tabú. Passada una estona, els nodes tabú ho deixen de ser i ja es poden investigar aquests canvis.

Per exemple, parlant del TSP, un possible canvi de les dues primeres ciutats pot ser:

['Salamanca', 'Santander', 'Valencia', 'Sevilla', 'Madrid', 'Santiago', 'Malaga', 'Granada'] ⇨ ['Santander', 'Salamanca', 'Valencia', 'Sevilla', 'Madrid', 'Santiago', 'Malaga', 'Granada']

Ara bé, com es guarda en memòria aquest canvi? Una possibilitat és utilitzar una matriu que representi els canvis que es poden fer entre totes les ciutats. D'aquesta manera, a l'inici tots els valors són 0, però quan es fa un canvi, la posició que relaciona les dues ciutats canviades passaria a tenir un 3, si aquest fos el nivell de persistència. Després de fer el canvi mostrat, la matriu quedaria com es veu a la taula de la *Il·lustració 19*.

2	3	4	5	6	7	8	
3	0	0	0	0	0	0	**1**
	0	0	0	0	0	0	**2**
		0	0	0	0	0	**3**
			0	0	0	0	**4**
				0	0	0	**5**
					0	0	**6**
						0	**7**

Il·lustració 19. Taula de persistència dels canvis

A la pròxima iteració el valor del canvi 1-2 passarà a ser 2 i al nou canvi que s'hagi fet se li donarà el valor 3. La cerca tabú repeteix aquest procediment successivament durant el nombre d'iteracions que es vulgui, depenent de la qualitat de la solució que s'intenti trobar. Però a l'hora d'implementar aquest algorisme és més fàcil guardar la memòria amb una llista que contingui el nom de les ciutats canviades i el seu valor tabú. Per exemple, pel canvi que s'ha fet es guardaria a la llista de canvis tabú això: ["Salamanca_Santander", 3]. El pseudocodi queda com es mostra a la *Il·lustració 20*.

```
Cerca_tabú(estat_actual):
      millor_estat = estat_actual
      canvis_tabú = llista de canvis i persistència d'aquests
      persistència_tabú = nivell de persistència
      v_refredament = velocitat de refredament
      iteracions = nombre d'iteracions que es volen fer

      Mentre iteracions > 0:
            valor_actual = avaluar(estat_actual)

            per cada veí de estat_actual:
                  estat_nou = escollir un dels estats veïns
                  si estat_nou no és tabú:
```

```
                        si estat_nou millora a estat_actual:
                              estat_actual = estat_nou
                              emmagatzemar canvi a canvis_tabú
                              si estat_actau és millor que millor_estat:
                                    millor_estat = estat_actual
                              sortir del bucle
                        si estat_nou no millora a estat_actual:
                              si estat_nou és millor que millor_estat:
                                    estat_actual = estat_nou
                                    emmagatzemar canvi a canvis_tabú
                                    millor_estat = estat_actual
                                    sortir del bucle
            disminuir persistència dels canvis_tabú
```

Il·lustració 20. Pseudocodi cerca tabú

La cerca tabú implementada al problema del viatjant dona el resultat que es veu a la *Il·lustració 21*.

```
In [41]: runfile('C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Búsqueda informada/
TSP_busqueda_tabu.py', wdir='C:/Users/TOM/OneDrive/Documentos/TDR IA/Algoritmos/Búsqueda
informada')
['Madrid', 'Valencia', 'Granada', 'Malaga', 'Sevilla', 'Salamanca', 'Santiago', 'Santander']
Distància total: 2376.373171393628
Temps d'execució': 0.0156238079 segons
```

Il·lustració 21. Retorn de l'algorisme de cerca tabú

5.7. Anàlisi dels resultats

Al llarg de l'explicació dels algorismes també s'han anat mostrant els resultats. Tots els algorismes han donat la mateixa ruta: ['Madrid', 'Valencia', 'Granada', 'Malaga', 'Sevilla', 'Salamanca', 'Santiago', 'Santander'], amb 2376 km. Com que també s'ha executat l'algorisme de poder computacional i dona el mateix resultat, es sap segur que la ruta trobada és el màxim absolut, ja que aquest algorisme avalua totes les rutes possibles.

Ara bé, el temps d'execució sí que ha variat substancialment entre els diferents algorismes. Aquí es mostren els resultats:

	Algorisme	Temps d'execució
Algorismes sense intel·ligència artificial	Poder computacional (cerca en profunditat)	4 minuts i 34 segons
Algorismes amb intel·ligència artificial	*Hill climbing iteratiu*	0,01 segons
	Simulated annealing	0,047 segons
	Cerca tabú	0,015 segons

També s'han fet proves dels algorismes intel·ligents amb 15 ciutats i aquests són els resultats:

Algorisme	**Distància del recorregut**	**Temps d'execució**
Hill climbing iteratiu	2971,73 km	1,3 segons
Simulated annealing	2971,73 km	2,36 segons
Cerca tabú	3239,22 km	1,17 segons

A la primera taula es veu com els algorismes d'intel·ligència artificial triguen moltíssim menys en trobar l'òptim global. L'algorisme de poder computacional ha de fer molts més processos i arriba a trigar més de 4 minuts per només 8 ciutats.

Per últim, es pot veure que els algorismes intel·ligents troben una solució, que pot ser més o menys bona però segueix sent un màxim, fins i tot amb 15 ciutats. En canvi, si intentéssim trobar la millor solució calculant totes les possibilitats, el procés duraria mesos o fins i tot anys. Degut al caràcter exponencial de l'espai d'estats, per cada ciutat que s'augmenta, l'algorisme exhaustiu triga moltíssim més en trobar la solució.

Es podria fer un anàlisi molt més profund dels diferents algorismes. Fent moltes més proves i manipulant els factors dels algorismes es podria demostrar quin dels algorismes és més òptim. Però tota aquesta part d'anàlisi del rendiment queda fora del treball.

6. EL JOC DE LES DAMES

Les dames són un grup de jocs de taula molt populars arreu del món. La primera variant va sorgir al s.XII de la unió del tauler dels escacs amb les regles de l'alquerc[29]. Durant els primers anys el joc era conegut pel nom *fierges,* procedent de la paraula *fers* atribuida a la reina dels escacs. Més endavant ja es va començar a anomenar *dame* i finalment *jeu de dames,* apel·latiu actual del joc.

Al llarg de les dècades el reglament de les dames ha anat evolucionant i han sorgit moltes variants. Algunes de les més importants són les dames poloneses o les espanyoles. Parts del reglament com el comportament de la reina, la necessitat de matar o fins i tot les dimensions del tauler varien segons la modalitat que es jugui.

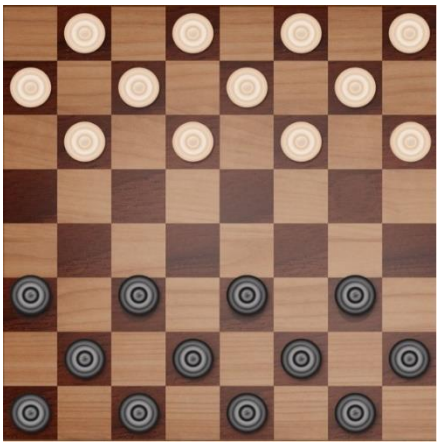
L'objectiu d'aquest apartat és elaborar un programa que sigui capaç de jugar al joc de les dames implementant sistemes d'intel·ligència artificial.

6.1. Instruccions i funcionament

Com s'ha explicat a l'apartat anterior, existeixen moltes variants del joc. Per poder començar a analitzar el joc primer s'ha de definir un reglament pel joc. Cercant per internet s'ha trobat molta ambigüitat envers al reglament del joc i les seves modalitats. Finalment s'ha triat la modalitat de les dames espanyoles, que són les més conegudes a la nostra regió. A continuació s'explicarà el reglament de les dames que més endavant s'implementarà al programa.

DISPOSICIÓ:

- Tauler d'escacs amb 64 caselles (8x8), blanques i negres alternativament.
- Inicialment cada jugador té 12 peces col·locades de manera que estiguin com es veu a la imatge
- El tauler s'ha d'orientar de tal manera que els dos jugadors tinguin una casella blanca a l'extrem dret de la primera fila.



Il·lustració 22. Disposició del tauler a l'inici de la partida

[29] L'alquerc és un joc de taula provinent del Mig Orient amb més de 3000 anys d'antiguitat.

COMPORTAMENT DE LES PECES:

- Les dames (peces inicials) només es mouen una casella endavant en diagonal.
- Quan hi ha una peça del contrincant a la casella on la dama podria moure's, hi ha la possibilitat de capturar aquesta peça, passant-li per sobre i col·locant-se a la casella immediatament següent en diagonal, sempre que estigui buida.
- Quan una dama arriba a l'última filera del tauler, primera filera del contrincant, es corona reina i se li col·loca una altra peça del mateix color a sobre per diferenciar-se de les dames.
- Les reines tenen l'avantatge de poder moure's i capturar altres peces cap enrere, i es poden moure totes les caselles que vulguin. També poden ser capturades per qualsevol peça enemiga, sigui dama o reina.
- Si estan en disposició de capturar, les dames o reines tenen l'obligació de fer-ho. Després de capturar una peça, si segueixen estant en disposició de capturar ho hauran de fer a la mateixa tirada. Si en una mateixa tirada hi ha més d'una opció per capturar, el jugador pot decidir quina executar. Finalitzada la jugada, es retiren les peces capturades.
- Si un jugador no captura quan té l'oportunitat, se l'haurà d'avisar que el moviment que vol fer no és vàlid i haurà de fer un dels moviments que maten.

DESENVOLUPAMENT I FINAL DE LA PARTIDA:

- A l'inici de la partida les peces es col·loquen com es veu a la imatge i les negres surten. Normalment es fa a sorts quin dels dos jugadors jugarà amb cada color.
- Cada jugador farà la seva tirada després de l'altre.
- L'objectiu del joc es capturar totes les peces del contrincant. El jugador que ho aconsegueixi abans guanya la partida.
- Si un jugador té totes les peces bloquejades perd la partida.
- Si cap dels jugadors té suficients peces al tauler per poder guanyar la partida, es declara taules.

6.2. Anàlisi del joc

Abans de començar a programar s'ha d'analitzar el problema. El que es busca és que el programa serveixi per enfrontar un humà amb l'ordinador i sigui interactiu, així que haurà de disposar de tres parts principals:

a) Un sistema que demani el moviment al jugador humà, i si és vàlid l'introdueixi al tauler.
b) Un sistema que desenvolupi l'arbre de possibles jugades, esculli el millor moviment que trobi i l'introdueixi al tauler
c) Un sistema que mostri el tauler interactiu, en el qual sigui còmode introduir els moviments

Per poder comprovar si un moviment és vàlid o per construir l'arbre de possibles jugades es necessita una funció molt important: ***get_mov_possibles***. Aquesta funció retornarà tots els moviments possibles a partir d'un tauler. Una vegada es tenen aquests moviments es pot comprovar si el moviment introduït per l'humà està entre la llista generada i si ho està, fer el moviment.

També es farà servir per desenvolupar l'arbre de jugades possibles. Tenint els moviments possibles a partir del tauler inicial es poden introduir els moviments i a partir dels nous taulers tornar a buscar els moviments possibles i així successivament. Així s'anirà generant l'arbre i s'escollirà la millor jugada que es trobi.

Ara bé, també ens cal una funció que decideixi quin és el millor moviment de l'arbre generat. Aquí és on intervé la intel·ligència artificial: l'***algorisme minimax*** és ideal per complir aquesta funció. Per últim, el minimax necessitarà la **funció d'avaluació** per diferenciar entre un estat del tauler bo per l'ordinador i un de dolent.

Finalment, el sistema per poder interactuar amb el programa molt més fàcilment, serà una **interfície gràfica**. Aquesta interfície ha de mostrar els moviments al tauler, permetre introduir els moviments i comunicar si s'ha introduït un moviment incorrecte.

Ara que ja s'ha desglossat el problema en diferents parts, s'entrarà més en detall en cada un d'aquests sistemes que s'utilitzaran per elaborar el programa final.

6.3. Funció get_mov_possibles

La forma més senzilla d'elaborar la funció *get_mov_possibles* és fer que vagi mirant les posicions del tauler. Quan es trobi amb una peça, ha de trucar a una altra funció que emmagatzema els moviments possibles de la peça en qüestió a la llista de moviments possibles. Així doncs, quan acabi de mirar totes les posicions del tauler, s'hauran guardat tots els moviments possibles a la llista. El pseudocodi d'aquest procediment es veu a la *Il·lustració 23*.

```
Moviments possibles = llista
per cada posició al tauler:
      si a la posició hi ha una dama del jugador:
            emmagatzemar moviments possibles de la dama a Moviments possibles
            (funció get_mov_dama)
      si a la posició hi ha una reina del jugador:
            emmagatzemar moviments possibles de la reina a Moviments possibles
            (funció get_mov_reina)
```

Il·lustració 23. Pseudocodi de la funció get_mov_possibles

Ara bé, s'ha de pensar com funcionarà la funció que acumula els moviments possibles d'una dama o reina. La manera més intuïtiva d'imaginar-ho seria una estructura de condicionals[30]. D'aquesta manera la funció anirà mirant si la peça pot moure en les diferents direccions o si pot matar.

Però per tal que aquesta funció sigui senzilla i efectiva també se li haurà d'afegir recursivitat[31] per poder tractar amb els moviments de la reina o els moviments dobles de la dama. Així doncs, si es troba un moviment que captura una peça, des de la funció *get_mov_reina* o *get_mov_dama* es tornarà a trucar a la mateixa funció per saber si des de la nova posició (havent capturat una peça) es pot tornar a capturar.

[30] Un condicional en programació és una instrucció que comprova una condició i executa una part del codi depenent de si aquesta condició és certa o falsa.

[31] La recursivitat és el procés pel qual una funció es truca a ella mateixa.

6.4. Algorisme minimax

Ara que ja està programada la funció *get_mov_possibles*, s'ha de buscar una forma de fer-la servir per desenvolupar l'arbre d'estats del tauler. Existeix un algorisme que encaixa perfectament amb aquest objectiu, ja que està pensat per jocs amb característiques similars a les dames: l'algorisme minimax.

El nom prové de la idea que es denomina MAX als estats del tauler on és el torn de l'ordinador i MIN als estats del tauler on és el torn del jugador humà. Els torns de l'ordinador es diuen MAX perquè maximitza les possibilitats de que guanyi i els del humà es diuen MIN perquè minimitza les seves possibilitats de guanyar.

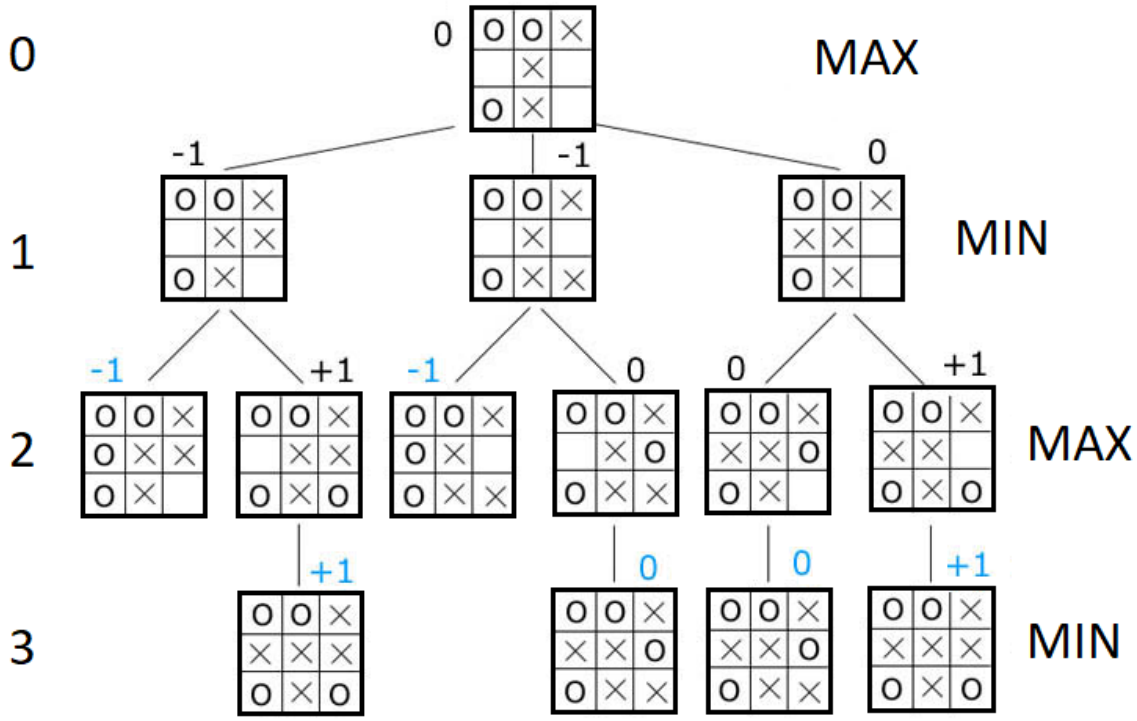
L'algorisme minimax és un tipus de cerca en profunditat, així que anirà recorrent l'arbre verticalment fins que arribi a un node terminal, estat del tauler en el qual s'acaba la partida o s'arriba a la profunditat màxima[32]. Quan arribi a un d'aquests nodes, s'utilitza la funció d'avaluació (es defineix al pròxim apartat) per saber el valor que té. Per situar-se, la funció d'avaluació retornarà un valor més gran per un estat del tauler més favorable per l'ordinador i un valor més petit per un estat que ho sigui menys.

Els valors dels nodes terminals s'aniran propagant als seus nodes pares. Si el node pare és MIN, s'escollirà el menor valor de tots els seus fills. Això vol dir que l'ordinador sempre dona per suposat que l'humà triarà la millor opció (la que li va pitjor a l'ordinador). En canvi, si el node pare és MAX, es propagarà el valor més gran dels fills, perquè es busca triar la millor opció per a l'ordinador.

El graf mostrat a continuació és un exemple de com funcionaria l'algorisme minimax pel joc 3 en ratlla des de l'estat inicial, donat del tauler. Com es pot observar, la funció d'avaluació retorna +1 si l'ordinador guanya, -1 si perd i 0 si empaten, l'ordinador és les X i l'humà les O. En aquest cas, com que la partida s'acaba en poques jugades, tots els nodes terminals suposen el final de la partida i poden estar a diferents nivells de profunditat.

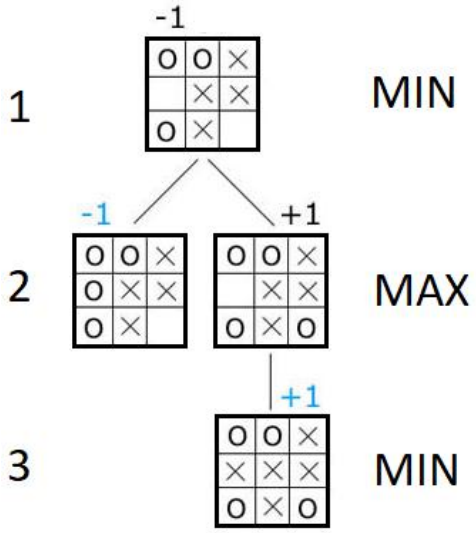
[32] Per tal de limitar la cerca, s'introduirà una profunditat màxima de jugades. Per exemple, si el límit és 8 de profunditat, quan ja s'hagin visitat 8 jugades en profunditat ja no es baixarà més en profunditat.

El valor dels nodes terminals està en blau. Des de l'estat inicial la funció *get_mov_possibles* dona tres moviments, que es corresponen amb els tres espais lliures que queden al tauler. Així doncs, ens trobem amb tres branques principals.



Il·lustració 24. Arbre de l'algorisme minimax

Si s'observa la branca de l'esquerra, es pot veure com al segon nivell de profunditat ja hi ha un node terminal en el qual l'ordinador perd. L'altre node terminal de la branca es troba al nivell 3 i és una victòria de l'ordinador. El valor +1 es propaga al node pare ja que és l'únic fill. Ara ens trobem amb dos nodes al nivell 2: un amb valor +1 i l'altre amb valor -1. Com que el nivell 1 és MIN i el node pare està a aquest nivell, es quedarà amb el valor -1.



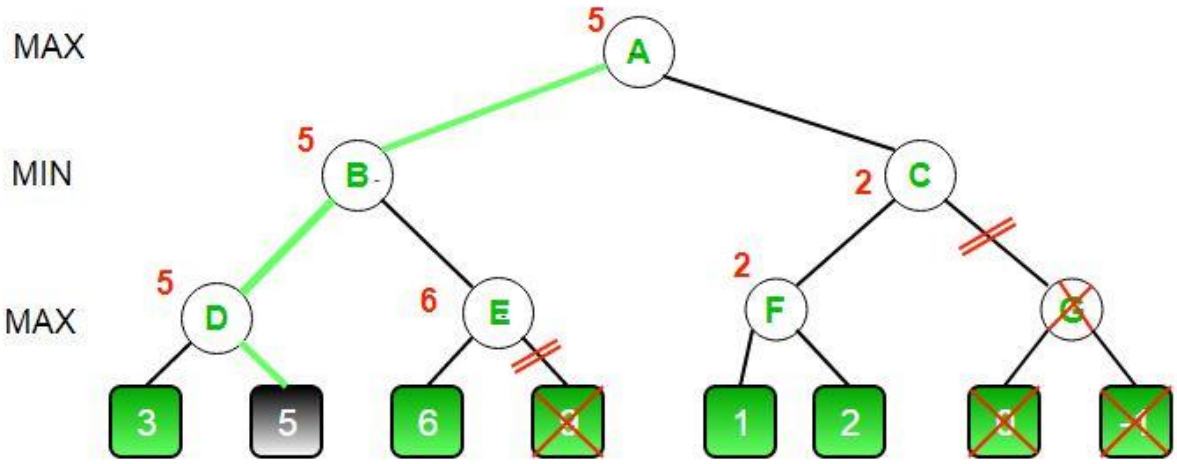
Il·lustració 25. Branca esquerra de l'arbre minimax

Si es segueix el mateix procediment amb les tres branques, s'arriba a que els tres primers nodes tenen valors -1, -1 i 0. Com que el node arrel és de tipus MAX, es queda amb el valor 0. Això vol dir que el moviment que es fa per arribar al tercer

node (amb valor 0), és el millor que pot fer l'ordinador per tal de no perdre la partida. Però si el jugador humà juga el millor possible, l'ordinador només podrà aconseguir un empat.

6.4.1. Poda alfa-beta

Un cop definit el comportament de l'algorisme minimax, es pot fer una optimització molt important. Si s'observen alguns casos de possibles arbres es pot veure que hi ha vegades que és redundant observar algunes branques. La *Il·lustració 26* és un arbre minimax en el qual es veuen representades dues podes:

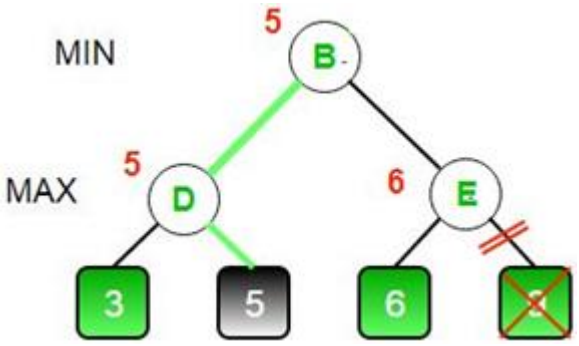


Il·lustració 26. Arbre minimax amb poda

Per començar, es veu com els nodes terminals poden prendre un valor qualsevol, no cal que sigui 0, +1 o -1. Seguim el que faria l'algorisme per comprendre com es fa la poda:

Com que el minimax explora l'arbre en profunditat, baixarà per la primera branca. Al tercer nivell de profunditat ja es troba un node terminal, que té valor 3. Com de moment és l'únic node fill del node D, el valor 3 es propaga a aquest. Però a continuació es troba un altre node fill que és terminal, amb valor 5. Com que el node D és MAX, selecciona el valor 5 entre els seus dos fills, que és el més gran.

Com que ja s'han explorat tots els nodes fills de D, el valor d'aquest node es propaga cap el seu pare momentàniament, el node B. Però com D no és l'únic node fill de B, s'explora el node E. El primer node fill que troba E és un node terminal amb valor 6, així que a priori el node E adopta el valor 6.

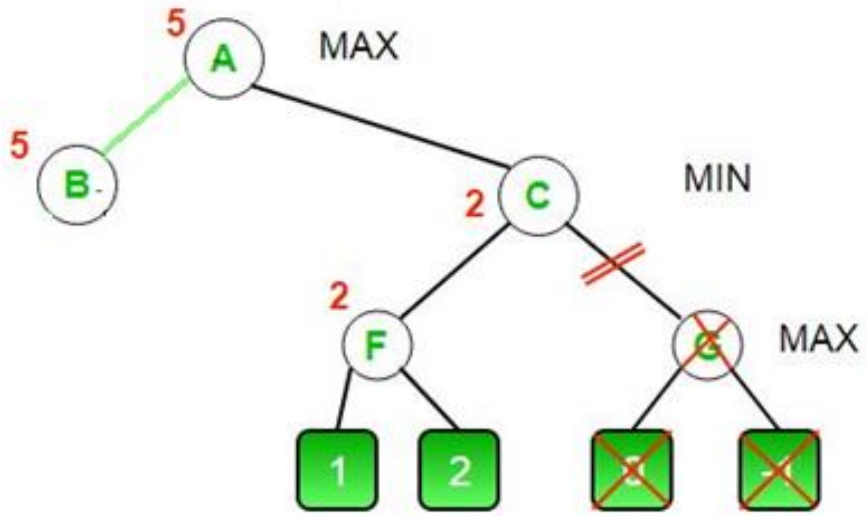


Il·lustració 27. Branca esquerra de l'arbre amb poda

En aquest punt es pot observar que el node E encara que tingui fills amb valors inferiors a 6, sempre es quedarà amb el valor 6 o un de més gran, ja que és un node MAX. També es pot observar que com que el node B és MIN, agafarà el valor 5 que té de moment o valors inferiors. En aquesta situació és redundant explorar més nodes fills d'E, perquè tinguin el valor que tinguin el node B sempre es quedarà amb el valor 5 que ja té assignat.

A continuació s'analitzarà la branca de la dreta, tenint en compte que l'algorisme ja ha donat el valor 5 al node B. A aquesta branca es dona un cas semblant:

En primer lloc l'algorisme baixarà en profunditat fins al node F i després d'avaluar els seus dos nodes fills terminals, es propagarà el valor 2 (el node F és MAX). A continuació el valor 2 es propaga al node C momentàniament (a l'espera de que s'avaluïn la resta dels seus fills).



Il·lustració 28. Branca dreta de l'arbre amb poda

En aquest moment es dona una altra situació de poda. Com que el node A és de tipus MAX, seleccionarà un valor 5 o més gran. En canvi, el node C, com que és MIN,

seleccionarà un valor 2 o més petit. En conclusió, el node A és impossible que adopti un valor de la branca de la dreta, així que encara que no s'hagi acabat d'explorar, es poda la branca i no s'analitza el node G.

6.4.2. Implementació del minimax amb poda alfa-beta

L'algorisme minimax s'implementa a través d'una cerca en profunditat recursiva. A part d'això, també ha de propagar els dels valors dels nodes terminals als nodes pares fins arribar al node arrel.

Però per fer la poda, s'han d'acumular dos valors més: alfa i beta, que serveixen per identificar les situacions de poda descrites a l'apartat anterior. Alfa s'identificarà amb l'últim valor que s'ha propagat a un node MAX, així que a l'inici rebrà el valor $-\infty$ i anirà augmentant. D'altra banda, la beta s'identifica amb l'últim valor que s'ha propagat a un node MIN, tindrà un valor inicial de $+\infty$ i anirà disminuint.

El sistema dels valors alfa-beta va comparant per cada node si el nou valor és més gran o més petit que beta o alfa. Si es tracta d'un node MAX, si es compleix la següent condició es fa poda:

```
si valor ≥ β:
```

I si es tracta d'un node MIN, si es compleix la següent condició es fa poda:

```
si valor ≤ α:
```

Així doncs, el pseudocodi de l'algorisme minimax amb poda alfa-beta en un cas general seria el que es mostra a la *Il·lustració 29*.

```
definir Minimax_αβ(node, profunditat, α, β)
      si profunditat == 0 o si node és terminal:
            retornar valor node.funció_avaluació()
      si node és MAX:
            valor  =  -∞
            per cada node_fill possible:
                  valor = màxim(valor, Minimax_αβ(node_fill, profunditat-1, α, β))
                  si valor ≥ β:
                        break (poda β)
                  α = màxim(α, valor)
            retornar valor
      si node és MIN:
            valor  =  +∞
            per cada node fill possible:
                  valor = mínim(valor, Minimax_αβ(node_fill, profunditat-1, α, β))
                  si valor ≤ α:
                        break (poda α)
                  β = mínim (β, valor)
            retornar valor

Minimax_αβ(node_original, PROFUNDITAT, -∞, +∞)    (Trucada inicial)
```

Il·lustració 29. Pseudocodi general de l'algorisme minimax

Aquest pseudocodi s'ha d'adaptar al joc de les dames. El seu funcionament no és l'indicat; és necessari que retorni el node amb el moviment escollit ja realitzat, no el valor del moviment escollit. Així doncs, s'han d'emmagatzemar d'alguna manera els nodes generats al primer nivell, i que es retorni el node que contingui el moviment escollit, que és el que ens interessa realment. El pseudocodi amb l'arranjament es veu a la *Il·lustració 30.*

```
definir Minimax_αβ(node, profunditat, α, β, node_retorn)
      si profunditat == 0 o si node és terminal:
            retornar valor node.funció_avaluació()
      si node és MAX:
            valor  =  -∞
            per cada node_fill possible:
                  valor_tmp = Minimax_αβ(node_fill, /  profunditat-1, α, β, None)
                  si node_retorn no és None:
                        si valor_tmp >= valor:
                              valor = valor_tmp
                              node_retorn = node_fill
                  si valor ≥ β:
                        break (poda β)
                  α = màxim(α, valor)
            retornar valor
      si node és MIN:
            valor  =  +∞
            per cada node fill possible:
                  valor = mínim(valor, Minimax_αβ(node_fill, profunditat-1, α, /
                  β, None))
                  si valor ≤ α:
                        break (poda α)
                  β = mínim (β, valor)
            retornar valor
node_retorn = node
Minimax_αβ(node_original, PROFUNDITAT, -∞, +∞, node_retorn)    (Trucada inicial)
```

Il·lustració 30. Pseudocodi de l'algorisme minimax pejoc de les dames

La forma que s'ha trobat per solucionar aquest problema és posar una comprovació afegida a dins dels processos que es fan pel node MAX (degut a que el primer node és sempre MAX). A tots els nodes MAX es comprovarà si es tracta del nivell 0 de profunditat i si ho és s'anirà guardant el node que esculli a una variable. D'aquesta manera no es canvia molt el funcionament, la funció segueix retornant el valor, però queda guardat el node escollit a una variable que se li passa a la trucada inicial per referència[33].

[33] A la funció minimax, se li dona accés a una variable on guardarà el node escollir. Després de l'execució de la funció, es pot accedir a aquesta variable per veure quin moviment ha escollit.

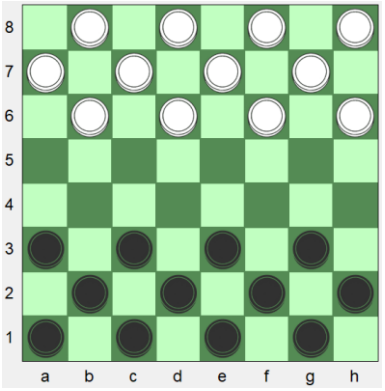
6.5. Funció d'avaluació

Com ja s'ha esmentat, la funció minimax necessita una funció d'avaluació per poder retornar un nombre a partir d'un estat determinat del tauler. D'aquesta manera el valor s'anirà propagant i finalment l'algorisme triarà el moviment que esdevé la posició del tauler més favorable a x moviments vista (depenent dels nivells que es vulgui que baixi per l'arbre).

Així doncs, s'ha de definir la funció d'avaluació. La forma més intuïtiva és donar-li un valor a les dames i un a les reines, i a les peces del jugador humà donar-li els mateixos valors però en negatiu. Per exemple: se li dona valor 1 a la dama i 8 a la reina. Si l'estat del tauler està representat amb una llista de 32 nombres que es corresponen amb les posicions (de baix a dalt) llavors la funció només ha de sumar tots els valors de la llista. Per al tauler inicial seria així:

Tauler = [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

La funció d'avaluació suma tots els nombres de la llista i retorna 0, un valor comprensible donat que els dos jugadors estan iguals.



Il·lustració 31. Tauler inicial

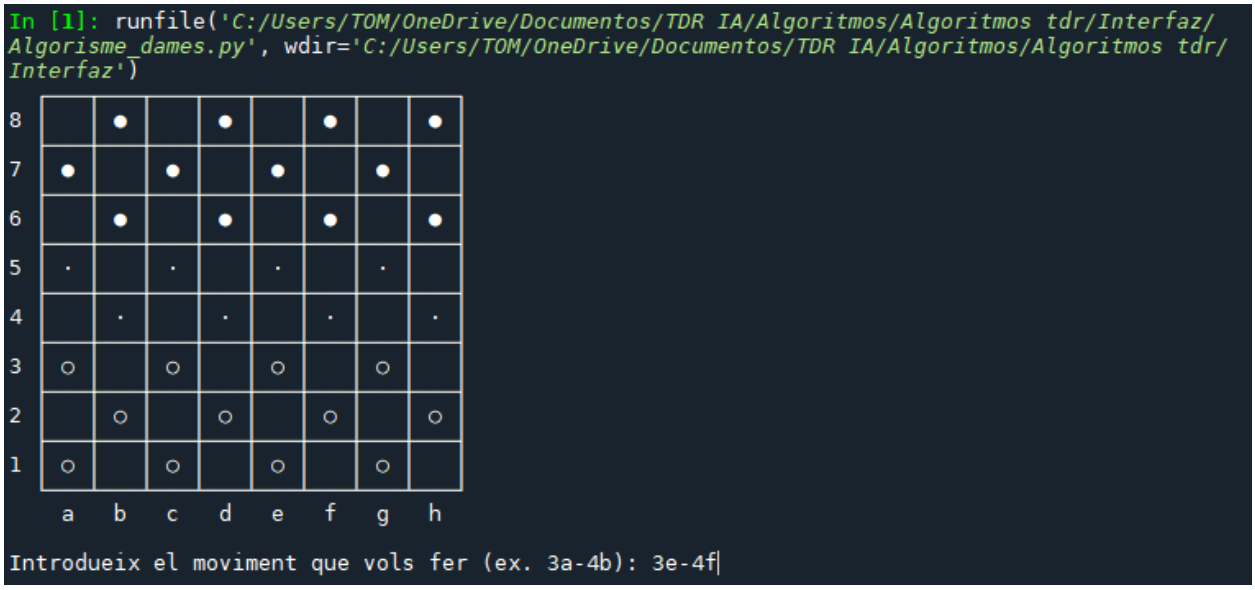
Es podrien tenir molts més factors en compte:

- Donar-li més valor a les peces segons a quina posició del tauler es trobin.
- Donar-li un valor diferent a les peces segons si al tauler en queden moltes o poques.
- Donar-li un valor diferent a les peces segons com està actuant el contrincant.

Al pròxim apartat es treballa com millorar la funció d'avaluació.

6.6. Interfície gràfica

Ara que ja s'han definit tots els components imprescindibles queda trobar la forma de que el programa sigui interactiu. Una solució provisional és programar una funció que mostri d'alguna manera el tauler a l'intèrpret interactiu per així poder jugar. El problema és que d'aquesta manera cada vegada que es fa un moviment es mostra un tauler nou. La partida costa una mica de seguir i els moviments s'han d'introduir a través de les coordenades. Tot i això, és la solució més fàcil que permet jugar i la primera que s'ha implementat. El tauler es veu així:



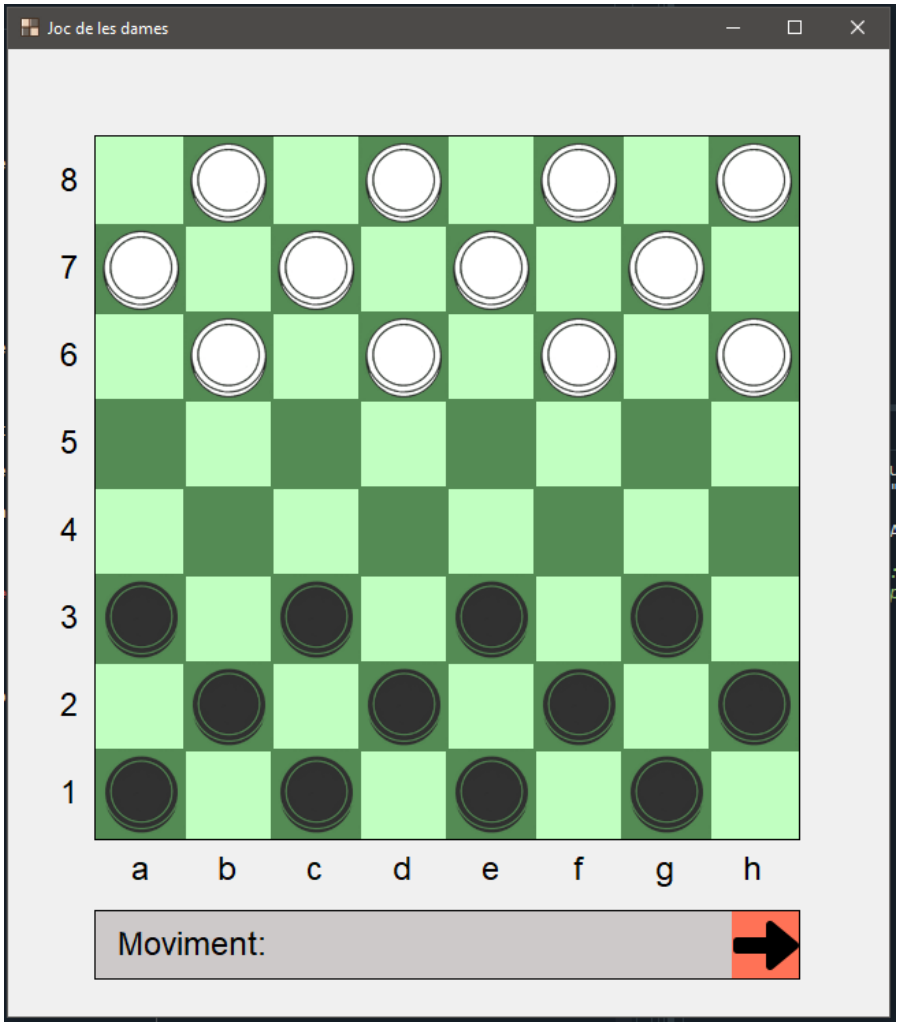
Il·lustració 32. Tauler mostrat a l'intèrpret interactiu

Ara bé, la solució bona que permet jugar còmodament és programar una interfície gràfica. Això es fa a través d'una biblioteca que s'importa[34] al mòdul[35]. Aquesta biblioteca es diu *Tkinter* i és la biblioteca per interfícies gràfiques que més es fa servir per Python. El que ens permet aquesta biblioteca és crear una finestra, la qual s'obre quan s'executa el mòdul. A la finestra es poden afegir botons, text, imatges...

[34] Les biblioteques són com "paquets" de funcions que es poden importar per així fer-les servir al mòdul que es vulgui.

[35] Un mòdul és un arxiu que conté un programa escrit.

Tots aquests elements es defineixen quan s'inicialitza la finestra, es determina el tipus (botó, imatge...) i se li donen totes les característiques. Buscant documentació sobre *Tkinter* és relativament fàcil programar-ho. La interfície gràfica pel joc de les dames ja acabada queda així:



Il·lustració 33. Interfície gràfica interactiva

Les caselles són totes botons i quan es fa clic a una d'elles es marca i es comença a acumular el moviment. Quan ja s'ha introduït el moviment si es fa clic a la fletxa (que és un altre botó) es comprova si és correcte i si ho és es mou la peça. Mentrestant al requadre on posa "Moviment:" es van acumulant les coordenades del moviment introduït.

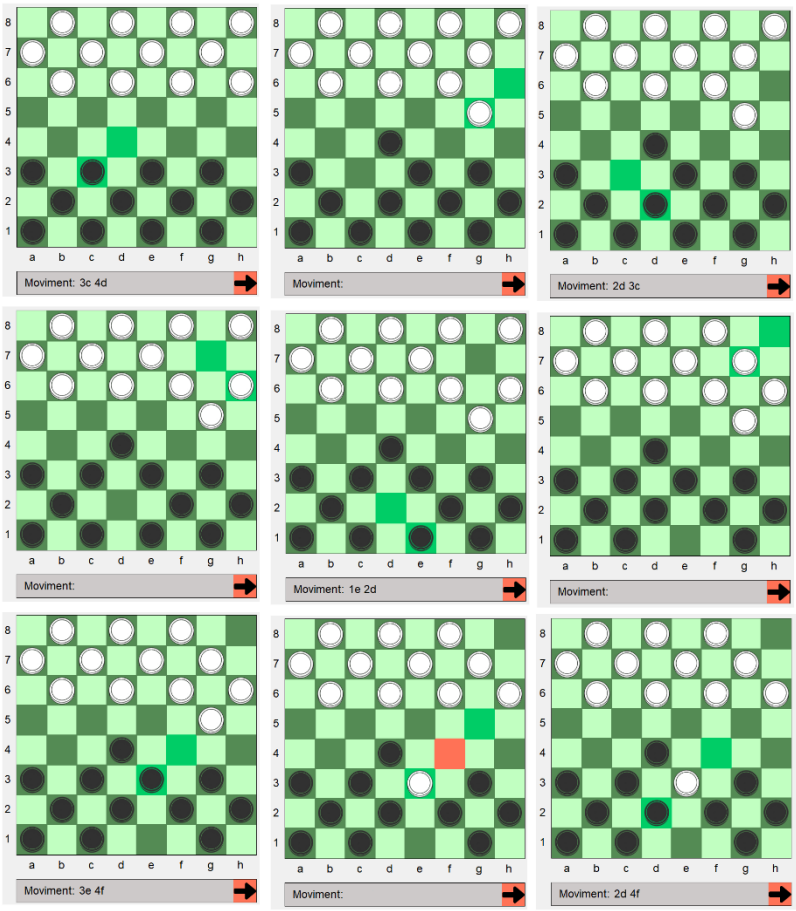
Quan s'ha fet un moviment automàticament es truca a la funció minimax per buscar quin moviment fa l'ordinador. Quan es troba el moviment, s'aplica al tauler i l'humà ja pot tornar a tirar. Els moviments es marquen d'un altre to de verd i les captures, de color vermell, per facilitar el seguiment del transcurs de la partida.

6.7. El programa final del joc de les dames

El programa final és la implementació de tots els sistemes explicats anteriorment. Està dividit en 3 mòduls diferents, on hi ha les diferents parts del programa. Al primer mòdul hi ha l'algorisme minimax que fa servir les classes i objectes definits a un segon mòdul. L'últim mòdul es correspon amb la interfície gràfica i utilitza funcions del mòdul minimax. Aquest és el que s'executa per començar a jugar.

Quan s'executa el programa s'obre una pestanya que mostra el tauler inicial. A través de la interfície gràfica es pot jugar còmodament la partida. Quan un dels jugadors guanya es bloquegen els botons i es mostra al requadre de text qui ha guanyat la partida.

Després de fer algunes proves s'ha comprovat que l'algorisme només funciona ràpid si es juga a 6 nivells de profunditat o menys. En aquestes condicions, es requereix concentració per guanyar la partida. És un programa funcional que respon ràpid i les partides són dinàmiques però el nivell de dificultat no és molt alt.



Il·lustració 34. Exemple del funcionament del programa

CONCLUSIONS

7. CONCLUSIONS DE LA PART PRÀCTICA TSP

En un primer moment, un dels objectius d'aquesta pràctica era programar els algorismes per solucionar el TSP amb estructures de classes i objectes. S'han començat a escriure aquests algorismes exitosament, amb l'ajuda de l'algorisme del manual. Però quan s'ha comprovat si els nous algorismes són òptims temporalment, resulta que tarden el triple per trobar la mateixa solució. En comparació amb els programes del manual, els que s'han programat amb classes i objectes són molt més lents.

Per aquesta raó finalment s'han fet servir els algorismes ja escrits per fer aquesta pràctica. Aquesta decisió s'ha pres tenint en compte que l'algorisme de les dames és la part pràctica principal i requerirà més temps de programació. En conclusió, el TSP s'ha fet servir per comparar els algorismes d'IA amb l'algorisme basat únicament en el poder computacional. Aquest objectiu, que és el principal, s'ha mantingut.

Després d'analitzar els resultats, es pot afirmar que el poder que tenen els sistemes d'intel·ligència artificial és molt superior al simple poder computacional. Fent servir els algorismes intel·ligents es redueix el temps d'execució 20.000 vegades.

Tot i això, podria semblar que és factible fer servir l'algorisme ingenu per aquest tipus de problemes, ja que, tot i que triga molt més que els altres, 4 minuts és un temps factible per trobar la solució a un problema. Però realment els sistemes que no tenen cap tipus d'intel·ligència són molt limitats davant d'un mínim de complexitat del problema. Això queda comprovat amb la prova d'executar l'algorisme de poder computacional amb més de 10 ciutats; l'algorisme trigaria un temps quasi il·limitat.

En canvi, els algorismes intel·ligents són molt interessants per trobar una solució factible a un problema molt més complex. Tal i com s'ha comprovat, els tres algorismes troben màxims pel TSP amb 15 ciutats, on els estats possibles són un nombre de 12 xifres. Aquests sistemes funcionen tant bé perquè la intel·ligència i el poder computacional de l'ordinador fan una combinació que acaba sent potentíssima.

Així doncs, la intel·ligència artificial queda comprovat que és la millor opció per tractar problemes d'optimització complexos com el TSP.

8. CONCLUSIONS DEL JOC DE LES DAMES

Com ja s'ha explicat, el programa de les dames funciona bé per un nombre moderat de nivells de profunditat. Això fa que es compleixi l'objectiu principal d'aquesta pràctica. S'ha pogut exemplificar el funcionament de sistemes intel·ligents per un problema complex. A més a més, s'ha elaborat un joc funcional que permet enfrontar a l'ordinador i veure fins a quin punt el programa és intel·ligent.

La eficàcia del programa no és la millor possible perquè es pot guanyar amb paciència i concentració. Tot i això, si l'ordinador comença guanyant resulta molt difícil remuntar la partida. Quan obté un mínim d'avantatge respecte al jugador en poc temps mata moltes peces i guanya molt ràpid.

Els programes mai s'acaben, així que es podrien fer moltes millores per augmentar la seva intel·ligència. Implementant l'algorisme ideal pel joc, el programa ja és capaç de calcular moltíssimes possibilitats i jugar relativament bé. Això porta a pensar que si es fessin més afegits a la funció d'avaluació es podria millorar encara més el seu funcionament. Però aquesta part queda fora del treball, ja que l'objectiu era obtenir un programa intel·ligent a través de la programació.

En un primer moment hi havia la intenció d'afegir l'apartat de millora de la funció d'avaluació al treball, però resulta que per fer-ho s'hauria de fer un anàlisi exhaustiu del funcionament i eficàcia del programa. Millorar la funció d'avaluació és més difícil del que s'esperava, així que no s'ha inclòs al treball per la seva complexitat i per falta de temps. De tota manera, segueix sent un tema molt interessant, així que s'explica el plantejament i estudis relacionats al segon annex: *Proposta de treball: millora de la funció d'avaluació*.

BIBLIOGRAFIA I WEBGRAFIA

9. BIBLIOGRAFIA

- GARCÍA SERRANO, Alberto: *Inteligencia Artificial,* RC libros, Madrid, 2ª ed. Revisada, 2016, 258 páginas.
- OBÓN, Xavier et. al.: *Todos los juegos del mundo,* Enciclopedias Planeta, Barcelona, 1ªed., 1994, 448 páginas.

9.1. Fonts informàtiques

- BLOG DE INTELIGENCIA ARTIFICIAL: Tema 3: estrategias de búsqueda no informada. https://jraquelm2.wixsite.com/ia2raquelmurillo/single-post/2015/05/04/TEMA-3-ESTRATEGIAS-DE-B%C3%9ASQUEDA-NO-INFORMADA (consulta: 6/9/2020)
- FERNANDO SANCHO CAPARRINI: Búsquedas No Informadas. http://www.cs.us.es/~fsancho/?e=95 (consulta: 6/9/2020)
- GREEKSFORGREEKS: Travelling Salesman Problem. https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/ (consulta: 26/7/2020)
- INAOEP.MX: 4.2 Cerca Local (Local Search). https://ccc.inaoep.mx/~emorales/Cursos/Busqueda/node58.html (consulta: 12/8/2020)
- INFOBAE: Presente y futuro de la inteligencia artificial: cómo va a cambiar nuestro mundo. https://www.infobae.com/tendencias/innovacion/2018/05/20/presente-y-futuro-de-la-inteligencia-artificial-como-va-a-cambiar-nuestro-mundo/ (consulta: 3/6/2020)
- INSTITUTO DE INGENIERÍA DEL CONOCIMIENTO: Procesamiento del lenguaje natural ¿qué es?. https://www.iic.uam.es/inteligencia/que-es-procesamiento-del-lenguaje-natural/ (Consulta: 9/5/2020)
- MATEMATICASIES: Grafos. Descripción. https://matematicasies.com/Grafos-Definicion (consulta: 16/6/2020)

- NATIONAL GEOGRAPHIC ESPAÑA: Breve historia visual de la Inteligencia Artificial. https://www.nationalgeographic.com.es/ciencia/breve-historia-visual-inteligencia-artificial_14419/13 (consulta: 10/5/2020)
- PRESENTACIÓN UPC: Resolución de problemas. https://www.cs.upc.edu/~bejar/ia/transpas/teoria/2-BH1-introduccion_busqueda.pdf (consulta: 3/6/2020)
- PYTHON.ORG: https://www.Python.org/ (consulta: 16/6/2020)
- ROUTIFIC: Understanding the Travelling Salesman Problem (TSP). https://blog.routific.com/travelling-salesman-problem (consulta: 26/7/2020)
- STUDOCU: Tema 7. Razonamiento automático. https://www.studocu.com/ca-es/document/universitat-autonoma-de-barcelona/inteligencia-artificial/apuntes/tema-7-razonamiento-automatico/2444529/view (consulta: 9/5/2020)
- UPC.EDU: Cerca Local. https://www.cs.upc.edu/~bejar/ia/transpas/teoria/2-BH3-Busqueda_local.pdf IA: Teoria de grafs. https://ca.wikipedia.org/wiki/Teoria_de_grafs (consulta: 18/6/2020)
- VIQUIPÈDIA: Dades massives. https://ca.wikipedia.org/wiki/Dades_massives (consulta: 3/6/2020)
- VIQUIPÈDIA: Dames. https://ca.wikipedia.org/wiki/Dames (consulta: 2/9/2020)
- VIQUIPÈDIA: Intel·ligència Artificial. https://ca.wikipedia.org/wiki/Intel%C2%B7lig%C3%A8ncia_artificial (consulta: 27/4/2020).
- VIQUIPÈDIA: Teoria de grafs. https://ca.wikipedia.org/wiki/Teoria_de_grafs (consulta: 18/6/2020)
- WIKIPEDIA: Agente Inteligente (Intel·ligència Artificial). https://es.wikipedia.org/wiki/Agente_inteligente_(inteligencia_artificial) (consulta: 9/5/2020)
- WIKIPEDIA: Historia de la Inteligencia Artificial. https://es.wikipedia.org/wiki/Historia_de_la_inteligencia_artificial (consulta: 10/5/2020)
- WIKIPEDIA: Inteligencia Artificial. https://es.wikipedia.org/wiki/Inteligencia_artificial (consulta: 27/4/2020).
- WIKIPEDIA: Problema del viajante. https://es.wikipedia.org/wiki/Problema_del_viajante (consulta: 26/7/2020)

ÍNDEX D'IL·LUSTRACIONS I TAULES

10. IL·LUSTRACIONS

GLOSSARI

11. GLOSSARI DE CONCEPTES ESPECÍFICS

Agents intel·ligents	Entitats capaces de rebre percepcions a través de sensors i processar-les mitjançant una sèrie de normes i condicions que es basen en la racionalitat. L'agent donarà una resposta depenent de si les dades són més o menys completes i del temps que disposa. Últimament s'estan investigant els *sistemes multiagent*, agents intel·ligents amb diferents capacitats que cooperen per solucionar un problema.
Algorisme exhaustiu	Un algorisme exhaustiu és un algorisme que investiga tots els nodes possibles per tal de trobar la solució a un problema.
Algoritme o algorisme	Un algorisme és un conjunt d'instruccions, passos o processos que tenen l'objectiu de trobar una solució a un problema. Als algorismes informàtics aquestes instruccions es troben escrites en un llenguatge de programació al llarg d'un codi.
Arbre (graf)	Els grafs "arbres" s'anomenen així per la seva estructura: d'un node en surten un número determinat més i d'aquests també en surten, i així successivament. És com les branques d'un arbre que es van dividint.

Branca d'un arbre	Una branca d'un arbre d'estats en programació és tota la ruta que es genera a partir d'un node fins al fons de l'arbre.
Condicional	Un condicional en programació és una instrucció que comprova una condició i executa una part del codi depenent de si aquesta condició és certa o falsa.
Domini d'un problema	El domini d'un problema és un conjunt de conceptes interrelacionats que enuncia el problema i que són necessaris per proposar una solució adequada.
Espai d'estats	L'espai d'estats és el conjunt de tots els estats possibles per un problema.
Estat d'un problema	Un estat és un conjunt de conceptes o dades que descriuen el problema en un moment determinat. Per exemple, si el problema fos un joc de taula, un estat és una posició de les peces al tauler.
Estats veïns	Els estats veïns d'un estat són aquells que són semblants a un estat i deriven d'aquest.
Expandir un node	En programació, expandir un node vol dir generar els seus nodes fills.
Funció (en programació)	Una funció en programació és un conjunt de línies de codi que executen un seguit d'accions i normalment retornen alguna cosa. Aquesta és la manera de dividir el codi per tal que sigui més ordenat i encapsulat.

Funció d'avaluació	La funció d'avaluació del problema determina com és d'òptim cada un dels possibles estats (solucions).
Funció heurística	La funció heurística és una funció que avalua si un node fill s'apropa més a la solució que el node pare, tenint en compte la informació que proporciona l'enunciat d'un problema.
Graf	Un graf és un gràfic d'un conjunt de vèrtexs o nodes i arestes que els uneixen. Serveix per modelitzar relacions entre parelles d'objectes. Això s'explica amb més profunditat al següent apartat.
Màxim global	El màxim global és el punt de la representació d'una funció on s'arriba al valor màxim de l'eix d'ordenades (altura màxima).
Màxim local	Un màxim local és el punt de la representació d'una funció on s'arriba al valor màxim per un interval determinat de la funció.
Node	Un node és una representació d'un objecte, conjunt de dades o estat d'un problema
Node arrel	El node arrel és el primer node d'un arbre, del qual deriven la resta de nodes.
Node pare i node fill	Un node fill és un node que prové d'un altre node, que serà el node pare d'aquest.
Operador	Un operador és una acció que es pot fer sobre un node per arribar a un node diferent.

Poder computacional	El poder computacional és la capacitat que té un ordinador per executar processos, tractar amb dades...
Procés iteratiu	Un procés és iteratiu quan es repeteix varies vegades amb l'objectiu de trobar una solució d'un problema.
Recursivitat	La recursivitat és el procés pel qual una funció es crida a ella mateixa.
Xarxes neuronals artificials	És un sistema de la IA que imita el comportament de les neurones biològiques.
Xarxes probabilístiques	Sistema d'IA que ajuda a trobar una solució a un problema encara que hi hagi incertesa respecte a la solució. En trets generals, el que fan les xarxes probabilístiques és buscar la solució que té més possibilitats de ser la correcta o la millor.

ANNEXOS

A.1. CODI DEL JOC DE LES DAMES COMENTAT

La feina de programació del treball es concentra al programa del joc de les dames. Per això és interessant mostrar al complet el codi del programa. D'aquesta manera es dona opció al lector amb coneixements de programació d'analitzar amb profunditat el codi.

El programa del joc de les dames està dividit en tres arxius dependents entre ells. El primer arxiu s'anomena "Algorisme_minimax.py" i conté la funció minimax i les funcions que s'executen al torn de l'ordinador i al del jugador. Es podria dir que aquest és l'arxiu principal. En segon lloc a l'arxiu "Def_classes.py" es defineixen les classes "Moviment", "NodeMAX" i "NodeMIN", que es fan servir a la funció "Minimax". Per últim, l'arxiu "Classe_interficie" conté el codi que forma la interfície gràfica.

A.1.1. Algorisme_minimax.py

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 12 12:21:16 2020

@author: TOM
"""
import sys
import random
from Def_classes import NodeMAX, NodeMIN #S'importen les classes NodeMAX i NodeMIN

INFINIT = sys.maxsize
MENYS_INFINIT = -sys.maxsize-1
PROFUNDITAT = 4

#Funció que rep l'objecte oNode i el moviment, retorna el nou Node modificat
#sempre que el moviment sigui vàlid.
def juga_huma(oNode, moviment):
    moviment = oNode.comprova_moviment(moviment)
    if moviment == None:
        return None
    else:
        nouNode = oNode.get_fill(moviment)
        return nouNode

#Funció Minimax que busca un moviment per l'ordinador, el seu funcionament
#s'explica a l'apartat de l'algorisme Minimax.
def Minimax(oNode, profunditat, alfa, beta, node_retorn):
    if oNode.game_over():
        if oNode.guanyador() == "ordinador":
            return INFINIT
        if oNode.guanyador() == "jugador":
            return MENYS_INFINIT
    elif profunditat == 0:
        return oNode.funcio_avaluacio()

    mov_possibles = oNode.get_mov_possibles()

    if type(oNode) == NodeMAX:
        valor = MENYS_INFINIT
        num_iguals = 2
```

```
41
42          for moviment in mov_possibles:
43              nouNode = oNode.get_fill(moviment)
44
45              if node_retorn != None:
46                  if len(mov_possibles) == 1:
47                      node_retorn.importa_dades(nouNode)
48                      return None
49
50              valor_tmp = Minimax(nouNode, profunditat-1, alfa, beta, None)
51              ##PARCHE:
52              if node_retorn != None:
53                  if valor_tmp == valor:
54                      if random.random() > (1/num_iguals):
55                          valor = valor_tmp
56                          node_retorn.importa_dades(nouNode)
57                      num_iguals += 1
58                  elif valor_tmp > valor:
59                      valor = valor_tmp
60                      node_retorn.importa_dades(nouNode)
61                      num_iguals = 2
62              else:
63                  if valor <= valor_tmp:
64                      valor = valor_tmp
65
66
67              if valor >= beta:
68                  return valor
69              alfa = max(alfa, valor)
70
71          return valor
72
73      elif type(oNode) == NodeMIN:
74          valor = INFINIT
75
76          for moviment in mov_possibles:
77              nouNode = oNode.get_fill(moviment)
78              valor = min(valor, Minimax(nouNode, profunditat-1, alfa, beta, None))
79
80              if valor <= alfa:
81                  return valor
82              beta = min(beta, valor)
83
84          return valor
85
86  #Funció que rep el objecte oNode i el retorna modificat realitzant el moviment de l'ordinador
87  def juga_ordinador(oNode):
88
89      #Es genera un nou objecte de la classe NodeMIN, ja que el nou node serà MIN
90      node_retorn = NodeMIN([0 for j in range(32)])
91
92      #Es truca a la funció Minimax, que retorna la millor puntuació i
93      #canvia el node_retorn per referència
94      punt = Minimax(oNode, PROFUNDITAT, MENYS_INFINIT, INFINIT, node_retorn)
95      return node_retorn
```

A.1.2. Def_classes.py

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Wed Sep  9 14:00:10 2020
4
5   @author: TOM
6   """
7   MAX_reina = 8
8   MAX_dama = 1
9   MIN_dama = -1
10  MIN_reina = -8
11
12  import copy
13  direccions_casellesMAX = [[-1, 4, -1, -1], [4, 5, -1, -1], [5, 6, -1, -1], [6, 7, -1, -1], \
14                            [8, 9, 0, 1], [9, 10, 1, 2], [10, 11, 2, 3], [11, -1, 3, -1], \
15                            [-1, 12, -1, 4], [12, 13, 4, 5], [13, 14, 5, 6], [14, 15, 6, 7], \
16                            [16, 17, 8, 9], [17, 18, 9, 10], [18, 19, 10, 11], [19, -1, 11, -1], \
17                            [-1, 20, -1, 12], [20, 21, 12, 13], [21, 22, 13, 14], [22, 23, 14, 15], \
18                            [24, 25, 16, 17], [25, 26, 17, 18], [26, 27, 18, 19], [27, -1, 19, -1], \
19                            [-1, 28, -1, 20], [28, 29, 20, 21], [29, 30, 21, 22], [30, 31, 22, 23], \
20                            [-1, -1, 24, 25], [-1, -1, 25, 26], [-1, -1, 26, 27], [-1, -1, 27, -1]]
```

```
direccions_casellesMIN = [[-1, -1, 4, -1], [-1, -1, 5, 4], [-1, -1, 6, 5], [-1, -1, 7, 6], \
                          [1, 0, 9, 8], [2, 1, 10, 9], [3, 2, 11, 10], [-1, 3, -1, 11], \
                          [4, -1, 12, -1], [5, 4, 13, 12], [6, 5, 14, 13], [7, 6, 15, 14], \
                          [9, 8, 17, 16], [10, 9, 18, 17], [11, 10, 19, 18], [-1, 11, -1, 19], \
                          [12, -1, 20, -1], [13, 12, 21, 20], [14, 13, 22, 21], [15, 14, 23, 22], \
                          [17, 16, 25, 24], [18, 17, 26, 25], [19, 18, 27, 26], [-1, 19, -1, 27], \
                          [20, -1, 28, -1], [21, 20, 29, 28], [22, 21, 30, 29], [23, 22, 31, 30], \
                          [25, 24, -1, -1], [26, 25, -1, -1], [27, 26, -1, -1], [-1, 27, -1, -1]]
ESQ = 0
DRE = 1
ESQ_ENR = 2
DRE_ENR = 3

#Es defineix la classe Node mare de les classes NodeMAX i NodeMIN
class Node:
    #Mètode que inicialitza l'objecte
    def __init__(self, dades):
        self.dades_tauler = dades

    #Mètode que importa les dades d'un node a un altre
    def importa_dades(self, node_importat):
        self.dades_tauler = node_importat.dades_tauler

    #Mètodes que es defineixen a les classes heretades
    def es_peça_contraria(self, posicio):
        return
    def es_peça_propia(self, posicio):
        return
    def es_dama_contraria(self, posicio):
        return
    def es_dama_propia(self, posicio):
        return
    def es_reina_contraria(self, posicio):
        return
    def es_reina_propia(self, posicio):
        return
    def get_direccions(self):
        return
    def get_mata(self, posicio):
        return
    def posicio_matades(self, posicio):
        return
    def es_ultima_fila(self, posicio):
        return
    def es_d_propia(self, num):
        return
    def reina_propia(self):
        return

    #Mètode que busca els moviments possibles d'una dama i els acumula a la llista recorreguts
    def get_jugades_dama(self, recorregut, recorreguts):
        #direccions = [[ESQ, moviment_esq, mata_esq], [DRE, moviment_dre, mata_dre]]
        direccions = [[ESQ, False, False], [DRE, False, False]]
        if recorregut.es_primer_mov():
            for direccio in direccions:
                if self.es_pot_moure(recorregut.casella_inici(), direccio[0]):
                    direccio[1] = True
                    nou_recorregut = recorregut.clone()
                    nou_recorregut.canvi_dades([self.get_pos(recorregut.casella_inici(), direccio[0]),\
                                                direccio[0], -1, 0])
                    recorreguts.append(nou_recorregut)
        for direccio in direccions:
            if not direccio[1]: #if not moviment_esq:
                es_pot, nou_recorregut = self.es_pot_matar(recorregut.casella_inici(), direccio[0], recorregut)
                if es_pot:
                    direccio[2] = True
                    self.get_jugades_dama(nou_recorregut, recorreguts)

        if not recorregut.es_primer_mov() and not direccions[0][2] and not direccions[1][2]:
            recorreguts.append(recorregut)

    #Mètode que busca els moviments possibles d'una reina i els acumula a la llista recorreguts
    def get_jugades_reina(self, recorregut, recorreguts):

        #direccions = [[ESQ, mata_esq ], [DRE, mata_dre], \
        #              [ESQ_ENR, mata_esq_enr], [DRE_ENR, mata_dre_enr]]

        direccions = [[ESQ, False], [DRE, False], \
                      [ESQ_ENR, False], [DRE_ENR, False]]
```

```
100
101             if recorregut.es_primer_mov():
102                 for direccio in direccions:
103                     casella_inicial = recorregut.casella_inici()
104
105                     while self.es_pot_moure(casella_inicial, direccio[0]):
106                         nou_recorregut = recorregut.clone()
107                         casella_inicial = self.get_pos(casella_inicial, direccio[0])
108                         nou_recorregut.canvi_dades([casella_inicial, direccio[0], -1, 0])
109                         recorreguts.append(nou_recorregut)
110
111                     es_pot, nou_recorregut = self.es_pot_matar(casella_inicial, direccio[0], recorregut)
112                     if es_pot:
113                         self.get_jugades_reina(nou_recorregut, recorreguts)
114
115             if not recorregut.es_primer_mov():
116                 for direccio in direccions:
117                     if direccio[0] != self.dir_inversa(recorregut.get_dades_mov()[-1][1]):
118                         es_pot, nou_recorregut = self.reina_es_pot_matar(recorregut.casella_inici(),\
119                                                                          direccio[0], recorregut)
120                         if es_pot:
121                             direccio[1] = True
122                             self.get_jugades_reina(nou_recorregut, recorreguts)
123
124                 if not direccions[0][1] and not direccions[1][1] and not direccions[2][1] and not direccions[3][1]:
125                     recorreguts.append(recorregut)
126
127         #Mètode que retorna la direcció inversa a la donada
128         def dir_inversa(self, direccion):
129             if direccion == 0:
130                 return 3
131             elif direccion == 1:
132                 return 2
133             elif direccion == 2:
134                 return 1
135             else:
136                 return 0
137
138         #Mètode que retorna la posició que resulta de moure des d'una casella en una direcció
139         def get_pos(self, posicio_inicial, direccio):
140             return self.get_direccions()[posicio_inicial][direccio]
141
142         #Mètode que comprova si es pot moure en una direcció des d'una casella inicial
143         def es_pot_moure(self, posicio_inicial, direccio):
144             pos = self.get_pos(posicio_inicial, direccio)
145             if pos == -1:
146                 return False
147             else:
148                 return(self.dades_tauler[pos] == 0)
149
150         #Mètode que comprova si es pot matar des d'una posició en una direcció
151         def es_pot_matar(self, posicio_inicial, direccio, recorregut):
152             pos_diag = self.get_pos(posicio_inicial, direccio)
153             if pos_diag != -1:
154                 pos_2diag = self.get_pos(pos_diag, direccio)
155                 if pos_2diag != -1:
156                     nou_recorregut = recorregut.clone()
157
158                     if self.es_peça_contraria(pos_diag) and self.es_pos_buida(pos_2diag):
159                         nou_recorregut.canvi_dades([pos_2diag, direccio, pos_diag, self.dades_tauler[pos_diag]])
160                         if recorregut.no_esta_morta(pos_diag):
161                             return True, nou_recorregut
162
163             return False, None
164
165         #Mètode que comprova si una reina pot matar en una direcció des d'una posició
166         def reina_es_pot_matar(self, posicio_inicial, direccio, recorregut):
167
168             while self.es_pot_moure(posicio_inicial, direccio):
169                 posicio_inicial = self.get_pos(posicio_inicial, direccio)
170
171             es_pot, nou_recorregut = self.es_pot_matar(posicio_inicial, direccio, recorregut)
172             if es_pot:
173                 return True, nou_recorregut
174             else:
175                 return False, None
176
177         #Mètode que comprova si un moviment és vàlid
178         def comprova_moviment(self, moviment):
179             recorreguts = self.get_mov_possibles()
180
181             moviment_valid = None
182             for recorregut in recorreguts:
183                 if recorregut.coincideix_mov(moviment):
184                     moviment_valid = recorregut
185                     break
```

```
        return moviment_valid

    #Mètode que calcula tots els moviments possibles des d'un tauler determinat
    def get_mov_possibles(self):
        recorreguts = []
        for i in range(32):
            if self.es_dama_propia(i):
                self.get_jugades_dama(Moviment([[i]]), recorreguts)
            elif self.es_reina_propia(i):
                self.get_jugades_reina(Moviment([[i]]), recorreguts)

        mata = False
        for recorregut in recorreguts:
            if recorregut.mata():
                mata = True
                break
        if mata:
            recorreguts_mata = []
            for recorregut in recorreguts:
                if recorregut.mata():
                    recorreguts_mata.append(recorregut)
            recorreguts = recorreguts_mata

        return recorreguts

    #Mètode que comprova si una posició està buida
    def es_pos_buida(self, posicio):
        return self.dades_tauler[posicio] == 0

    #Mètode que comprova si la partida s'ha acabat
    def game_over(self):
        guanya = True
        for i in range(32):
            if self.es_peça_propia(i):
                guanya = False
                break

        mov_possibles = self.get_mov_possibles()

        flag = (not guanya and (len(mov_possibles) > 0))
        return(not flag)

    #Mètode que retorna el gunyador de la partida
    def guanyador(self):
        if type(self) == NodeMIN:
            return("ordinador")
        if type(self) == NodeMAX:
            return("jugador")

    #Mètode que genera un node fill a partir d'un node i un moviment
    def get_fill(self, moviment):
        posicions = moviment.tradueix_mov()
        peces_matades = moviment.peces_matades()

        copia_dades = copy.copy(self.dades_tauler)
        if self.es_ultima_fila(posicions[-1]) and self.es_d_propia(copia_dades[posicions[0]]):
            copia_dades[posicions[-1]] = self.reina_propia()
        else:
            copia_dades[posicions[-1]] = copia_dades[posicions[0]]
        copia_dades[posicions[0]] = 0
        if peces_matades != None:
            for peça in peces_matades:
                copia_dades[peça] = 0

        return self.crea_node_fill(copia_dades)

    def funcio_avaluacio(self):
        return(sum(self.dades_tauler))



#Definició de la classe heretada NodeMAX
class NodeMAX(Node):

    def es_d_propia(self, num):
        return(num == MAX_dama)

    def reina_propia(self):
        return(MAX_reina)
```

```
267
268     def es_ultima_fila(self, posicio):
269         return(posicio in [0, 1, 2, 3])
270
271     def es_peça_contraria(self, posicio):
272         return self.dades_tauler[posicio] < 0
273
274     def es_peça_propia(self, posicio):
275         return self.dades_tauler[posicio] > 0
276
277     def es_dama_contraria(self, posicio):
278         return self.dades_tauler[posicio] == MIN_dama
279
280     def es_dama_propia(self, posicio):
281         return self.dades_tauler[posicio] == MAX_dama
282
283     def es_reina_contraria(self, posicio):
284         return self.dades_tauler[posicio] == MIN_reina
285
286     def es_reina_propia(self, posicio):
287         return self.dades_tauler[posicio] == MAX_reina
288
289     def get_direccions(self):
290         return direccions_casellesMIN
291
292     def crea_node_fill(self, dades):
293         return NodeMIN(dades)
294
295 #Definició de la classe heretada NodeMIN
296 class NodeMIN(Node):
297
298     def es_d_propia(self, num):
299         return(num == MIN_dama)
300
301     def reina_propia(self):
302         return(MIN_reina)
303
304     def es_ultima_fila(self, posicio):
305         return(posicio in [31, 30, 29, 28])
306
307     def es_peça_contraria(self, posicio):
308         return self.dades_tauler[posicio] > 0
309
310     def es_peça_propia(self, posicio):
311         return self.dades_tauler[posicio] < 0
312
313     def es_dama_contraria(self, posicio):
314         return self.dades_tauler[posicio] == MAX_dama
315
316     def es_dama_propia(self, posicio):
317         return self.dades_tauler[posicio] == MIN_dama
318
319     def es_reina_contraria(self, posicio):
320         return self.dades_tauler[posicio] == MAX_reina
321
322     def es_reina_propia(self, posicio):
323         return self.dades_tauler[posicio] == MIN_reina
324
325     def get_direccions(self):
326         return direccions_casellesMAX
327
328     def crea_node_fill(self, dades):
329         return NodeMAX(dades)
330
331
332 #Definició de la classe Moviment
333 class Moviment:
334
335     def __init__(self, recorregut):
336         self.dades_mov = recorregut
337
338     def es_primer_mov(self):
339         return len(self.dades_mov) == 1
340
341     def casella_inici(self):
342         return self.dades_mov[-1][0]
343
344     def get_dades_mov(self):
345         return self.dades_mov
346
347     def coincideix_mov(self, moviment):
348         mov = []
349         for posicio in self.dades_mov:
350             mov.append(posicio[0])
351         return mov == moviment
```

```
352
353     def mata(self):
354         peces_matades = self.peces_matades()
355         return(len(peces_matades) > 0)
356
357     def peces_matades(self):
358         matades = []
359         for posicio in self.dades_mov:
360             if posicio != self.dades_mov[0]:
361                 if posicio[2] != -1:
362                     matades.append(posicio[2])
363         return matades
364
365     def clone(self):
366         return(copy.deepcopy(self))
367
368     def canvi_dades(self, noves_dades):
369         self.dades_mov.append(noves_dades)
370
371     def tradueix_mov(self):
372         moviment_traduit = []
373         for i in self.dades_mov:
374             moviment_traduit.append(i[0])
375         return moviment_traduit
376
377     def no_esta_morta(self, pos_peça_matar):
378         for i in range(1, len(self.dades_mov)):
379             if self.dades_mov[i][2] == pos_peça_matar:
380                 return False
381         return True
```

A.1.3. Classe_interficie.py

```
1   #-*- coding: utf-8 -*-
2   """
3   Created on Sat Oct 10 16:23:16 2020
4
5   @author: TOM
6   """
7   from tkinter import Tk,Text,Button,END,re, Frame, Canvas
8   from PIL import ImageTk
9   from Def_classes import NodeMAX, NodeMIN
10  import Algorisme_minimax as CI
11
12  MAX_reina = 8
13  MAX_dama = 1
14  MIN_dama = -1
15  MIN_reina = -8
16
17  COORD = [[0, 7], [2, 7], [4, 7], [6, 7], \
18           [1, 6], [3, 6], [5, 6], [7, 6], \
19           [0, 5], [2, 5], [4, 5], [6, 5], \
20           [1, 4], [3, 4], [5, 4], [7, 4], \
21           [0, 3], [2, 3], [4, 3], [6, 3], \
22           [1, 2], [3, 2], [5, 2], [7, 2], \
23           [0, 1], [2, 1], [4, 1], [6, 1], \
24           [1, 0], [3, 0], [5, 0], [7, 0]]
25
26
27  class Interficie:
28      def __init__(self, finestra, Node_inicial):
29
30          #S'inicialitza la finestra, la llista de botons i dues
31          #variables booleanes que es faran servir més endevant
32          self.finestra=finestra
33          self.finestra.title("Joc de les dames")
34          self.finestra.iconbitmap("Icona_pestanya.ico")
35          self.llista_botons = []
36          self.click_bloquejat = False
37          self.tauler_marcat = False
38
39          #S'inicialitza el quadre que determina les mesures de la pestanya
40          self.quadre = Canvas(finestra, width=640, height=704)
41          self.quadre.pack()
42
43          #S'inicialitza el quadre que fa de fons del tauler
44          self.quadre.create_rectangle(63, 63, 578, 578, fill="DarkSeaGreen1")
45
46          #S'inicialitzen dues llistes que emmagatzemen els moviment mentre es
47          #va introduint, una el té en coord i l'altra en números
48          self.mov_acumulat = []
49          self.mov_acumulat_num = []
50
51          #S'inicialitza el requadre on es mostra el text
52          self.Requadre_text = self.quadre.create_rectangle(63, 680, 578, 630, fill="snow3")
53
54          #Es posa el text Moviment: dins el requadre i es crea una altra cadena de text buida
```

```
        self.ID_text_Moviment = self.quadre.create_text(80, 655, anchor="w", text = "Moviment:", font=("Helvetica",18))
        self.ID_text_requadre = self.quadre.create_text(198, 655, anchor="w", text = "", font=("Helvetica",18))

        #Es creen els 32 botons
        for i in range(32):
            oBoto = Boto(self, i)
            self.llista_botons.append(oBoto)

        #Es carreguen les dades del primer node
        self.carregar_dades(Node_inicial, False)

        #S'inicialitza la flexa de fer els moviments
        self.Boto_fer_mov = Button(finestra, image=Fletxa, border=0, bg="Coral1", height=47, width=47, command=lambda:self.fer_mov())
        self.Boto_fer_mov.place(x=529, y=631)

        #Es posen els números i les lletres al voltant
        llista_num = ["8","7","6","5","4","3","2","1"]
        for i in range(8):
            self.quadre.create_text(44, 96+64*i, text=llista_num[i], width=32, font=("Helvetica",18))
        llista_lletres = ["a","b","c","d","e","f","g","h"]
        for i in range(8):
            self.quadre.create_text(96+64*i, 600, text=llista_lletres[i], width=32, font=("Helvetica",18))


    #Mètode que carrega les dades d'un node corresponent a un tauler determinat
    def carregar_dades(self, nouNode, es_mov_ordinador):
        for i in range(32):
            if self.llista_botons[i].codi_boto != nouNode.dades_tauler[i]:
                if es_mov_ordinador:
                    self.llista_botons[i].marcar_mov(nouNode.dades_tauler[i])

                self.llista_botons[i].canvi_dada(nouNode.dades_tauler[i])


    #Mètode que mostra un text al requadre
    def show_text(self, noutext, es_mov):
        if es_mov:
            self.quadre.itemconfig(self.ID_text_requadre, text=self.mov_acumulat)
            if len(self.mov_acumulat) > 9:
                self.mov_acumulat = []
                self.mov_acumulat_num = []
        else:
            self.quadre.itemconfig(self.ID_text_requadre, text=noutext)


    #Mètode que és la instrucció del botó fer_mov
    def fer_mov(self):
        if len(self.mov_acumulat_num) > 0:

            #Es truca a juga_huma
            global oNode
            nouNode = CI.juga_huma(oNode, self.mov_acumulat_num)

            #Si el moviment no es valid es mostra moviment incorredte i es reinicia l'acumulació del moviment
            if nouNode == None:
                self.show_text("moviment incorrecte.", False)
                self.mov_acumulat =[]
                self.mov_acumulat_num =[]
                self.tauler_marcat = True

            #Si no passa això:
            else:
                #Es renten les caselles marcades i es carreguen les dades del nou tauler
                for boto in self.llista_botons:
                    boto.oButton.config(bg="PaleGreen4")
                oNode = nouNode
                self.carregar_dades(oNode, False)

                #Es reinicia el moviment acumulat i es borra el text
                self.mov_acumulat =[]
                self.mov_acumulat_num =[]
                self.show_text("", False)
                self.click_bloquejat = True

                #Es mostra al quadre de text que es el torn de l'ordinador i s'actualitza la pestanya
                self.quadre.itemconfig(self.ID_text_Moviment, text="Torn ordinador")
                self.finestra.update()

                #Es comprova si algú ha guanyat:
                if oNode.game_over():
                    self.show_text("Guanyador: " + oNode.guanyador(), False)
                    self.quadre.itemconfig(self.ID_text_Moviment, text="")

                #Si ningu ha guanyat es truca a juga_ordinador i es carreguen les dades
                else:
                    oNode = CI.juga_ordinador(oNode)
                    self.carregar_dades(oNode, True)

                    #Es comprova si algú ha guanyat:
                    if oNode.game_over():
                        self.show_text("Guanyador: " + oNode.guanyador(), False)
                        self.quadre.itemconfig(self.ID_text_Moviment, text="")

                    #Es configura tot pel torn de l'humà
                    else:
                        for boto in self.llista_botons:
                            self.click_bloquejat = False
                            self.quadre.itemconfig(self.ID_text_Moviment, text="Moviment:")
```

```
153
154 #Classe botó
155 class Boto:
156     #S'inicialitzen les propietats del botó
157     def __init__(self, Interficie, i):
158         self.oButton = Button(Interficie.finestra, image=IMG_buit, border=0, bg="PaleGreen4", height=64, width=64, \
159                               command=lambda:self.click())
160         self.oButton.place(x=((COORD[i][0]+1)*64), y=((COORD[i][1]+1)*64))
161         self.marcat = False
162         self.codi_boto = 0
163         self.pos = i
164         self.oInterficie = Interficie
165
166     #Mètode que canvia la peça que hi ha a un botó
167     def canvi_dada(self, dada_nova):
168         if dada_nova == MAX_reina:
169             self.codi_boto = MAX_reina
170             self.oButton.config(image=IMG_reinaB)
171
172         elif dada_nova == MIN_reina:
173             self.codi_boto = MIN_reina
174             self.oButton.config(image=IMG_reinaN)
175
176         elif dada_nova == MAX_dama:
177             self.codi_boto = MAX_dama
178             self.oButton.config(image=IMG_damaB)
179
180         elif dada_nova == MIN_dama:
181             self.codi_boto = MIN_dama
182             self.oButton.config(image=IMG_damaN)
183
184         else:
185             self.codi_boto = 0
186             self.oButton.config(image=IMG_buit)
187
188     #Mètode que tradueix el moviment de coord a nombre
189     def tradueix_mov(self, mov):
190         coord = ["1a", "1c", "1e", "1g", "2b", "2d", "2f", "2h", "3a", "3c", "3e", "3g", "4b", "4d", "4f", \
191                  "4h", "5a", "5c", "5e", "5g", "6b", "6d", "6f", "6h", "7a", "7c", "7e", "7g", "8b", "8d", "8f", "8h"]
192         return coord[mov]
193
194     #Mètode que és la instrucció dels botons que són les caselles del tauler
195     def click(self):
196         #Acumula la posició a mov_acumulat
197         if not self.oInterficie.click_bloquejat:
198             self.oInterficie.mov_acumulat_num.append(self.pos)
199             self.oInterficie.mov_acumulat.append(self.tradueix_mov(self.pos))
200
201             #Renta les posicions que s'havien marcat del mov de l'ordinador
202             if self.oInterficie.tauler_marcat:
203                 for boto in self.oInterficie.llista_botons:
204                     boto.oButton.config(bg="PaleGreen4")
205                 self.oInterficie.tauler_marcat = False
206
207             #Es marca el botó  on s'ha fet click
208             self.oButton.config(bg="springgreen3")
209             self.oInterficie.show_text(self.tradueix_mov(self.pos), True)
210
211     #Mètode que marca el moviment que fa l'ordinador
212     def marcar_mov(self, canvi_fitxa):
213         if self.codi_boto<0:
214             self.oButton.config(bg="coral1")
215         else:
216             self.oButton.config(bg="springgreen3")
217         self.oInterficie.tauler_marcat = True
218
219 dades_tauler = [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1]
220 #dades_tauler = [1,0,0,1, 1,0,0,0, 1,1,-8,0, 0,0,0,0, 0,-1,-1,0, -1,-1,0,1, -1,0,0,0, -1,-1,0,8]
221 #dades_tauler = [0,0,0,0, 8,0,0,0, -1,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,-8,0, 0,0,-8,0]
222 oNode = NodeMIN(dades_tauler)
223 finestra_principal = Tk()
224
225 IMG_damaB = ImageTk.PhotoImage(file="Dama_blanca.png")
226 IMG_damaN = ImageTk.PhotoImage(file="Dama_negra.png")
227 IMG_reinaB = ImageTk.PhotoImage(file="Reina_blanca.png")
228 IMG_reinaN = ImageTk.PhotoImage(file="Reina_negra.png")
229 IMG_buit = ImageTk.PhotoImage(file="Buida.png")
230 Fletxa = ImageTk.PhotoImage(file="Fletxa.png")
231
232 joc_dames = Interficie(finestra_principal, oNode)
233 finestra_principal.mainloop()
```

A.2. PROPOSTA DE TREBALL: MILLORA DE LA FUNCIÓ D'AVALUACIÓ

La millora i anàlisi de la funció d'avaluació és molt important per al programa de les dames, ja que és una part que té molta rellevància en la manera de jugar de l'ordinador. Degut a això, en un primer moment es van voler fer proves per saber amb quina funció juga millor l'ordinador, per tal de millorar el comportament del programa. A continuació s'explica el plantejament proposat i s'introdueixen alguns estudis relacionats amb el tema.

A.2.1. Plantejament de la investigació i estudis relacionats

Amb el programa elaborat fins ara hi ha dos factors determinants que fan que l'ordinador jugui millor o pitjor: la funció d'avaluació i els nivells de profunditat de moviments que explora l'algorisme. Aquest últim factor no té molt marge de canvi, ja que si es posen massa nivells triga molt en decidir el moviment i el joc es fa avorrit. Ja tenint el programa acabat i fent proves, s'ha observat que a partir de 6 nivells de profunditat el programa va massa lent. Així doncs, s'ha triat el 6 com a màxim de nivell de profunditat.

Ara bé, la funció d'avaluació sí que es pot millorar gairebé tot el que es vulgui, depenent del temps que s'inverteixi. En primer lloc, la solució ideal que augmentaria moltíssim la intel·ligència del programa seria fer que l'algorisme fos adaptatiu. Si això s'implementés, la computadora aniria canviant la funció d'avaluació durant el transcurs de la partida, com ho fa un humà quan juga. Per exemple, no té el mateix valor una reina a l'inici de la partida, quan encara no n'hi ha, que una reina quan els dos jugadors ja en tenen dues. Però és molt difícil fer aquesta millora sense modificar tota l'estructura de l'algorisme, així que aquesta opció queda descartada.

Pel que fa a la funció d'avaluació en si, es poden tenir en compte molts més factors que els que s'estan considerant:

- Es pot avaluar si les peces de la primera fila s'han mogut. A les dames, com més s'enredereix el moviment d'aquestes millor. Si les peces pròpies bloquegen les posicions de la primera fila el contrincant no pot fer reines.

- Sempre és preferible moure les peces en direcció al centre del tauler. Si es mouen les peces als extrems del tauler la seva mobilitat es veu reduïda ja que es troben amb els límits i només es poden moure en una direcció. Així doncs, també es podria considerar la mobilitat de les peces del tauler.
- Per últim, si es té desavantatge quan queden poques peces és pitjor que quan es té desavantatge amb moltes peces. Això també es pot tenir en compte.

A més a més, encara s'ha d'analitzar quin és el millor valor que se li ha de donar a les dames i a les reines. De moment, se li ha donat un valor de 1 a les dames i de 8 a les reines, però no té perquè ser la millor combinació.

En definitiva, s'ha de fer tot un conjunt de proves per analitzar si el programa millora quan s'implementen tots aquests factors. Per fer-ho, s'ha d'aïllar el factor que es vol avaluar i adaptar el programa per tal que jugui contra ell mateix moltes partides. Un dels jugadors tindrà les capacitats "bàsiques" i l'altre tindrà en compte el factor a analitzar. En altres paraules, s'aïlla la variable per saber l'eficiència del canvi.

Aquest procés és similar al que duen a terme els humans per aprendre: quan es juguen moltes partides i s'aprèn quines accions fan que es guanyi més, l'individu millora la seva capacitat per jugar a un joc.

Les primeres proves que s'han fet, en igualtat de condicions entre els dos jugadors, han donat un resultat inesperat. De les 500 partides que ha jugat l'ordinador contra si mateix, 302 partides les ha guanyat el jugador que no fa el primer moviment, 153 el primer en tirar i 45 empats. Això demostra que per les normes del joc determinades al programa i la funció d'avaluació definida, quan l'ordinador juga contra ell mateix, qui comença té desavantatge. Aquí es complica bastant la investigació.

Per començar s'haurien d'investigar i comparar diferents funcions d'avaluació. A partir d'aquí s'ha de trobar una funció d'avaluació que posi en igualtat de condicions els dos jugadors de l'ordinador, per poder experimentar els altres factors de manera fiable. Fent diferents proves es trobaria quins factors i valors de les peces fan que l'ordinador sigui més intel·ligent. Però això ja seria un altre treball de recerca sencer, per això no s'ha fet aquesta part experimental i es deixa com una proposta de treball.

A internet s'han trobat alguns treballs d'investigació professionals sobre el tema de la funció d'avaluació que també poden ser molt útils per a la investigació.

Per exemple, hi ha un treball d'investigació que estudia els efectes sobre el comportament del programa en assignar diferents valors a les peces, mirar quantes peces hi ha a la primera fila, qui està controlant el centre del tauler i les peces amenaçades. Al "Final Project Report Optimal Heuristic For Checkers"[36], escrit per Kevin Gregor i Marcus Boeck-Chenevier, s'exposen els resultats d'aquesta investigació. Es conclou que aplicant totes les heurístiques enumerades el percentatge de victòries augmenta molt.

Un altre estudi sobre el tema és el "*Study on the Evaluation Function Parameters of the Checkers Game Program on Weka Platform"*[37] , elaborat per Li Shuqin, Xue Weiming i Yuan Xiaohua. En aquest estudi s'avaluen els paràmetres de la funció d'avaluació a la Weka, una plataforma d'aprenentatge automàtic d'accés obert. A través d'aquest sistema d'aprenentatge s'analitza en profunditat l'efecte que té fer canvis als paràmetres de la funció.

[36] Informe trobat a la pàgina web:
https://github.com/kevingregor/Checkers/blob/master/Final%20Project%20Report.pdf
[37] Estudi trobat a la página web:
https://www.ijntr.org/download_data/IJNTR01070003.pdf