Resolució de problemes i cerca

PID_00267992

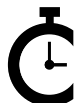
Vicenç Torra i Reventós

Revisió a càrrec de

Jordi Delgado Pin

Andrés Cencerrado Barraqué

Temps mínim de dedicació recomanat: 8 hores



Universitat Oberta de Catalunya

Vicenç Torra i Reventós

Jordi Delgado Pin

Andrés Cencerrado Barraqué

La revisió d'aquest recurs d'aprenentatge UOC ha estat coordinada pel professor: Carles Ventura Royo (2019)

Tercera edició: setembre 2019
© Vicenç Torra i Reventós, Jordi Delgado Pin, Andrés Cencerrado Barraqué
Tots els drets reservats
© d’aquesta edició, FUOC, 2019
Av. Tibidabo, 39-43, 08035 Barcelona
Realització editorial: FUOC

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

Índex

Introducció 5

Objectius 7

1. Introducció a la resolució de problemes i cerca 9
- 1.1. Espai d'estats i representació d'un problema 9
  - 1.1.1. Algunes classes generals de problemes: satisfacció de restriccions i planificació 15
  - 1.1.2. Algunes consideracions addicionals: la importància d'una representació adequada i la qüestió del cost 19
  - 1.1.3. Anàlisi pràctica del problema de les vuit reines 20
  - 1.1.4. Representació d'un problema: implementació amb Python del trencaclosques lineal 21

2. Construcció d'una solució 25
- 2.1. La implementació 28
  - 2.1.1. Implementació amb Python 32
  - 2.1.2. L'arbre de cerca: representació i funcions 33
  - 2.1.3. Els nodes: representació i funcions 36
  - 2.1.4. El problema: representació i funcions 39
- 2.2. Algunes consideracions addicionals 39

3. Estratègies de cerca no informada 42
- 3.1. Cerca en amplada 42
  - 3.1.1. Implementació 44
- 3.2. Cerca en profunditat 44
  - 3.2.1. Cerca en profunditat limitada 46
- 3.3. Exemple pràctic de cerca de solució 52

4. Cost i funció heurística 55
- 4.1. Cerca de cost uniforme 56
- 4.2. Cerca amb funció heurística: cerca àvida 58
- 4.3. Cerca amb funció heurística: algorisme A* 61
  - 4.3.1. Algunes qüestions de la funció heurística 64
  - 4.3.2. Consistència de l'heurístic 65
  - 4.3.3. Implementació 68
- 4.4. Altres mètodes de cerca heurística 71

5. Cerca amb adversari: els jocs 73
- 5.1. Decisions perfectes 73
  - 5.1.1. La poda α-β 85

5.2. Decisions imperfectes 92
5.3. Jocs amb elements d’atzar 94

Activitats.. 97

Exercicis d'autoavaluació.. 97

Solucionari.. 100

Glossari.. 103

Bibliografia... 104

Annex.. 105

Introducció

Heuristic Search Hypothesis

«The solutions to problems are represented as symbol structures. A physical symbol system exercises its intelligence in problem solving by search –that is, by generating and progressively modifying symbol structures until it produces a solution structure.»

A. Newell; H. A. Simon (1976). «Computer Science as Empirical Inquiry: Symbols and Search». *Communications of the ACM* (vol. 3, núm. 19, pàg. 113-126).

En el primer mòdul de l'assignatura hem vist que, d'acord amb la hipòtesi de Newell, l'activitat intel·ligent s'aconsegueix mitjançant l'ús de patrons de símbols per a representar els aspectes significatius del domini del problema, operacions d'aquests patrons per a generar les solucions potencials i cerca per a seleccionar una solució entre aquestes possibilitats.

Vegeu també

Recordeu la hipòtesi de Newell presentada en el mòdul «Què és la intel·ligència artificial» d'aquesta assignatura.

En aquest mòdul ens concentrem en el tercer element: la cerca. Veurem com formalitzar un problema de manera que ens defineixi un espai en què intentarem trobar la solució. Veurem que hi ha diferents tipus de problemes i com es pot fer la cerca en els espais respectius.

Per a fer tot això, aquest mòdul està dividit en els quatre apartats següents:

a) En el primer, es veurà com es pot formular un problema i com la formulació defineix l'anomenat *espai d'estats*. Veurem alguns exemples de problemes i la seva formulació. A més, donarem la representació d'un problema.

b) A continuació, en el segon apartat es mostra com podem trobar una solució a partir de la formulació del problema. Es planteja un esquema que permet fer cerques en l'espai d'estats de moltes maneres diferents. A més, l'esquema és general, de manera que permet resoldre problemes de molts tipus diferents. S'inclou aquí una implementació amb Python dels algorismes de cerca que complementa la donada en el primer apartat per a un problema concret.

c) L'esquema general introduït en l'apartat «Construcció d'una solució» no concreta algunes qüestions relacionades amb l'estratègia a l'hora de seleccionar cada pas de la cerca. En els apartats «Estratègies de cerca no informada» i «Cost i funció heurística», es veuen algunes de les estratègies per al cas en què per a resoldre un problema només fa falta aplicar una seqüència d'accions. Primer es veuran mètodes que com a única informació fan servir només l'espai d'estats (són les estratègies de cerca no informada de l'apartat «Estratègies de cerca no informada») i després els mètodes que fan servir informació addicional com ara el cost o les funcions heurístiques (apartat «Cost i funció heurística»).

d) El mòdul segueix amb el cas dels jocs quan hi ha un adversari. Això és, com triar quin moviment s'ha de fer tenint en compte que juguem amb un contrincant. Primer es considera el cas ideal en què no hi ha problemes ni de memòria ni de temps, i després es planteja què fer en cas de tenir aquestes restriccions.

Objectius

En aquest mòdul assolireu els objectius següents:

1. Descobrir que les solucions d'un problema estan en l'espai d'estats.

2. Aprendre que diferents problemes es poden resoldre amb uns mateixos mètodes.

3. Formular els problemes de manera que sigui possible trobar les solucions.

4. Conèixer algorismes per a resoldre diferents tipus de problemes.

5. Entendre el funcionament dels programes de jocs i copsar la problemàtica de tenir recursos limitats.

1. Introducció a la resolució de problemes i cerca

Com s'ha dit en la introducció, l'objectiu d'un programa d'intel·ligència artificial és resoldre problemes a partir del coneixement de què es disposa. Per a poder-ho fer, cal formalitzar el procés de resolució. Això és, formalitzar com trobem una solució. La formalització es fa a partir de la modelització de les possibles situacions amb què es pot trobar un sistema i les accions que aquest sistema pot fer en el seu entorn de treball. Com veurem, l'espai d'estats és el que modelitza aquest conjunt de situacions possibles i les accions que el relacionen. També veurem que és en aquest espai on es fa una cerca a fi de trobar una solució mitjançant els anomenats *algorismes de cerca*.

En aquest apartat considerarem la formalització d'un problema per a poder-hi aplicar després els algorismes de cerca. A més, considerarem uns exemples a fi d'il·lustrar aquesta formalització. Alguns dels exemples es tornaran a fer servir en els propers apartats.

1.1. Espai d'estats i representació d'un problema

Per tal que un sistema pugui aplicar els algorismes de cerca a un problema concret, cal modelitzar el seu entorn de treball de manera que sigui possible trobar una solució a partir de la situació inicial en què està. Per a fer aquesta modelització, cal considerar un seguit d'elements: quin és l'entorn del sistema, què pot fer aquest sistema per a canviar l'entorn, etc.

La modelització sempre ha de portar cap a estructures semblants per tal que problemes de naturalesa molt diferent es puguin resoldre amb les mateixes eines. Això és, que se'ls puguin aplicar després els mètodes generals existents (alguns dels quals es veuen en aquest mòdul).

Exemples de modelitzacions

A fi d'il·lustrar el procés de modelització considerarem els exemples següents:

a) El trencaclosques lineal

Donada una permutació qualsevol de la seqüència [1, 2, 3, 4], determinar com en construiríem una altra que satisfaci una determinada propietat (per exemple, que estigui ordenada de gran a petita, que representi un nombre múltiple de 2, etc.). Considerarem que, donada una permutació, per a construir-ne una de nova podem intercanviar només dos nombres que estiguin en posicions consecutives de la seqüència. Per exemple, com s'ordena la seqüència [3, 4, 1, 2]?

b) Camí mínim

Donat un mapa de carreteres i dues poblacions, trobar la ruta de distància mínima que permet anar de la primera a la segona. Per exemple, d'acord amb un mapa, com hem d'anar d'un lloc A a un B fent el mínim nombre de quilòmetres possibles?

c) Integració simbòlica de funcions

Donada una expressió numèrica en què apareix un símbol d'integral, trobar una expressió equivalent en la qual aquest símbol no aparegui. Per a fer la integració suposem que disposem d'un conjunt de «regles d'integració» que descriuen com transformar expressions en d'altres d'equivalents. Per exemple, quina és la integral de $x^2e^x + x^3$?

d) Demostració de teoremes

Donada una expressió, demostrar-ne la validesa. La demostració es basa en l'existència d'un conjunt d'axiomes inicials. Per exemple, és cert $a \times (b + c) = a \times b + a \times c$?

La modelització del problema constarà de tres passos: la modelització de l'entorn en què es mou el sistema, la modelització de les accions del sistema i la definició del problema. A continuació, estudiem cadascun d'aquests passos amb detall.

1) Modelització de l'entorn en què es mou el sistema

Un sistema necessita conèixer l'entorn en què està en un instant concret. Atès que aquest entorn canvia –en particular, alguns canvis són a causa de les actuacions del mateix sistema– haurem de poder representar totes les situacions possibles amb què un sistema es pot trobar. La representació del món en un instant concret s'anomena *estat*. D'acord amb això i per a considerar totes les situacions possibles, la modelització haurà de tenir en compte un conjunt d'estats. Corresponen a totes les situacions possibles en tots els instants.

Per a fer la modelització, hem de respondre les preguntes: Què és un estat? Quins són els estats possibles? Tot i això, però, per a la implementació haurem de decidir com es representa un estat.

L'entorn en un moment donat es modelitza amb el que s'anomena *estat*. Cal saber quins són els estats possibles amb què treballarà un sistema. Hem de determinar com representar (implementar) un estat.

D'acord amb això, com que en el cas del trencaclosques lineal, les possibles situacions amb què ens podem trobar són les diferents seqüències, cada seqüència és un estat possible.

Estats possibles en el trencaclosques lineal

En l'exemple del trencaclosques lineal les seqüències [1, 2, 3, 4], [2, 3, 4, 1], [3, 1, 2, 4] són estats possibles. De fet, el conjunt d'estats possibles és el conjunt de totes les seqüències possibles construïbles amb les permutacions de les quatre xifres. Com que el nombre de seqüències possibles formades amb els quatre dígits {1,2,3,4} és 4!, tindrem que els possibles estats són 24 i són, evidentment, els següents: {[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3], [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1], [3, 2, 1, 4], [3, 2, 4, 1], [3, 1, 2, 4], [3, 1, 4, 2], [3, 4, 2, 1], [3, 4, 1, 2], [4, 2, 3, 1], [4, 2, 1, 3], [4, 3, 2, 1], [4, 3, 1, 2], [4, 1, 2, 3], [4, 1, 3, 2]}.

Normalment, la definició d'un problema porta implícites una sèrie de restriccions que remarquen la importància de distingir entre estats possibles i estats vàlids. Un cop es determina la manera de representar un problema, els estats possibles seran aquells que siguin representables. No obstant això, si la definició del problema restringeix algunes característiques, ens podrem trobar amb

estats que són representables, però no són vàlids. Per exemple, si la definició del trencaclosques lineal esmentés que els dos últims dígits mai no poden sumar menys de 5, tindríem que l'estat [4, 3, 2, 1] és un estat representable (la nostra representació permet *expressar-ho*), però no seria un estat vàlid.

2) Modelització de les accions del sistema

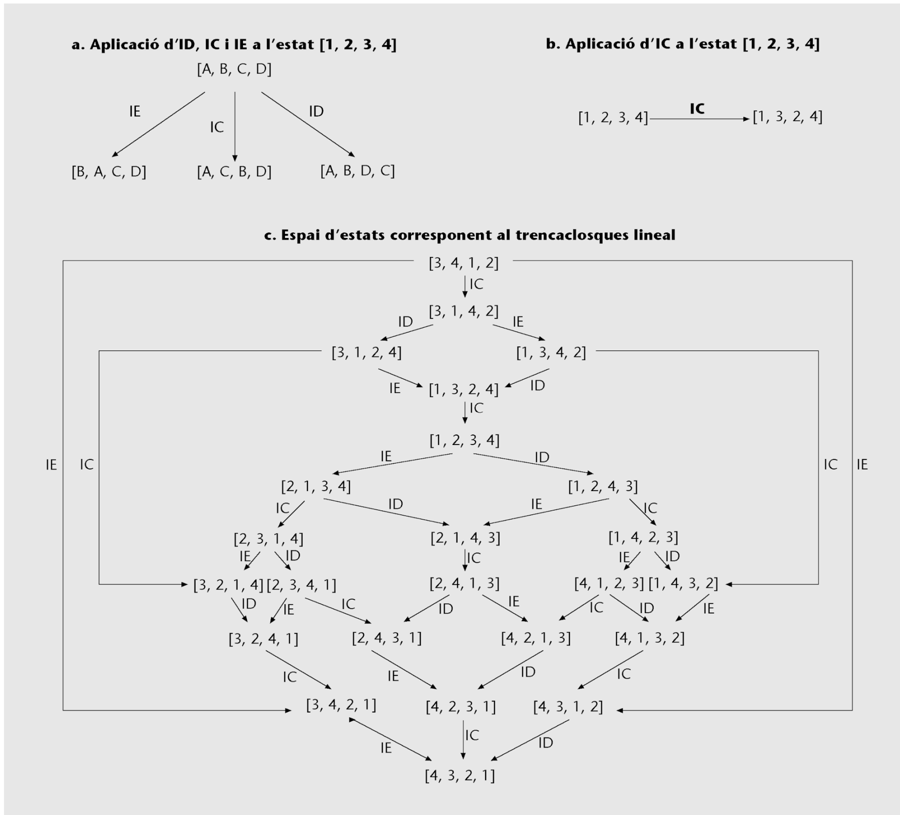
El sistema actua en el seu entorn mitjançant un seguit d'accions. Per a poder trobar el camí cap a la solució a partir de la situació actual necessitem saber quines són les accions que pot fer el sistema i quin efecte tenen aquestes accions en el seu entorn. Atès que el que fan les accions és modificar l'entorn, les podem veure com a formes de passar d'una situació –un estat– a una altra –un altre estat. Per aquesta raó, les accions es modelitzen com a transicions d'un estat a un altre. Al nombre d'accions que es poden aplicar en els estats d'un problema, l'anomenarem *factor de ramificació*. Aquest factor ens serà d'utilitat a l'hora de comparar mètodes de cerca.

Les **accions** d'un sistema es modelitzen com a transicions entre estats. El conjunt de tots els estats possibles i les accions que actuen en aquests estats defineix l'**espai d'estats**. Les accions són dutes a terme mitjançant **operadors** que, com veurem, en funció de com es defineixen podran realitzar més d'una acció diferent.

El **factor de ramificació** és el nombre d'accions que es poden aplicar en els estats d'un problema.

Els estats i les accions defineixen l'anomenat *espai d'estats*. L'espai d'estats es pot representar gràficament com un graf dirigit en què els nodes corresponen als estats i les arestes corresponen a les accions. Un camí en aquest graf indica quina seqüència d'accions permet transformar al sistema una situació en una altra. Trobar una solució correspon a fer una cerca en aquest espai d'estats.

Figura 1



Possibles accions del sistema en el trencaclosques lineal

En el sistema del trencaclosques lineal hi ha tres accions possibles corresponents als tres intercanvis que podem fer. Anomenarem les accions ID, IC, IE que correspondran, respectivament, als intercanvis dret, central i esquerre. Així, donat l'estat [A, B, C, D], l'acció IE ens portarà a l'estat [B, A, C, D], l'acció IC ens portarà a l'estat [A, C, B, D] i l'acció ID a l'estat [A, B, D, C]. En l'esquema a de la figura 1 es fa una representació gràfica d'aquestes accions. La representació gràfica de l'aplicació de l'acció IC a l'estat [1, 2, 3, 4] és a l'esquema **b**. L'espai d'estats corresponent a aquest problema apareix a **c**.

L'entitat encarregada de dur a terme una o diverses accions rep el nom d'operador. La manera com definim un operador determinarà quines i quantes accions diferents pot executar, depenent de si aquest està parametritzat o no, és a dir, de si pot rebre uns paràmetres o arguments que configurin la manera com executa l'acció.

Així, per exemple, les accions ID, IC i IE es poden realitzar mitjançant tres operadors específics diferents que no cal que rebin paràmetres, atès que cadascun ha estat definit per a dur a terme una sola acció, de manera unívoca (fer intercanvi de posició dels dos dígits de la dreta, del centre o de l'esquerra, respectivament). Tanmateix, aquestes tres accions les pot dur a terme un sol operador `intercanvi(posició)` que rep un argument indicant quin parell de dígits s'han d'intercanviar, si els de la dreta, els del centre o els de l'esquerra.

3) Definició del problema

Per a poder trobar la solució del problema necessitem, a més de la modelització dels estats i de les accions, la definició d'allò que volem resoldre. Això és, quin és el problema. Per a fer-ho, necessitem dues coses:

a) D'una banda, hem de saber quina és la situació actual –l'anomenat *estat inicial*.

b) De l'altra, necessitem saber allà a on volem arribar –el que correspon a l'estat objectiu. Atès que sovint no hi ha una única solució al problema, això és, no hi ha un únic estat que compleixi els requeriments, l'objectiu es defineix mitjançant una funció. Aquesta funció, anomenada *funció objectiu*, aplicada a un estat, retorna un valor booleà, que serà cert quan aquest estat satisfaci els requeriments, i fals, quan no els satisfaci. En aquest sentit, l'objectiu és una propietat que només satisfà alguns estats.

Observació

Cal tenir cura, doncs, de no confondre el factor de ramificació amb el nombre d'operadors. El factor de ramificació coincideix amb el nombre d'accions que es poden aplicar a un estat, que no cal que coincideixi amb el nombre d'operadors definits.

La **definició del problema** necessita una descripció de l'estat inicial i una funció objectiu. Les **funcions objectiu** comproven la satisfacció dels requeriments exigits a un estat solució.

Definició del problema del trencaclosques lineal

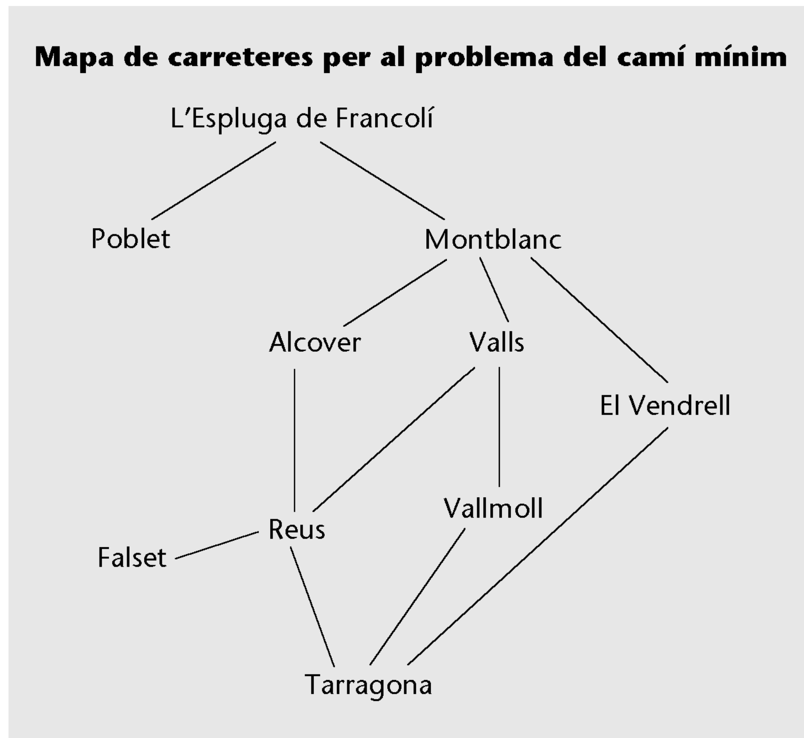
En el cas del trencaclosques lineal, un exemple de problema és el següent: donada la seqüència [2, 4, 1, 3], determinar els moviments per a aconseguir una seqüència que sigui múltiple de dos. En aquest cas, l'estat inicial és [2, 4, 1, 3] i la funció objectiu aplicada a un estat correspon a provar si la seqüència acaba en 2 o en 4. Un altre problema és, donada la seqüència anterior, determinar els moviments per a aconseguir la seqüència ordenada [1, 2, 3, 4]. En aquest cas tenim, com abans, que l'estat inicial és [2, 4, 1, 3] i la funció objectiu serà comprovar que l'estat és [1, 2, 3, 4].

Modelització del problema del camí mínim

La modelització d'aquest problema es farà de manera anàloga al problema del trencaclosques lineal.

Comencem modelitzant l'entorn del sistema. Aquí, les diferents poblacions corresponen als diversos estats amb què es pot trobar el sistema i les accions són les carreteres que connecten dues poblacions diferents (que ens permeten passar d'un estat a un altre). Si considerem el mapa de carreteres per al problema del camí mínim tindrem que els estats possibles són les poblacions que hi apareixen. Les accions que ens permeten canviar d'estat són els segments de carreteres. Per tant, en un estat tindrem tants operadors com camins surtin d'una població. Per exemple, si considerem el mapa de carreteres per al problema del camí mínim tindrem que a l'estat «Montblanc» li podem aplicar quatre operadors que ens porten als estats «L'Espluga de Francolí», «Alcover», «Valls» i «El Vendrell». De fet, es pot veure aquest mapa com l'espai d'estats. Observeu que aquest mapa té la mateixa estructura que l'espai d'estats corresponent al trencaclosques lineal representat anteriorment. El problema serà definir on estem i on volem anar.

Figura 2



Fins ara hem parlat de la modelització dels problemes en termes d'estats i d'accions d'aquests estats amb la idea posada en un estat com a representació d'un entorn, diguem-ne físic, d'un sistema. En general, però, l'abstracció de la idea d'*estat* i d'*acció* permet treballar en altres dominis. No cal que hi hagi un sistema que modifiqui l'entorn, n'hi ha prou amb les idees d'*estat* i *transformació de l'estat*.

Exemples de tipus d'aplicacions

En el cas del sistema per a la integració simbòlica, pot ser que el que realment ens interessi és la integral d'una determinada funció i no pas com el sistema aconsegueix trobar aquesta integral. De la mateixa manera, en el cas de la demostració de teoremes ens pot interessar saber si una determinada propietat es dedueix a partir d'uns axiomes (si és cert $a \times (b + c) = a \times b + a \times c$) i no pas els pas-sos que calen per a arribar a demostrar-la. Això darrer, suposant, evidentment, que el sistema opera correctament.

Exemples d'abstracció de la idea d'*estat*

1) Modelització del problema d'integració simbòlica de funcions

Atès que el sistema ha de tractar amb expressions, definim *estat* com una expressió numèrica. Així, el conjunt d'estats possibles és el conjunt de totes les expressions que permet la notació utilitzada. Per exemple, si considerem sumes, restes, multiplicacions, exponenciacions, etc., podrem tenir expressions com ara x, x^2, $3x$, $5x^2$, e^x, e^{5x} ... En relació amb les accions, tenim que aquestes corresponen a cadascuna de les tècniques d'integració que el sistema preveu. Per exemple, que $(1/2)x^2$ és la integral d'x. L'estat inicial és una expressió qualsevol i un estat final és aquell en què no hi ha cap símbol d'integral.

2) Modelització del problema de demostració de teoremes

Una manera de tractar aquest problema és considerar l'estat com un conjunt de fórmules lògiques. Així, un estat és allò que sabem en un instant concret que es pot demostrar a partir dels axiomes. En aquest cas, les accions corresponen a afegir noves fórmules a un estat que es poden deduir de les primeres.

Atesa una modelització d'un problema, un algorisme de cerca determina la seqüència d'accions que permeten passar de l'estat inicial a un estat objectiu. Aquesta seqüència d'accions s'anomena *pla* o *camí*. Sovint la seqüència d'accions i l'estat objectiu a què s'arriba seran la solució al problema. Ara bé, a vegades només interessa saber quin és l'estat objectiu o, fins i tot, n´hi ha prou amb saber que aquest existeix.

Així, en general tindrem dos tipus d'aplicacions:

1) Aplicacions en què interessa saber com s'arriba a un estat objectiu.

2) Aplicacions en què només interessa saber si és possible arribar a l'estat objectiu.

Tot i que habitualment fem servir el terme *solució* per a denotar només l'estat objectiu, quan ens trobem amb el primer tipus d'aplicacions també el fem servir per a indicar el camí que hi porta.

1.1.1. Algunes classes generals de problemes: satisfacció de restriccions i planificació

Fins ara hem vist com la resolució de problemes s'aplica a alguns exemples concrets. Ara considerem dues classes de problemes i veurem com se'ls hi apliquen aquestes tècniques.

Un tipus particular de problema és l'anomenat *problema de satisfacció de restriccions* (CSP[1]). Tots els problemes d'aquesta família tenen la mateixa forma. Tenim un conjunt de variables a què les hi hem d'assignar un valor. Aleshores, un estat es defineix com un conjunt d'assignacions a unes quantes variables. D'acord amb això, un operador correspon a l'assignació d'un valor a una variable. La solució d'aquests problemes es pot obtenir utilitzant els algorismes de cerca i triant mètodes adequats per a la selecció de quina variable li assignem.

(1) *Constraint satisfaction problem* és l'expressió anglesa per a *problema de satisfacció de restriccions*.

Aquesta classe de problemes té interès perquè molts problemes es poden formalitzar d'aquesta manera (per exemple, trobar una configuració que satisfaci unes restriccions, o també un problema de joc com ara el de situar *n* reines en un tauler). De fet, els problemes de satisfacció de restriccions s'han aplicat a molts problemes reals.

El telescopi espacial Hubble

La programació de les observacions del telescopi espacial Hubble fa servir aquesta formulació de satisfacció de restriccions.

Un altre tipus de problema és el de la planificació. La tasca d'un planificador és la de trobar una seqüència d'accions (un pla) que permeti fer una tasca concreta al sistema. La seqüència descriu quines accions s'han de fer i en quin moment.

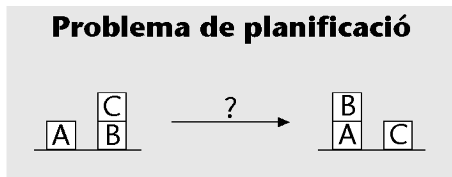
Exemple de planificació

Considerem la ubicació de tres objectes A, B i C i volem determinar els moviments necessaris per a posar cada objecte a la posició requerida. En la figura 3 es mostra on hi ha els objectes i com es volen deixar. Per a aconseguir aquest resultat, el sistema pot agafar un objecte empilat i deixar-lo sobre la superfície o agafar un objecte i posar-lo sobre un altre.

L'espai de plans

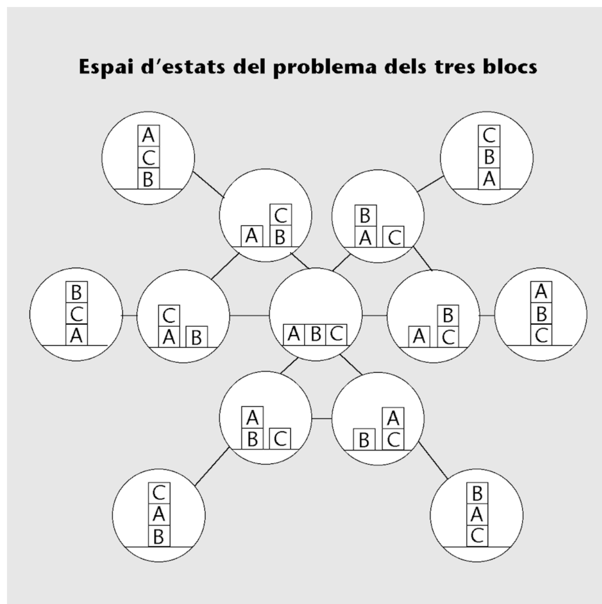
Actualment es prefereix fer la cerca en l'espai de plans en lloc de fer-ho en l'espai d'estats.

Figura 3



La manera més senzilla de tractar la planificació és considerant que cada possible situació dels objectes és un estat. En aquest cas, les accions que relacionen els estats són les accions que pot fer l'agent per a actuar en el món. En la figura 4, es mostra l'espai dels estats corresponent a un problema de planificació d'acord amb aquesta formalització. L'espai d'estats s'ha definit usant els tres blocs ja utilitzats abans: A, B i C. El problema considera els operadors `taula (objecte)` –que deixa l'`objecte` sobre la taula– i `damunt (objecte 1, objecte 2)` –que pren l'`objecte 1` i el situa damunt l'`objecte 2`. L'estat inicial i l'objectiu per a aquest problema s'han donat en la figura 3.

Figura 4. Exemple d'espai d'estats



La figura mostra l'espai dels estats del problema de tres blocs. Font: adaptat de D. S. Weld (1994). «An introduction to least commitment planning» A: *Magazine.*

A continuació, es presenten amb més detall els problemes de satisfacció de restriccions.

Problemes de satisfacció de restriccions

Un problema de satisfacció de restriccions es defineix mitjançant un conjunt de variables, un domini per a cada variable i un conjunt de restriccions que les variables han de satisfer. El problema està resolt quan totes les variables tenen un valor assignat i, a més, se satisfan totes les restriccions.

Exemples de problemes de satisfacció de restriccions

En primer lloc, veurem el problema ben conegut de situar vuit reines en un tauler sense que es matin. Després, considerem el problema conegut com el problema criptoaritmètic.

El problema de les vuit reines: donat un tauler d'escacs (de 8 × 8) s'han de posar vuit reines de manera que no es matin entre si. Això és, que dues reines no coincideixin ni en la mateixa fila, ni en la mateixa columna, ni tampoc en una diagonal.

El problema criptoaritmètic: es considera una expressió aritmètica corresponent a la suma de dos nombres i el seu resultat. Tanmateix, per a cada nombre, en lloc de donar-li els dígits, se li dona una codificació en termes de lletres de l'alfabet. Així, tenim una expressió com ara:

```
  DOS
+TRES
-----
 CINC
```

Suposant que per a cada dígit només hi ha una codificació (no hi poden haver dues lletres diferents a què se'ls assigni el mateix valor), el problema consisteix a trobar una assignació a les lletres de l'alfabet, de manera que la suma quadri. És a dir, en el cas de l'exemple, una solució seria assignar el valor 1 a la lletra E, el valor 2 a la lletra S i la resta d'assignacions com s'indica a continuació. Amb aquesta assignació, la suma anterior correspon a: 952 + 3.812 = 4.764.

```
1 2 3 4 5 6 7 8 9 0

E S T C O N I R D _
```

A més d'aquests dos exemples de problemes de joc, també es poden plantejar d'aquesta manera problemes de planificació temporal (disseny d'horaris) o problemes de configuració (quins components s'han de posar).

Ara definim per a aquesta classe de problemes els tres elements que calen per a resoldre un problema. Si tenim en compte aquests elements, podrem aplicar a un problema d'aquesta classe els algorismes de cerca que veurem en els propers apartats. Recordem que els tres elements són: modelització de l'entorn en què es mou el sistema, modelització de les accions del sistema i definició del problema.

Veurem aquí com la modelització es fa sobre la base d'unes variables i unes restriccions sobre els valors. Primer considerem el cas general i després concretarem per a dos casos d'exemple. Passem, doncs, a concretar els tres elements anteriors:

1) Modelització de l'entorn en què es mou el sistema. L'estat es representarà mitjançant la llista de variables del problema i els valors que tenen assignats aquestes variables. En un estat concret, només es tindrà un subconjunt de totes les variables.

2) Modelització de les accions del sistema. Com que el que volem és que al final cada variable tingui associat un valor, les accions que permetem són les d'assignar un valor a una variable. Per tant, tindrem tantes accions com valors de variables hi hagi.

3) Definició del problema. Inicialment cap variable no té un valor assignat. Volem aconseguir que totes les variables tinguin un valor assignat i que, a més, es compleixin les restriccions. Per tant, l'estat inicial seran les variables sense assignar i la funció objectiu serà comprovar que totes les variables estiguin assignades i que els valors compleixin les restriccions.

Exemples de modelització de problemes de satisfacció de restriccions

1) Problema de les vuit reines

Una manera de modelitzar aquest problema seguint el que s'ha explicat fins ara és considerar vuit variables de manera que cada variable correspongui a la columna en què hi ha la reina de l'*i*-èssima fila. Així, si considerem les variables x_1, x_2, x_3, x_4, x_5, x_6, x_7 i x_8 podem entendre x_1 com la columna en què hi ha la reina de la fila 1, x_2 com la columna en què hi ha la reina de la fila 2. Així, en general, x_i és la columna en què hi ha la reina de la fila *i*-èssima. Com que cada reina pot estar situada, en principi, en qualsevol de les vuit columnes, el domini de cada variable és el conjunt {1, 2, 3, 4, 5, 6, 7, 8}. Amb aquestes variables, un estat correspondrà a tenir valors associats a unes quantes variables (això, de fet, es pot veure com a equivalent a tenir unes quantes reines posades). Una acció serà assignar un valor a una variable que encara no tenia un valor assignat. L'estat inicial serà no tenir cap variable assignada i l'estat final serà tenir valors assignats a totes les variables i que, a més, es compleixin les restriccions (que les reines no es matin).

La formulació de les restriccions serà:

- Que dues reines no estiguin a la mateixa fila: la construcció del problema ja no permet aquest cas.

- Que dues reines no estiguin a la mateixa columna:
  $x_i \neq x_j$ per a tot $i \neq j$

- Que dues reines no estiguin a la mateixa diagonal: hi ha dos casos en què dues reines estan en la mateixa diagonal. Són els de la figura 5 (casos 1r. i 2n.) quan $a = b$. Noteu que suposem que la fila j és més gran que l'i ($j > i$). El primer cas serà si:

$$x_i - x_j = j - i$$

i el segon cas serà si:

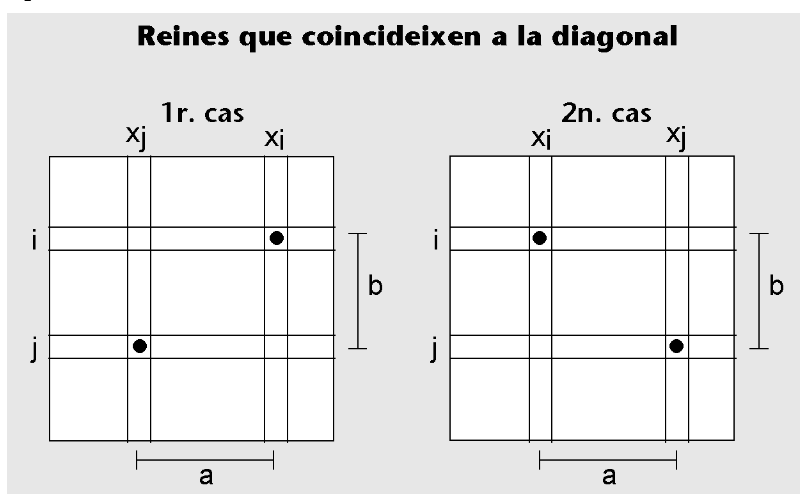
$$x_j - x_i = j - i$$

Per tant, per a exigir que dues reines no estiguin en la mateixa diagonal necessitem:

$$\text{si } j > i,\ x_i - x_j \neq j - i$$

$$\text{si } j > i,\ x_j - x_i \neq j - i$$

Figura 5



2) Problema criptoaritmètic

També es pot formalitzar de manera similar. En aquest cas, hem de tenir una variable per a cadascuna de les lletres que apareixen en l'expressió. Com que cada lletra es pot fer correspondre a un dels dígits del 0 al 9, el domini de cada variable serà el conjunt {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. En aquest cas, com abans, els operadors són triar una variable i una assignació. Finalment, per a definir el problema criptoaritmètic necessitem concretar l'estat inicial i la funció objectiu. L'estat inicial serà no tenir cap variable assignada i la funció objectiu serà que cada variable (cada lletra) tingui un valor i que no hi hagi dues variables a què se'ls hi hagi assignat el mateix valor. Les restriccions seran les que es dedueixen de l'expressió (tenint en compte que quan fem la suma ens en podem emportar): (S + S) mod 10 = C; O + E + [(S + S) div 10] = N; ...

1.1.2. Algunes consideracions addicionals: la importància d'una representació adequada i la qüestió del cost

A causa que els algorismes de cerca es basen en l'espai d'estats que es defineix, interessa que l'espai sigui el més reduït possible per tal que la cerca sigui el menys costosa possible. A vegades, un canvi de representació pot reduir enormement una cerca.

Un exemple del món real que un canvi de representació pot reduir la cerca és el del problema de la planificació. Abans hem considerat que una representació es basa en l'espai d'estats. Actualment es considera la resolució del problema basada en una cerca en l'espai de plans, ja que això és més eficient. En aquest cas, l'estat inicial és un pla buit (sense cap acció) i l'objectiu és aconseguir un pla que porti a una situació que satisfaci les condicions que nosaltres imposem. Un estat és un pla que s'ha especificat només parcialment i les accions són operacions de refinament del pla (per exemple, afegir una acció al pla).

A l'hora de definir un problema hi ha un altre aspecte que a vegades s'ha de tenir en compte: el cost. En alguns problemes, no tots els camins són igual de bons ni tampoc totes les accions. Les accions que fa el sistema es poden avaluar en termes d'una funció de cost. El cost pot correspondre a la dificultat de fer l'acció o al preu que s'ha de pagar per a portar-la a terme. El cost d'un pla es correspon a com sigui de bona una seqüencia d'accions i, sovint, és la suma dels costos de les accions que hi prenen part. Un exemple ben conegut d'això és quan volem determinar com anar d'una població a una altra i tenim diferents alternatives (rutes). En aquest cas, el cost pot correspondre al nombre de quilòmetres entre les dues poblacions, al temps necessari per a fer una ruta o al cost econòmic de fer el desplaçament.

Solució òptima del trencaclosques lineal

Si volem trobar la solució amb el nombre d'inversions més baix, podem assignar a cada acció un cost unitat i, a un camí, la suma dels costos de les accions que hi prenen part. Com que això darrer correspondrà a la longitud del camí podrem prevaler aquelles solucions amb menys inversions.

La qüestió del cost ens permetrà avaluar els diferents algorismes de cerca. Així, direm que una solució és òptima si el camí de l'estat inicial a l'objectiu és mínim (menor que tots els altres camins). Un algorisme serà òptim si troba una solució òptima.

1.1.3. Anàlisi pràctica del problema de les vuit reines

La modelització escollida per a abordar un problema és clau a l'hora de dur a terme qualsevol operació o intent de resolució. Així doncs, cal tenir cura amb cadascuna de les definicions i decisions que es prenen en la modelització, ja que sovint un aspecte (per exemple, la representació dels estats) té una conseqüència directa en d'altres (per exemple, l'elecció de l'estratègia de cerca d'una solució).

Vegeu també

Aprofundirem en el tipus d'estratègies de cerca en el apartat «Construcció d'una solució».

Basant-nos en la modelització que acabem de donar del problema de les vuit reines, ens podem fer algunes preguntes que seran determinants per a poder implementar un procediment de cerca de solucions:

- **Quants estats possibles té el nostre problema?**
  Atès que tenim vuit variables, amb nou possibles valors per a cadascuna (un per columna, més el valor nul que indica la variable sense assignació), tenim 9^8 diferents possibles estats.

- **Són tots els estats possibles vàlids?**
  Els 9^8 possibles estats que acabem d'esmentar són els estats que la nostra modelització **permet representar**. Això no vol dir que tots els estats siguin vàlids, ja que la formalització del problema imposa unes restriccions molt clares que molts d'aquests estats no compliran (per exemple, qualsevol estat en què dues o més variables tinguin el mateix valor associat).

- **Quins són els operadors de què disposem? Com canvien els estats?**
  L'única acció que es defineix en la modelització proposta és «assignar un valor a una variable que encara no tenia un valor assignat». Així doncs, comptem amb un únic operador, que realitzarà aquesta acció. Aquest operador necessitarà rebre com a arguments la variable a modificar i el valor que hi assignarà, de manera que transformarà un estat en un altre mitjançant la modificació de la variable.
  A nivell conceptual, també és vàlid considerar que, com que tenim vuit variables, podem disposar de vuit operadors, en què cadascun d'aquests està «dedicat» a una sola variable.

- **Tots els estats són accessibles des de qualsevol altre estat?**
  No tots els estats seran accessibles des de qualsevol altre estat, atès que l'acció modelada no permet assignar un valor a una variable que ja té un valor assignat. Apart d'aquest detall, es pot considerar que el mateix operador té en compte per si mateix si l'estat que produirà és vàlid o no. Si ho té en compte i, en conseqüència, el nostre operador no pot generar estats invàlids, llavors el nombre d'estats accessibles des de qualsevol altre estat és molt menor que en el cas que l'operador pugui produir estats invàlids.

Com es pot observar, en molts casos aquests plantejaments estan subjectes a la implementació concreta que es faci del problema. Generalment, les definicions de problemes deixen marge per a prendre diferents decisions que poden ser igualment vàlides i respectar les restriccions del problema, tant implícites com explícites.

D'aquestes decisions dependrà, en bona mesura, el rendiment que tindran els nostres algorismes, així que és important dur a terme una bona reflexió a nivell conceptual de la solució que es proposa, tenint en compte les estratègies de resolució que estudiem en els propers apartats.

Vegeu també

Per a entendre el codi de l'apartat, és aconsellable fer una ullada a l'annex que podeu trobar al final del mòdul. Explica el tractament de les llistes de Python que hem adoptat en el codi que trobareu d'ara endavant.

1.1.4. Representació d'un problema: implementació amb Python del trencaclosques lineal

A continuació, farem la representació del problema del trencaclosques lineal amb Python. La seva implementació correspon a definir tots aquells elements que necessitem per a poder aplicar posteriorment els algorismes de cerca. En particular, i tal com s'ha descrit en el subapartat precedent, necessitem els elements següents:

- **Modelització de l'entorn**. Això és, hem de saber com representarem un estat (s'ha dit ja que un estat correspondrà a una permutació de les quatre xifres).

- **Modelització de les accions del sistema**. Hem de definir funcions per a cadascun dels operadors disponibles. Per tant, hem de definir tres funcions, una per a cadascun dels tres intercanvis possibles.

- **Definició del problema**. Hem de saber com representarem l'estat inicial i hem de disposar d'una funció que, aplicada a un estat, ens retorni cert quan aquest sigui un estat objectiu.

Passem, ara, a la definició d'aquests elements:

1) Modelització de l'entorn

Definició de l'estructura per a representar un estat. El més simple és representar un estat mitjançant una llista de quatre nombres corresponents als quatre dígits d'una seqüència. Així, amb Python representarem l'estat mitjançant la seqüència [A, B, C, D].

2) Modelització de les accions del sistema

Definim les funcions `mov_ie`, `mov_ic` i `mov_id` corresponents als intercanvis possibles. Les funcions s'apliquen a l'estat. Per tant, reben l'estat actual com a paràmetre i retornen l'estat nou. Així, tindrem:

```
def mov_ie (estat, info):
    return [cadr(estat), car(estat), caddr(estat), cadddr(estat)]

def mov_ic (estat, info):
    return [car(estat), caddr(estat), cadr(estat), cadddr(estat)]

def mov_id (estat, info):
    return [car(estat), cadr(estat), cadddr(estat), caddr(estat)]
```

L'esquema general que segueixen totes tres funcions és similar: construïm una llista a partir dels quatre elements que defineixen la nova seqüència. La diferència entre les tres funcions és l'ordre en què es prenen els elements. En tots els casos, com que l'estat és una llista, per a aconseguir el primer element n'hi ha prou amb agafar el cap de la llista. Així, tenim que `car(estat)` és el primer element. En canvi, per a aconseguir el segon element, prenem la cua de la llista (tots menys el primer, o sigui, del segon fins el darrer) i d'aquesta cua en prenem el primer element (o sigui, el segon de la llista original). Això és `car(cdr(estat))`. La funció `cadr` és equivalent a aquestes dues crides.

Informació addicional

A més d'un estat les funcions reben un altre paràmetre que anomenem *informació addicional* que, de moment, no considerem, però que ens serà d'utilitat més endavant, en l'apartat «Cerca amb profunditat limitada» dedicada a la seva implementació.

De manera similar, per a aconseguir el tercer element, apliquem dues vegades la funció cua (`cdr`) a l'estat, de manera que obtenim la llista formada pel tercer i quart element. Com que el primer element d'aquesta llista és el tercer, aplicant-hi `car` tindrem el tercer element. Això és: `car(cdr(cdr(estat)))` o, el que és equivalent: `caddr(estat)`. Per a obtenir el quart element hem de fer el mateix que en el cas anterior, però ara aplicant la funció `cdr` una vegada més. Així, tenim que el quart element serà: `car(cdr(cdr(cdr(estat))))` o, equivalentment: `cadddr(estat)`. Si resumim, tenim que per a aconseguir els quatre elements que componen la llista hem de fer:

```
1r element   car (estat)
2n element   cadr(estat)
3r element   caddr(estat)
4t element   cadddr(estat)
```

Els operadors del trencaclosques lineal els emmagatzemem en una llista de parells d'elements que retornarà la funció `tl_operadors` (on `tl` correspon a trencaclosques lineal). Per a cada operador tenim el seu nom i la funció que ens permet obtenir el nou estat a partir d'un estat inicial. Així, per a l'intercanvi esquerra tenim `'ie'` que és el nom de l'operador i la funció corresponent. `mov_ie` representa la funció). De la mateixa manera, tenim `['ic', mov_ic]` per a l'intercanvi central i `['id', mov_id]` per a l'intercanvi dret. Així, la llista d'operadors serà:

```
def tl_operadors():
    return [['ie', mov_ie],
            ['ic', mov_ic],
            ['id', mov_id]]
```

3) Definició del problema

Necessitem saber com representem l'estat inicial i disposar d'una funció objectiu. La representació de l'estat inicial és clara un cop ha estat triada la representació dels estats. Així, l'estat inicial és la seqüència [2, 4, 1, 3].

Quant a la funció objectiu, tenim que quan les solucions del problema són aquells estats on la seqüència acaba en 2 o 4 una possible funció objectiu és aquesta:

```
def tl_funcio_objectiu_parell(estat):
    return cadddr(estat) == 2 or cadddr(estat) == 4
```

Aquesta funció es complirà quan el darrer element de la seqüència `cadddr(estat)` és igual a 2 o igual a 4. Si considerem que només hi ha un estat solució i aquest és el de la seqüència [1, 2, 3, 4] tindrem que una funció objectiu pot ser:

```
def tl_funcio_objectiu(estat):
    return estat == [1,2,3,4]
```

Aquesta funció simplement compararà l'estat amb l'únic estat objectiu.

Ara, amb tots aquests elements podem muntar el problema. Això ho fem definint una funció anomenada *problema*:

```
def problema():
    return [tl_operadors(),
            (lambda info_node_pare, estat, nom_operador: []),
            [4,3,2,1],
            (lambda estat: estat == [1,2,3,4]),
            (lambda estat: [])]
```

En la definició del problema considerem els quatre elements següents: els operadors del problema (en aquest cas `tl-operadors()`), l'estat inicial (la llista `[4,3,2,1]`), la funció objectiu `lambda estat: estat == [1,2,3,4]` i una funció que més endavant ens serà d'utilitat que associa a cada estat informació addicional (de moment, com que no la fem servir la definim de manera que retorni `[]` per a qualsevol estat). El problema es representa mitjançant una llista amb aquests quatre elements amb l'ordre donat.

2. Construcció d'una solució

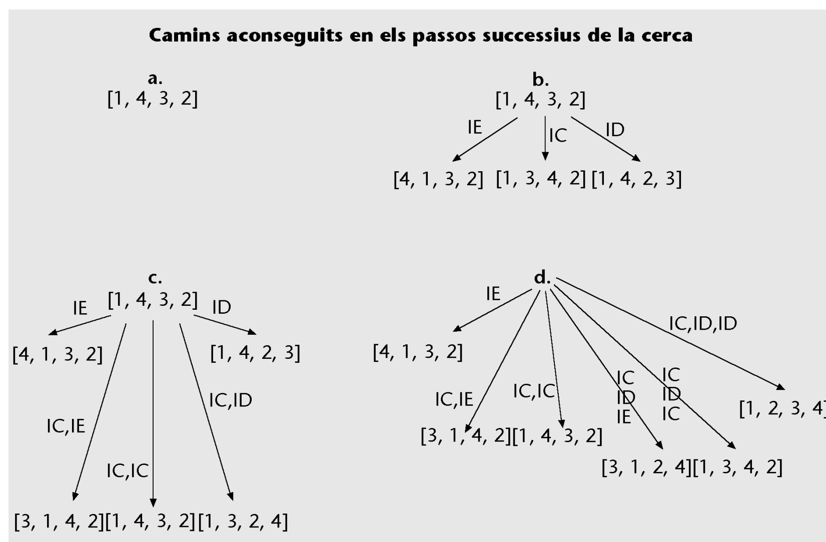
Un cop hem definit el problema hem de passar a trobar-ne la solució. Com que sabem d'allí on partim (l'estat inicial) i allà on volem anar (un estat que satisfaci la funció objectiu), el que hem de fer és trobar un camí que ens hi porti. La cerca correspon a trobar aquest camí en el graf que representa l'espai d'estats.

Un algorisme de cerca construeix el camí (la solució) en passos successius. La idea general és disposar d'un conjunt de camins (solucions) parcials i a cada pas prolongar-los (millorar-los) una mica més. Això s'aconsegueix triant un dels camins del conjunt i considerant aquelles accions que el sistema pot aplicar al darrer estat del camí. Cada acció ens determina un nou camí. Així, prenem un camí i, per a cada acció que es pugui aplicar a l'estat terminal, construïm un nou camí. El camí correspon al camí original més la transició corresponent a l'acció seleccionada.

Procés de construcció de la solució en el trencaclosques lineal

Reprenem el problema del trencaclosques lineal i ho il·lustrem:

Figura 6



Considerem el problema amb un estat inicial igual a [1, 4, 3, 2] i com a objectiu la seqüència [1, 2, 3, 4]. En aquest cas, comencem amb un únic camí buit (sense cap acció) que comença i acaba en l'estat inicial. Aquesta primera situació es representa en el pas [a]. Com que aquest estat no és la solució al problema, considerem quines accions es poden aplicar a aquest estat i a quins estats ens porten: totes tres (IE, IC, ID) són aplicables i ens porten, respectivament, als estats: [4, 1, 3, 2], [1, 3, 4, 2] i [1, 4, 2, 3]. Amb aquesta informació podem construir tres camins alternatius (formats cadascun amb una única acció) que porten als tres estats nous. En el pas **b** hi ha la representació d'aquests tres camins i els estats on s'arriba a partir de l'estat inicial [1, 4, 3, 2].

Prosseguim seleccionant un dels tres camins i aplicant a l'estat terminal les accions que li són adequades (en aquest problema, ho són totes tres). Suposem que el camí seleccionat és el segon (el que mena a [1, 3, 4, 2]), per tant, quan considerem les tres accions obtenim

tres nous camins que porten, respectivament, a [3, 1, 4, 2], [1, 4, 3, 2] i [1, 3, 2, 4]. Una representació gràfica dels camins que tenim en aquest punt es donen en el pas **c**.

El pas següent segueix el mateix esquema. Hem de considerar els camins existents, triar-ne un i aplicar-hi les accions adequades a l'estat terminal. En aquest moment hi ha cinc camins a considerar (els que ja s'han considerat no s'han de considerar de nou, ja que refarien camins que ja tenim). Són els que porten a: [4, 1, 3, 2], [3, 1, 4, 2], [1, 4, 3, 2], [1, 3, 2, 4] i [1, 4, 2, 3]. Si ara seleccionem el camí fins a l'estat [1, 3, 2, 4] tindrem que entre els nous camins n'hi ha un que porta a l'estat objectiu. En el pas d es donen tots els camins que tenim en aquest punt. Es pot veure que el camí IC, ID, IC porta a la seqüència que volíem [1, 2, 3, 4].

Per tal que la part comuna dels camins no aparegui repetida i es pugui veure allò que els camins comparteixen, la implementació de la cerca s'acostuma a representar mitjançant una estructura d'arbre. Així, tenim que l'arrel de l'arbre correspon a l'estat inicial, els nodes de l'arbre són els estats, els arcs corresponen a les accions i les fulles de l'arbre corresponen als estats terminals dels camins. D'aquesta manera, quan resseguim les branques de l'arbre des del node inicial a una fulla, estem repassant un camí des de l'estat inicial a un de terminal passant per tots els estats intermedis. A la vegada, podem conèixer les accions que el sistema ha de fer per a portar a terme els canvis d'estat i aconseguir arribar a un estat fulla (són els arcs de l'arbre).

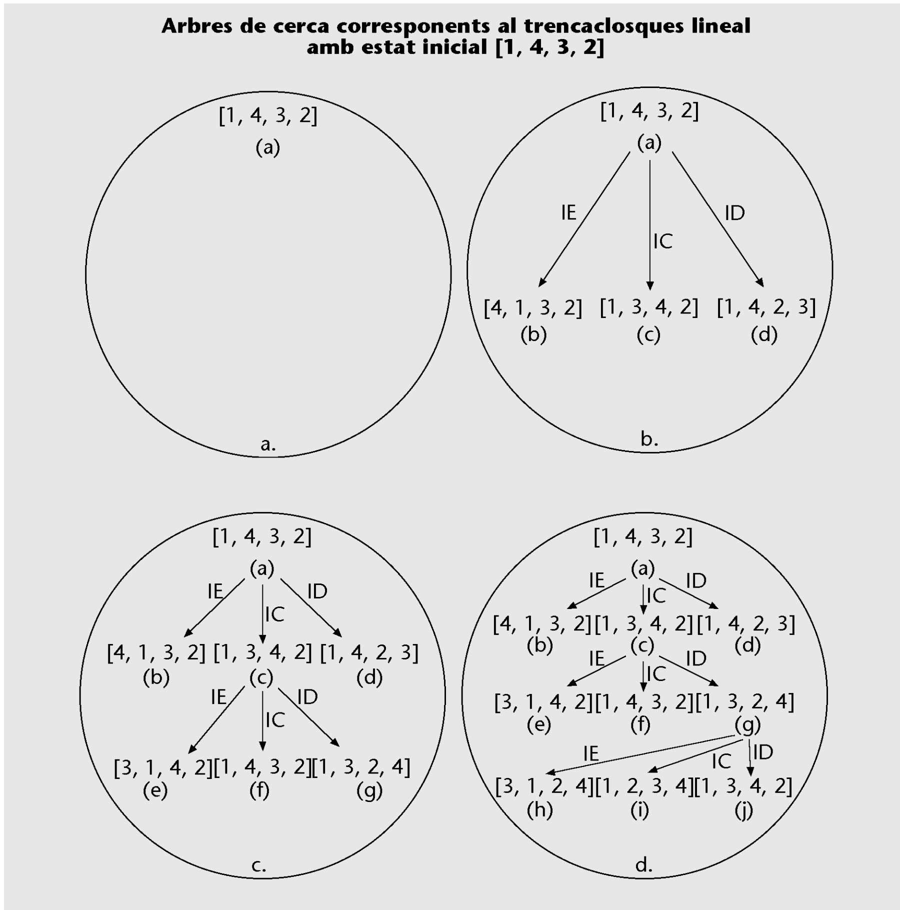
La tria d'un camí correspon a triar un node fulla i l'aplicació de les accions a l'estat terminal correspon a aplicar les accions a l'estat associat al node fulla. L'aplicació de les accions proporciona un conjunt d'estats que són afegits a l'arbre com a descendents del node terminal. El node ja tractat deixarà així de ser fulla. D'aquesta manera, en tot moment, les fulles de l'arbre corresponen a camins que encara no s'han considerat, mentre que tots els nodes que no són a les fulles corresponen a nodes ja tractats. En un moment donat de la cerca anomenarem *frontera de l'arbre de cerca* al conjunt de nodes que no s'han tractat. La consideració d'un node de la frontera i l'aplicació de les accions que li són adequades s'anomena *expansió del node*.

L'expansió d'un node consisteix a aplicar els operadors a l'estat associat a un node.

> Anomenarem *frontera de l'arbre de cerca* el conjunt de nodes no expandits.

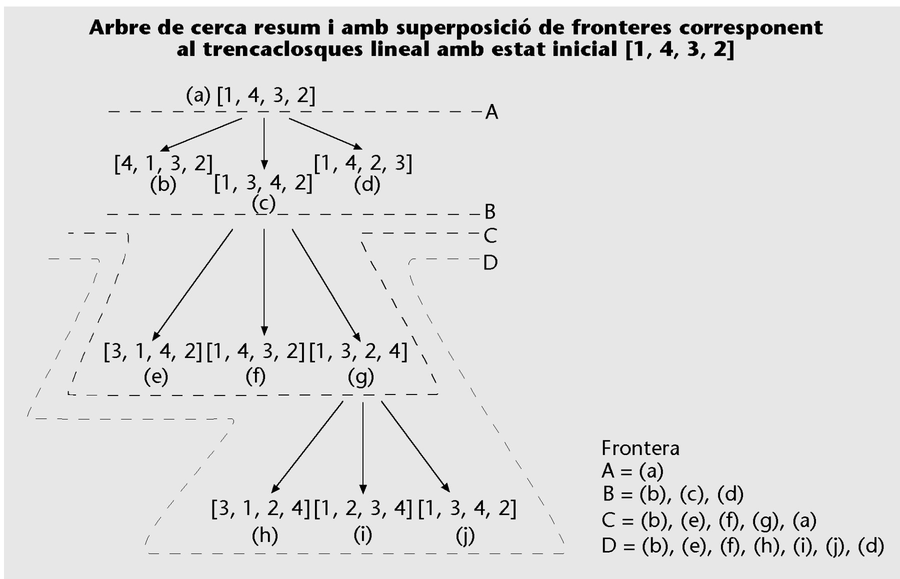
En la figura 7, es donen els arbres de cerca que obtenim seguint els mateixos passos que hem considerat abans. Els nodes de l'arbre disposen d'un identificador del node (a), (b), ... (j) i de la informació dels estats. Els arcs apareixen etiquetats amb el nom de l'acció que permet passar d'un estat al següent.

Figura 7



En la figura 8, apareix un altre cop el darrer arbre de cerca obtingut en els passos successius, ara sobreposant-hi els conjunts frontera. Són les línies A, B, C i D. Les fronteres són els punts on s'ha de triar entre els camins existents. El node terminal del camí escollit és expandit i l'arbre creix amb els nous estats que s'han generat.

Figura 8



Es pot veure que en els successius arbres de la figura 7, els camins des del node arrel a les fulles corresponen als camins que s'han considerat en els passos a, b, c i d. Així tenim que en el pas a hi ha un únic camí (que és buit) del node inicial a si mateix. En el pas b tenim tres camins, tots corresponents a fer una única acció, que porten a [4, 1, 3, 2], [1, 3, 4, 2] i [1, 4, 2, 3] respectivament. Aquests camins són els que apareixen en el segon arbre de la figura 7 que representa els arbres de cerca obtinguts en els passos successius. En el pas (c) tenim els camins que corresponen als que porten als nodes (b), (e), (f), (g), (d) del tercer arbre de la figura 7 que representa els arbres de cerca obtinguts en els passos successius. El mateix passa amb el pas d i el quart arbre de la figura 7.

Tot i que l'arbre de cerca té força semblances amb l'espai d'estats (de fet, un node i la seva expansió es pot sobreimposar a l'espai d'estats), hi ha una diferència important: el graf corresponent a l'espai d'estats té tants nodes com estats hi ha, mentre que en el cas de l'arbre de cerca això no és així, hi ha estats que poden no aparèixer i d'altres aparèixer més d'una vegada. Per exemple, en la figura 8, l'estat [1, 4, 3, 2] apareix en els nodes (a) i (f). El mateix passa amb les arestes. En la cerca podem tornar a considerar un operador aplicat a un estat a què ja havíem aplicat abans. Aquest seria el cas si el node (f) fos expandit.

En la descripció de l'esquema hi ha alguns elements que queden pendents de clarificació. El més important és la tria del node que hem d'expandir. De fet, és el punt més crític de l'algorisme ja que una bona tria ens permetrà arribar més ràpidament a la solució estalviant-nos l'expansió de molts altres nodes. En els apartats següents es descriuen maneres de fer aquestes tries. Abans, però, passarem a veure una implementació de l'esquema general de la cerca.

2.1. La implementació

Per a seguir l'esquema plantejat fins ara, la implementació d'un algorisme de cerca necessita unes estructures de dades que corresponguin a l'arbre de cerca i també les funcions per a construir aquest arbre i operar amb aquest. A continuació, fem una implementació d'aquests elements primer en alt nivell i després amb llenguatge Python. La implementació donada és genèrica, i, per tant, intenta ser apta per a tots els algorismes. Això fa que no sempre sigui l'òptima.

Passem a continuació a la implementació començant per la representació de l'arbre de cerca. L'arbre necessita, en primer lloc, l'estructura de dades corresponent a un node. L'estructura ha de contenir la informació que necessitem per a reconstruir el camí que ens porta des del node inicial a un node solució. De fet, com que fins que un node no és una solució no sabem quin de tots els possibles camins és el bo, és més senzill considerar la reconstrucció del camí des del node solució al node inicial. Sobre la base d'això, considerem que un node ha de contenir la informació següent: quin altre node l'ha generat (quin és el seu node pare) i com l'ha generat (quin és l'operador que aplicat a l'estat del pare retorna l'estat del node que estem considerant). A més, per a simplificar el procés de construcció de l'arbre afegirem l'estat corresponent

a cada node. També hi inclourem un identificador del node. Amb això darrer tindrem que per a saber qui és el pare d'un node n'hi ha prou amb emmagatzemar l'identificador del pare. A banda de tota aquesta informació, alguns algorismes de cerca necessiten informació addicional dels nodes. Aquesta informació també es guardarà al node.

D'acord amb tot això tenim que l'estructura de dades d'un node disposa de la informació següent:

```
Estructura de dades node:
    identificador: etiqueta identificadora
    estat: representació del node (array, struct, etc.)
    node-pare: identificador del pare
    operador-generador: identificador de l'operador
    informacio-addicional: estructura amb informació extra
```

El nombre de nodes que hi ha en el camí des del node inicial o el cost del camí són exemples d'informació addicional.

L'estructura de dades corresponent a l'arbre de cerca haurà de contenir informació de tots els nodes, diferenciant aquells que ja han estat expandits d'aquells que encara no ho han estat. Això s'aconsegueix utilitzant dues estructures: una per a cada conjunt de nodes.

1) Conjunt de nodes ja expandits

Com que les operacions que fem amb aquest conjunt són les d'afegir nodes a l'estructura (quan expandim un node, aquest haurà de passar de l'estructura de nodes a expandir la d'expandits) i obtenir el node que correspon a un determinat identificador (per a poder resseguir el camí hem de poder trobar el pare d'un node a partir del seu identificador). Podem fer servir simplement una llista.

Llista en comptes de taula

De fet, una implementació amb una taula –o una taula de dispersió (segons quins identificadors es facin servir)– és més eficient. Tanmateix, considerem una llista per a reduir els elements necessaris del llenguatge Python.

2) Conjunt de nodes a expandir

A aquest conjunt li hem d'aplicar les operacions de selecció d'un node (això és, triar-ne un que serà el que expandirem) i afegir un conjunt de nodes a l'estructura (un cop un node s'ha expandit, els seus fills han de ser inclosos a l'estructura per a ser tractats més tard). Per a simplificar l'operació de selecció, s'implementarà l'estructura mitjançant una cua ordenada (una cua amb prioritats). D'aquesta manera, sempre seleccionarem el primer de la cua. En aquest cas, quan afegim nous nodes s'hauran de posar amb cura per tal que quedin en la posició més adequada segons un criteri determinat i fixat prèviament.

D'aquesta manera l'estructura de dades de l'arbre de cerca ha de ser:

```
Estructura de dades arbre de cerca:
    Nodes a expandir: cua amb prioritat de nodes
    Nodes ja expandits: llista de nodes
```

Un cop tenim l'estructura de dades corresponent a l'arbre de cerca, vegem l'algorisme per a generar l'arbre:

```
funcio cerca(problema, estrategia-de-cerca,
            arbre-de-cerca-inicial) retorna solucio es
    arbre-de-cerca := arbre-de-cerca-inicial;
    solucio-trobada := fals;
    mentre no (solucio-trobada) i
            hi-ha-nodes-per-expandir(arbre-de-cerca) fer
        node := seleccionar-node(arbre-de-cerca);
        afegir-node-a-ja-expandits(node, arbre-de-cerca);
        si solucio(problema, node)
            llavors solucio-trobada := cert;
            sino  nous-nodes := expandir(node, problema);
                  afegir-nodes-a-per-expandir (nous-nodes,
                                               estrategia-de-cerca,
                                               arbre-de-cerca);
        fsi;
    fmentre;
    si solucio-trobada
        llavors retorna solucio(arbre-de-cerca, node);
        sino retorna no-hi-ha-solucio;
    fsi;
ffuncio;
```

La funció rep com a paràmetres tres elements: el problema, l'estratègia de cerca i l'arbre de cerca inicial:

1) El problema que conté la informació corresponent a l'estat inicial, la funció objectiu i els operadors.

2) L'estratègia de cerca que representa tot allò que es refereix a la tria d'un nou node.

3) L'arbre inicial que està format per un únic node que correspon a l'estat inicial. De fet, com que hem suposat que la llista de nodes a expandir és una cua amb prioritat i sempre es tria el primer, l'estratègia és la funció que determina com ordenar els nodes de la llista i no com es fa la selecció.

A partir d'aquests paràmetres, la funció s'executa i ho farà sempre que no s'hagi trobat ja la solució (això ho marcarà la variable booleana `solucio-trobada`) i mentre quedin nodes a expandir. Com que aquests nodes estan emmagatzemats a l'arbre de cerca, la funció `hi-ha-nodes-per-expandir` s'aplica a la variable que conté l'arbre.

En general, el procediment a seguir és el següent:

1) Seleccionar el node (els possibles nodes són a l'arbre).

2) Passar-lo al conjunt de nodes ja expandits (aquests nodes també s'emmagatzemen a dins l'arbre de cerca).

3) Processar-lo segons si és una solució o no. Per a comprovar si és una solució, farem servir a més del node la funció objectiu (que s'emmagatzema en la variable problema). Si el node no és una solució haurem d'expandir-lo (aquí farem servir el problema per a saber quins operadors podem aplicar a l'estat associat al node) i afegir els nodes nous a l'arbre de cerca (la manera d'afegir-los dependrà de l'estratègia per tal que aquells que s'hagin de seleccionar primer quedin més ben situats a la cua amb prioritat).

En cas d'acabar i d'haver trobat una solució, el camí des de l'estat inicial a la solució es construirà a partir del node solució i de l'arbre de cerca (l'arbre conté tots els nodes ja expandits i, per tant, tots els nodes des del node inicial al final).

L'algorisme descrit aquí segueix, en línies generals, el que s'ha explicat informalment en el subapartat precedent, tot i que hi ha una diferència important. En l'explicació de l'exemple, hem aturat el seguiment quan expandint el node [1, 3, 2, 4] hem trobat que un dels seus fills ja correspon a un estat solució. La funció de cerca que presentem aquí no s'aturarà en aquest punt. Un cop expandit el node [1, 3, 2, 4] i obtinguts els tres fills [3, 1, 2, 4], [1, 2, 3, 4] i [1, 3, 4, 2] aquests seran afegits a l'arbre com a nodes a expandir.

A continuació, la funció procedirà amb la selecció d'un altre node. Fixeu-vos que la condició que activa la variable `solucio-trobada` és que el node seleccionat correspongui a un estat objectiu i no que hi correspongui un dels `nous-nodes`. Per tant, no serà fins que seleccionant un node, aquest sigui la solució que la cerca s'atura. Aquesta implementació que no segueix l'apuntada anteriorment es fa per tal que alguns dels algorismes de cerca que veurem més endavant siguin òptims. És a dir, no solament trobi una solució, sinó que en el cas que n'hi hagi més d'una aquesta sigui l'òptima.

2.1.1. Implementació amb Python

A continuació, fem una implementació amb Python de la funció corresponent a l'esquema general de cerca, juntament amb la de les funcions auxiliars i de les estructures de dades necessàries. Comencem amb la funció cerca.

```
def cerca (problema, estrategia, arbre):
    if (not candidats(arbre)):
        return ['no_hi_ha_solucio']
    else:
        node      = selecciona_node(arbre)
        nou_arbre = elimina_seleccio(arbre)
        if solucio(problema, node):
            return cami(arbre,node)
        else:
            return cerca(problema,
                         estrategia,
                         expandeix_arbre(problema,
                                         estrategia,
                                         nou_arbre,
                                         node))
```

La versió Python de l'esquema general de cerca utilitza els mateixos paràmetres ja descrits abans: el problema, l'estratègia i l'arbre. De tota manera, la forma de la funció és lleugerament diferent, ja que aquesta que es presenta aquí és recursiva. Així, primer es comprova la condició d'acabament: si no hi ha cap node que es pugui expandir haurem acabat (el cas de trobar la solució es tracta més endavant, aleshores la funció ja retorna el resultat i la recursió acaba).

La funció `candidats` és la que mira si hi ha nodes a expandir dins l'arbre de cerca i, si no n'hi ha, s'acaba i es retorna la llista amb un únic element `['no-hi-ha-solucio']`. Quan hi ha nodes a expandir se'n selecciona un i es defineix el nou arbre en què aquest node es treu de la llista de nodes a expandir. La selecció la fa la funció `selecciona_node` i el nou arbre sense el node que se selecciona retorna la funció `elimina_seleccio`. Si el node seleccionat és la solució (la funció `solucio` comprova això amb l'ajuda del problema i el node en qüestió), aleshores s'ha de retornar el camí fins al node solució. Això ho fa la crida `cami(arbre,node)`. Si no és així es fa una crida recursiva a la funció cerca amb el nou arbre resultat de l'expansió del node seleccionat. Tenim que `expandeix_arbre` és la funció que ens calcula l'expansió del node incorporant els nous nodes a l'arbre.

La funció `cerca` es cridarà des d'una funció que només té com a paràmetres el problema i l'estratègia i que, a partir del problema, crea l'arbre inicial. Aquest arbre estarà format per l'estat inicial definit en el problema.

```
def fer_cerca (problema, estrategia):
    return cerca(problema,
                 estrategia,
                 arbre_inicial(estat_inicial(problema),
                               info_inicial(problema)))
```

Abans de passar a les funcions relatives a l'arbre de cerca, considerem la definició de la funció `solucio`. Aquesta funció, donat un node i un problema, mira si el primer és una solució del segon. La funció té la definició següent:

```
def solucio (problema, node):
    ff = funcio_objectiu(problema)
    return ff(estat(node))
```

Aquí utilitzem el fet que la funció objectiu del problema és una funció. Per tant, només cal aplicar la funció a l'estat associat al node. Amb `estat(node)` obtenim l'estat corresponent al node i amb `funcio_objectiu(problema)` obtenim la funció que ens dirà si l'estat és una solució o no. Si el problema és el del trencaclosques lineal definit en el subapartat «Anàlisi pràctica del problema de les vuit reines», tindrem que la funció `ff` és `tl_funcio_objectiu`:

```
def tl_funcio_objectiu(estat):
    return estat == [1,2,3,4]
```

2.1.2. L'arbre de cerca: representació i funcions

Ara passem a la representació de l'arbre de cerca. Fem servir la mateixa representació descrita en l'apartat «La implementació», una cua amb prioritat pels nodes a expandir i una llista pels nodes ja expandits. Tant la cua com la llista es representaran mitjançant una llista Python. Així tenim el següent:

```
Estructura de dades arbre de cerca::=
            [llista_nodes_a_expandir, llista_nodes_ja_expandits]
```

Sobre la base d'aquesta estructura podem definir algunes de les funcions que han aparegut i que retornen algun dels elements de l'arbre. S'inclouen aquí altres funcions que ens faran falta més tard:

```
def nodes_a_expandir (arbre):
    return car(arbre)

def nodes_expandits (arbre):
    return cadr(arbre)

def selecciona_node (arbre):
    return car(nodes_a_expandir(arbre))

def candidats (arbre):
    return bool(nodes_a_expandir(arbre))
```

La definició d'aquestes funcions es basa en la selecció de l'element corresponent d'acord amb l'estructura. Així, les funcions `nodes_a_expandir` i `nodes_expandits` retornen les llistes de nodes que hi ha a l'arbre i `selecciona_node` retorna el primer node dels que hi ha a expandir. `candidats` comprova que la llista de nodes a expandir no sigui buida. Només cal subratllar que la selecció del node agafarà sempre el primer element de la llista corresponent als nodes a expandir perquè, com ja s'ha dit, aquesta llista correspon a una cua amb prioritats.

Una altra funció relacionada amb l'arbre –i que apareix en la funció `cerca` quan trobem la solució– és la que ens retorna el camí corresponent al node.

```
def cami (arbre, node):
    if not id_pare(node):
        return []
    lp = cami(arbre, node_arbre(id_pare(node), arbre))
    return lp + [operador(node)]
```

Aquesta funció reconstrueix el camí del node que se'ns passa fins a la solució. Atès que donat un node en podem conèixer el seu pare, tenim que tot el camí el podem escriure com el camí del node inicial fins al pare del node actual i l'operador que ens ha permès passar del pare al node actual. Així, `cami(arbre,node_arbre(id_pare(node),arbre))` correspon al camí fins al pare i `[operador(node)]` és una llista amb l'operador que, aplicat al pare, genera el `node`. La funció fa servir la funció `id_pare` que donat un node

retorna l'identificador del pare i una altra anomenada `node_arbre` que donat un identificador –i l'arbre– ens troba el node corresponent. Com que aquesta darrera funció forma part de l'estructura arbre la veurem tot seguit:

```
def node_arbre (id_node, arbre):
    check_node = lambda node: ident(node) == id_node
    a_expandir = member_if(check_node, nodes_a_expandir(arbre))
    if bool(a_expandir):
        return car(a_expandir)
     return find_if(check_node, nodes_expandits(arbre))
```

Aquesta funció, per a trobar el node, primer mira si el node està entre els nodes a expandir i, si és així (si la variable `a_expandir` no és nul·la), fa la selecció del node entre els que s'han d'expandir. Si no és així, la selecció es farà entre els nodes ja expandits. Si el node tampoc no està a la segona llista, la segona selecció retornarà `[]`. Les funcions que fem servir aquí de `member_if` i `find_if` comproven i seleccionen si hi ha un element que satisfà una propietat (definides a l'apèndix).

Ara passem a la funció `expandeix_arbre` que expandeix l'arbre. Es pot veure que primer es calculen els nodes que s'han d'expandir i que després seran incorporats a l'arbre amb la funció `construeix_arbre`. La funció que construeix l'arbre necessita, a més de l'arbre, el node que hem expandit i els nous nodes generats. A més, per tal que a l'hora d'encuar els nous nodes a la cua de nodes a expandir això es faci correctament, necessitem l'estratègia. Com que considerem que l'estratègia és una funció, la cridem indicant-li com a paràmetres la cua de nodes a expandir ja existents i els nous nodes.

En quant a l'expansió del node, aquesta funció fa servir, a més dels nodes, els operadors del problema i –per a alguns tipus de cerca– certa informació addicional associada al problema.

```
def expandeix_arbre (problema, estrategia, arbre, node):
    nous_nodes_a_expandir = expandeix_node(node,
                                            operadors(problema),
                                            funcio_info_addicional(problema))
    return construeix_arbre(arbre,
                            estrategia,
                            node,
                            nous_nodes_a_expandir)

def construeix_arbre (arbre, estrategia, node_expandit, nous_nodes_a_expandir):
    elm = estrategia(car(arbre), nous_nodes_a_expandir)
    return cons(elm, [cons(node_expandit, cadr(arbre))])
```

Per a completar les funcions que actuen en l'arbre ens manca la que elimina el node seleccionat anomenada `elimina_seleccio`, i la que ens genera l'arbre inicial del problema.

```
def elimina_seleccio (arbre):
    return cons(cdr(nodes_a_expandir(arbre)), cdr(arbre))

def arbre_inicial (estat, info):
    infres = info(estat)
    node = construeix_node(gensym(), estat, [], [], [])
    tmp = [node + infres]
    return [tmp]
```

La funció `elimina_seleccio` construeix un nou arbre (una llista) a partir dels nodes a expandir (després d'eliminar el primer aplicant `cdr` a la llista resultant de cridar `nodes_a_expandir`) i dels nodes ja expandits (la cua de l'arbre). La funció `arbre_inicial` construeix un nou arbre on només hi ha un node a expandir que té com a estat el que se li indica a la funció. Aquesta funció crida la funció que calcula la informació addicional per a l'estat inicial.

2.1.3. Els nodes: representació i funcions

La representació dels nodes segueix el que s'ha proposat anteriorment: un identificador, l'estat, l'identificador del node pare i l'operador que l'ha generat. A més, tot i que ara per ara no fa falta, considerarem que és possible que el node contingui més camps corresponents a informació addicional (per exemple, per a guardar-hi el cost d'arribar al node). Per a mantenir tota aquesta informació farem servir una llista en què els elements que hi apareguin correspondran als especificats anteriorment i amb el mateix ordre. Així, tindrem que l'estructura és com segueix:

```
node ::=
    [Identificador, Estat, Identificador-Node-Pare,
    Operador-generador, Altres...]
```

Sobre la base d'això definim un conjunt de funcions consultores que donat un node ens retornin els elements d'aquests nodes que podem necessitar en la cerca. Aquestes funcions són:

```
def ident (node): return car(node)
```

```
def estat (node): return cadr(node)

def id_pare (node): return caddr(node)

def operador (node): return car(cdddr(node))

def info (node): return cdr(cdddr(node))
```

La funció `info` (d'informació) ens retornarà simplement la cua final de la llista. Així, si hi ha més d'un element, la funció ens retornarà tots els que hi hagi. Les altres funcions ens retornen cadascuna un únic element, malgrat que aquest pugui ser una llista. Aquest segon cas serà el que tindrem si estem cercant solucions per al problema del trencaclosques lineal, en què un estat és una llista.

A més de les funcions per a fer consultes d'un node, necessitem una funció que ens permeti construir un node. Això ens fa falta en el moment de l'expansió (crida a la funció `expandeix_node`) i de crear l'arbre inicial (crida a la funció `arbre_inicial`). A aquesta funció li indiquem quatre elements i una llista amb la informació addicional. Com abans, aquesta darrera informació és una llista per a no limitar el nombre d'elements.

```
def construeix_node (ident, estat, id_pare, op, info):
    return [ident, estat, id_pare, op] + info
```

Així, per exemple, si volem generar el node inicial corresponent a l'estat [2, 4, 1, 3] farem la crida:

```
construeix_node(gensym(), [2, 4, 1, 3], [], [], [])
```

Si volem construir el mateix node però de manera que hi afegim informació dient que el cost d'aquest node és zero i que el seu estat correspon a l'estat inicial, podem fer:

```
construeix_node(gensym(), [2, 4, 1, 3], [], [], [0, estat_inicial])
```

Com hem vist, a més d'aquestes funcions cal una funció que faci l'expansió d'un node. Aquesta tasca la fa la funció `expandeix_node`, una funció que crida a la funció `expandeix_arbre` que hem vist més amunt.

```
def expandeix_node (node, operadors, funcio):
    def elimina_estats_buits (llista_nodes):
        return remove_if(lambda node: estat(node) == 'buit',
                         llista_nodes)

    st        = estat(node)
    id_node   = ident(node)
    info_node = info(node)
    aux       = []
    for op in operadors:
        nou_simbol = gensym()
        ff         = cadr(op)
        ffapp      = ff(st,info_node)
        aux.append(construeix_node(nou_simbol,
                                   ffapp,
                                   id_node,
                                   car(op),
                                   funcio([st, info_node],
                                          ffapp,
                                          car(op))))
    return elimina_estats_buits(aux)
```

Aquesta funció construirà un node per a cada operador (aplicant cada operador al node que estem expandint) i, a continuació, eliminarà aquells que són buits (o erronis). L'eliminació la fa la funció `elimina_estats_buits`, que es defineix localment. Aquesta funció és per a resoldre el cas dels moviments que no es poden fer en un determinat estat (per exemple, eliminar una peça quan aquesta no existeix).

La constant `'buit'` defineix què s'entén per un estat que no s'ha pogut generar. La funció que aplica cada operador a l'estat és una funció `lambda` que per a cada operador genera un node amb la funció `construeix_node`. Cada nou node té un identificador generat amb `gensym()`, el nou estat que s'obté amb `ff(st, info_node)` on `ff` és la funció a `cadr(op)`, l'identificador del node que és el seu pare (l'identificador del node en curs `id_node`), el nom de l'operador `car(op)` i la nova informació associada al node.

2.1.4. El problema: representació i funcions

A l'hora d'aplicar l'esquema de cerca a un exemple concret, s'ha considerat que el problema té l'estructura següent:

```
problema ::=
    [operadors, funcio, estat-inicial,
        funcio-objectiu, info-inicial...]
```

i d'acord amb això s'han definit les funcions `operadors`, `funció_info_addicional`, `estat_inicial`, i `info_inicial` tal com segueix:

```
def operadors (problema): return car(problema)

def funcio_info_addicional (problema): return cadr(problema)

def estat_inicial (problema): return caddr(problema)

def funcio_objectiu (problema): return car(cdddr(problema))

def info_inicial (problema): return car(cdr(cdddr(problema)))
```

2.2. Algunes consideracions addicionals

L'esquema de cerca presentat aquí no detalla quina és l'estratègia que hem de fer servir per a seleccionar el node que hem d'expandir. En els apartats que segueixen es presenten diferents alternatives per a fer-ho. Per a poder-les avaluar, utilitzarem diverses propietats. Per exemple, podem considerar la completesa, l'optimalitat, la complexitat pel que fa al temps i la complexitat pel que fa a l'espai. Les dues primeres són per a saber si quan hi ha una solució, la trobem, i si quan n'hi ha més d'una, trobem la solució de més qualitat. En aquest cas, cal que el problema defineixi la manera d'avaluar les solucions (per exemple, mitjançant les funcions de cost). Les dues darreres corresponen al temps i la memòria que necessiten els algorismes per a trobar la solució. Aquests costos en temps i memòria s'expressen en funció del factor de ramificació, la longitud del camí solució, etc.

A més d'aquestes propietats, n'hi ha una altra que interessa quan la cerca s'aplica en un entorn de temps real. Són els **algorismes tot-temps**[2]. Aquests mètodes de cerca retornen una solució per a qualsevol assignació de temps de

(2) En anglès, *anytime algorithms*.

computació i s'espera que retornin una solució més bona com més temps se'ls deixa. Per a alguns d'aquests algorismes se sap que, amb prou temps, donarien la solució òptima.

Les propietats esmentades es defineixen a continuació:

1) Completesa: un algorisme de cerca és complet si, quan un problema té solució, l'algorisme la troba.

2) Optimalitat: un algorisme de cerca és òptim, quan en un problema amb diferents solucions, sempre troba la solució de més «qualitat».

3) Complexitat pel que fa al temps: el temps que triga l'algorisme a trobar la solució.

4) Complexitat pel que fa a l'espai: la memòria que necessita l'algorisme per a poder trobar la solució.

5) Algorismes tot-temps: un algorisme de cerca és tot-temps quan pot trobar una solució per a qualsevol assignació de temps de computació.

La qüestió dels estats repetits

Ja heu vist que l'arbre de cerca i l'espai d'estats tenen força semblances però que no són iguals perquè el graf corresponent a l'espai d'estats té tants nodes com estats, però l'arbre de cerca no, perquè hi pot haver estats que apareixen més d'un cop. El fet de tenir estats repetits provoca que els algorismes de cerca siguin ineficients perquè expandeixen alguns subarbres més d'un cop i, a més, pot provocar que l'algorisme no acabi perquè un estat s'expandeix de manera repetida. Per a evitar això, es pot comprovar si un estat ja ha aparegut abans, però això introdueix un sobrecost addicional al mètode. Per tant, hi ha d'haver un compromís entre el sobrecost de comprovar si un estat és repetit i que l'algorisme acabi. En general, es poden considerar tres alternatives per a evitar la repetició d'estats. Les presentem ordenades de menys a més cost de comprovació:

1) No permetre el retorn a l'estat d'on venim: cal que la funció que expandeix un node comprovi que l'estat corresponent al node generat no coincideix amb el pare del node que estem expandint. Aquesta comprovació es pot fer en temps constant.

2) No permetre generar camins amb cicles: cal que la funció que expandeix un node comprovi que l'estat corresponent al node generat no coincideix amb cap dels antecedents del node que estem expandint. Aquesta comprovació es farà en temps lineal (en relació amb la longitud del camí).

3) No generar cap estat que ja hagi estat generat: això obliga a guardar tots els estats de l'arbre de cerca. Per tant, la complexitat serà de l'ordre de O(s) on s és el nombre d'estats en l'espai d'estats.

És important subratllar que no és indispensable emmagatzemar els nodes ja expandits per a representar l'arbre de cerca, ni tampoc per a poder reconstruir el camí quan tenim la solució. Per tant, emmagatzemar aquests nodes representa un cost addicional en memòria.

En general, l'ús d'una tècnica o una altra dependrà del problema. De fet, com més estats repetits apareguin, més útil serà emmagatzemar-los i utilitzar el temps per a comprovar que un estat no hagi aparegut abans. A més, s'ha de tenir en compte que segons quina sigui la implementació triada per als estats i el nombre d'estats diferents, es poden tenir mètodes més o menys costosos per a comprovar si un estat ja ha estat visitat o no.

3. Estratègies de cerca no informada

L'esquema general de cerca plantejat anteriorment no concreta quin node hem d'expandir o, el que és equivalent, com hem d'ordenar els nodes en la llista de nodes a expandir. A continuació, veurem alguns dels mètodes que es poden utilitzar per a fer aquesta expansió. Primer, en aquest apartat, veurem mètodes que com a única informació per a decidir quin node cal expandir i només fan servir els operadors que es poden aplicar a un determinat estat. Així, veurem la cerca en amplada i profunditat. En el proper apartat, en canvi, es veuran mètodes que fan servir informació addicional relativa al problema.

Ara veurem dos algorismes de cerca: cerca en amplada i cerca en profunditat.

3.1. Cerca en amplada

La cerca en amplada correspon a fer un recorregut del graf d'estats per nivells. Això és, primer es visiten tots aquells estats a què es pot accedir des de l'estat inicial aplicant-hi només un operador, després es visiten tots aquells estats a què s'hi accedeix aplicant dos operadors a l'estat inicial, seguint amb aquells que necessiten l'aplicació de tres operadors per a accedir-hi i, així, successivament.

Formalment tenim que l'algorisme espera a expandir un node d'un nivell determinat fins al moment en què tots els nodes del nivell anterior ja s'han expandit. Això es pot implementar com un refinament de l'esquema general de cerca en què l'estratègia és emmagatzemar el nou conjunt de nodes (els expandits a partir del node seleccionat) després de tots els nodes que encara queden pendents. Noteu que, en un moment donat, els nodes encara pendents només poden ser de dos nivells consecutius. O són del nivell d o del nivell $d + 1$. Si n'hi ha dels dos nivells els del $d + 1$ quedaran darrere dels del nivell d. D'acord amb això, tenim que, en general, l'algorisme accedeix a un node de nivell d i els nodes resultat de l'expansió (nodes del nivell $d + 1$) quedaran al final de la llista.

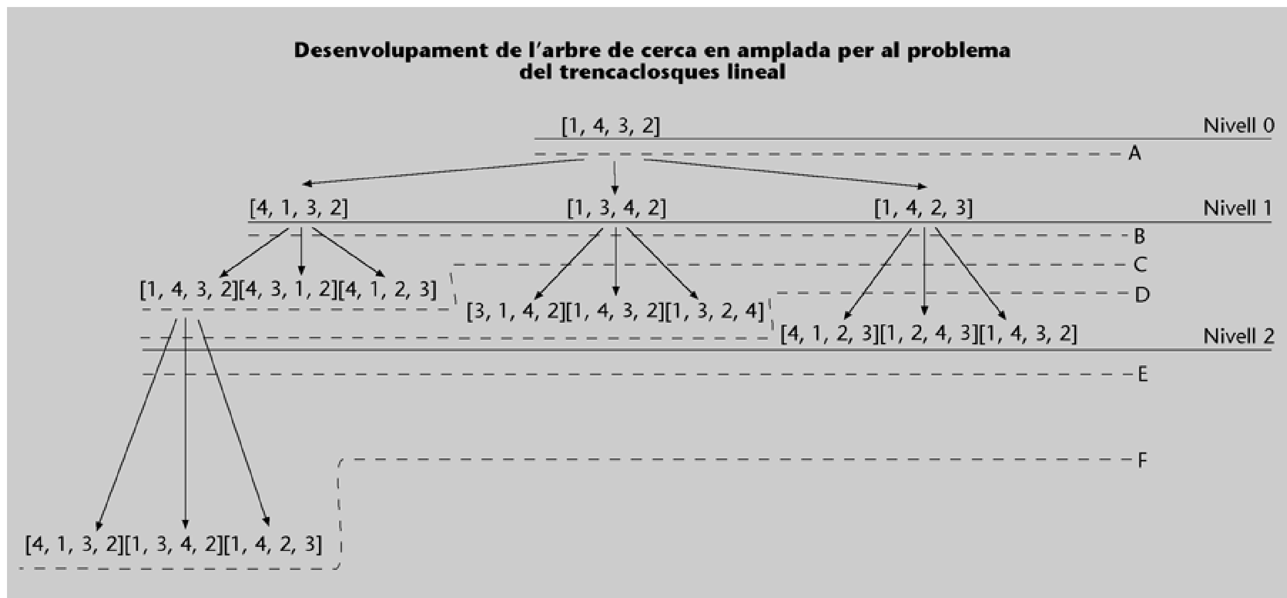
En la figura 9 apareix el desenvolupament de l'arbre de cerca partint de l'estat inicial [1, 4, 3, 2]. Evidentment tenim que ara la frontera és A. La inicialització de l'arbre ens dona un únic node que correspon a l'estat [1, 4, 3, 2]. L'expansió d'aquest estat ens dona tres nous estats que són [4, 1, 3, 2], [1, 3, 4, 2], [1, 4, 2, 3].

Ara la frontera de l'arbre de cerca és B. A continuació, expandirem el primer d'aquests nodes i obtindrem la frontera C. Seguidament, quan seleccionem un nou node a expandir, hem d'agafar un dels que hi ha en el nivell més

endarrerit. Prenem el node amb l'estat [1, 3, 4, 2], que quan l'expandim dona la frontera D. El node següent que seleccionem és el [1, 4, 2, 3], que és l'únic que està en un nivell endarrere. Quan l'expandim, obtenim la frontera E.

Per a implementar aquest mecanisme amb l'esquema general de cerca, hem d'afegir els nodes resultants de l'expansió al final de la llista de nodes a expandir. Per tant, la funció estratègia és concatenar la nova llista pel final.

Figura 9



Com que en la cerca en amplada els nodes es van expandint per nivells, es fa un recorregut sistemàtic de l'arbre. Això fa que, quan hi ha una solució, l'algorisme la trobarà quan arribi al nivell corresponent. Per això, aquest esquema de cerca és complet. L'optimalitat, en canvi, només ocorre si el cost del camí és una funció no decreixent de la profunditat del node. Això és, que si hem trobat una solució al nivell d, no n'hi hagi cap de més bona a profunditats més grans que d.

La complexitat pel que fa al temps de l'algorisme és exponencial, així, si tenim que el factor de ramificació (el nombre d'operadors) és b i que la solució està a la profunditat d, tindrem que per a trobar la solució haurem d'haver expandit tots els nodes fins al nivell d. Per tant, com que $1 + b + b^2 + b^3 + ... + b^d = O(b^d)$ la complexitat serà ordre de b^d.

Per a determinar la complexitat en quant a la memòria, hem de tenir en compte el nombre de nodes que tindrem en la llista de nodes a expandir. En aquest mètode, el nombre de nodes dependrà del nivell. De fet, en el nivell i-èssim tindrem b^i nodes (podeu veure que en la figura anterior tenim 3^0 nodes en la

frontera del nivell 0 – la frontera A, 3^1 en la del nivell 1 – la frontera B, 3^2 en la del nivell 2 – la frontera E, ...). Com que la solució està al nivell *d*, quan trobem la solució hi haurà a la memòria de l'ordre de $O(b^d)$ nodes.

3.1.1. Implementació

Quan utilitzem l'estratègia de cerca en amplada hem d'expandir tots els nodes d'un determinat nivell abans de passar als nodes del nivell següent. Per a implementar això, n'hi ha prou amb què la funció que implementa l'estratègia col·loqui els *n* nodes resultat de l'expansió darrere dels que tenim a la llista de nodes a expandir.

Això es pot implementar amb Python mitjançant una concatenació de la llista corresponent als nodes a expandir i la llista amb els nous nodes:

```
def tl_estrategia_amplada (nodes_a_expandir, nous_nodes_a_expandir):
    return nodes_a_expandir + nous_nodes_a_expandir
```

Així, la cerca en amplada la podem definir com:

```
def cerca_amplada (problema):
    return fer_cerca(problema, tl_estrategia_amplada)
```

3.2. Cerca en profunditat

La cerca en profunditat sempre expandeix els nodes que estan en un nivell més profund dins l'arbre de cerca. Només quan s'arriba a un node a què no se li pot aplicar cap operador es triarà un node d'un altre camí.

Per a implementar aquest mètode amb l'esquema general de cerca, afegirem els nodes resultat de l'expansió davant dels nodes a expandir. D'aquesta manera, els nodes situats a més profunditat estaran sempre davant i se seleccionaran primer. Així, la implementació amb Python de la funció d'estratègia corresponent serà:

```
def tl_estrategia_profunditat (nodes_a_expandir, nous_nodes_a_expandir):
    return nous_nodes_a_expandir + nodes_a_expandir
```

Amb la funció d'estratègia podem definir la funció de cerca en profunditat i, a continuació, fer la crida amb el problema del trencaclosques lineal que hem definit en el subapartat «Anàlisi pràctica del problema de les vuit reines»:

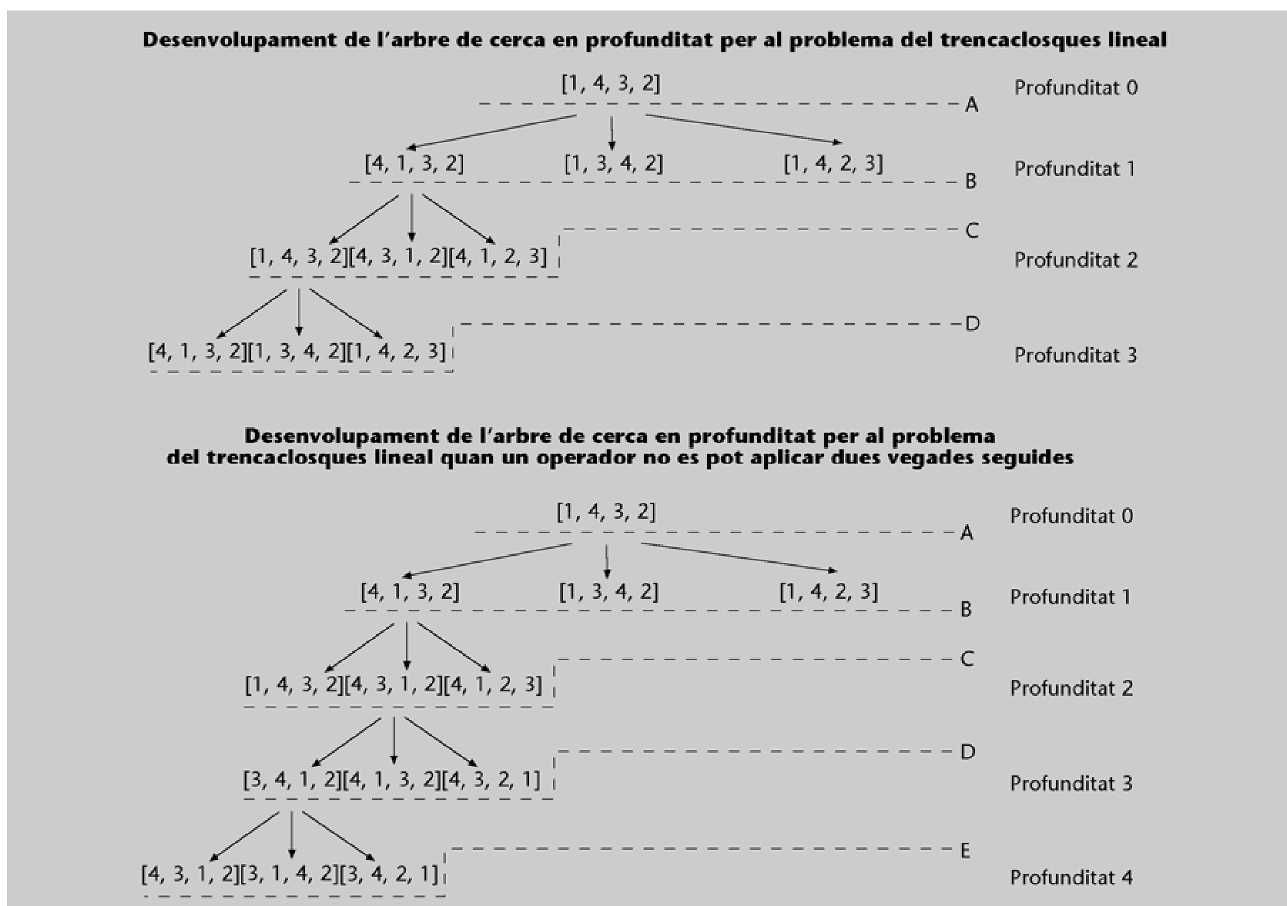
```
def cerca_profunditat (problema):
    return fer_cerca(problema, tl_estrategia_profunditat)
```

Per a fer la cerca en profunditat del problema farem:

```
cerca_profunditat(problema())
```

Aquesta crida, però, no retorna cap resultat perquè queda penjada la funció concatenant moviments IE.

Figura 10



En la figura 10, es representa el desenvolupament de l'arbre de cerca en profunditat corresponent al problema del trencaclosques lineal definit a la variable `problema`. Es pot veure que la cerca va aprofundint en un mateix camí. A més, també es pot veure que el mecanisme de cerca no és complet perquè

en determinades condicions no acaba. Aquest és el cas de la figura, on el sistema comença a repetir els estats [1, 4, 3, 2], [4, 1, 3, 2]. En el segon esquema de la mateixa figura, es representa el desenvolupament de l'arbre de cerca en profunditat corresponent al mateix problema quan els operadors es restringeixen de manera que no es puguin aplicar dues vegades seguides (o el que és equivalent, no es pugui repetir l'estat del pare). Si seguim l'expansió de l'arbre veurem que també s'entra en un cicle.

Malgrat el problema que la cerca no sigui completa, la cerca en profunditat té un gran avantatge en relació amb la cerca en amplada: té una complexitat menor pel que fa a memòria. Es pot veure en els dos arbres de la figura 10 que la frontera dels arbres de cerca successius no són exponencials com en el cas de la cerca en amplada, sinó que són lineals en relació amb la profunditat. Així, en general, si tenim que un problema té un factor de ramificació b i disposem d'operadors fins a una profunditat m (suposem que es pot trobar la solució a una profunditat menor que m), tindrem que la complexitat pel que fa a l'espai és de $O(bm)$. Vegeu que en el primer arbre de la figura 10 tenim una profunditat igual a tres: $(3 - 1) + (3 - 1) + 3$ nodes a expandir i que, en la figura que representa el desenvolupament de l'arbre de cerca en profunditat, quan un operador no es pot aplicar dues vegades seguides, tenim una profunditat igual a quatre: $(3 - 1) + (3 - 2) + (3 - 1) + 3$ nodes.

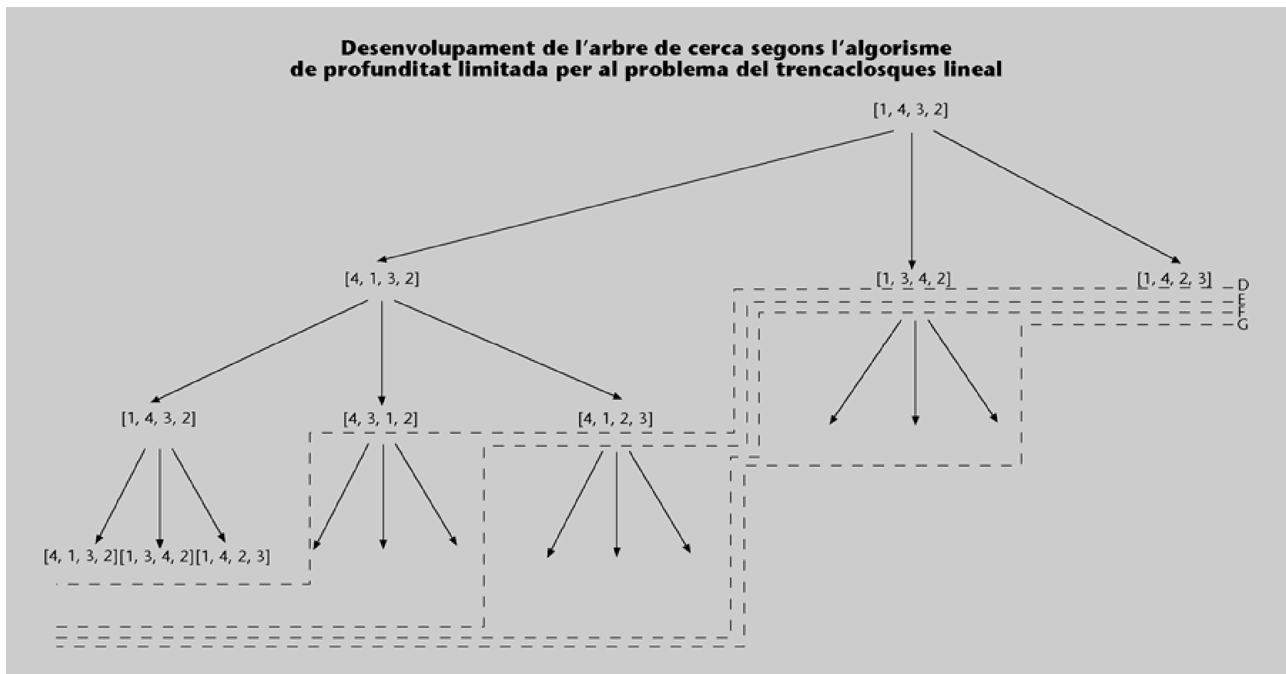
La complexitat pel que fa al temps, en canvi, serà semblant al cas de la cerca en amplada, $O(b^m)$, tot i que es pot trobar la solució amb menys temps. Això serà així perquè l'arbre s'anirà obrint a mesura que arribem a nodes que no es poden expandir. Això, però, no afectarà la memòria perquè dels nodes ja expandits (i encara menys si ja sabem que aquests nodes no porten a la solució) no cal emmagatzemar-ne l'estructura.

Com que el cost de memòria d'aquest esquema de cerca és substancialment més bo que el de la cerca en amplada, s'han desenvolupat mètodes per a aprofitar aquest cost, però sense caure en la no completesa del mètode. A continuació veurem dues aproximacions.

3.2.1. Cerca en profunditat limitada

En aquest cas s'aplica l'algorisme de cerca en profunditat, però fixant la profunditat màxima a què es pot arribar. Així, l'algorisme funciona de la mateixa manera que el mètode descrit més amunt, però ara quan s'arriba a la profunditat prefixada, en lloc d'aplicar l'operador, es considera que aquest no es pot aplicar. Això força a recular i a agafar un node d'un nivell menys profund.

Figura 11



En la figura 11, es presenta el desenvolupament de l'arbre de cerca vist anteriorment per a un cas del problema del trencaclosques lineal amb una profunditat màxima igual a 3. Es donen les fronteres de l'arbre de cerca que seguiran les que ja hem donat en la figura 10. Es pot veure que en tots els camins la cerca s'atura a un mateix nivell.

El fet d'introduir la profunditat màxima permet que en determinades situacions el mètode sigui complet (malgrat que no sempre ho serà). Així, tindrem que la cerca amb profunditat màxima amb una profunditat *p* és completa quan hi ha una solució a menys profunditat. La complexitat per a aquest mètode és anàloga al cas anterior, prenent ara *p* en lloc d'*m* (*m* era la profunditat màxima que permetien els operadors). Per tant, la complexitat pel que fa al temps serà $O(b^p)$ i la complexitat pel que fa a l'espai serà $O(bp)$.

A continuació, s'indiquen les funcions que implementen aquest mètode de cerca. El control de la profunditat el fem associant a cada node la seva profunditat i impedint que el node s'expandeixi quan ja estigui a la profunditat màxima (si la profunditat és superior o igual al `limit` –definim això com una variable global– el moviment no és permès i la funció retorna aleshores un moviment buit). Aquesta associació s'implementa mitjançant la informació addicional del node que ja havíem comentat en el subapartat «Anàlisi pràctica del problema de les vuit reines».

Moviment buit

La funció d'expandir els nodes `expandeix_node` ja té en compte quan un node no es pot expandir, com ja hem vist en el subapartat «La implementació».

Així tindrem que els operadors seran:

```
limit = 0 ## variable global, valor inicial arbitrari
```

```
def mov_ie_profLim (estat, info):
    if car(info) < limit: return [cadr(estat), car(estat), caddr(estat), cadddr(estat)]
    return 'buit'

def mov_ic_profLim (estat, info):
    if car(info) < limit: return [car(estat), caddr(estat), cadr(estat), cadddr(estat)]
    return 'buit'

def mov_id_profLim (estat, info):
    if car(info) < limit: return [car(estat), cadr(estat), cadddr(estat), caddr(estat)]
    return 'buit'

def tl_operadors_profLim():
    return [['ie', mov_ie_profLim], ['ic', mov_ic_profLim], ['id', mov_id_profLim]]
```

Aquí `'buit'` és una constant que defineix l'estat que no s'ha pogut generar (o estat erroni).

Amb aquests operadors es pot definir el problema d'una manera similar a com s'ha fet abans (subapartat «Anàlisi pràctica del problema de les vuit reines»). S'ha de tenir en compte que ara hem d'inicialitzar el node corresponent a l'estat inicial de manera que tingui associat el valor de la seva profunditat (això és, zero). Aquesta funció serà: `lambda estat: [estat, [0]]`. També hem de definir una funció que calculi la profunditat del nou node a partir del node que expandim. Això ho defineix la funció:

```
def auxf(info_node_pare, estat, nom_operador):
    info_node = cadr(info_node_pare)
    return [estat, [1 + caadr(info_node)]]
```

La resta de la definició és com abans, tenint en compte que ara els operadors són el resultat de cridar `tl_operadors_profLim`.

```
def problema_profLim():
    def auxf(info_node_pare, estat, nom_operador):
        info_node = cadr(info_node_pare)
        return [estat, [1 + caadr(info_node)]]

    return [tl_operadors_profLim(),
            auxf,
            [4,3,2,1],
            (lambda estat: estat == [1,2,3,4]),
            (lambda estat: [estat, [0]])]
```

Amb aquestes funcions definim la cerca en profunditat limitada a partir d'un problema i un límit de la manera següent:

```
def cerca_profunditat_limitada (problema, lim):
    global limit
    limit = lim
    return fer_cerca(problema, tl_estrategia_profunditat)
```

Per a cridar la funció amb el problema que hem definit i una profunditat màxima de deu hem de fer:

```
cerca_profunditat_limitada(problema_profLim(), 10)
```

Aquesta crida donarà el resultat següent:

```
['ie', 'ie', 'ie', 'ie', 'ie', 'ic', 'ie', 'id', 'ic', 'ie']
```

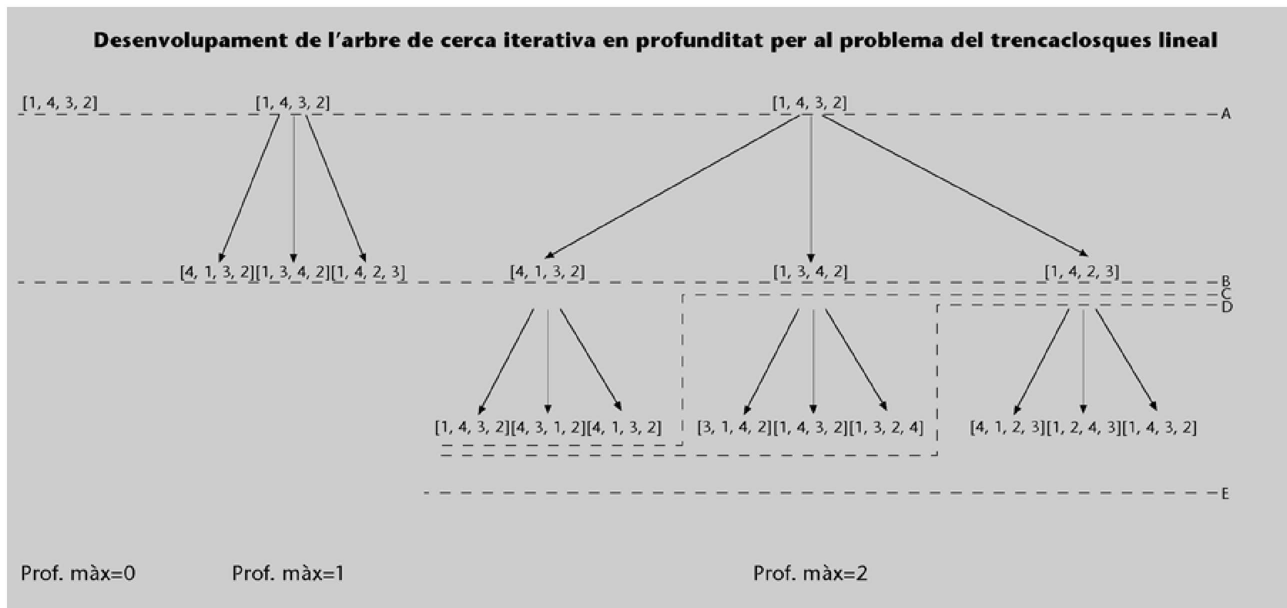
Els primers cinc moviments IE es deuen al fet que, en realitat, fem una cerca en profunditat i, com que està limitada, tira enrere i troba una solució fent servir altres operadors. Recordeu que la cerca en profunditat per a aquest mateix problema quedava penjada aplicant sempre l'operador IE.

Atès que, en general, no es coneix a quin nivell hi ha la solució, és difícil i problemàtic fixar *a priori* el nivell màxim per l'algorisme. Una alternativa a aquest problema és l'anomenada *cerca iterativa amb profunditat*.

Cerca iterativa amb profunditat

L'estratègia consisteix a fer diverses cerques amb una profunditat limitada, utilitzant cada vegada valors creixents de la profunditat màxima.

Figura 12



Desenvolupament de l'arbre de cerca iterativa en profunditat per al problema del trencaclosques lineal

En la figura 12, es presenten els diferents arbres de cerca (i el seu desenvolupament) per al problema del trencaclosques lineal. En cadascun dels arbres tenim una cerca en profunditat limitada amb una profunditat màxima diferent.

Evidentment, aquest procediment és complet, perquè arribarà un moment en què la profunditat màxima permesa igualarà la profunditat en què hi ha la solució.

Pel que fa als costos, cal diferenciar entre el temps i la memòria. Com que per a cada profunditat màxima la cerca comença de nou, la memòria necessària correspon a la que es necessita per a la cerca amb profunditat limitada quan la profunditat correspon a la profunditat de la solució. Així, tenim que el cost de memòria serà $O(b\ d)$. Per a determinar la complexitat pel que fa al temps s'ha de tenir en compte que ara hi ha nodes que s'expandeixen més d'un cop. Si suposem que la solució és a una profunditat d tindrem que haurem d'expandir tots els arbres fins a una profunditat màxima igual a d. Per tant, el nombre de nodes que expandirem serà:

$$1 + b + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

Aquesta expressió es pot reescriure com:

$$(d + 1) \cdot 1 + d \cdot b + (d - 1) \cdot b^2 + \ldots + 3 \cdot b^{d-2} + 2 \cdot b^{d-1} + b^d$$

Per tant, el cost serà: $O(b^d)$.

El quocient de les dues expressions ens donarà el sobrecost de repetir la cerca. Aquest quocient és aproximadament $1 + 1/b$, la qual cosa mostra que el sobrecost és petit ($100/b$ % aproximadament) i que, com més gran sigui b, més petit serà. Per exemple si prenem $b = 10$ i una profunditat màxima de 5, tindrem que la cerca iterativa en profunditat visitarà 123.456 nodes, mentre que si visitem els nodes només una vegada (el cas de la cerca en amplada) en visitarem 111.111. Això representa un sobrecost de l'11 %. Amb $b = 20$ i la mateixa profunditat visitarem 3.545.706 amb la cerca iterativa i 3.368.421 amb la cerca en profunditat. Per tant, tindrem que el quocient és 1,052 i el sobrecost serà del 5,2 %.

Sobrecost

Aquests càlculs són, evidentment, suposant que la cerca amb profunditat màxima la fem amb una profunditat igual a b. Això només serà possible si coneixem b; sinó faríem una cerca a major profunditat per a assegurar-nos de trobar-la i, per tant, l'estalvi (si n'hi ha) seria molt menor.

Per a implementar la cerca iterativa en profunditat utilitzarem una funció que fa la cerca iterativa des d'una determinada profunditat (és la funció `cerca_iterativa_profunditat_desde_k`). Aleshores, la cerca iterativa començarà des de la profunditat zero i si en aquesta profunditat no hi ha la solució començarà la cerca des de la profunditat següent. Una implementació d'aquestes dues funcions és la següent:

```
def cerca_iterativa_profunditat (problema):
    return cerca_iterativa_profunditat_desde_k(problema,0)

def cerca_iterativa_profunditat_desde_k (problema, lim):
    resultat = cerca_profunditat_limitada(problema, lim)
    if resultat == ['no_hi_ha_solucio']:
        return cerca_iterativa_profunditat_desde_k(problema, lim+1)
    return resultat
```

La crida a la funció serà utilitzant el problema on l'expansió té en compte la profunditat màxima:

```
cerca_iterativa_profunditat(problema_profLim())
```

Aquesta crida dona la solució següent:

```
['ie', 'ic', 'ie', 'id', 'ic', 'ie']
```

Podeu veure que no hi pot haver cap altra solució amb un camí de longitud menor. Si existís, s'hauria trobat abans atès que en els passos previs la longitud màxima estava limitada a 5, 4, 3, 2, 1 i 0. Per tant, qualsevol solució d'aquestes longituds s'hauria trobat.

3.3. Exemple pràctic de cerca de solució

Reprenent el problema de les vuit reines del apartat «Introducció a la resolució de problemes i cerca», mirem l'evolució de l'algorisme de cerca en profunditat per a la cerca d'una solució a partir d'aquest estat inicial:

					R		
	R						
						R	
R							
		R					

Per a simplificar, agruparem les variables x_1, x_2, x_3, x_4, x_5, x_6, x_7 i x_8 en un *array* o llista $x = [\, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 \,]$. Aleshores, l'evolució de l'algorisme de cerca en profunditat seria la següent:

```
Pas 0:

Estat inicial: x=[5,1,6,0,2,NULL,NULL,NULL]

Nodes a expandir (solament l'estat inicial):

[ [5,1,6,0,2,NULL,NULL,NULL] ]

Pas 1:

Seleccionem primer node de la llista a expandir.

Comprovem si aquest node és la solució. No ho és. L'expandim.

Nodes a expandir:

[ [5,1,6,0,2,4,NULL,NULL], [5,1,6,0,2,7,NULL,NULL],
[5,1,6,0,2,NULL,7,NULL], [5,1,6,0,2,NULL,NULL,3] ]

Pas 2:

Seleccionem primer node de la llista a expandir.

Comprovem si aquest node és la solució. No ho és. L'expandim.

Afegirem els nous nodes per davant dels anteriors.

Nodes a expandir:

[ [5,1,6,0,2,4,7,NULL], [5,1,6,0,2,4,NULL,3],
[5,1,6,0,2,7,NULL,NULL], [5,1,6,0,2,NULL,7,NULL],
[5,1,6,0,2,NULL,NULL,3] ]

Pas 3:

Seleccionem primer node de la llista a expandir.

Comprovem si aquest node és la solució. No ho és. L'expandim.

Afegirem els nous nodes per davant dels anteriors.

Nodes a expandir:

[ [5,1,6,0,2,4,7,3], [5,1,6,0,2,4,NULL,3],
[5,1,6,0,2,7,NULL,NULL], [5,1,6,0,2,NULL,7,NULL],
[5,1,6,0,2,NULL,NULL,3] ]

Pas 4:

Seleccionem primer node de la llista a expandir.

Comprovem si aquest node és la solució.

És la solució. Problema resolt.

Solució: [5,1,6,0,2,4,7,3]
```

És important destacar que, com que es pot comprovar, a l'hora d'expandir només considerem els nodes resultants vàlids (satisfan les restriccions del problema). Així doncs, l'algorisme ha trobat la solució següent:

<table>
<tr><td></td><td></td><td></td><td></td><td></td><td>R</td><td></td><td></td></tr>
<tr><td></td><td>R</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>R</td><td></td></tr>
<tr><td>R</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>R</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>R</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>R</td></tr>
<tr><td></td><td></td><td></td><td>R</td><td></td><td></td><td></td><td></td></tr>
</table>

4. Cost i funció heurística

En els esquemes de cerca vistos fins ara, la informació utilitzada a l'hora de decidir quin node és aquell que cal expandir es limita a saber quins operadors podem aplicar a un determinat node. Ara presentem uns nous esquemes que utilitzen informació addicional. En primer lloc, considerem l'ús del cost associat a un operador i, a continuació, l'ús de les anomenades *funcions heurístiques*.

El cost d'un operador ja ha estat comentat en el subapartat «Algunes consideracions addicionals: la importància d'una representació adequada i la qüestió del cost» i correspon a una avaluació dels operadors en termes de preu, dificultat, temps, etc. Una solució, o dit de manera més precisa, el camí a una solució es pot avaluar sobre la base dels costos dels generadors que apliquem. Així, el cost d'un camí es considerarà la suma dels costos dels operadors que componen el camí. El cost de les solucions ens permeten discriminar-los i, normalment, ens interessarà la solució de cost mínim (o una d'aquestes solucions si n'hi ha més d'una amb el mateix cost).

Funcions de cost de tres exemples

Considerem, a continuació, dos dels exemples ja coneguts i un de cerca en bases de dades de pagament. Per a cadascun d'aquests exemples li indiquem funcions de cost.

1) Trencaclosques lineal

Podem considerar que tots els operadors del trencaclosques tenen el mateix cost (cost unitari). Així, el cost d'un camí correspondrà a la longitud del camí i una solució amb menys moviments serà considerada més bona que una que en tingui més.

2) Camí mínim

Com que l'objectiu és trobar la ruta de distància mínima, el cost d'un operador el definim com la longitud de la carretera corresponent (en aquest problema, triar un operador és triar una carretera). Evidentment, el cost d'un camí serà la longitud d'aquest camí (suma dels segments de carretera seleccionats). Un problema equivalent al del camí mínim serà considerar el camí de preu mínim. En aquest cas, haurem de tenir una avaluació dels preus en euros per a cada carretera.

3) Cerca en bases de dades de pagament

Quan per a trobar una determinada informació en una base de dades, hi hem d'accedir més d'un cop fent diverses consultes i, per a cada consulta, hem de pagar una certa quantitat, tindrem que cada consulta correspon a un operador i que el cost dels operadors és el preu de fer aquestes consultes. De manera anàloga als casos anteriors, el cost de la solució correspondrà a la suma dels costos dels accessos.

Vegeu també

Vegeu els exemples del trencaclosques lineal i del camí mínim en el subapartat «Espai d'estats i representació d'un problema» d'aquest mòdul.

Ara veurem alguns mètodes que, a fi de trobar la solució, utilitzen el cost o una informació similar.

4.1. Cerca de cost uniforme

Per a ordenar els nodes de la llista de nodes a expandir, l'algorisme de cost uniforme utilitza el cost del seu camí des de l'estat inicial. Així, quan s'extreu el primer node de la llista, tenim que aquest és sempre el node terminal del camí de menys cost.

Donat un node n de l'arbre d'expansió, denotarem amb $g(n)$ el cost del camí de l'estat inicial al node n.

Figura 13

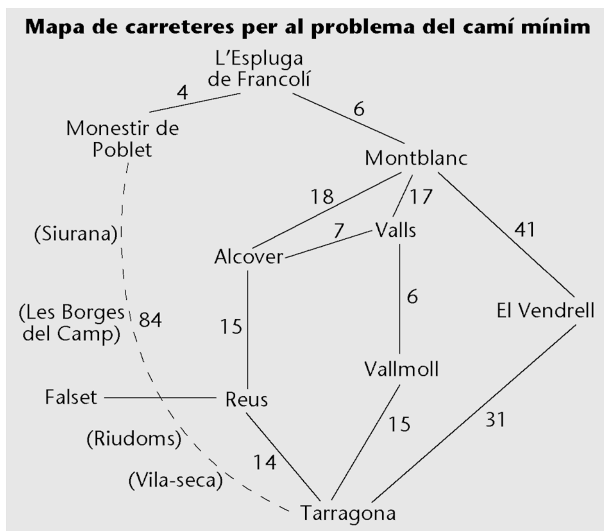
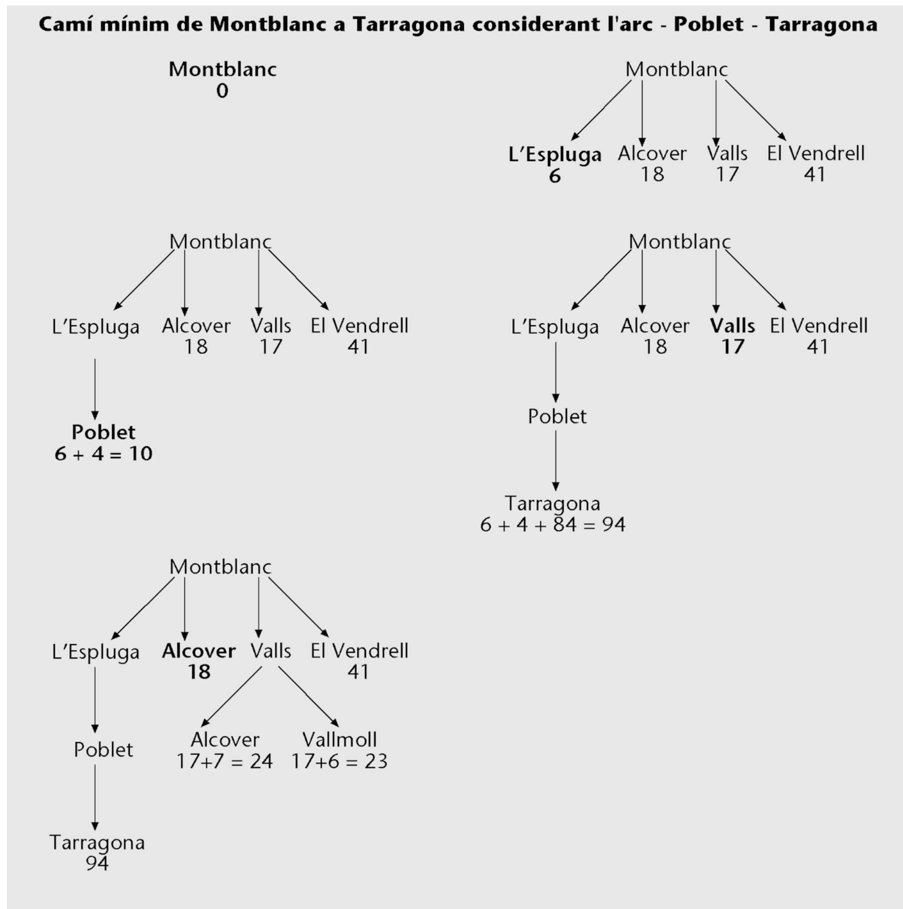


Figura 14



Cerca del cost uniforme per al problema del camí mínim de Montblanc a Tarragona quan fem servir el mapa de carreteres considerant l'arc de Poblet a Tarragona.

Exemple del camí mínim

Considerem, a continuació, el camí mínim de Montblanc a Tarragona quan els operadors del problema són aquells que es poden deduir del mapa de carreteres per al problema del camí mínim. Aquest camí es desenvolupa parcialment en la figura que representa la cerca del cost uniforme i la llista de nodes a expandir apareix amb detall a la taula següent. Per a simplificar l'exemple, hem suposat que en una població no podem desfer el darrer camí (no es pot visitar el pare d'un node). Així, si el darrer arc considerat és el de Montblanc a Valls, no podem passar de Valls a Montblanc.

Vegeu també

Vegeu el subapartat «Algunes consideracions addicionals».

Com es veu en la figura i en la taula, la cerca del cost uniforme comença expandint Montblanc i obtenint 4 poblacions. Cada població és avaluada amb la funció de cost (en aquest cas, la distància amb Montblanc) i inserida en la llista de nodes a expandir. Després, se seleccionarà el poble amb un valor menor (l'Espluga) i s'expandirà. Com que no permetem tornar al node pare, no podem refer el darrer camí i així només obtenim un nou estat: Poblet. El cost d'aquest estat serà el de l'Espluga més el cost de l'operador de l'Espluga a Poblet (la distància de l'arc). Així, tenim que Poblet està avaluat en 6 + 4 = 10. Seguidament, expandirem el node amb un cost menor que és Poblet.

Observació

Recordeu que en el subapartat «La implementació» s'havia dit que, per a acabar, cal esperar que el node seleccionat sigui la solució i que aquesta sigui la solució òptima. Aquest és el cas de la cerca de cost uniforme com veiem aquí.

La figura 14 fa els primers passos de l'expansió. Es pot veure que, tot i que l'algorisme a expandir Poblet genera l'estat Tarragona, aquest no es dona com a solució. Això és així perquè l'esquema general de cerca presentat en el subapartat «La implementació» només acaba quan és el node seleccionat a expandir aquell que és la solució i aquest no és el cas aquí. Ara tenim que el node seleccionat és Poblet i Tarragona quedarà a la cua de nodes a expandir.

Llista de nodes

Llista de nodes pendents d'expandir per al problema del camí mínim de Montblanc a Tarragona quan fem servir el mapa de carreteres i la cerca de cost uniforme, considerant la carretera Poblet-Tarragona.

La taula següent dona les fronteres que anem obtenint en els passos successius fins a trobar la solució. Tenim els estats que apareixen a la frontera i, per a cadascun d'aquests, entre parèntesis, la seva funció de cost.

Apareixen subratllats els nodes seleccionats que s'expandeixen. L'ordre dels nodes de la llista no és el que hi hauria a la cua amb prioritat, sinó que segueixen la frontera de la figura. La cua amb prioritat els ordenaria segons el cost que apareix entre parèntesis.

```
1.Montblanc (0)

2.L'Espluga (6), Alcover (18), Valls (17), El Vendrell (41)

3.Poblet (10), Alcover (18), Valls (17), El Vendrell (41)

4. Tarragona (94), Alcover (18), Valls (17), El Vendrell (41)

5. Tarragona (94), Alcover (18), Alcover (24), Vallmoll (23), El
Vendrell (41)

6. Tarragona (94), Valls (25), Reus (33), Alcover (24), Vallmoll (23),
El Vendrell (41)

7. Tarragona (94), Valls (25), Reus (33), Alcover (24), Tarragona
(38), El Vendrell (41)

8. Tarragona (94), Valls (25), Reus (33), Montblanc (42), Reus (39),
Tarragona (38), El Vendrell (41)

9. Tarragona (94), Montblanc (42), Vallmoll (31), Reus (33), Montblanc
(42), Reus (39), Tarragona (38), El Vendrell (41)

10. Tarragona (94), Montblanc (42), Tarragona (46), Reus (33), Mont-
blanc (42), Reus (39), Tarragona (38), El Vendrell (41)

11. Tarragona (94), Montblanc (42), Tarragona (46), Falset (68),
Tarragona (47), Montblanc (42), Reus (39), Tarragona (38), El Vendrell
(41)
```

Si resseguim l'algorisme fins a trobar la solució, podem observar que troba el camí òptim. De fet, la cerca de cost uniforme sempre troba la solució òptima (això és, troba el camí de menys cost) quan el cost d'un camí no decreix mai –quan els costos dels operadors són sempre positius. Dit d'una altra manera, no decreix el cost quan s'afegeixen més nodes (és una funció monòtona).

L'optimalitat de l'algorisme és a causa del fet que la cerca s'atura quan se selecciona un node i aquest és la solució i no quan troba un estat objectiu d'estats resultat d'una expansió. Quan l'algorisme acaba, voldrà dir que no hi ha cap camí parcial amb un cost menor que el del camí que hem trobat cap a la solució. Per tant, si els costos són sempre positius qualsevol altre camí sempre tindrà un cost més gran.

Solució òptima

En l'exemple del camí mínim, hem considerat que mai no podíem desfer el darrer tros de camí. És important subratllar que si haguéssim permès retornar a l'estat pare també hauríem trobat la solució òptima.

> La cerca del cost uniforme és completa i òptima quan la funció del cost és sempre positiva.

Bucles de cost positiu

Noteu que el fet de tenir bucles no impedeix trobar la solució (sempre que siguin bucles de cost positiu) perquè quan iterem sempre acabaran sobrepassant el cost de qualsevol altre camí. Això provocarà que també es desenvolupi el camí òptim.

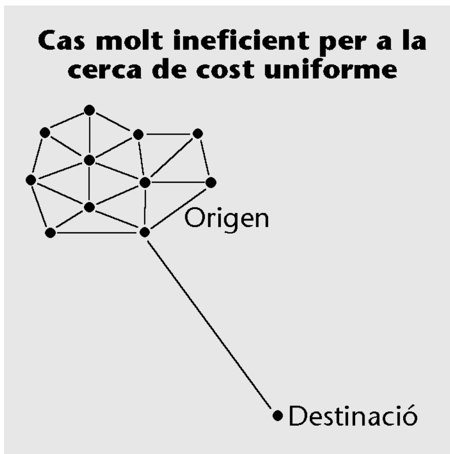
La cerca del cost uniforme es pot veure com una generalització de la cerca en amplada: quan els costos de tots els operadors són iguals tindrem una cerca en amplada. De fet, la cerca que hem presentat aquí té propietats semblants.

4.2. Cerca amb funció heurística: cerca àvida

La cerca del cost uniforme és ineficient en alguns casos perquè l'expansió no té en compte si ens estem acostant o no a un estat solució. Així, en l'exemple considerat abans, de Montblanc passem a l'Espluga i s'expandeix aquest abans d'expandir-ne qualsevol altre. En particular s'expandeix abans que Valls quan aquesta darrera població és més propera a Tarragona. En general, si tenim una situació com la que apareix en la figura que representa un cas molt inefici-

ent, tindrem que la cerca del cost uniforme expandirà molts nodes abans no s'arriba a la destinació. Només passarà a l'estat solució, quan tots els camins parcials superin el cost de l'arc directe.

Figura 15



El problema que aquí es planteja apareix perquè la cerca del cost uniforme només té en compte a l'hora de decidir quin node expandeix el seu cost des de l'estat inicial. No es té en compte que del nou node s'ha de prosseguir el camí fins a la solució. Així, en el cas de decidir entre expandir l'Espluga i Valls, si considerem que Valls està més a prop de Tarragona que l'Espluga, el podríem expandir primer. Tanmateix, com que en general no podem saber fins a quin punt és a prop un estat de la solució, farem servir una funció que ens fa una estimació de la proximitat del node a la solució. Una funció d'aquestes característiques es diu una *funció heurística*.

Una **funció heurística** és una funció que, aplicada a un node, estima el cost del millor camí entre aquest node i un estat solució. Farem servir $h(n)$ per a representar les funcions heurístiques.

Evidentment tindrem que les funcions heurístiques depenen del problema (de les característiques dels operadors i dels estats que satisfan la funció objectiu). Habitualment, tindrem que les funcions són positives i definides de manera que quan són aplicades a un estat solució retornin el valor zero ($h(n)$=0 si n correspon a un estat objectiu).

Exemples de funcions heurístiques

1) Trencaclosques lineal

El nombre de peces mal posades és una funció heurística. Com més peces hi hagi fora de lloc, més moviments caldrà fer per a col·locar-les.

2) Camí mínim

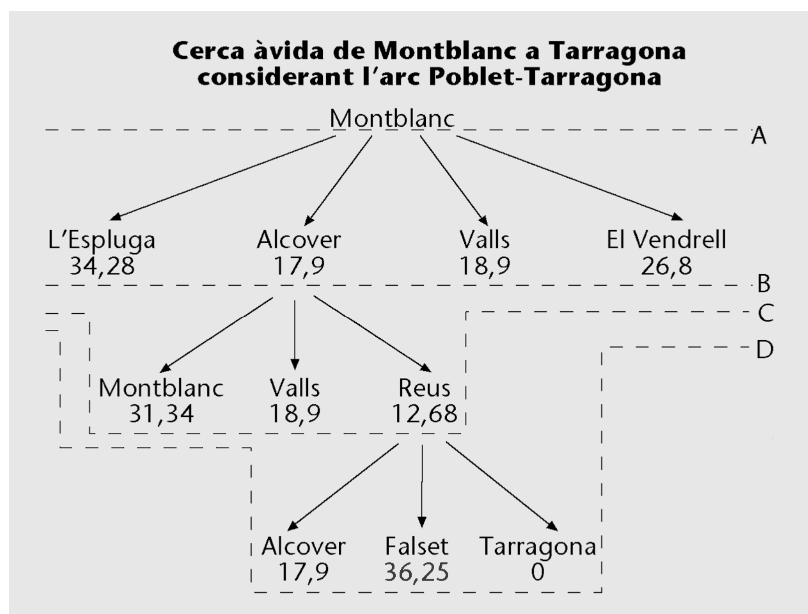
La distància en línia recta entre una població i la població on volem anar es pot fer servir com a funció heurística. La distància que hem de recórrer acostuma a ser més gran com més gran és la distància en línia recta.

L'ús de la funció heurística com a criteri en l'ordenació dels nodes en la llista dels nodes a expandir correspon a la cerca àvida (o del primer millor). El fet de triar el node més proper en passos successius provoca generalment que un cop triat un camí aquest es vagi seguint fins a la solució. Per això, la cerca àvida s'assembla a la cerca en profunditat. A més, té els mateixos problemes que la cerca en profunditat en el sentit que no sempre troba la solució i, quan la troba, no cal que sigui la solució òptima.

Taula 1

Funció heurística		
Població	**Destinació = Tarragona**	**Destinació = Falset**
Alcover	17,9	32,5
L'Espluga	34,28	37,5
Falset	36,25	0
Montblanc	31,34	37,5
Poblet	34,2	35,5
Reus	12,68	24,25
Tarragona	0	36,25
Vallmoll	14,47	36,25
Valls	18,9	37,5
El Vendrell	26,8	60

Figura 16



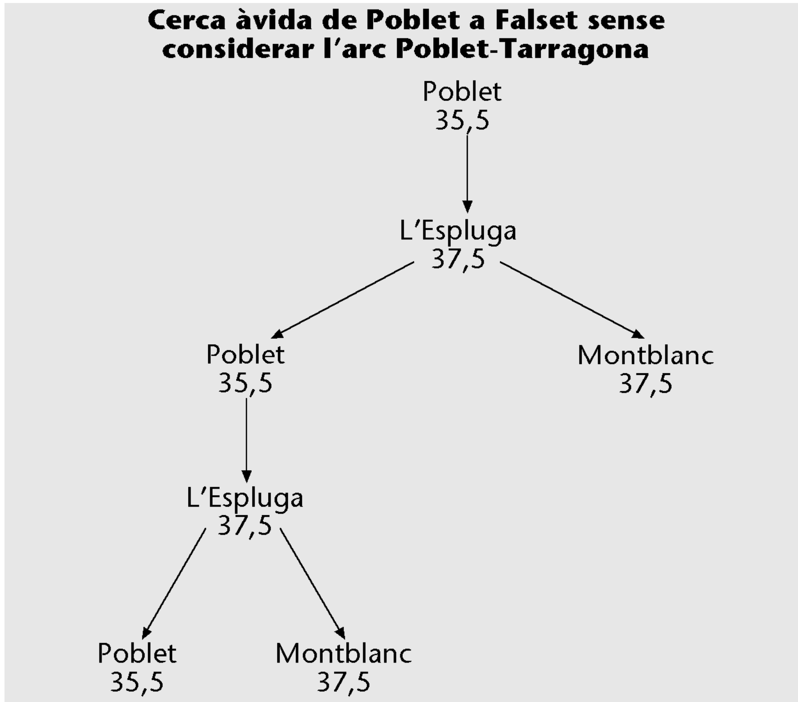
Exemples d'aplicació de la cerca àvida

Seguint amb l'exemple, en la figura 16 es presenta l'evolució de l'arbre de cerca quan partint de Montblanc es vol arribar a Tarragona. Per a aplicar l'algorisme, es fan servir

els valors de la funció heurística que apareixen a la taula de la funció heurística. Aquests valors són els que es fan servir per a ordenar els nodes que hi ha a la frontera i seleccionar a cada iteració el node a expandir. Es pot veure que la solució trobada no és l'òptima. El cost del camí és més gran que en el cas de la cerca amb cost uniforme.

En la figura 17, es veu l'evolució de l'arbre de cerca quan partint de Poblet es vol anar a Falset. En aquest exemple no considerem l'arc Poblet –Tarragona. L'exemple mostra que, en aquest cas, l'algorisme de cerca entra en un bucle sense fi perquè l'Espluga i Poblet són sempre els estats amb un valor de la funció heurística menor.

Figura 17



Amb aquests dos exemples s'ha vist que la cerca àvida no és ni completa ni òptima. Pel que fa als costos, tenim que si m és la profunditat màxima, on hi ha la solució, els costos tant de temps com d'espai són per al cas pitjor $O(b^m)$. De tota manera el cost real depèn de la funció heurística triada i pot fer que tant el cost en espai com en temps disminueixi substancialment. Per exemple, en el cas del problema de Montblanc a Tarragona, hem vist que el nombre de nodes que hem hagut d'expandir només són tres.

4.3. Cerca amb funció heurística: algorisme A*

Com s'ha vist, la cerca àvida tria aquell node que estima mínim el cost d'un node a un estat objectiu i, per tant, escurça el que resta encara de cerca. Aquesta tria, però, fa que el mètode no sigui ni complet ni òptim. Per altra banda, la cerca del cost uniforme minimitza el cost del camí del node origen fins al node en curs. D'aquesta manera es té sempre el millor cost, amb independència del que falti per a arribar a la solució. En aquest sentit, els dos algorismes de cerca són complementaris. Definint una funció f per a cada node n com:

$$f(n) = g(n) + h(n)$$

on la $g(n)$ és el cost des de l'estat inicial al node n i $h(n)$ és l'estimació d'aquest node a la solució, tenim que $f(n)$ és una estimació del cost del camí que va de l'estat inicial a un de solució passant pel node actual n. La selecció del node a expandir, d'acord amb aquesta funció, és l'anomenat *algorisme A**.

La figura 18 representa gràficament el significat d'aquesta funció per als nodes de la frontera. Així, una estimació del cost de l'estat inicial a un estat objectiu passant per ($f(n)$) es pot definir com el cost des de l'estat inicial a n (aquesta part és $g(n)$) més l'estimació del cost des d'n fins a l'estat objectiu ($h(n)$).

Representació gràfica de f(n)

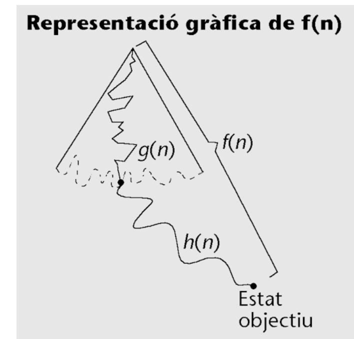


Figura 18

L'optimalitat i completesa de l'algorisme A* depèn de la funció heurística. Tenim que és un algorisme òptim i complet quan la funció heurística no sobreestima mai el cost d'arribar a l'objectiu. Si la funció h satisfà aquesta propietat direm que és una funció heurística admissible.

> Una funció heurística és admissible si mai no sobreestima el cost. L'algorisme A* amb una funció heurística admissible és complet i òptim.

Podem dir que una funció heurística admissible és una funció optimista, ja que sempre ens estima el cost a la baixa. Això és, fa creure que el cost és menor del que realment és.

Exemples de funcions heurístiques admissibles i no admissibles

1) Camí mínim

La distància en línia recta és una heurística admissible, ja que la distància real sempre serà més gran que la distància en línia recta.

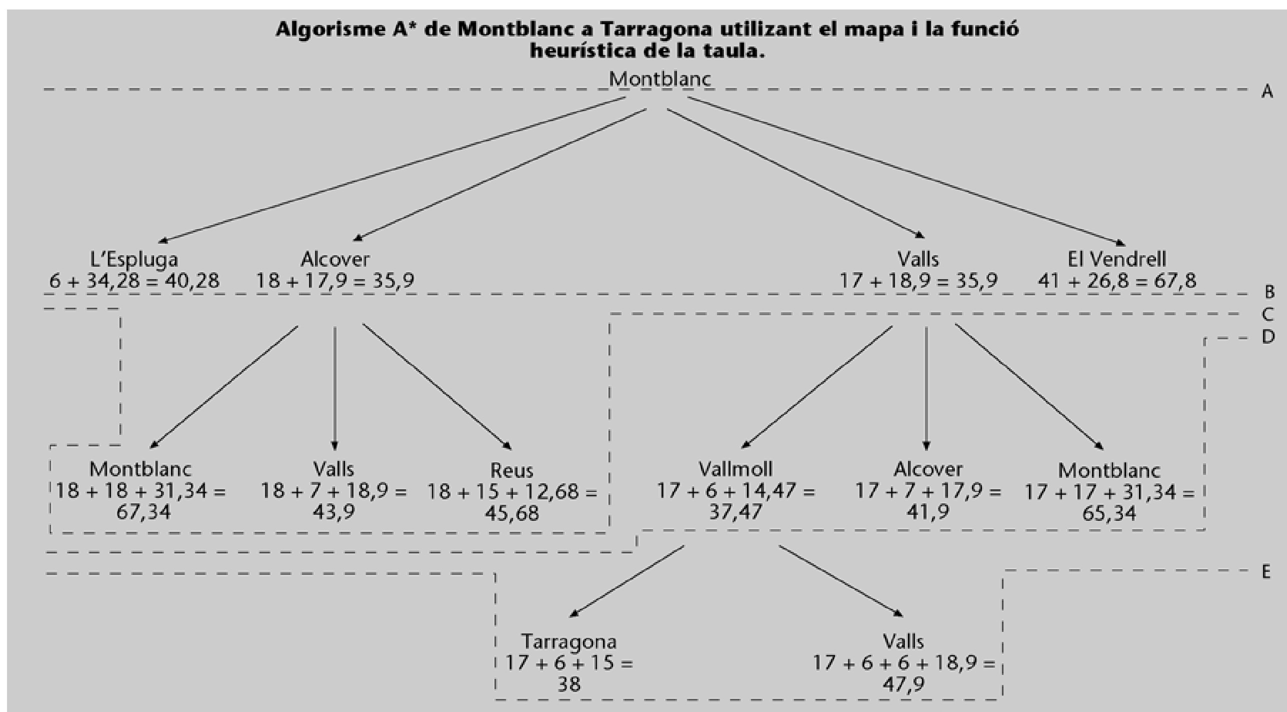
2) El trencaclosques lineal

El nombre de peces mal posades és una funció heurística, però no és admissible perquè podem tenir dues peces mal posades i que es pugui arribar a la solució amb un sol moviment (una única permutació). En canvi, el nombre de peces mal posades dividit entre dos sí que és una funció admissible. Noteu que per a resoldre taulers amb dues peces mal posades, necessitem com a mínim un moviment; per a resoldre taulers amb tres peces, necessitem com a mínim dos moviments; i pels de quatre peces mal posades, també necessitem dos moviments com a mínim. En la taula, trobem els valors que retornaria la funció heurística en tres casos diferents i el nombre de moviments necessaris.

Taula 2

Valor de la funció heurística i nombre de moviments mínim per a tres exemples de tauler de trencaclosques lineal			
Tauler	**Nombre de peces mal posades**	**Heurística**	**Nombre de moviments mínim**
[b, a, c, d]	2	1	1
[b, c, a, d]	3	1,5	2
[b, a, d, c]	4	2	2

Figura 19



En la figura 19, es desenvolupa l'algorisme A* pel cas del camí mínim amb origen a Montblanc i destinació a Tarragona. Com que cada vegada començarem amb l'arbre de cerca amb un únic node corresponent a Montblanc, quan expandim aquest node obtenim les quatre poblacions connectades segons el mapa. Cadascuna d'aquestes poblacions és avaluada amb la funció d'avaluació que consisteix a sumar el cost de l'arc de Montblanc a la població i l'heurística de la població a Tarragona. D'aquesta manera s'obtenen els valors 6 + 34,28 = 40,28 per a l'Espluga, 18 + 17,9 = 35,9 per a Alcover, 17 + 18,9 = 35,9 per a Valls i 41 + 26,8 = 67,8 per al Vendrell. A continuació, quan seleccionem el millor node obtenim Alcover (també s'hagués pogut escollir Valls, ja que té el mateix valor) i, per tant, serà aquest node el que expandirem. L'expansió genera les poblacions de Montblanc, Valls i Reus. L'avaluació per a cadascuna d'aquestes correspondrà al cost de Montblanc a Alcover, més el cost d'Alcover a la població i afegint-hi l'heurística de la població a Tarragona. Continuem expandint el node Valls que té associat el valor mínim (35,9). L'expansió ens afegeix a l'arbre els nodes de Vallmoll, Alcover i Montblanc. Tot seguit se selecciona el

node amb una avaluació menor (Vallmoll) i s'expandeix. Quan l'expandim, s'obtenen dos estats (Tarragona i Valls) que seran avaluats i inserits a la llista de nodes a expandir. Seguidament, se seleccionarà el més prometedor que ja és un estat solució (Tarragona), i, per tant, l'algorisme acaba.

4.3.1. Algunes qüestions de la funció heurística

S'ha dit que una funció d'avaluació admissible implica que l'algorisme sigui òptim i complet. A més d'influir en aquests aspectes, les funcions heurístiques influeixen, evidentment, en els costos de temps i d'espai.

A fi d'avaluar les diferents heurístiques d'un problema, podem fer servir l'anomenat *factor de ramificació efectiu* que denotem amb b^*.

> El **factor de ramificació efectiu** es defineix de la manera següent: si la quantitat de nodes expandits per l'algorisme en un problema és N, i la profunditat en què hi ha la solució és d, aleshores el factor de ramificació efectiu b^* és el valor per tal que el corresponent arbre uniforme (equilibrat) de profunditat d tingui exactament N nodes.

Això és, b^* és el valor que compleix:

$$N = 1 + b^* + (b^*)_2 + \dots + (b^*)^d$$

Com que els resultats experimentals diuen que atesa una heurística el factor b^* és força semblant en tots els problemes, per a avaluar una heurística concreta podem considerar uns quants problemes petits i estimar b^*. Una heurística anirà bé quan b^* s'acosta a 1.

Un altre concepte important és el de la *dominància*. Direm que una funció heurística H domina una altra h quan $H(n) \geq h(n)$ per a tots els nodes n. En aquestes condicions, es pot demostrar que A* amb la funció H és més eficient que A* amb h. Això és així perquè tot node expandit amb H s'expandirà també amb h, però no cal que tot node expandit a h ho sigui també a H. Per tant, pot ser que h obligui a expandir més nodes.

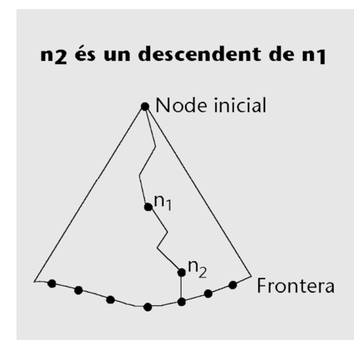


Figura 20

La dominància i la seva influència en l'eficiència de l'algorisme fa que siguin interessants els dos aspectes següents:

Demostrem, a continuació, que tot node que s'expandeixi amb l'heurística H també s'expandeix amb h: suposem que la solució té un cost k. Aleshores, per a arribar a aquesta solució hem d'expandir tots els nodes amb cost + heurística menor que k. Si tenim un node n que s'expandeix amb l'heurística H, vol

dir que el cost (origen, n) + $H(n) < k$. En aquest cas, si $h(n) < H(n)$, aleshores també tenim el cost (origen, n) + $h(n) < k$. Per tant, amb l'heurística h també s'expandirà.

1) Quan en un arbre de cerca el valor d'f mai no decreix mai, direm que l'heurística és monòtona. Això és, que per a tot parell de nodes n_1 i n_2 on n_2 és un descendent de n_1 (vegeu la figura 20, on n_2 és un descendent de n_1), es compleix $f(n_1) \leq f(n_2)$.

Quan aquesta condició no se satisfà, podem definir una nova funció f de la manera següent: $f'(n_2) = \max(f(n_1), g(n_2) + h(n_2))$.

Aquesta definició equival a incrementar el valor de la funció heurística per al node n_2, i, per tant, correspon a definir una nova heurística que domina la primera. A més, aquesta transformació construeix una funció heurística admissible quan l'heurística inicial ja ho és. Això és així perquè sabem que tot camí que passi per n_1 ha de tenir un cost més gran que $f(n_1)$, per la condició d'admissibilitat d'h. Per tant, el cost real del camí que passa per n_2 i n_1 també ha de ser més gran que $f(n_1)$, amb independència que $f(n_1) \geq f(n_2)$ i del valor $f(n_2)$. Per això podem incrementar el valor de la funció aplicada al darrer node fins a $f(n_1)$.

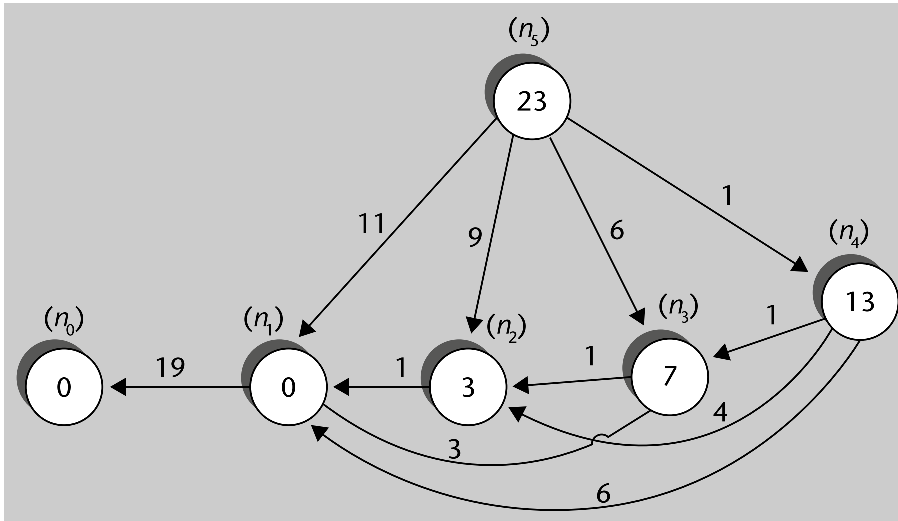
2) Si tenim $h_1, h_2, ..., h_m$ funcions heurístiques admissibles, aleshores podem definir una nova heurística h de la manera següent: $h(n) = \max(h_1(n), h_2(n), ..., h_m(n))$. Aquesta funció h domina les altres i l'algorisme serà, per tant, més eficient.

4.3.2. Consistència de l'heurístic

Un heurístic d'un graf és *consistent* si és un heurístic admissible amb la propietat següent: per a tot parell de nodes d'un graf x i y, si hi ha un camí del node x al node y, aleshores $h(x) \leq \text{cost}(x, y) + h(y)$, on cost($x$, y) és el cost d'anar d'x a y. Judea Pearl va demostrar que podem restringir y a ser un veí d'x per a donar una definició més intuïtiva, tot i que equivalent: el fet d'anar d'un node a un veí seu no ha de fer decréixer h més del que creix g. Direm que un heurístic es *inconsistent* quan no és consistent. Fixem-nos que l'admissibilitat forma part de la definició de consistència, per tant, que un heurístic sigui consistent implica, per definició, que és admissible i que l'A* trobarà la solució òptima si fem servir aquest heurístic. També observem que si consistent $\Rightarrow$ admissible, aleshores *no admissible* $\Rightarrow$ *inconsistent*.

Alberto Martelli va investigar precisament aquest problema, i va definir una família de grafs G_i, on $\{i \geq 3\}$, anomenats *Grafs de Martelli* amb determinades propietats especials. Donat el graf de Martelli G_5:

Figura 21



on el nombre de les arestes és el cost d'anar d'un node a l'altre i l'heurístic és el nombre dins cada node. Així, $h(n_5) = 23$, $h(n_4) = 13$, etc. L'objectiu és trobar el camí de cost més petit per anar d'n_5 a n_0.

Demostrar que h és admissible és senzill, ja que el cost òptim d'arribar de qualsevol node a n_0 no pot ser inferior a 19, és a dir, $h^*(n_j) \geq 19$, per a tot j entre 1 i 5. Això és a causa de l'aresta que va d'n_1 a n_0. I tots els heurístics són menors que 19: $h(n_j) < 19 \leq h^*(n_j)$, per a tots els nodes excepte n_5. El camí òptim per a n_5 té un cost 23, com veurem després quan apliquem l'A*, però que és força evident. Com que $23 = h(n_5) \leq h^*(n_5) = 23$, h és admissible. La inconsistència d'h es veu, ja que $13 = h(n_4) > \text{cost}(n_4, n_3) + h(n_3) = 1 + 7$. Hi ha més inconsistències.

Fixem-nos que a l'algorisme A* utilitzem dues llistes de nodes oberts i tancats, revisant els costos en cas de millorar-los. En canvi, si mireu el pseudocodi de la Wikipedia, trobareu un algorisme on, quan s'expandeix un node, si algun dels seus veïns és el conjunt de tancats, ignorem el veí en qüestió, sense revisar per a res els seus costos.

Considerarem dos algorismes A* diferents:

- **A* complet:** Quan considerem els successors del node actual, si en trobem un que ja està en el conjunt de tancats, calculem el seu cost i si millora el cost que tenia quan el vam tancar, traiem el successor del conjunt de tancats i el tornem a posar al conjunt d'oberts amb el cost revisat.

- **A* simplificat:** Quan considerem els successors del node actual, si en trobem un que ja està en el conjunt de tancats l'ignorem. Aquest és el cas del pseudocodi mencionat que podeu trobar a la Wikipedia.

L'A* simplificat només funciona bé si l'heurístic és consistent. Si apliquem l'A* simplificat a G_5, veurem que el camí que troba és el camí d'n_5 a n_1, i d'n_1 a n_0, amb cost 30, que no és l'òptim. Si apliquem l'A* complet trobem el camí òptim, de cost 23 (exercici pel lector: apliqueu aquests algorismes manualment).

Detall de l'A* simplificat i complet aplicats a G_5

Si considerem l'A* simplificat en el graf anterior G_5, aleshores les llistes de nodes oberts i tancats que obtenim en cada iteració són els següents (el node subratllat és el node de la llista de nodes oberts que s'expandeix en la iteració següent pel fet de ser el de cost mínim):

Oberts(0): (n_5, 0 + 23) → (n_5, 23)
Tancats(0):

Oberts(1): (n_1, 11 + 0), (n_2, 9 + 3), (n_3, 6 + 7), (n_4, 1 + 13) → (n_1, 11), (n_2, 12), (n_3, 13), (n_4, 14)
Tancats(1): (n_5, 23)

Oberts(2): (n_0, 11 + 19 + 0), (n_2, 12), (n_3, 13), (n_4, 14) → (n_0, 30), (n_2, 12), (n_3, 13), (n_4, 14)
Tancats(2): (n_1, 11), (n_5, 23)

Oberts(3): (n_0, 30), (n_3, 13), (n_4, 14)
Tancats(3): (n_1, 11), (n_2, 12), (n_5, 23)

Oberts(4): (n_0, 30), (n_4, 14)
Tancats(4): (n_1, 11), (n_2, 12), (n_3, 13), (n_5, 23)

Oberts(5): (n_0, 30)
Tancats(5): (n_1, 11), (n_2, 12), (n_3, 13), (n_4, 14), (n_5, 23)

Per tant, veiem que com que no es recalculen els costos pels nodes ja visitats, l'A* simplificat decideix que el camí per anar del node 5 al node 0 és el següent:

$$n_5 \rightarrow n_1 \rightarrow n_0$$

El cost d'aquest camí és 30 (11 + 19) i no és el camí mínim existent entre els nodes n_5 i n_0. En canvi, utilitzant l'A* complet, sense restriccions, obtenim un desplegament de la cerca més llarg, però correcte:

Oberts(0): (n_5, 0 + 23) → (n_5, 23)
Tancats(0):

Oberts(1): (n_1, 11 + 0), (n_2, 9 + 3), (n_3, 6 + 7), (n_4, 1 + 13) → (n_1, 11), (n_2, 12), (n_3, 13), (n_4, 14)
Tancats(1): (n_5, 23)

Oberts(2): (n_0, 11 + 19 + 0), (n_2, 12), (n_3, 13), (n_4, 14) → (n_0, 30), (n_2, 12), (n_3, 13), (n_4, 14)
Tancats(2): (n_1, 11), (n_5, 23)

Oberts(3): (n_0, 30), (n_1, 9 + 1 + 0), (n_3, 13), (n_4, 14) → (n_0, 30), (n_1, 10), (n_3, 13), (n_4, 14)
Tancats(3): (n_2, 12), (n_5, 23)

Oberts(4): (n_0, 9 + 1 + 19), (n_0, 30), (n_3, 13), (n_4, 14) → (n_0, 29), (n_3, 13), (n_4, 14)
Tancats(4): (n_1, 10), (n_2, 12), (n_5, 23)

Oberts(5): (n_0, 29), (n_1, 6 + 3 + 0), (n_2, 6 + 1 + 3), (n_4,14) → (n_0, 29), (n_1, 9), (n_2, 10), (n_4, 14)
Tancats(5): (n_3, 13), (n_5, 23)

Oberts(6): (n_0, 29), (n_0, 6 + 3 + 19 + 0), (n_2, 10), (n_4, 14) → (n_0, 28), (n_2, 10), (n_4, 14)
Tancats(6): (n_1, 9), (n_3, 13), (n_5, 23)

Oberts(7): (n_0, 28), (n_1, 6 + 1 + 1 + 0), (n_4, 14) → (n_0, 28), (n_1, 8), (n_4, 14)
Tancats(7): (n_2, 10), (n_3, 13), (n_5, 23)

Oberts(8): (n_0, 28), (n_0, 6 + 1 + 1 + 19 + 0), (n_4, 14) → (n_0, 27), (n_4, 14)
Tancats(8): (n_1, 8), (n_2, 10), (n_3, 13), (n_5, 23)

Oberts(9): (n_0, 27), (n_1, 1 + 6 + 0), (n_2, 1 + 4 + 3), (n_3, 1 + 1 + 7) → (n_0, 27), (n_1, 7), (n_2, 8), (n_3, 9)
Tancats(9): (n_4, 14), (n_5, 23)

Oberts(10): (n_0, 27), (n_0, 1 + 6 + 19 + 0), (n_2, 8), (n_3, 9) → (n_0, 26), (n_2, 8), (n_3, 9)
Tancats(10): (n_1, 7), (n_4, 14), (n_5, 23)

Oberts(11): (n_0, 26), (n_1, 1 + 4 + 1 + 0), (n_3, 9) → (n_0, 26), (n_1, 6), (n_3, 9)
Tancats(11): (n_2, 8), (n_4, 14), (n_5, 23)

Oberts(12): (n_0, 26), (n_0, 1 + 4 + 1 + 19 + 0), (n_3, 9) → (n_0, 25), (n_3, 9)
Tancats(12): (n_1, 6), (n_2, 8), (n_4, 14), (n_5, 23)

Oberts(13): (n_0, 25), (n_1, 1 + 1 + 3 + 0), (n_2, 1 + 1 + 1 + 3) → (n_0, 25), (n_1, 5), (n_2, 6)
Tancats(13): (n_3, 9), (n_4, 14), (n_5, 23)

Oberts(14): (n_0, 25), (n_0, 1 + 1 + 3 + 19 + 0), (n_2, 6) → (n_0, 24), (n_2, 6)
Tancats(14): (n_1, 5), (n_3, 9), (n_4, 14), (n_5, 23)

Oberts(15): (n_0, 24), (n_1, 1 + 1 + 1 + 1 + 0) → (n_0, 24), (n_1, 4)
Tancats(15): (n_2, 6), (n_3, 9), (n_4, 14), (n_5, 23)

Oberts(16): (n_0, 24), (n_0, 1 + 1 + 1 + 1 + 19 + 0) → (n_0, 23)
Tancats(16): (n_1, 4), (n_2, 6), (n_3, 9), (n_4, 14), (n_5, 23)

Quan se selecciona el (n_0, 23) l'A* complet finalitza i troba que el camí entre els nodes n_5 i n_0 és el següent:

$$n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_0$$

El cost d'aquest camí és 23 (1 + 1 + 1 + 1 + 19) i es tracta del camí òptim entre els nodes n_0 i n_5. Noteu un parell de detalls en el pas a pas de l'algorisme: els nodes revisitats amb un cost menor s'eliminen de la llista de nodes tancats i els nodes duplicats de la llista de nodes oberts es manté únicament el node de cost menor. Un exemple del primer cas el podem trobar en la llista de nodes Oberts(3), on hi ha (n_1, 10) i, per tant, s'elimina (n_1, 11) de la llista de tancats. Un exemple del segon cas el trobem en la llista de nodes Oberts(4), on tenim (n_0, 29) i (n_0, 30) i només mantenim (n_0, 29).

4.3.3. Implementació

A continuació, es fa una implementació amb Python de la cerca A*. Comencem definint el problema associant informació addicional als nodes, com en el cas de la cerca en profunditat limitada. Ara tindrem associat a cada node el valor del cost del node i el d'*f*(*n*) (la funció heurística més el cost del node). En cada expansió recalcularem aquests valors a partir del que teníem en el node pare i de l'heurística del nou estat. La funció per a calcular aquests valors per a cada node i la funció per a definir-lo per l'estat inicial es defineixen en el problema anomenat `problema_cercaAstar()`.

```
def problema_cercaAstar():
    tl_ops = tl_operadors()
    def aux_func(info_node_pare, estat, operador):
        estat_pare = car(info_node_pare)
        g          = caadr(info_node_pare)
        g_mes_pare = g + 1
        g_mes_h    = g_mes_pare + heuristic(estat)
        return [g_mes_pare, g_mes_h]
```

```
    estat_inicial = [4,3,2,1]
    check_estat_final = lambda estat: estat == [1,2,3,4]
    return [tl_ops, aux_func, estat_inicial,
            check_estat_final,
            lambda estat: [0, heuristic(estat)
```

Per a provar l'algorisme, necessitem una funció heurística. En construïm una de molt dirigida cap a aconseguir el millor resultat.

```
def heuristic(estat):
    if estat == [4,3,2,1]: return 6
    elif estat == [4,3,1,2]: return 5
    elif estat == [4,1,3,2]: return 4
    elif estat == [1,4,3,2]: return 3
    elif estat == [1,3,4,2]: return 2
    elif estat == [1,3,2,4]: return 1
    elif estat == [1,2,3,4]: return 0
    else: return 10
```

A més d'aquestes definicions, necessitem la funció corresponent a l'estratègia:

```
def tl_estrategia_Astar (nodes_a_expandir, nous_nodes_a_expandir):
    unio = nous_nodes_a_expandir + nodes_a_expandir
    return quicksort(selecciona_estimacio(unio),unio)
```

Aquesta funció comença unint els nodes que teníem a expandir amb els nous nodes i, després, els ordena mitjançant una funció que implementa el mètode d'ordenació `quicksort`. Aquesta funció rep dues llistes com a paràmetres, una amb els nodes a ordenar i una altra (de la mateixa longitud) amb els valors que han de guiar l'ordenació. Així, si fem:

```
quicksort([10, 4, 3],['n1', 'n2', 'n3'])
```

ens retornarà:

```
['n3', 'n2', 'n1']
```

ja que 'n_1' s'associa al valor 10, 'n_2' al valor 4 i 'n_3' al valor 3.

El `quicksort` es compon amb la funció `selecciona_estimacio` que construeix una llista amb els valors *f*(*n*) per a cada node de la nova llista de nodes a expandir (la llista `unio_nodes`).

```
def selecciona_estimacio (unio_nodes):
    return mapcar(lambda node: caddr(cdddr(node)), unio_nodes)
```

Aquesta funció selecciona el segon element del camp d'informació addicional, ja que és aquí on es guarda *f*(*n*).

```
def quicksort (per_ordenar, elems):
    if not elems:
        return []
    pivot = car(per_ordenar)
    elemp = car(elems)
    petits = selecciona_menorigual(pivot, cdr(per_ordenar), cdr(elems))
    grans = selecciona_major(pivot, cdr(per_ordenar), cdr(elems))
    return quicksort(selecciona_estimacio(petits),petits) +
            cons(elemp, quicksort(selecciona_estimacio(grans),grans))
```

La funció `quicksort` segueix l'esquema habitual: selecciona el primer element de la llista a ordenar (que fa de pivot); separa els elements més petits que el pivot d'aquells que són més grans; ordena cadascun d'aquests dos conjunts i, finalment, retorna una nova llista que concatena els petits ordenats, l'element pivot i els grans ja ordenats.

Les funcions que seleccionen els nodes amb una estimació més petita o més gran que el node pivot són aquestes:

```
def selecciona_menorigual (pivot, per_ordenar, elems):
    if not per_ordenar:
        return []
    ll = selecciona_menorigual(pivot,cdr(per_ordenar),cdr(elems))
    if car(per_ordenar) <= pivot:
        return cons(car(elems),ll)
    return ll

def selecciona_major (pivot, per_ordenar, elems):
    if not per_ordenar:
        return []
```

```
    ll = selecciona_major(pivot,cdr(per_ordenar),cdr(elems))
    if car(per_ordenar) > pivot:
        return cons(car(elems),ll)
    return ll
```

Amb aquestes definicions podem definir la cerca de la manera següent:

```
def cerca_Astar (problema):
    return fer_cerca(problema, tl_estrategia_Astar)
```

L'aplicació d'aquesta funció al problema que hem definit abans dona el resultat següent:

```
cerca_Astar(problema_cercaAstar())
['id', 'ic', 'ie', 'ic', 'id', 'ic']
```

En aquest apartat, no hem fet les implementacions de la cerca uniforme i àvida, però seguirien el mateix esquema presentat aquí.

4.4. Altres mètodes de cerca heurística

Un problema que comparteixen la majoria dels algorismes de cerca és que han de guardar a la memòria tots els nodes que en els passos successius de l'algorisme s'han de considerar per a la seva expansió. Normalment, la memòria necessària per a emmagatzemar aquests nodes creix de manera lineal en relació amb el temps d'execució i de manera exponencial en relació amb la dimensió del problema. Això significa que molts dels algorismes no es poden aplicar en problemes de dimensió elevada. Darrerament, s'han dissenyat versions d'algorismes de cerca clàssics que intenten resoldre aquest problema. Una de les alternatives és aprofitar les idees de la cerca iterativa amb profunditat limitada (vegeu el subapartat «Cerca en profunditat limitada»). L'aplicació a l'algorisme A* defineix els algorismes anomenats *IDA**[3]: es poda una branca de l'arbre de cerca segons un llindar que es va modificant en iteracions successives. La primera versió d'aquests algorismes fou de l'any 1985 (de Korf), tot i que hi ha versions posteriors.

(3) Acrònim de l'anglès *iterative-deepening* A*.

Atès que els mètodes com els IDA* infrautilitzen la memòria disponible, s'han desenvolupat altres mètodes que l'aprofiten, però evitant el *col·lapse* del sistema. Són els mètodes anomenats *reducció de nodes*[4]. Quan la memòria està completament plena, els fills d'aquell node que té la pitjor avaluació d'entre els que hi ha a memòria s'esborren i el valor del node pare és modificat. Se

(4) De l'anglès *node retraction.*

li assigna el valor del fill amb un valor menor. D'aquesta manera el node té una estimació segons l'expansió que s'hi ha fet. D'acord amb tot això, aquests algorismes combinen fases d'expansió i de reducció de nodes.

A més d'aquests mètodes, hi ha altres alternatives per a resoldre els problemes d'espai com ara la cerca en el perímetre (un tipus de cerca bidireccional) i la cerca tabú. Un dels mètodes de cerca en el perímetre comença fent una cerca cap enrere (per exemple, una cerca en amplada) a partir d'un estat objectiu fins que gairebé s'ocupa tota la memòria. A partir d'aquest moment, s'aplica una cerca cap endavant des de l'estat inicial fins que troba un estat de la frontera de l'arbre de l'altra cerca.

Amb aquesta cerca s'optimitza l'ús de la memòria i, a més, el cost de la cerca és menor que fer-la només en una direcció. Tanmateix, fer la cerca cap enrere no sempre és possible. De vegades, hi ha molts estats objectius i no podem seleccionar-ne un, d'altres vegades no és fàcil aplicar els operadors cap enrere. Aquest seria el cas del problema de la integració. En d'altres casos, com en els exemples del trencaclosques lineal o el problema del camí mínim, fer la cerca enrere és més fàcil. En aquests casos, els operadors són reversibles (podem definir l'operador que va cap enrere) i, a més, hi ha un únic estat final.

5. Cerca amb adversari: els jocs

Dins la intel·ligència artificial, els jocs són una de les aplicacions més antigues perquè són problemes abstractes i completament formalitzables. Un dels primers programes d'escacs va ser desenvolupat al voltant de l'any 1950 per C. Shannon (el pare de la teoria de la informació).

Des d'un punt de vista formal, la dificultat principal per a prendre decisions en els jocs és la presència d'un adversari, ja que aquest intenta fer aquells moviments que ens són menys favorables. Això introdueix incertesa en el model. Un altre tipus d'incertesa que apareix en alguns jocs és el component aleatori (per exemple, els daus). Els algorismes per a prendre decisions en els jocs han de tenir en compte aquests dos components.

Tot i que els jocs estan resolts des d'un punt de vista formal (hi ha un algorisme per a condicions ideals de temps de computació i de memòria), des del punt de vista computacional això no és tan senzill. Els jocs són problemes difícils de resoldre quan volem una solució amb un temps raonable i que utilitzi la memòria disponible en una certa màquina. Això és així perquè els arbres de cerca que caldria expandir per a donar la solució òptima són immensos. Per exemple, en el cas dels escacs una partida mitjana correspon a uns cent moviments (cinquanta per jugador) i la mitjana del seu factor de ramificació és al voltant de trenta-cinc (quan es considera un tauler es poden fer, de mitjana, uns trenta-cinc moviments diferents). Això correspon a un arbre de cerca amb 35^{100} nodes diferents (que correspon a $2{,}55 * 10^{154}$ nodes). Evidentment, no podem expandir tot l'arbre i després aplicar l'algorisme que ens doni el moviment més bo. Per tant, ens trobem amb el fet que la incertesa en els jocs no és perquè tinguem una manca d'informació sinó perquè no es poden calcular les conseqüències exactes d'un moviment a causa de problemes de temps i de memòria.

A continuació, descriurem els algorismes de cerca per a jocs. Començarem amb els que donen una decisió perfecte perquè representen que ni el temps ni l'espai són recursos limitats. Després, passarem a tenir en compte les restriccions dels recursos. Per acabar, es considerarà l'element aleatori.

5.1. Decisions perfectes

La formalització d'un joc quan no s'ha de considerar cap element de sort és similar a la formalització d'un problema de cerca com els vistos fins ara. Per tant, haurem de definir: la modelització de l'entorn en què es mou el sistema (els estats), la modelització de les accions del sistema i la definició del problema. Passem, ara, a definir cadascun dels elements.

1) Els estats. En el nostre cas, això correspondrà a representar la informació corresponent a un instant del joc. Per exemple, si és un joc amb un tauler i unes fitxes, haurem de representar el tauler i com estan col·locades les fitxes.

2) Les accions. Això correspondrà a modelitzar els moviments que poden fer els jugadors.

3) El problema. Com en qualsevol problema de cerca, vindrà definit per l'estat inicial i la funció objectiu. En un joc, l'estat inicial correspondrà a la posició del joc (per exemple, la situació del tauler) en un moment donat en què volem que el nostre programa prengui una decisió. És important subratllar que la posició inicial no sempre serà la mateixa (i, evidentment, no correspon a la situació del joc al començament de la partida) perquè cada vegada que el programa hagi de decidir tindrem una posició inicial diferent. La funció objectiu la descompondrem en dues funcions: un test d'acabament i una funció d'utilitat. El test d'acabament aplicat a un estat ens dirà si encara es pot continuar jugant. La funció d'utilitat aplicada a un estat ens l'avaluarà dient-nos fins a quin punt és bo.

Una vegada feta la formalització, si el problema fos un típic problema de cerca, n'hi hauria prou amb trobar un camí des de l'estat inicial a un estat objectiu. Aquí, però, això no és així perquè el camí que planifiquem per a trobar la millor solució serà destarotat per l'altre jugador.

Un algorisme que ens permet donar la millor jugada tenint en compte les opcions dels jugadors i les possibles evolucions de la partida un cop s'ha tirat és l'anomenat *algorisme de minimax*. Aquest algorisme és per a jocs de dos jugadors en què no hi intervé cap element de sort. A més, l'algorisme representa que no tenim restriccions ni de memòria ni de temps i que, per tant, podem explorar totes les alternatives dels jugadors.

El mecanisme general d'aquest algorisme es basa en una funció d'utilitat que obté valors grans quan guanya un jugador i petits quan guanya l'altre. D'aquesta manera, per a un determinat jugador l'algorisme triarà aquella opció que li permet aconseguir optimitzar la utilitat.

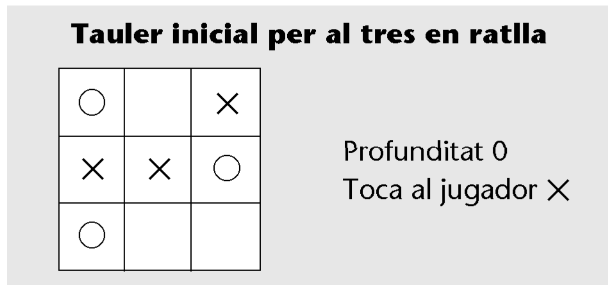
Abans de passar a veure una implementació de l'algorisme, veurem com es pot calcular la millor solució per un joc i un tauler concrets.

Modelització del joc del tres en ratlla

Considerarem el joc del tres en ratlla (amb jugadors *x* i *o*, com és habitual, i en la versió en què els jugadors van fent torns posant les seves fitxes fins que un dels dos guanya (posant tres de les seves fitxes fent una línia) o el tauler està completament ple. Si el tauler és ple i cap dels dos jugadors ha guanyat,

direm que els dos jugadors empaten. Estimem un problema en què la partida ja ha començat i li toca al jugador creu: prendrem el tauler que es mostra en la figura 22.

Figura 22



La modelització del problema del tres en ratlla és força evident:

1) Estats. Els taulers que poden aparèixer en el joc definiran el conjunt d'estats possibles. Així tindrem que són estats tots els taulers de 3 × 3 en què les posicions estan buides o plenes amb alguna de les dues fitxes possibles (*o* o *x*).

2) Accions. Seran accions a considerar les d'afegir una peça al tauler en una posició buida.

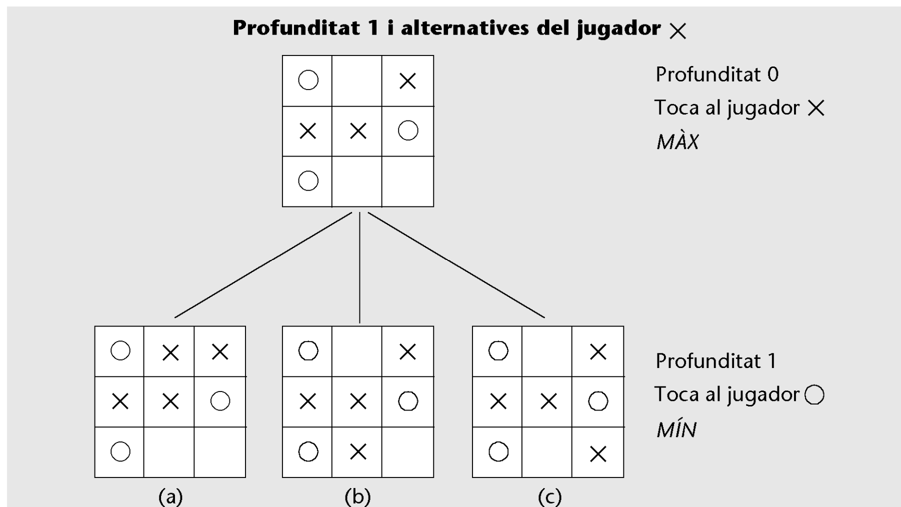
3) El problema. L'estat inicial serà un tauler concret. En el nostre cas, prendrem el del tauler representat en la figura 22. El test d'acabament serà, evidentment, comprovar si algú ja ha fet tres en ratlla o si ja no es poden posar més fitxes.

Com s'ha dit abans, la funció d'utilitat ha de permetre avaluar els estats de manera que si guanya un jugador s'obtinguin valors grans i per l'altre petits. Aquí suposarem aquí que el jugador *x* és a qui li interessa una puntuació alta (anomenarem a aquest jugador MÀX – de màxim). Aleshores, farem que la funció ens retorni un valor d'1 si guanya *x*. En cavi, al jugador *o* el denominarem MÍN (de mínim) i definirem la funció d'utilitat de manera que si guanya MÍN retorni -1. En el cas que els dos jugadors quedin empatats, la funció d'utilitat retornarà 0. Aquesta definició és consistent amb la idea que el jugador *x* prefereix guanyar a empatar, i empatar a perdre (com que és el jugador MÀX prefereix el valor d'1 al valor 0, i el valor 0 al –1, això és, sempre l'interessa el valor més gran). De manera anàloga, *o* prefereix guanyar a empatar i empatar a perdre (prefereix el valor de –1 al 0, i el 0 a l'1).

És important assenyalar que la funció d'utilitat només s'aplica a estats terminals (quan el test d'acabament es compleix): taulers on guanya un jugador o quan estan empatats perquè no es pot posar cap altra peça. La funció d'utilitat no es pot aplicar (no té sentit) quan hi ha taulers amb posicions buides en què no guanya ningú.

Seguim amb l'exemple del joc del tres en ratlla:

Figura 23



L'algorisme de minimax pren les decisions considerant les diferents alternatives. Ara repasseguirem la manera en què es prenen les decisions utilitzant el tauler inicial:

1) En aquest cas, tenim que el jugador *x* pot fer tres moviments diferents que porten als tres taulers de la figura anterior, amb profunditat 1 i on es representen les alternatives del senyor *x* –denotarem aquests taulers amb (a), (b) i (c).

2) Un cop tenim aquestes alternatives, les hem d'avaluar per a poder decidir quina és la millor.

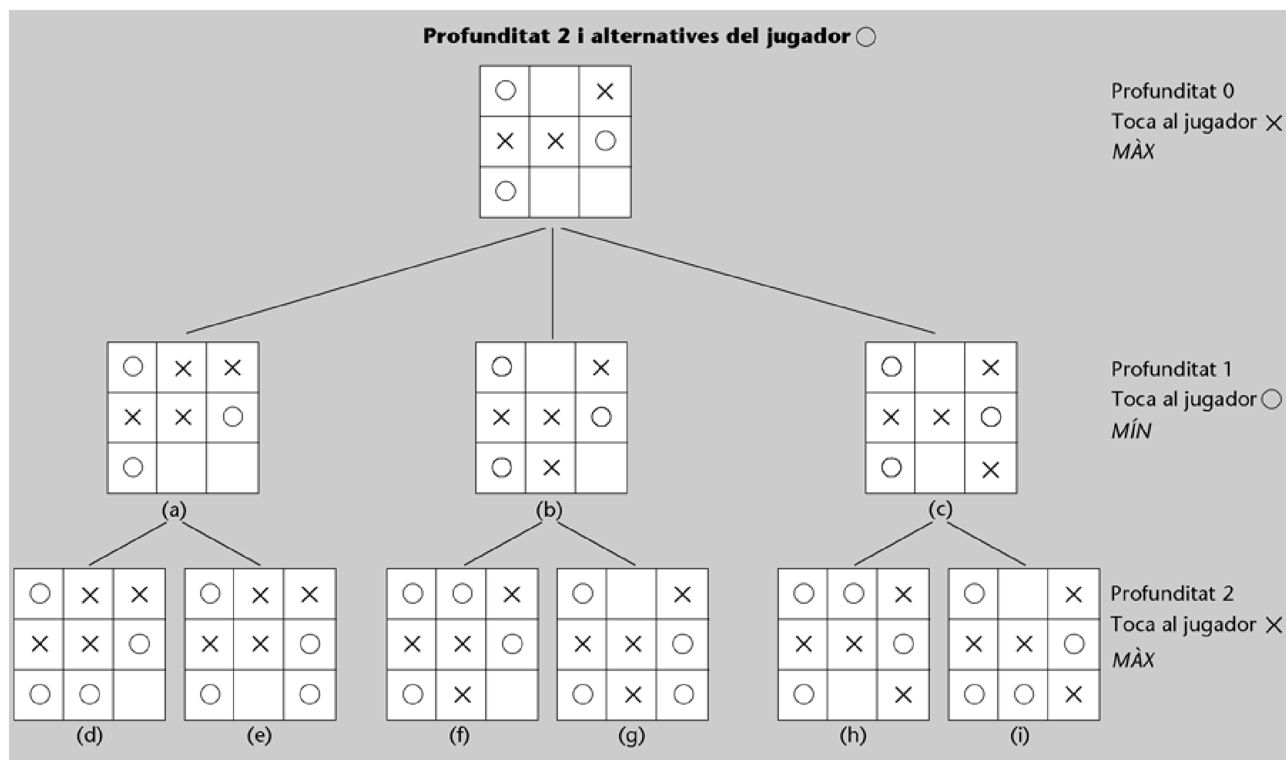
3) Quan siguin avaluades, d'entre els tres taulers triarem aquell que tingui un valor més gran (perquè *x* és el jugador màxim).

Desafortunadament, l'avaluació dels tres taulers no és trivial perquè no són posicions en què un jugador guanyi (tampoc no empaten). Així que no es pot aplicar la funció d'utilitat.

Quan l'avaluació no es pot fer mitjançant la funció d'utilitat, podem aplicar el mateix esquema que hem aplicat al tauler inicial: considerar quines jugades pot fer el jugador, avaluar les jugades i triar la que va millor al jugador. En el nostre cas, això representa que per a cadascun dels tres taulers hem de considerar tots els moviments que pot fer el jugador –en aquest cas és el jugador *o*– i avaluar-los.

L'expansió d'aquests tres taulers es mostra en la figura amb profunditat 2 i on es representen les alternatives del senyor *o*. Obtenim per a cada tauler dos nous taulers. Per exemple, per al tauler (a) obtenim els nous taulers (d) i (e).

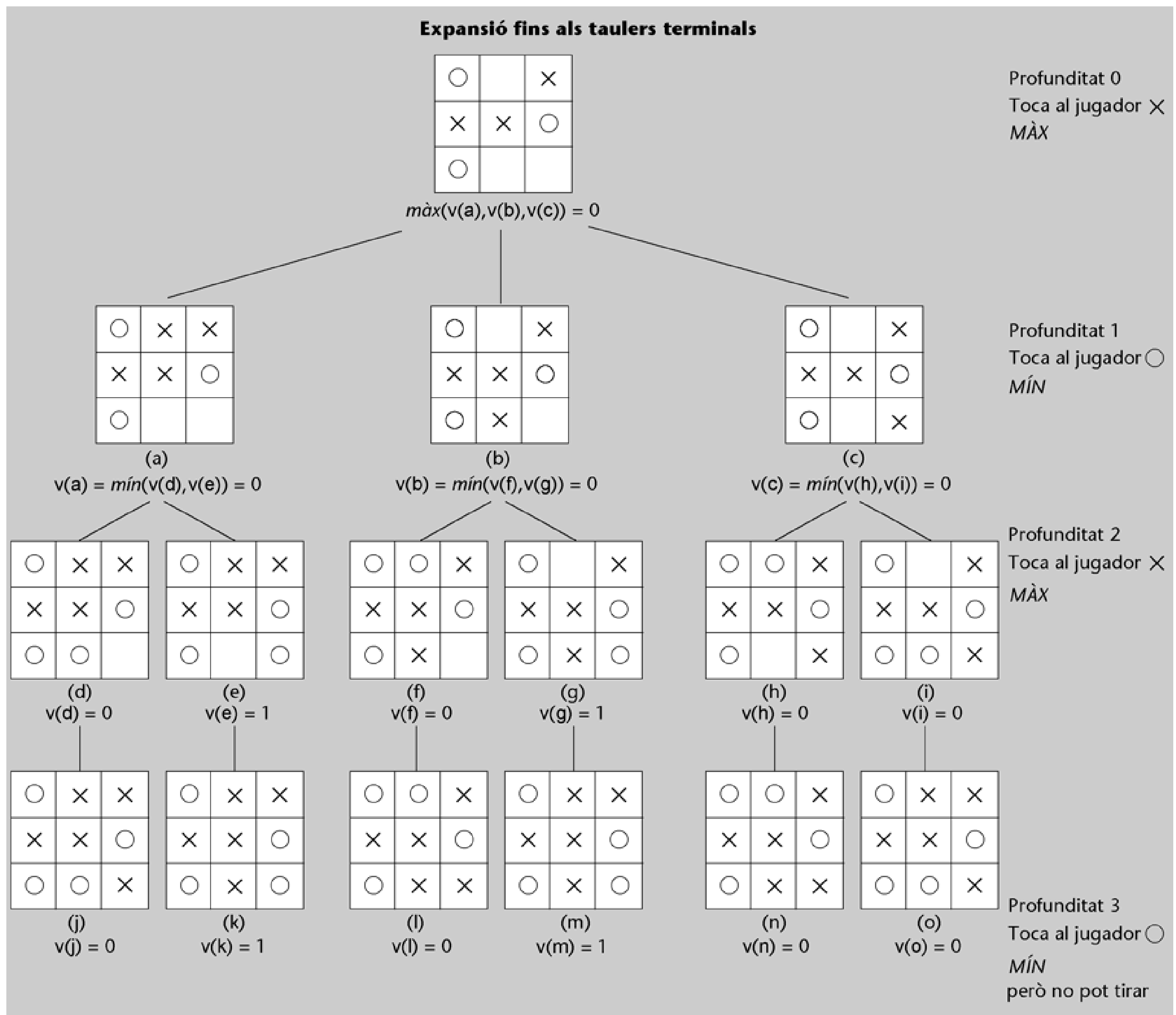
Figura 24



Si els nous taulers fossin terminals, els avaluaríem. Per a determinar l'avaluació dels taulers (a), (b) i (c) que ens quedava pendent triarem en cada cas el millor dels valors d'acord amb el jugador que ha de jugar. Com que ara el jugador és *o*, triarem el moviment que té un valor més petit. Així, per a determinar el valor del tauler (a) calcularem el mínim dels valors dels taulers (d) i (e). De la mateixa manera:

$$v(\text{b}) = \min(v(\text{f}), v(\text{g})) \text{ i } v(\text{c}) = \min(v(\text{h}), v(\text{i})).$$

Figura 25



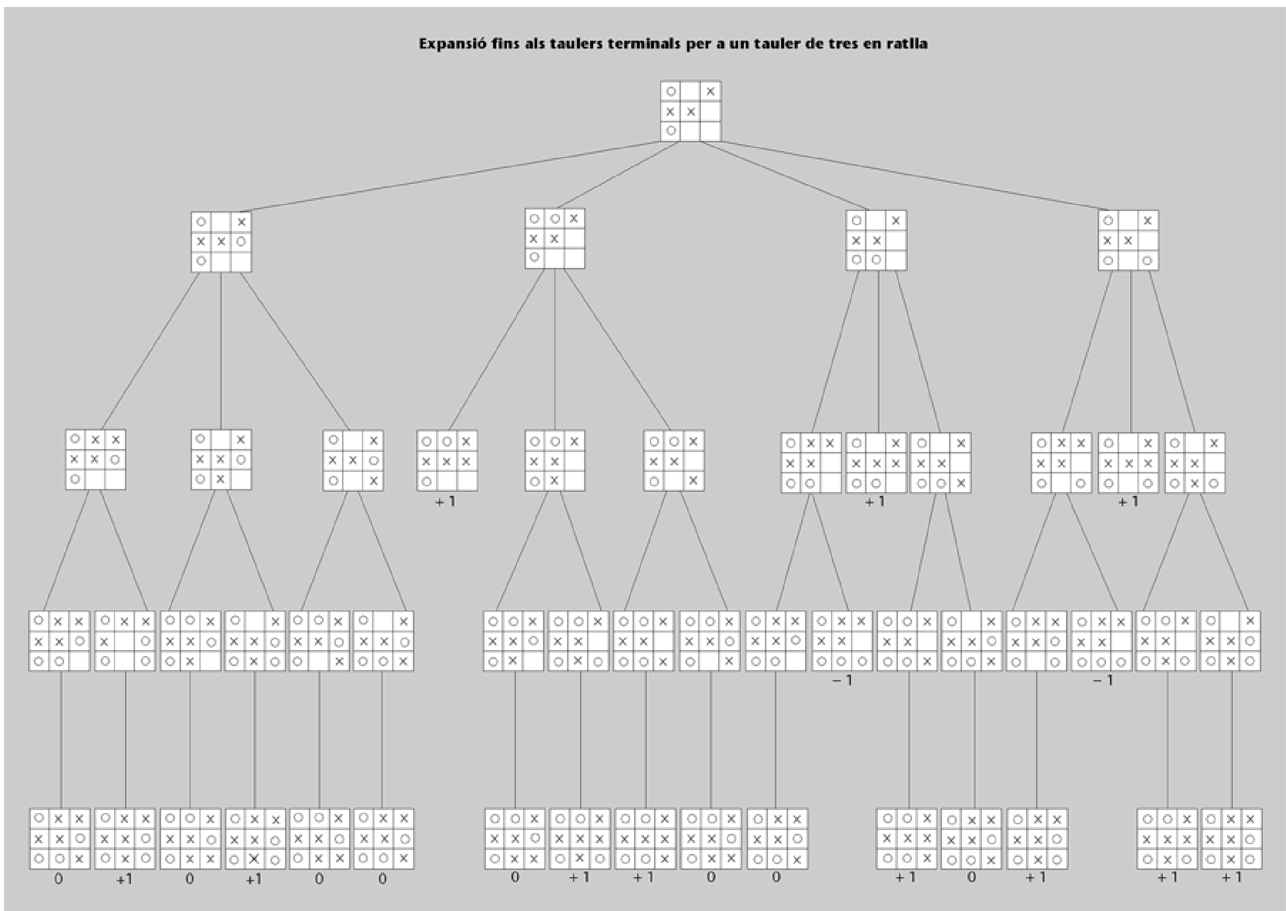
Desafortunadament, en aquest cas els nous taulers tampoc no són terminals i no podem aplicar la funció d'utilitat. L'alternativa és tornar a aplicar el mateix procediment: calcular els possibles moviments, avaluar-los i triar el millor (d'acord amb qui li toca tirar). La figura 25 mostra l'expansió dels darrers estats de la figura amb profunditat 2 i on es representen les alternatives del senyor *x*. Ara, per a cada tauler que expandim (d), (e), (f), (g), (h) i (i) només hi ha un moviment possible (és a dir, només hi ha una posició buida per a cada tauler) i, a més, els taulers que es generen són terminals. Per tant, els podem avaluar amb la funció d'utilitat. La figura mostra aquestes avaluacions: v(j) = v(l) = v(n) = v(o) = 0 perquè corresponen a taulers en què s'empata, i, en canvi, v(k) = v(m) = 1 perquè hi guanya el jugador *x*.

Un cop determinat el valor dels taulers obtinguts a profunditat 3, n'hem de calcular l'avaluació a profunditat 2. El jugador *x* (a qui li toca jugar a profunditat 2) ha de triar l'alternativa més bona (la del valor màxim) però, com que en aquest cas només n'hi ha una, l'avaluació dels taulers (d), (e), (f), (g), (h) i (i) correspondrà a les dels taulers (j), (k), (l), (m), (n) i (o).

Un cop avaluats els taulers de profunditat 2, avaluem els de profunditat 1. Els havíem deixat pendents perquè no es podia aplicar la funció d'utilitat als de profunditat 2. Com s'ha dit, v(a) = min(v(d), v(e)), v(b) = min(v(f), v(g)) i v(c) = min(v(h), v(i)) per a reflectir que considerem què farà el jugador *o* en cada cas: triar d'entre les diferents alternatives la que porta a un valor més petit. Ara bé, aquesta tria és significativa només per als taulers (a) i (b), perquè per al tauler (c) les dues opcions tenen la mateixa avaluació. En el tauler (a), el jugador triaria (d) a fi de no perdre (el tauler (d) està avaluat en 0 i el tauler (e), en 1). En el tauler (b) es triaria (f) per la mateixa raó (el tauler (f) està avaluat en 0 i (g) en 1).

Un cop tenim l'avaluació dels taulers (a), (b) i (c) el jugador *x* pot triar entre les tres alternatives que es plantejaven inicialment (profunditat 0). En aquest cas, i suposant que els dos jugadors juguen sempre el millor possible, es veu que el resultat final serà d'empat: en els tres casos s'arriba a l'empat. Per tant, qualsevol moviment és adequat.

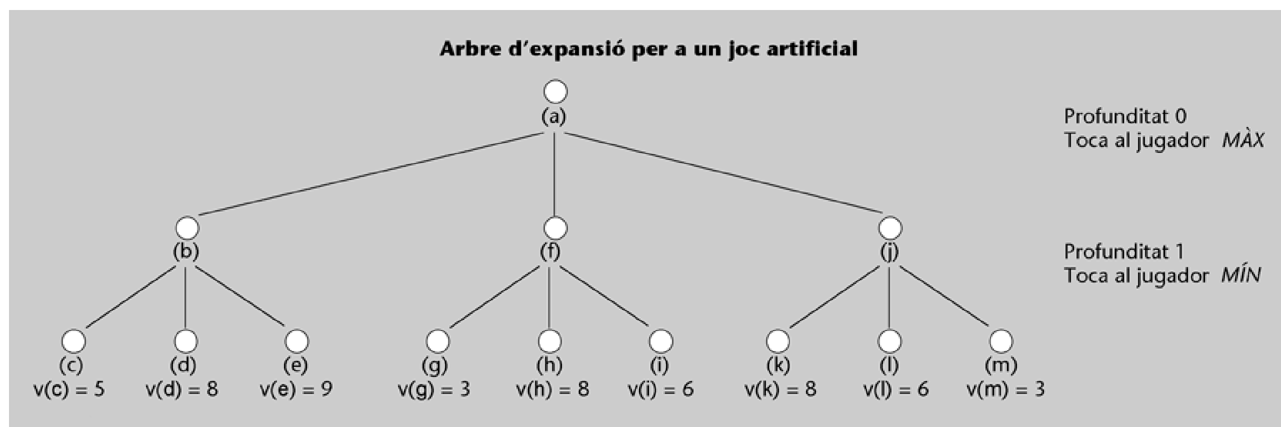
Figura 26



La figura 26 presenta l'arbre d'expansió per a un tauler similar al de la figura 22 però amb una fitxa menys. En aquest cas, es pot veure que l'avaluació de l'arbre permet triar al jugador *x* un moviment que li permet no perdre la partida. En aquest arbre, el tauler (a) és el de la figura 22 i, per tant, la seva avaluació correspon a l'avaluació que apareix en la figura que presenta l'expansió fins als taulers terminals (figura 25).

La figura 27 mostra l'arbre d'expansió corresponent a un altre joc. Es consideren dos nivells i la funció d'utilitat aplicada a tots els nodes del darrer nivell. Si suposem que el jugador a qui li toca moure en l'estat inicial (a) és el jugador MÀX, tindrem que amb l'esquema plantejat el jugador MÀX triarà com a millor opció la corresponent a l'estat (b).

Figura 27



La implementació

A continuació, fem la formalització de l'algorisme de Minimax. Aquest algorisme correspon al procés que s'ha seguit més amunt per a decidir el millor moviment per al tauler de la figura 22.

```
def decisio_minimax (toca, peca_min, peca_max, tauler):
    f_aux_max = lambda tf: [valor_minimax(peca_max, \
                                          peca_min, \
                                          peca_max, tf), tf]
    f_aux_min = lambda tf: [valor_minimax(peca_min, \
                                          peca_min, \
                                          peca_max, tf), tf]
    if guanya(peca_min, tauler) or guanya(peca_max, tauler) or ple(tauler):
        return funcioUtilitat(peca_min, peca_max, tauler)
    elif toca == peca_min:
        return cadr(selecciona_min(mapcar(f_aux_max, \
                                          opcions_tauler(peca_min, tauler))))
    else:
        return cadr(selecciona_max(mapcar(f_aux_min, \
                                          opcions_tauler(peca_max, tauler))))
```

La funció rep com a paràmetres un tauler, les dues peces que juguen (`peca_min` i `peca_max`) i la peça a què li toca jugar (`toca`).

Màxim i mínim

Les peces correspondran, respectivament, al màxim i al mínim, tot i que, de fet, quan es fa la crida això no afectarà el resultat. Donarà el mateix `decisio_minimax('o', 'o', 'x', tauler)` que `decisio_minimax('o', 'x', 'o', tauler`. Noteu que si fem `decisio_minimax('x', 'o', 'x', tauler)` amb el tauler `[['o', '_', 'x'], ['x', 'x', 'o'], ['o', '_', '_']]` –el cas de la figura 25– les avaluacions dels taulers són com apareixen en la figura. En aquest cas, quan es triï el moviment d'x, es triarà un tauler amb el valor més gran. En canvi, si la crida és `decisio_minimax 'x', 'x', 'o', tauler)` tindrem que les avaluacions dels taulers seran les mateixes però canviades de signe. Però en aquest cas, com que `'x'` és la peça del jugador mínim

(peca_min) i 'o' és la del jugador màxim (peca_max), a cada nivell s'intercanvien els papers i això fa que la tria sigui la mateixa. Tingueu en compte que la funció d'utilitat que es fa més endavant dona –1 per a la peça mínima amb independència de si és 'o' o 'x'.

La funció mirarà primer si es pot posar una peça i, si és així, generarà tots els possibles moviments –això ho fa `opcions_tauler(peca_min, tauler)`– i els avaluarà. Per a fer això darrer s'aplica a cada tauler fill `tf` la funció:

```
lambda tauler_fill: [valor_minimax(peca_max, peca_min, \
                                   peca_max, tauler_fill), tauler_fill]
```

que genera per a cada tauler un parell amb l'avaluació del tauler i després el tauler mateix. La funció `selecciona_min` (respectivament, la funció `selecciona_max`) retorna el millor tauler. Això és, el que té una avaluació més petita (respectivament, el que té una avaluació més gran).

L'avaluació dels taulers la fa la funció que s'anomena `valor_minimax`. Aquesta funció mirarà si amb el tauler que tenim ja podem determinar el valor (si algun jugador guanya o si el tauler és ple). Si és així, es retorna el valor i, en cas contrari, es consideren les diferents alternatives (s'avaluaran fent una crida recursiva a la mateixa funció `valor_minimax`) i després es tria la millor. La tria, com abans, la farà `selecciona_min` i `selecciona_max`.

```
def valor_minimax (toca, peca_min, peca_max, tauler):
    f_aux_max = lambda tf: [valor_minimax(peca_max, peca_min, peca_max, tf), tf]
    f_aux_min = lambda tf: [valor_minimax(peca_min, peca_min, peca_max, tf), tf]
    if guanya(peca_min, tauler) or guanya(peca_max, tauler) or ple(tauler):
        return funcioUtilitat(peca_min, peca_max, tauler)
    elif toca == peca_min:
        return car(selecciona_min(mapcar(f_aux_max, \
                                  opcions_tauler(peca_min, tauler))))
    else:
        return car(selecciona_max(mapcar(f_aux_min, \
                                  opcions_tauler(peca_max, tauler))))
```

Les funcions per a seleccionar el tauler mínim i màxim apareixen definides a continuació. Aquestes funcions prenen una llista de parells valor-tauler i retornen el parell que té un valor més petit o més gran (segons sigui `selecciona_min` o `selecciona_max`). Les dues funcions tenen un funcionament semblant, agafen els dos primers elements i miren quin és el millor. Si és el segon, fan una crida recursiva a la mateixa funció un cop eliminat el primer. `cdr(parells_v_t)` eliminarà el primer element. En canvi, si el millor element és el primer faran la crida eliminant el segon. De fet, això es fa

construint una nova llista en què el primer element és el que abans era primer, `car(parells_v_t)` i la resta de la llista són tots els que teníem a partir del tercer `cddr(parells_v_t)`. Així, la nova llista és:

```
cons(car(parells_v_t),cddr(parells_v_t))
```

La condició d'acabament de la funció és quan la llista només té un element (quan la cua de la llista és buida – `cdr(parells_v_t)==[]`. En aquest cas el primer (i únic) element és la solució.

```
def selecciona_min (parells_v_t):
    if cdr(parells_v_t) == []:
        return car(parells_v_t)
    elif caar(parells_v_t) > caadr(parells_v_t):
        return selecciona_min(cdr(parells_v_t))
    else:
        return selecciona_min(cons(car(parells_v_t),cddr(parells_v_t))

def selecciona_max (parells_v_t):
    if cdr(parells_v_t) == []:
        return car(parells_v_t)
    elif caar(parells_v_t) < caadr(parells_v_t):
        return selecciona_max(cdr(parells_v_t))
    else:
        return selecciona_max(cons(car(parells_v_t),cddr(parells_v_t)))
```

Les funcions que hem explicat abans són generals per a implementar el minimax per a qualsevol joc. Ara indiquem les pròpies del tres en ratlla. Comencem amb la funció d'utilitat, la que mira si el tauler està ple i la de generar tots els possibles moviments per a un tauler donat. Per a fer aquestes funcions, hem de definir l'estructura de dades del tauler. La representació del tauler serà una llista en què cada element correspondrà a una fila. La representació de la fila també utilitzarà una llista. Així, tindrem que un tauler és una llista de llistes. Les posicions buides les representarem amb el símbol `'_'`. Per tant, el tauler de la figura inicial es representarà:

```
[['o', '_', 'x'], ['x', 'x', 'o'], ['o', '_', '_']]
```

Passem, ara, a la definició de les funcions:

```
def funcioUtilitat (peca_min, peca_max, tauler):
    if guanya(peca_min, tauler):
        return -1
    elif guanya(peca_max,tauler):
```

```
        return 1
    elif ple(tauler):
        return 0
    else: return None

def ple (tauler):
    return ('_' not in car(tauler)) and \
        ('_' not in cadr(tauler)) and \
        ('_' not in caddr(tauler))

def guanya (peca, tauler):
    return alguna_columna(peca,tauler) or \
        alguna_diagonal(peca,tauler) or \
        alguna_fila(peca,tauler)

def alguna_diagonal (peca, tauler):
    diag1 = (peca == car(car(tauler))) and \
            (peca == cadr(cadr(tauler))) and \
            (peca == caddr(caddr(tauler)))
    diag2 = (peca == caddr(car(tauler))) and \
            (peca == cadr(cadr(tauler))) and \
            (peca == car(caddr(tauler)))
    return diag1 or diag2

def alguna_fila (peca, tauler):
    fila1 = (peca == car(car(tauler))) and \
            (peca == cadr(car(tauler))) and \
            (peca == caddr(car(tauler)))
    fila2 = (peca == car(cadr(tauler))) and \
            (peca == cadr(cadr(tauler))) and \
            (peca == caddr(cadr(tauler)))
    fila3 = (peca == car(caddr(tauler))) and \
            (peca == cadr(caddr(tauler))) and \
            (peca == caddr(caddr(tauler)))
    return fila1 or fila2 or fila3

def alguna_columna (peca, tauler):
    col1 = (peca == car(car(tauler))) and \
           (peca == car(cadr(tauler))) and \
           (peca == car(caddr(tauler)))
    col2 = (peca == cadr(car(tauler))) and \
           (peca == cadr(cadr(tauler))) and \
           (peca == cadr(caddr(tauler)))
    col3 = (peca == caddr(car(tauler))) and \
           (peca == caddr(cadr(tauler))) and \
           (peca == caddr(caddr(tauler)))
    return col1 or col2 or col3
```

```
## Donat un tauler, genera tots els moviments possibles.
def opcions_tauler (peca, tauler):
    if tauler == []:
        return []
    caps = opcions_cap_tauler(peca, car(tauler))
    faux1 = lambda la_resta: cons(car(tauler), la_resta)
    faux2 = lambda un_cap: cons(un_cap, cdr(tauler))
    if caps == []:
        return mapcar(faux1,opcions_tauler(peca, cdr(tauler)))
    else:
        return mapcar(faux2, caps) + mapcar(faux1, opcions_tauler(peca, cdr(tauler)))

def opcions_cap_tauler (peca, cap_tauler):
    aux = lambda x: cons(car(cap_tauler), x)
    if cap_tauler == []:
        return []
    elif car(cap_tauler) == '_':
        llaux = mapcar(aux, opcions_cap_tauler(peca,cdr(cap_tauler)))
        return cons(cons(peca,cdr(cap_tauler)),llaux)
    else:
        llaux = mapcar(aux, opcions_cap_tauler(peca,cdr(cap_tauler)))
        return llaux
```

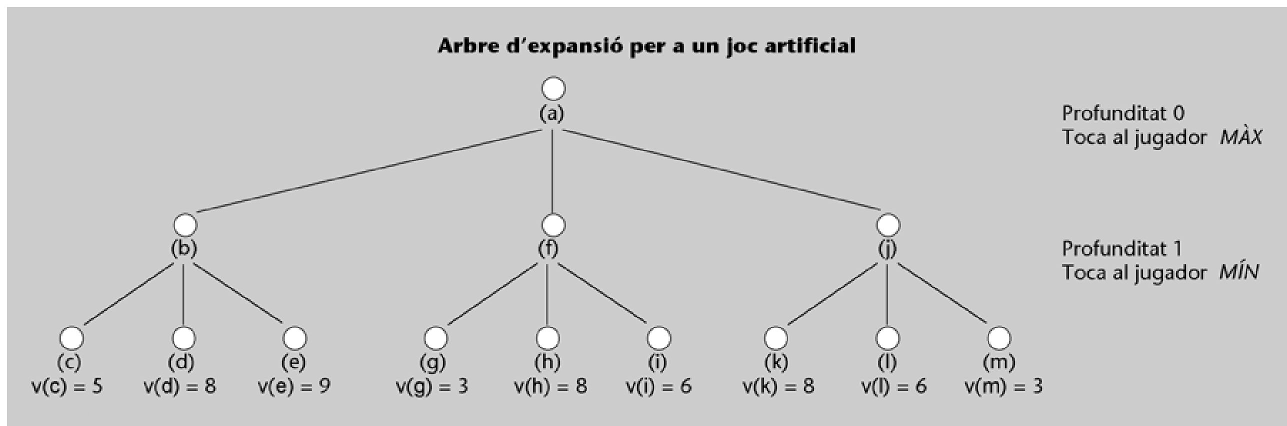
5.1.1. La poda α-β

Si fem una anàlisi acurada de l'exemple de l'arbre d'expansió per al joc artificial de la figura 27, trobarem que hi ha nodes de l'arbre que no caldria expandir perquè abans de visitar-los ja sabem del cert que mai no seran seleccionats. Quan ocorri aquest cas, l'algorisme de poda α-β no avaluarà les branques estalviant-nos la corresponent expansió de l'arbre.

Procediment minimax

Abans de passar a veure l'algorisme resseguim el procediment de minimax aplicat en la figura 27. Recordem aquesta figura:

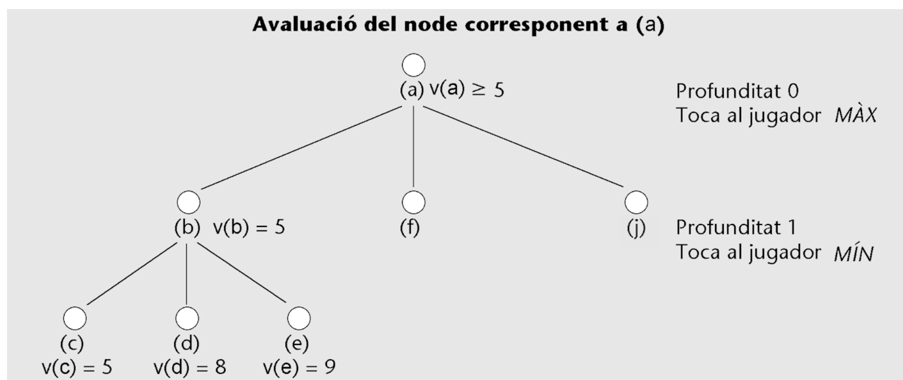
Figura 27



Si comencem l'avaluació de l'arbre (considerarem l'avaluació dels nodes d'esquerra a dreta) ens trobarem que quan expandim el node (a) obtenim els nodes (b), (f) i (j). Per a avaluar (b), l'hem d'expansionar i assignar a (b) el mínim dels valors dels tres fills. Així, v(b) = min(v(c), v(d), v(e)). Per tant, v(b) = 5.

Abans de passar a l'avaluació del node (f) podem observar que com que en el node (a) el jugador que ha de moure és MÀX, l'avaluació de (a) no serà mai més petita que 5, perquè si els nodes (f) i (j) s'avaluessin amb un valor més petit que 5, quan decidirà el moviment del node (a), MÀX triarà l'opció (b) en lloc d'(f) o (j). Posem v(a) ≥ 5 en el node (a) per a indicar això. La figura 28 representa aquesta situació.

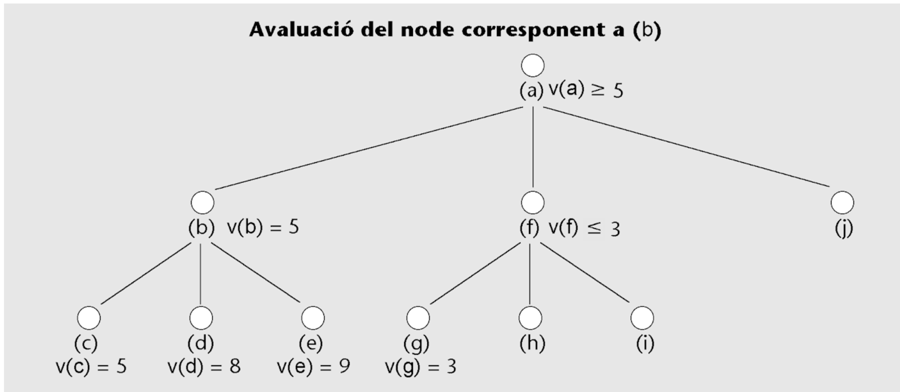
Figura 28. Avaluació del node (a)



Arbre d'expansió, avaluació del node corresponent a (b). Fita inferior per a l'avaluació d'(a).

Passem, ara, a avaluar el node (f). En aquest cas, comencem amb l'avaluació del node (g). Com que la seva avaluació és 3, podem afirmar que l'avaluació d'(f) serà més petita o igual que 3 ((f) és un estat que correspon al jugador MÍN). Per tant, podem escriure v(f) ≤ 3 (la figura 29 correspon a aquest nou pas).

Figura 29. Avaluació del node (b)



Fita inferior per a l'avaluació d'(a) i superior per a *v*(f).

Noteu que:

- si l'avaluació del node (h) és **més petita** que la de (g), llavors el jugador MÍN quan estigui a l'estat (f) triarà (h) en lloc de (g).

- si l'avaluació del node (h) és **més gran** que la de (g), llavors el jugador MÍN preferirà (g).

Poda de l'exemple de l'arbre d'expansió per al joc artificial

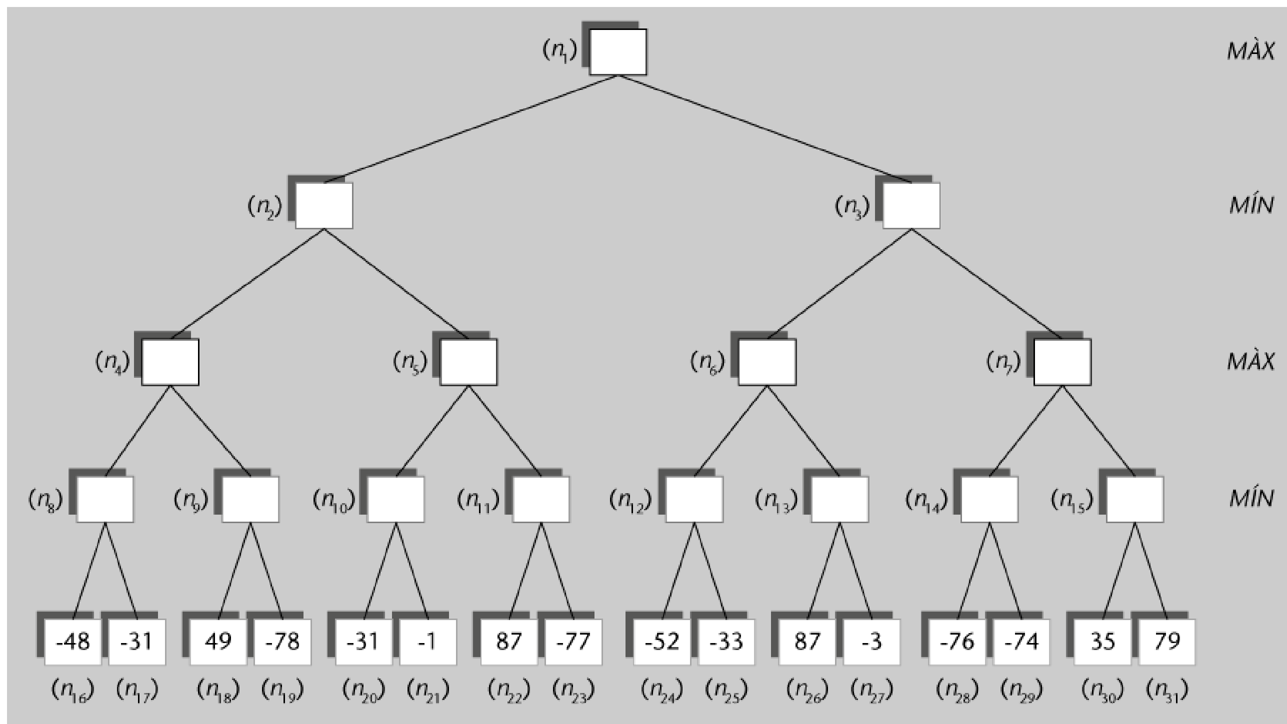
Un cop sabem que l'avaluació d'(f) és menor que 3, com que sabem que la del node (a) és més gran que 5 i que (a) és un node que correspon al jugador MÀX, també sabem que el jugador MÀX quan estigui al node (a) no triarà mai (f). Siguin quines siguin les avaluacions d'(h) i d'(i), l'avaluació d'(f) sempre serà més petita o igual que 3 i, per tant, el jugador MÀX triarà sempre (b) abans que (f). Per tant, no cal avaluar els nodes (h) i (i) perquè la seva avaluació és irrellevant a l'hora de prendre la decisió. Al fet de no considerar els nodes (h) i (i) s'anomena poda. En el cas de l'avaluació del node (j), comencem fent l'avaluació de (k). En aquest cas, trobem que és 8 i, per tant, $v(j) \leq 8$. Això, però, no ens permet podar perquè 8 és més gran que la fita que tenim ara per a *v*(a). Per tant, si es confirmés aquest valor, el jugador MÀX preferiria (j) a (b). A continuació, avaluem el node de (l) i ens dona 6. Tot i que el valor és menor que 8 –i, per tant, tindrem una modificació de la fita perquè el jugador MÍN preferirà (l) a (k) –aquest valor no ens permet podar l'arbre perquè aquesta branca és l'alternativa més bona en aquest moment per al jugador MÀX.

Seguidament, avaluem l'estat (m) i això ens retorna 3. Aquest valor deixarà $v(j) \leq 3$ i permetria –si hi haguessin altres nodes germans d'(m)– fer una poda. Aquest no és el cas i, per tant, no podem podar.

Exemple de minimax amb algorisme de poda α-β

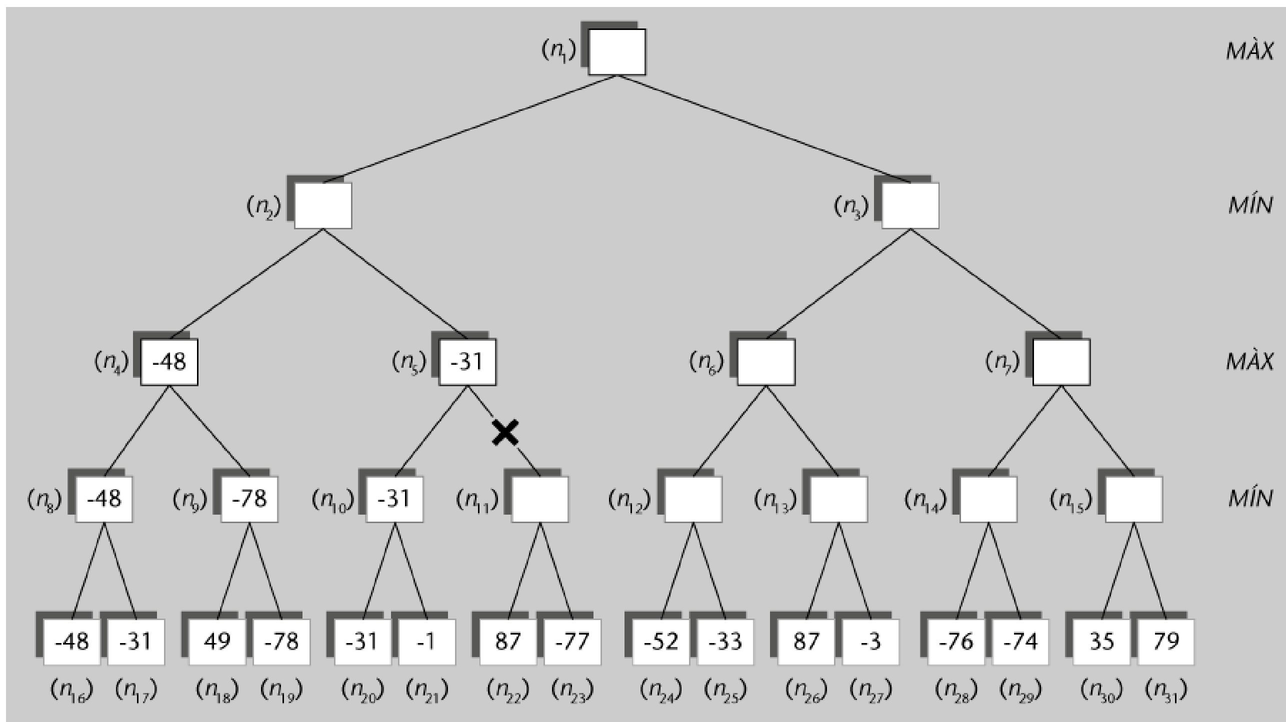
A continuació, veiem un altre exemple d'aplicació de l'algorisme minimax amb l'algorisme de poda α-β en l'exemple d'un joc artificial representat en la figura 30.

Figura 30



L'arbre es va resolent d'esquerra a dreta i de baix cap a dalt. Per exemple, en el primer pas, el valor del node (n_8) es tria entre el mínim dels valors dels nodes (n_{16}) i (n_{17}). Després, per al node (n_9) s'avalua primer el node (n_{18}) que té valor 49. Com que aquest valor és superior al valor assignat al node (n_8), el node (n_9) també explora el node (n_{19}) per si troba un valor més petit. Per tant, el node ($n_1$9) no és esporgat i, de fet, com que es tracta d'un valor més petit és assignat al node (n_9). A continuació, s'assigna a (n_4) el valor màxim dels nodes (n_8) i (n_9). Se segueix amb l'avaluació del node (n_{10}), el qual primer explora el node (n_{20}) amb valor -31. Com que -31 és més gran que -48 (el valor de (n_4)), l'algorisme també explora el node (n_{21}) per si troba un valor millor. No és el cas, així que s'acaba assignant el valor -31 al node (n_{10}). En aquest punt és quan trobem el primer moment en què es pot esporgar una branca de l'arbre, tal com es veu en la figura 31. Quan s'avalua el node (n_5), com que es tracta d'un node MÀX, aquest tindrà un valor a decidir entre el node (n_{10}) amb valor -31 i el node (n_{11}) amb un valor a explorar. Com que el node (n_2) és un node MÍN i el valor del node (n_4) és -48, el qual és més petit que -31, el node MÀX (n_5) no aconseguirà cap millora encara que el node (n_{11}) tingui un valor més gran que -31. Així doncs, el node (n_5) decideix esporgar la branca corresponent al node (n_{11}) i assignar el valor del node (n_{10}) al node (n_5).

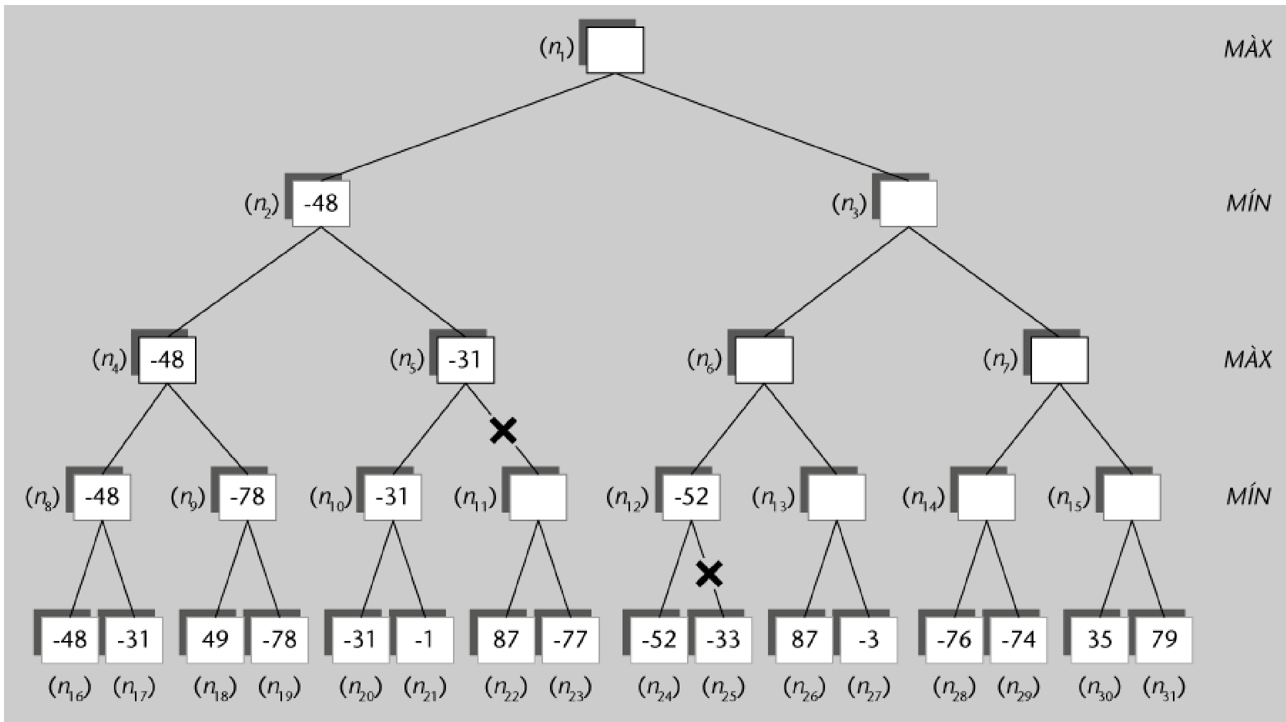
Figura 31



En les figures següents (figures 32, 33 i 34) només es visualitzen els passos en què es produeix alguna esporga i només s'explicarà el detall d'aquests passos.

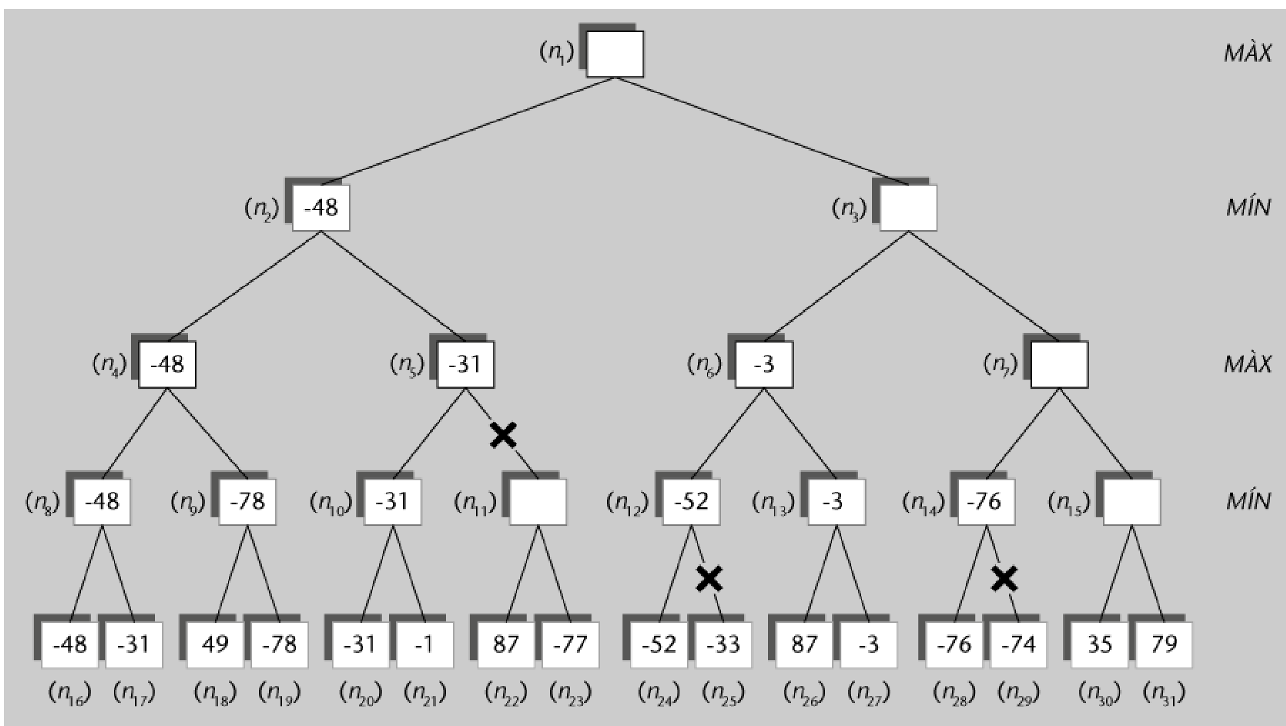
En la figura 32 es pot veure que, un cop el node (n_{12}) ha explorat el node (n_{24}), amb valor -52, es pot esporgar la branca corresponent al node (n_{25}) atès que el valor -52 ja és més petit que el valor que es té en el node (n_2) amb valor -48. Com que per la part esquerra de l'arbre, el millor valor que ha pogut pujar el jugador MÍN (amb l'oposició del jugador MÀX) ha estat -48 i a l'arrel hi ha el jugador MÀX, el jugador MÍN en té prou amb intentar pujar qualsevol valor que sigui inferior a -48 per la part dreta de l'arbre.

Figura 32



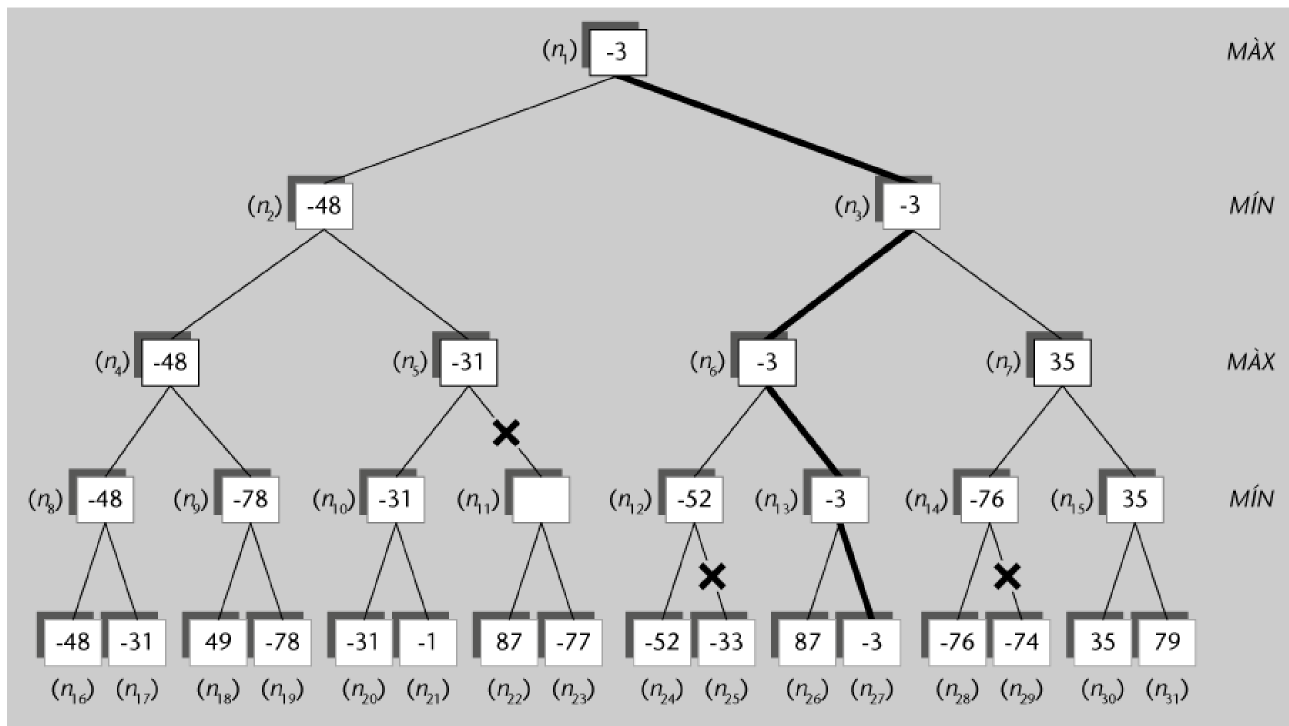
Més endavant, tal com es veu en la figura 33, i pel mateix motiu que anteriorment, el node (n_{14}) pot esporgar la branca corresponent al node (n_{29}) ja que quan ha avaluat el node (n_{28}) ha obtingut un valor (-76) que és més petit que el del node (n_2) amb valor -48.

Figura 33



Finalment, tal com es veuen la figura 34, no es produeix cap esporga més i s'acaben trobant les decisions òptimes que prendran cadascun dels jugadors (visualitzat en línia més gruixuda). Així doncs, en el primer torn el jugador MÀX triarà la branca dreta. A continuació, el jugador MÍN decidirà tirar la branca esquerra. Seguidament, el jugador MÀX escollirà la branca dreta. Finalment, el jugador MÍN optarà per la branca dreta.

Figura 34



La implementació

La implementació de l'algorisme de poda α-β es basa en dues constants `alfa` i `beta`. `alfa` correspon a una fita del millor que pot aconseguir el jugador MÀX i `beta` a una fita del millor que pot aconseguir MÍN. Per tant, `alfa` tendirà a créixer (el jugador MÀX canviarà la fita si pot aconseguir un valor més gran) i beta tendirà a decréixer (el jugador MÍN canviarà la fita si pot aconseguir un valor més petit).

Un jugador podarà una branca quan `alfa` ≥ `beta`. Això és així perquè el jugador MÀX no triarà mai un valor pitjor que aquell que ja té assegurat. Així, si hi ha un valor (`beta`) menor que `alfa`, al jugador MÀX no li interessa i poda. Per tant, si `beta` ≤ `alfa` es poda. Per altra banda, el jugador MÍN tampoc no triarà mai un valor pitjor que el que té assegurat. Així, si hi ha un valor (`alfa`) més gran que `beta`, no interessarà al jugador MÍN. Per tant, quan `alfa` ≥ `beta` es poda.

A continuació, indiquem l'algorisme de poda:

```
funcio poda-alfa-beta (node, alfa, beta) es
    si limit(cerca) llavors retorna valor(node)
    si nivell(node)=min llavors
        repetir fins (no quedin fills) o (alfa ≥ beta) fer
            r:=poda-alfa-beta(fill, alfa, beta)
            si r<beta llavors beta:=r
        fi repetir
        retorna beta
    sino ;; nivell(node)=max
        repetir fins (no quedin fills) o (alfa ≥ beta) fer
                r:=poda-alfa-beta(fill, alfa, beta)
                si r>alfa llavors alfa:=r
                fi repetir
            retorna alfa
            fsi
ffuncio
```

La crida a aquest procediment s'ha de fer assignant al node el corresponent a l'estat inicial, a `alfa` un valor molt petit (– infinit) i a `beta` un valor molt gran (infinit). Aquestes assignacions d'`alfa` i `beta` permetran que les fites es vagin modificant a mesura que s'expandeixen els nodes.

5.2. Decisions imperfectes

L'algorisme que hem presentat representa que cerquem tots els nodes fins a arribar als estats terminals. Això, però sempre no és possible. L'algorisme minimax té un cost exponencial ja que ha de recórrer tots els nodes de tots els nivells abans de prendre una decisió. Així, si la cerca es fa fins a una profunditat d i tenim un factor de ramificació de b, el minimax té un cost $O(b^d)$ –expandim tots els nodes fins a la profunditat d. Per tant, en la majoria dels casos no és possible expandir tot l'arbre. Aquí $b = 35$ i $d = 100$, per tant, s'haurien d'expandir de l'ordre de $35^{100} = 2{,}55 * 101^{54}$ nodes. De manera indicativa, en la taula que veurem hi ha els temps d'execució del tres en ratlla per a taulers amb diferent nombre de posicions buides. També es pot veure l'evolució exponencial en relació amb la profunditat a què ha d'arribar el minimax.

Escacs

Aquest és el cas dels escacs que s'ha comentat en la introducció.

La poda α-β redueix el nombre de nodes que es consideren, i, per tant, redueix el cost de l'algorisme. De fet, en general tenim que el seu cost en el cas ideal és $O(b^{d/2})$. Això permet reduir considerablement el temps d'execució.

Aquesta reducció correspon al cas ideal i necessita que els nodes estiguin ordenats de manera que sempre es pugui podar al màxim. En l'exemple que hem considerat en la figura 27, l'expansió dels nodes (f) i (j) dona 3 nodes que s'avaluen amb els mateixos valors (3, 8 i 6). Es pot observar que, mentre

que l'ordenació (3,8,6) permet podar, la de (8, 6, 3) no. De fet, en el cas del jugador MÍN ens interessarà trobar primer els valors més petits. Inversament, en el cas del jugador MÀX interessarà trobar primer els valors més grans. Desafortunadament, no és possible tenir *a priori* aquestes ordenacions perquè això representaria tenir els valors i això és, precisament, el que estem cercant.

Per a tractar el cas de tenir limitacions de temps i de memòria, es consideren les alternatives següents:

1) No desenvolupar tot l'arbre: aturar l'expansió dels nodes
2) Aplicar una funció d'avaluació heurística a les fulles de l'arbre.

La necessitat de la funció heurística apareix com un efecte secundari del fet de no expandir tot l'arbre. Com que no s'arriba als nodes terminals, no podem aplicar la funció d'utilitat perquè en els nodes intermedis de l'arbre no hi haurà una situació en què un dels dos jugadors guanyi o en què els dos jugadors estiguin empatats. Quan s'aturi el desenvolupament de l'arbre, trobarem nodes en què encara és possible fer moviments. La funció d'avaluació heurística donarà una estimació de la utilitat que es pot aconseguir a partir d'aquella posició.

Com que el rendiment del programa dependrà fortament de la funció herística, la funció haurà de reflectir les possibilitats de guanyar. Evidentment, la funció s'haurà de definir de manera que es pugui calcular de manera eficient i, a més, ha de coincidir en els nodes finals amb la funció d'utilitat. Una manera de definir aquestes funcions és que reflecteixi la probabilitat de guanyar per aquell estat.

Per a limitar la cerca, hi ha diverses alternatives. Una d'aquestes és fixar la profunditat màxima de manera que el temps no excedeixi el temps màxim disponible. Una altra alternativa és aplicar una variant de la cerca iterativa en profunditat de manera que quan el temps s'acaba es retorna la cerca més profunda completada. Es defineix el minimax amb profunditat limitada i, iterativament, es va recalculant cada vegada amb una profunditat més gran. Això correspon a un algorisme tot-temps (definit en el subapartat «Algunes consideracions addicionals»).

> El fet de fitar la cerca provoca que les decisions que es prenen no sempre són òptimes.

Cal dir que, quan hi ha restriccions de temps o memòria, també és possible aplicar la poda α-β. En aquest cas, a causa que el cost de l'algorisme és $O(b^{d/2})$ en lloc de $O(b^d)$ com en el minimax, tenim que per a un mateix temps, podrem explorar fins al doble de profunditat.

Taula 3. Taula de costos per a la funció de minimax aplicada al tres en ratlla

Tauler	Temps d'execució (s)	Nodes expandits
[['*', '*', '*'], ['*', '*', '*''], ['*', '*', '_']]	0.031	1
[['*', '*', '*'], ['*', '*', '*'], ['*', '_', '_']]	0,028	4
[['*', '*', '*''], ['*', '*', '*'], ['_', '_', '_']]	0,029	15
[['*', '*', '*'], ['*', '*', '_'], ['_', '_', '_']]	0,03	64
[['*', '*', '*'], ['*', '_', '_'], ['_', '_', '_']]	0,04	325
[['*', '*', '*'], ['_', '_', '_'], ['_', '_', '_']]	0.134	1884
[['*', '*', '_'], ['_', '_', '_'], ['_', '_', '_']]	0.428	12043
[['*', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]	2.2995	84040
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]	19.139	549945

Les posicions ocupades s'han indicat amb un asterisc (per a evitar que les posicions plenes provoquin tres en ratlla).

Procediment de càlcul de la taula

Aquesta taula s'ha calculat executant amb Linux `time python3 <fitxer.py>`, on el fitxer tenia una crida a `decisio_minimax('o', 'x', 'o', tauler)`, per a cadascun dels taulers indicats, amb un fitxer diferent per tauler. Per a saber la quantitat de nodes expandits, s'ha utilitzat una variable global `nodes_expandits` que s'incrementa per a cada node que es visita:

```
def valor_minimax (toca, peca_min, peca_max, tauler):
    global nodes_expandits
    nodes_expandits += 1
    f_aux_max = lambda tf: [valor_minimax(peca_max, peca_min, peca_max, tf), tf]
    f_aux_min = lambda tf: [valor_minimax(peca_min, peca_min, peca_max, tf), tf]
    ...
```

5.3. Jocs amb elements d'atzar

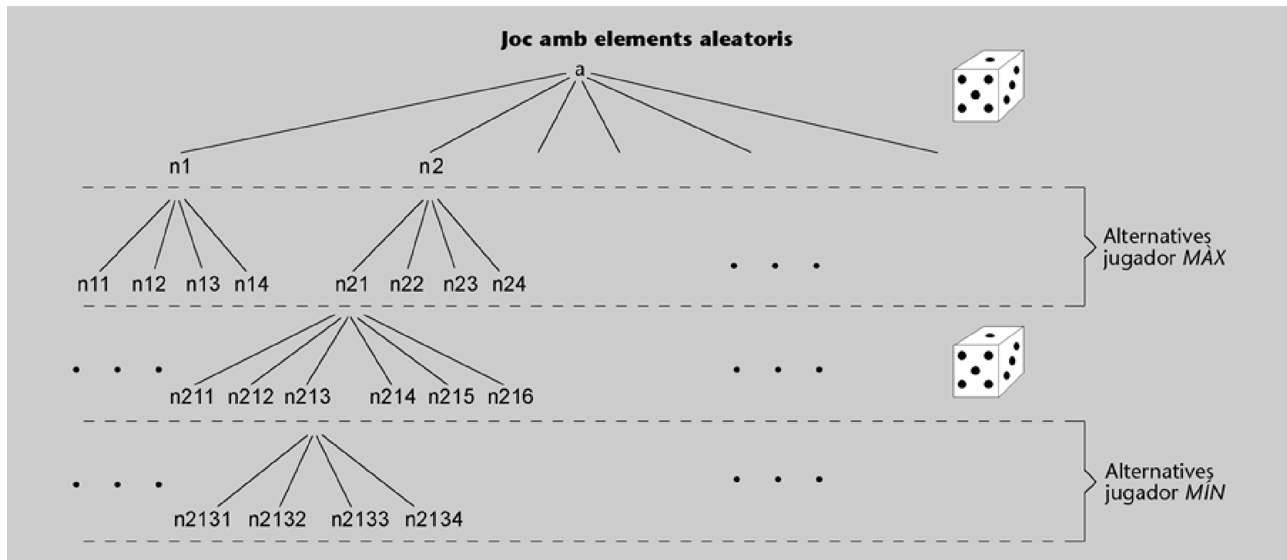
Quan en una partida s'ha de considerar un element aleatori (per exemple, un dau), el desenvolupament de l'arbre de cerca ha de tenir en compte aquest element. Així, cada vegada que li toca moure a un jugador, haurem de considerar les diverses possibilitats de l'element aleatori i, per a cadascuna d'aquestes, desenvolupar l'arbre corresponent. Un cop es disposa de les avaluacions de tots els fills, l'avaluació del node s'aconsegueix fent la mitjana de les avaluacions dels fills tenint en compte la probabilitat de cada component aleatori. D'acord amb això, el desenvolupament de l'arbre de cerca correspondrà a una successió de nodes corresponents a mínims, element aleatori, màxim, element aleatori, etc.

Exemple de joc amb element aleatori

Un exemple d'aquest procés es presenta en la figura 30. Considerem un problema en què abans de moure un jugador s'ha de tirar el dau i segons el que surt, es podrà fer un moviment o un altre. Estimem que, en cada posició del

joc, es poden fer quatre moviments. Així, en el cas de trobar-nos en la situació representada en el node a, tenim que el dau ens pot oferir sis alternatives diferents corresponents als sis valors que poden sortir. Per a cada puntuació del dau, es poden fer quatre moviments. Així, si en el dau sortís un 1, ens trobaríem en el node n_1, i els quatre moviments possibles ens porten a n_{11}, n_{12}, n_{13} i n_{14}, respectivament. La figura també mostra els casos en què en el dau hi surt 2, 3, ..., 6. Per a cadascun d'aquests nodes s'aplicarà el mateix procés.

Figura 35



Per a poder fer l'avaluació dels nodes, hem de tenir en compte, per un costat, quan un jugador correspon al jugador mínim o al màxim. A més, però, haurem d'incloure l'element de sort. Com s'ha dit, això es fa definint el valor esperat com la mitjana de les avaluacions dels nodes fills. Així, si el node "a" correspon al jugador màxim, tindrem que l'avaluació d'n_1 serà el màxim de les avaluacions d'n_{11}, n_{12}, n_{13} i n_{14}; l'avaluació d'n_2 serà el màxim de les avaluacions d'n_{21}, n_{22}, n_{23} i n_{24}; i així successivament pels altres. Usant aquestes avaluacions tindrem que l'avaluació d'"a" serà el valor esperat de les avaluacions dels nodes n_1, n_2, ..., n_6. Com que la probabilitat que surti el valor i $\in$ {1, 2, 3, 4, 5, 6} quan es tira el dau sempre és 1/6, l'avaluació del node a serà $1/6v(n_1)+ 1/6v(n_2) + 1/6v(n_3) + 1/6v(n_4) + 1/6v(n_5) + 1/6v(n_6)$ on $v(n)$ correspon a l'avaluació del node n. Un cop sabem com avaluar un node a partir de les avaluacions dels nodes fills, el procés de propagació de les avaluacions i de selecció del millor moviment és com en el cas del minimax.

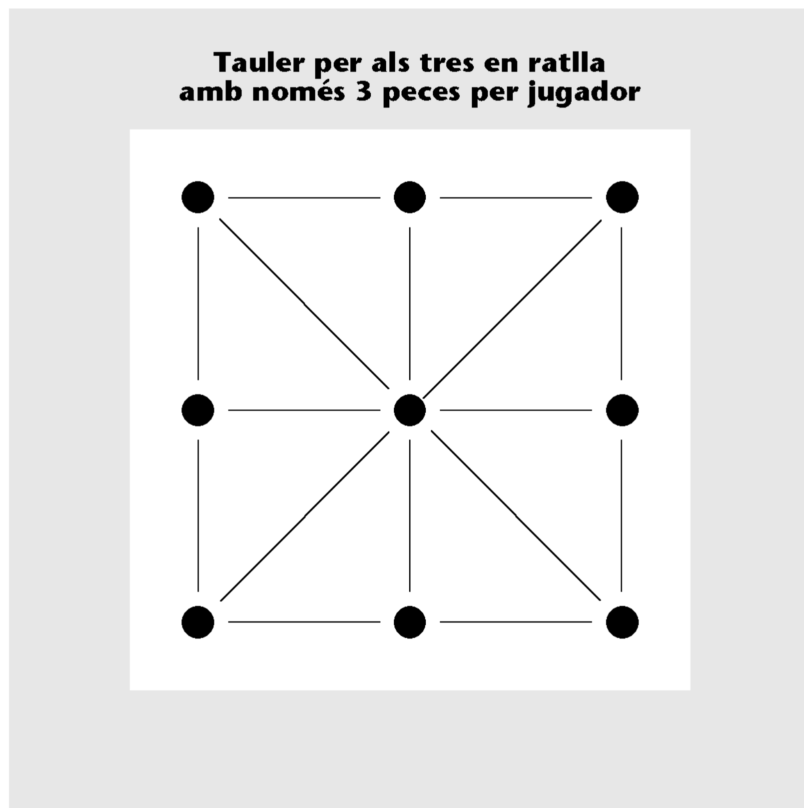
Activitats

1. Refeu la cerca del cost uniforme amb el mateix exemple considerat en el subapartat «Cerca de cost uniforme», però ara permetent que el fill d'un node tingui el mateix estat que el seu pare.

2. Utilitzant el mapa de carreteres per al problema del camí mínim (subapartat «Cerca de cost uniforme») i la taula d'heurístiques (subapartat «Cerca amb funció heurística: cerca àvida»), determineu el camí mínim de Vallmoll a Falset amb l'algorisme A*.

3. Considereu el joc de 2 jugadors següent: Hi ha 23 llumins a sobre la taula i, per torns, cadascun dels jugadors agafa 1, 2, 3, 4 o 5 llumins. Perd el que agafa el darrer llumí (quan un n'agafa com a mínim n'ha de quedar 1 a sobre la taula). Penseu com faríem el programa per tal que donats n llumins decideixi quants se n'ha d'agafar.

4. Considereu i implementeu el joc del tres en ratlla quan els jugadors tenen en tot moment 3 peces i es tracta de desplaçar-les en el tauler. El tauler és de la forma de la figura i una peça només es pot desplaçar d'una posició (un dels punts marcats en el dibuix) a una altra si hi ha un arc que uneixi les dues posicions.



Exercicis d'autoavaluació

1. Considereu el problema següent:

Donat un laberint, trobeu la seqüència de moviments que permeten anar de l'entrada a la sortida. Responeu les qüestions següents:

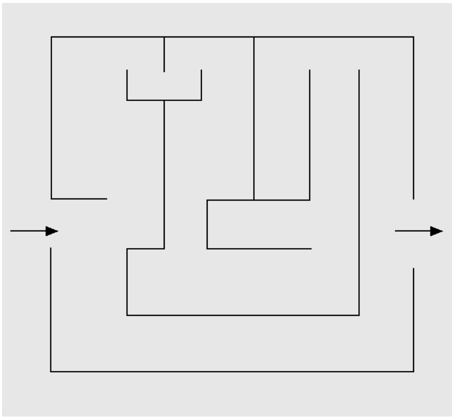
a) Formuleu el problema de manera que pugui ser resolt mitjançant els mètodes de cerca.

b) D'acord amb la formulació, descriviu l'espai d'estats del problema.

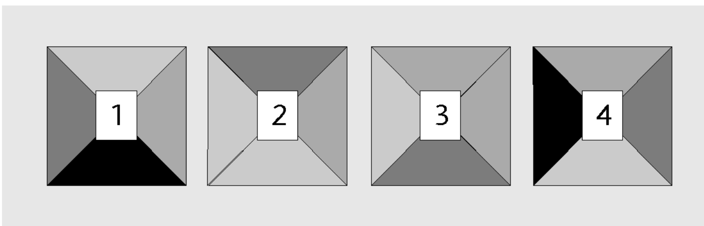
c) Mostreu com s'aplica aquesta formulació al laberint de la figura.

d) Quins mètodes de cerca podem utilitzar per a resoldre aquest problema?

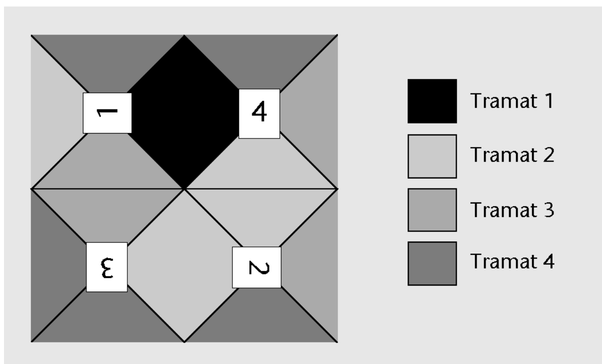
e) Si volem que el nombre de posicions del laberint travessades sigui mínim, podem aplicar l'algorisme A*? Indiqueu una funció heurística admissible per a aquest problema.



2. Considereu el problema de posar les quatre peces donades en la figura formant un quadrat. S'ha de complir que les trames de dues peces que es toquen han de ser iguals.



Una solució d'aquest problema és la de la figura següent. Fixeu-vos que en aquesta solució només apareixen les peces que hi havia a la figura de les quatre peces. Per tant, el problema consisteix a saber quina peça s'ha de posar en quina posició i l'orientació de la peça.



Contesteu les preguntes següents:

a) Formuleu el problema de manera que pugui ser resolt mitjançant els mètodes de cerca. Quin és el factor de ramificació de la vostra formulació?

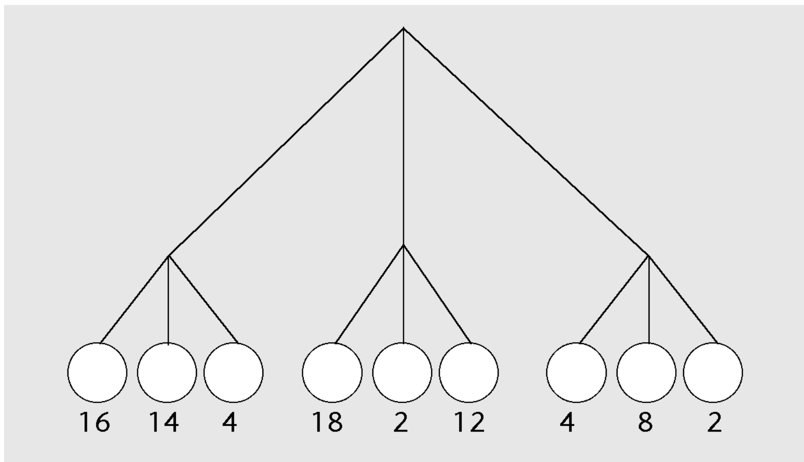
b) Quins algorismes de cerca podem aplicar per a resoldre aquest problema?

c) Amb la representació triada hi ha una solució amb el mètode de cerca en amplada? I amb la cerca en profunditat? I amb profunditat limitada? I en cerca iterativa amb profunditat limitada?

d) Es pot formular el problema com un problema de satisfacció de restriccions?

e) Es poden aplicar els algorismes genètics per a resoldre aquest problema?

3. Considereu com fer la poda α-β en l'arbre de la figura. Els nombres a sota dels nodes de l'arbre corresponen a la funció d'avaluació a les fulles. Suposem que el nivell més alt de l'arbre és de nivell MÀX i en el nivell inferior és de nivell MÍN. Aleshores, quins nodes es podaran?


16
14
4
18
2
12
4
8
2

Solucionari

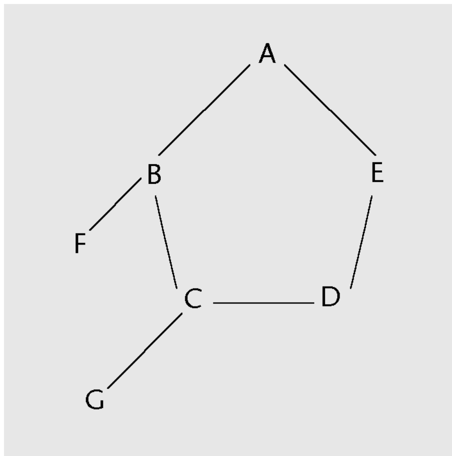
Exercicis d'autoavaluació

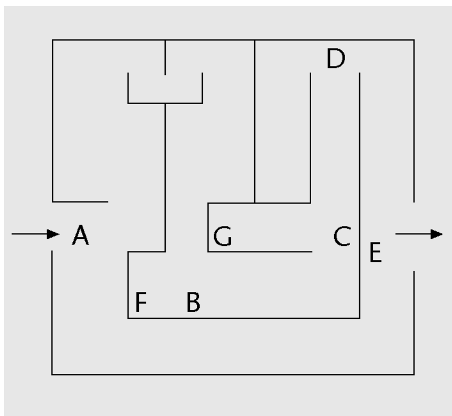
1. Les respostes a les qüestions corresponents al problema del laberint són aquestes:

a) Per a formular el problema hem de modelitzar l'entorn en què es mou el sistema, les seves accions i definir el problema.

- La modelització de l'entorn es farà representant el laberint en forma de graf de manera que a cada cruïlla (lloc on hi ha diversos camins) se li associa un node (també representem els camins cul de sac amb node), i les arestes del graf seran els camins que connecten dues cruïlles. Hem de subratllar que amb aquesta representació una vegada hem triat un camí, l'haurem de seguir fins a la propera cruïlla. D'aquesta manera, només tenim les diferents cruïlles com a estats del problema (vegeu la figura).
- Les accions que considerem són les de prendre un determinat camí que surt d'una cruïlla. Per tant, en una cruïlla podrem considerar tantes accions com camins hi surtin. Una acció ens porta d'un estat a un altre (d'una cruïlla a una altra).
- El problema és definit per la situació inicial (l'entrada al laberint) i la situació final (la sortida del laberint). A aquestes posicions se'ls hi assigna un node del graf que representa el problema (si és que no són cruïlles).

b) La formulació presentada fa que l'espai d'estats correspongui al graf definit més amunt en què els nodes corresponen a les cruïlles i les arestes als camins entre cruïlles. Trobar una solució correspon a trobar un camí entre el node que representa l'entrada i el que representa la sortida.





c) En el cas del laberint, el graf d'estats serà el de la figura anterior on A, B, ... són les cruïlles de la figura del laberint. En la figura següent se sobreposa el nom de les cruïlles al laberint.

d) Noteu que amb la formulació plantejada el problema és equivalent al dels camins mínims.

Per tant, per a resoldre aquest problema podem aplicar tots els mecanismes de cerca explicats en els apartats «Estratègies de cerca no informada» i «Cost i funció heurística» d'aquest mòdul. Hem de tenir en compte que a causa que hi pot haver cicles en el graf, la cerca en profunditat pot donar problemes.

e) És possible aplicar l'algorisme A* en aquest problema. Si volem que el nombre de posicions que la solució travessa siguin mínimes, hem d'associar a cada arc del graf el nombre de posicions travessades. Aquest valor serà el cost de cada aresta. En aquest problema, una funció heurística admissible serà, com en el cas del problema de camins mínims, la de la distància en línia recta a la solució (mesurant la distància com a posicions travessades).

2. Les respostes als apartats del segon exercici són aquestes:

a) Per a representar un estat podem suposar que sempre tenim les quatre peces posades formant el quadrat però que no sempre les peces adjacents tenen trames coincidents. A més, cada peça tindrà la seva orientació. Així, la figura que presenta una possible solució del problema la representarem dient quina peça està en aquella posició repassant les posicions d'esquerra a dreta i de dalt a baix i quantes rotacions de 90 graus ha fet cada peça. Així, tindrem que la situació de les peces correspondrà a la llista (1 4 3 2) i que les rotacions de les peces seran (3 1 2 0). Això darrer vol dir que la peça 1 s'ha rotat 3 vegades, la peça 2 s'ha rotat un cop, la peça 3 s'ha rotat dos cops i la peça 4 no s'ha rotat cap vegada.

Al nostre sistema li permetrem dos tipus d'accions: (1) intercanviar dues peces qualssevol i (2) rotar una peça 90 graus en el sentit de les agulles del rellotge. Així, d'operacions d'intercanvi en tindrem 6 (intercanviar la posició 1 amb la 2 o la 3 o la 4; intercanviar la posició 2 amb la 3 o la 4; intercanviar la 3 amb la 4) i de rotacions 4 (podem rotar la peça situada en la primera posició, la situada en la segona, la situada en la tercera i la situada en la quarta posició). D'acord amb això el factor de ramificació serà 10.

Per a formalitzar el problema podem considerar l'estat inicial com un estat en què hi ha les quatre peces amb una orientació. Per simplicitat, podem considerar la ubicació (1 2 3 4) i l'orientació (0 0 0 0). La funció objectiu serà comprovar si les trames adjacents són iguals.

Hem de subratllar que es podrien utilitzar altres representacions (per exemple, no sempre considerar les quatre peces ja posades formant el quadrats sinó que se'n tenen unes de posades i d'altres no) i que unes altres representacions ens portarien a unes altres accions (per exemple, accions per a posar o treure peces). De fet, la representació triada també permet altres tipus d'accions. En particular, podríem considerar rotacions en el sentit contrari a les agulles del rellotge.

b) La formulació anterior ens permet aplicar els algorismes de cerca no informada. La definició d'una funció de cost i una funció heurística ens permetrà aplicar algorismes de cerca informada. Una funció de cost pot ser el nombre de moviments.

c) La cerca en amplada trobarà la solució amb un menor nombre d'accions (siguin quines siguin les accions permeses). La cerca en profunditat no retornarà cap solució perquè cada vegada aplicarà la mateixa acció (es quedarà penjada) (llevat que controlem els estats repetits). La cerca en profunditat limitada només trobarà una solució si el límit és més gran que el nombre d'accions necessàries per a construir la solució. La cerca iterativa en profunditat també trobarà la solució amb menys nombre d'accions.

d) Sí que es pot formular el problema com un problema de satisfacció de restriccions. Una manera de fer-ho és considerar vuit variables, quatre corresponents a les peces que hi ha en les quatre posicions del tauler (per exemple, pos_1, pos_2, pos_3 i pos_4), i quatre posicions corresponents a la rotació de cada peça ($rotp_1$, $rotp_2$, $rotp_3$ i $rotp_4$). Així, la solució de la figura que presenta una solució del problema es pot representar com: $pos_1 = 1$, $pos_2 = 4$, $pos_3 = 3$ i $pos_4 = 2$. $rotp_1 = 3$, $rotp_2 = 1$, $rotp_3 = 2$, $rotp_4 = 0$. Les restriccions seran que una peça no pot estar en dues posicions a la vegada ($pos_i \neq pos_j$ quan $i \neq j$) i que els valors possibles per a les variables posi són 1, 2, 3 i 4 i els de les variables rotpi són 0, 1, 2 i 3. A més, s'han de considerar restriccions de les variables per tal que les trames adjacents siguin iguals.

3. En el fill de l'esquerra no es podrà podar res. L'avaluació dels seus tres fills (els nodes que quan s'avaluen donen 16, 14 i 4) donarà com a resultat el valor més petit de tots. Així que retornarà 4. Aquesta tria es fa tenint en compte que el node és de MÍN.

Quan considerem el segon fill, tindrem que el procediment podarà quan avaluant el segon node tenim que aquest és 2 i, per tant, menor que el valor de 4 que havíem obtingut en el primer fill.

El tercer fill no podrà podar perquè fins que no trobi un valor menor que el 4 del primer fill no pot podar. El tercer fill del tercer fill (el que té un valor de 2) permetria podar si hi hagués més fills després del darrer node, però com que no és el cas, no es podarà res.

Glossari

admissibilitat *f* Dit de la funció heurística que és admissible si mai no sobreestima el cost.

arbre de cerca *m* Representació en forma d'arbre del procés de cerca en un espai d'estats.

espai d'estats *m* Conjunt d'estats possibles i les seves relacions.

estat *m* Situacions possibles amb què es pot trobar un sistema.

expansió d'un node *m* Aplicació de l'estat associat a un node a tots els operadors de què disposem.

factor de ramificació *m* Nombre d'accions que es poden aplicar en els estats d'un problema.

frontera *f* Conjunt de nodes d'un arbre de cerca que encara no han estat expandits.

funció heurística *f* Funció que, aplicada a un node, estima el cost del millor camí entre aquest node i un estat solució.

Bibliografia

Bibliografia bàsica

Russell, S.; Norvig, P. (1995). *Artificial Intelligence: A modern approach*. Prentice-Hall.

Bibliografia complementària

Pearl, J. (1984). *Heuristics*. Addison-Wesley.

Annex

El codi d'aquest mòdul està fet amb Python i hem intentat simplificar al màxim el codi utilitzat. Hem utilitzat les llistes de Python com a estructura de dades fonamental i, per a poder treballar amb aquestes llistes d'una manera encara més simple, hem reproduït la manera de construir i accedir a les llistes característica del llenguatge de programació Lisp.

Si `L` és una llista, la funció `car(L)` retorna el primer element de la llista `L` (`L[0]`) o la llista buida `[]` si `L` és buida. La funció `cdr(L)` retorna una llista igual que `L`, però sense el primer element (`L[1:]`, retorna la llista buida si `L` té un sol element o si està buida). Construirem les llistes amb la funció `cons(x,L)` que amb Python es pot expressar com `[x] + L`.

```
Si L = [1, 2, 3, 4, 5]
    car(L) = 1
    cdr(L) = [2, 3, 4, 5]
i, en general, es verifica que:
    cons(car(L), cdr(L)) = L
```

També hem afegit algunes funcions auxiliars, com `gensym`, que genera identificadors (*strings*) amb un component aleatori per a garantir que no han aparegut abans, i funcions d'ordre superior, que apliquen funcions a llistes o a elements de llistes. Són fàcils d'entendre i pensem que el codi és prou autocontingut.

```
import sys
import random
sys.setrecursionlimit(1000000)
# No és el mateix que el gensym de Lisp, però és més que
# suficient pel que necessitem
def gensym ():
    return 'symb' + str(int(10000000*random.random())).rjust(7,'0')

def car(lst): return ([] if not lst else lst[0])

def cdr(lst): return ([] if not lst else lst[1:])

def caar(lst): return car(car(lst))

def cadr(lst): return car(cdr(lst))
```

```
def cdar(lst): return cdr(car(lst))


def cddr(lst): return cdr(cdr(lst))

def caddr(lst): return car(cdr(cdr(lst)))

def cdddr(lst): return cdr(cdr(cdr(lst)))

def caadr(lst): return car(car(cdr(lst)))

def cadadr(lst): return car(cdr(car(cdr(lst))))

def cadddr(lst): return car(cdr(cdr(cdr(lst))))

# Aquest cons no fa el mateix que el cons de Lisp, però ja ens va bé
def cons(elem, lst):
    tmp = lst.copy()
    tmp.insert(0,elem)
    return tmp

def member_if (prd, lst):
    ll = lst.copy()
    leng = len(lst)
    while (leng > 0):
        elem = ll[0]
        if prd(elem):
            return ll
        ll.pop(0)
        leng -= 1
    return []

def find_if (prd, lst):
    for elem in lst:
        if prd(elem):
            return elem
    return []

def remove_if (prd, lst):
    results = []
    for elem in lst:
        if not prd(elem):
            results.append(elem)
    return results

def mapcar (f, lst):
```

```
    return list(map(f,lst))
```