Ud2 - Resolució de Problemes per Mitjà de Cerques

CE Models d'inteligència artificial

Curs 2024-2025

Sebastian Ciscar

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica

Continguts

1 Resolució de Problemes per Mitjà de Cerques **2**
1.1 Introducció . . . 2
1.2 Índex . . . 2
1.3 Resolució de problemes . . . 2
1.3.1 Què és un problema? . . . 2
1.3.2 L'espai d'estats . . . 3
1.3.3 Algorismes de cerca en l'espai d'estats . . . 3
1.4 Cerques no informades . . . 4
1.4.1 Cerca independent del problema . . . 4
1.4.2 Cerca en amplària . . . 4
1.4.3 Cerca en profunditat . . . 5
1.4.4 Cerca en profunditat limitada . . . 6
1.4.5 Cerca en profunditat iterativa . . . 7
1.5 Cerca heurística . . . 7
1.5.1 Què és la heurística? . . . 7
1.5.2 Heurística admissible . . . 8
1.5.3 Heurística consistent . . . 8
1.5.4 Cerca A* . . . 8
1.5.5 Distancia de Manhattan . . . 9
1.6 Conclusió . . . 10
1.6.1 Llegend: . . . 13

1 Resolució de Problemes per Mitjà de Cerques

1.1 Introducció

Aquest document està dirigit als alumnes del curs d'especialització de Big Data i Intel·ligència Artificial. En ell es tractarà la resolució de problemes mitjançant diferents algorismes de cerca, explorant des de cerques no informades fins a cerques heurístiques avançades.

1.2 Índex

1. Resolució de problemes
    1. Què és un problema?
    2. L'espai d'estats
    3. Algorismes de cerca en l'espai d'estats
2. Cerques no informades
    1. Cerca independent del problema
    2. Cerca en amplària
    3. Cerca en profunditat
    4. Cerca en profunditat limitada
    5. Cerca en profunditat iterativa
3. Cerca heurística
    1. Què és la heurística?
    2. Heurística admissible
    3. Heurística consistent
    4. Cerca A*

1.3 Resolució de problemes

1.3.1 Què és un problema?

En el context de la intel·ligència artificial i la informàtica, un problema es pot definir com una situació inicial juntament amb una condició objectiu que volem assolir mitjançant una seqüència d'accions. Aquestes accions ens portaran des de la situació inicial fins a la situació objectiu.

Un problema sol estar compost de quatre elements principals: - **L'estat inicial**: La configuració del sistema en el punt de partida. - **L'objectiu o estat final**: La configuració que volem assolir. - **Les**

accions o operadors: Les possibles transicions que permeten moure's d'un estat a un altre. - **Les restriccions**: Les condicions que les solucions han de complir.

1.3.2 L'espai d'estats

L'espai d'estats és una representació abstracta de totes les possibles configuracions que poden assolir-se a partir de la situació inicial mitjançant les accions disponibles. Cada estat representa una configuració única i les transicions entre estats es defineixen mitjançant les accions.

Aquest espai es pot visualitzar com un graf on cada node representa un estat i cada aresta representa una transició entre estats. L'objectiu de la cerca és trobar un camí en aquest graf que porti de l'estat inicial a l'estat objectiu.

Per exemple, en el problema del trencaclosques de les vuit reines, cada estat és una configuració de les reines sobre el tauler, i les accions són moviments de les reines que no violen les regles del joc.

1.3.3 Algorismes de cerca en l'espai d'estats

Els algorismes de cerca s'utilitzen per explorar l'espai d'estats amb l'objectiu de trobar una seqüència d'accions que ens portin a la situació objectiu. Aquests algorismes poden ser classificats en cerques no informades i cerques heurístiques, depenent de si utilitzen o no informació addicional per guiar la cerca.

Els algorismes de cerca en l'espai d'estats poden ser de diferents tipus, depenent de com exploren el graf d'estats. Alguns dels algorismes més comuns inclouen:

- **Cerca en amplària**: Explora tots els nodes a un nivell de profunditat abans de passar als nodes del següent nivell.
- **Cerca en profunditat**: Explora tant com sigui possible al llarg de cada branca abans de retrocedir.
- **Cerca en profunditat limitada**: Similar a la cerca en profunditat, però amb un límit de profunditat.
- **Cerca en profunditat iterativa**: Realitza cerques en profunditat amb límits successivament creixents.
- **Cerca de cost uniforme**: Explora els nodes en funció del cost acumulat des de l'inici.
- **Cerca A***: Utilitza una funció de cost combinada que inclou tant el cost acumulat com una estimació heurística del cost restant.

1.4 Cerques no informades

1.4.1 Cerca independent del problema

Les cerques no informades, també conegudes com cerques a cegues, no utilitzen informació addicional sobre el problema més enllà de les definicions bàsiques dels estats i les transicions. Aquestes cerques exploren l'espai d'estats de manera sistemàtica.

Aquest tipus de cerques inclouen la cerca en amplària i la cerca en profunditat. Són independents del problema en el sentit que no fan ús de cap informació específica sobre la distribució dels estats ni sobre l'objectiu final. Això les fa generals però sovint ineficients, ja que poden explorar moltes parts de l'espai d'estats que no són rellevants per trobar la solució.

1.4.2 Cerca en amplària

La cerca en amplària explora tots els nodes a un determinat nivell de profunditat abans de passar als nodes del següent nivell. Aquesta tècnica garanteix trobar la solució més curta però pot ser ineficient pel que fa a l'ús de memòria.

Aquest algorisme funciona utilitzant una cua, afegint els nodes fills d'un node abans d'expandir el següent node de la cua. És complet, és a dir, troba una solució si n'hi ha una, i és òptim si el cost de cada pas és igual. La principal limitació és el seu alt requeriment de memòria, especialment en espais d'estats amb un gran factor de ramificació.

Pseudocodi:

```
function GRAPH-SEARCH (problema) return una solució o una fallada
  Inicialitzar la llista OPEN amb l'estat inicial del problema
  Inicialitzar la llista CLOSED a buit
  do
    if OPEN està buida
      then return fallada
    p <- pop (llista OPEN)
    afegir p a la llista CLOSED
    if p = estat final
      then return solució p
    generar fills de p
    per a cada fill n de p :
      aplicar f(n)
      if n no està en CLOSED then
```

```
        if n no està en OPEN o (n està repetit en OPEN i f(n) és millor que el v
          then inserir n en ordre creixent de f(n) en OPEN*
      else f(n) és millor que el valor del node repetit en CLOSED
        then escollir entre re-expandir n (inserir-ho en OPEN) o descartar-
lo
  enddo
* Com estem interessats a trobar únicament la primera solució, es pot eliminar n
```

1.4.3 Cerca en profunditat

La cerca en profunditat explora tant com sigui possible al llarg de cada branca abans de retrocedir. Aquest mètode utilitza menys memòria però pot no trobar la solució més curta i pot caure en cicles infinits si no es controla.

L'algorisme utilitza una pila per gestionar els nodes a explorar, afegint els fills del node actual al principi de la pila. És més eficient en termes de memòria comparat amb la cerca en amplària, però pot explorar camins molt llargs que no condueixen a una solució, especialment en espais d'estats amb molta profunditat.

Pseudocodi:

```
Cerca_en_profunditat(estat_inicial):
  crear pila buida S
  afegir estat_inicial a S
  mentre S no està buida:
    estat_actual = treure de S
    si estat_actual és l'estat_objectiu:
      retornar el camí a estat_actual
    per cada veí de estat_actual:
      si veí no està visitat:
        marcar veí com visitat
        afegir veí a S
  retornar cap_solució
```

1.4.3.1 Cerca en profunditat amb backtracking La cerca en profunditat amb backtracking és una variant de la cerca en profunditat que permet gestionar l'exploració de l'espai d'estats de manera més eficient. Aquest mètode utilitza una pila per gestionar els nodes a explorar i una llista per emmagatzemar els nodes visitats. Quan es troba una solució, es retrocedeix per trobar altres solucions possibles.

Aquesta tècnica és útil en problemes on hi ha múltiples solucions possibles o on es vol trobar la millor solució. Per exemple, en el problema de les vuit reines, es poden trobar totes les solucions possibles amb aquest mètode.

Pseudocodi:

```
Cerca_en_profunditat_backtracking(estat_inicial):
  crear pila buida S
  crear llista buida visitats
  afegir estat_inicial a S
  mentre S no està buida:
    estat_actual = treure de S
    si estat_actual és l'estat_objectiu:
      afegir estat_actual a solucions
    per cada veí de estat_actual:
      si veí no està en visitats:
        afegir veí a visitats
        afegir veí a S
  retornar solucions
```

1.4.4 Cerca en profunditat limitada

Aquest tipus de cerca en profunditat imposa un límit a la profunditat que es pot explorar, evitant així caure en cicles infinits. Tot i això, aquest límit pot impedir trobar una solució si aquesta es troba més enllà de la profunditat imposada.

La cerca en profunditat limitada és útil quan sabem que la solució es troba dins d'un cert nombre de passos des de l'estat inicial. El principal desavantatge és que si la solució està més enllà del límit establert, l'algorisme no la trobarà.

Pseudocodi:

```
Cerca_en_profunditat_limitada(estat_inicial, límit):
  crear pila buida S
  afegir estat_inicial a S amb profunditat 0
  mentre S no està buida:
    (estat_actual, profunditat_actual) = treure de S
    si estat_actual és l'estat_objectiu:
      retornar el camí a estat_actual
    si profunditat_actual < límit:
```

```
    per cada veí de estat_actual:
      si veí no està visitat:
        marcar veí com visitat
        afegir veí a S amb profunditat profunditat_actual + 1
retornar cap_solució
```

1.4.5 Cerca en profunditat iterativa

La cerca en profunditat iterativa combina els avantatges de la cerca en amplària i en profunditat. Realitza cerques en profunditat amb límits successivament creixents, assegurant que es trobarà la solució més curta.

Aquest mètode és complet i òptim com la cerca en amplària, però amb millors requeriments de memòria. Funciona realitzant repetides cerques en profunditat, incrementant el límit de profunditat cada vegada fins que es troba la solució. És especialment útil en espais d'estats molt grans.

Pseudocodi:

```
Cerca_en_profunditat_iterativa(estat_inicial):
  límit = 0
  mentre cert:
    resultat = Cerca_en_profunditat_limitada(estat_inicial, límit)
    si resultat no és cap_solució:
      retornar resultat
    límit = límit + 1
```

1.5 Cerca heurística

1.5.1 Què és la heurística?

Una heurística és una funció que estima el cost o la distància des d'un estat donat fins a la situació objectiu. Les cerques heurístiques utilitzen aquestes funcions per guiar la cerca cap a la solució de manera més eficient.

Les heurístiques es basen en coneixement addicional sobre el problema i poden fer que la cerca sigui molt més eficient. Per exemple, en un problema de trobar el camí més curt en un mapa, una heurística pot ser la distància en línia recta des de l'estat actual fins a l'objectiu.

1.5.2 Heurística admissible

Una heurística es considera admissible si mai sobreestima el cost de trobar la solució. Això assegura que la cerca guiada per aquesta heurística trobarà sempre una solució òptima.

Això vol dir que per a qualsevol estat, la funció heurística sempre retorna un valor que és igual o menor que el cost real d'arribar a l'objectiu. Les heurístiques admissibles són essencials per garantir que els algorismes de cerca, com A*, trobin la millor solució possible.

1.5.3 Heurística consistent

Una heurística és consistent si, per a cada estat, la seva estimació no excedeix el cost d'arribar a un estat veí més el cost estimat des d'aquest estat veí fins a la situació objectiu. Les heurístiques consistents garanteixen que la cerca A* sigui òptima i completa.

La consistència implica que la funció heurística h compleix la següent condició per a qualsevol estat n i qualsevol successor n' de n: h(n) <= c(n, n') + h(n'), on c(n, n') és el cost d'anar de n a n'. Aquesta propietat assegura que una vegada que un node és expandit, el seu cost total no canviarà.

1.5.4 Cerca A*

L'algorisme A* és una tècnica de cerca heurística que utilitza una funció de cost combinada que inclou tant el cost acumulat des de l'inici com l'estimació heurística fins a la situació objectiu. Aquesta combinació permet a l'algorisme trobar solucions òptimes de manera eficient.

A* utilitza una funció de cost f(n) = g(n) + h(n), on g(n) és el cost des de l'estat inicial fins a n, i h(n) és l'estimació heurística del cost des de n fins a l'objectiu. Aquest algorisme expandeix el node amb el menor valor de f(n), garantint així que trobi el camí més curt si h és admissible i consistent.

```
Cerca_A*(estat_inicial, estat_objectiu, heurística):
  crear cua buida de prioritat Q
  afegir estat_inicial a Q amb prioritat 0
  g(estat_inicial) = 0
  mentre Q no està buida:
    estat_actual = treure de Q
    si estat_actual és l'estat_objectiu:
      retornar el camí a estat_actual
    per cada veí de estat_actual:
      cost = g(estat_actual) + cost(estat_actual, veí)
```

```
    si veí no està en Q o cost < g(veí):
      g(veí) = cost
      f(veí) = g(veí) + heurística(veí, estat_objectiu)
      afegir veí a Q amb prioritat f(veí)
retornar cap_solució
```

1.5.5 Distancia de Manhattan

La distància de Manhattan és útil en problemes on els moviments només es poden fer en direccions horitzontals i verticals, com en un tauler de joc o un mapa de ciutat.

La distància Manhattan, també coneguda com a "distància taxicab" o "distància de bloc de ciutat", es calcula com la suma de les diferències absolutes entre les coordenades de dos punts en una quadrícula. És útil quan es pot moure només horitzontalment i verticalment, com és el cas en una ciutat amb carrers ortogonals com Manhattan.

En un mapa bidimensional, la distància de Manhattan entre dos punts (x1, y1) i (x2, y2) es calcula com |x1 - x2| + |y1 - y2|. Aquesta heurística és admissible i consistent, ja que sempre subestima el cost real i compleix la condició de consistència.

Aquest càlcul es pot generalitzar per a qualsevol dimensió, on simplement es sumen les diferències absolutes de les coordenades corresponents de cada dimensió.

A diferència de la distància euclidiana, que mesura la distància més curta en línia recta entre dos punts, la distància Manhattan segueix camins paral·lels a les coordenades, com en un sistema de carrers d'una ciutat. Aquesta distància és molt utilitzada en aplicacions de ciències de dades, reconeixement de patrons i en jocs com el "Snake", on els moviments són limitats a direccions ortogonals.

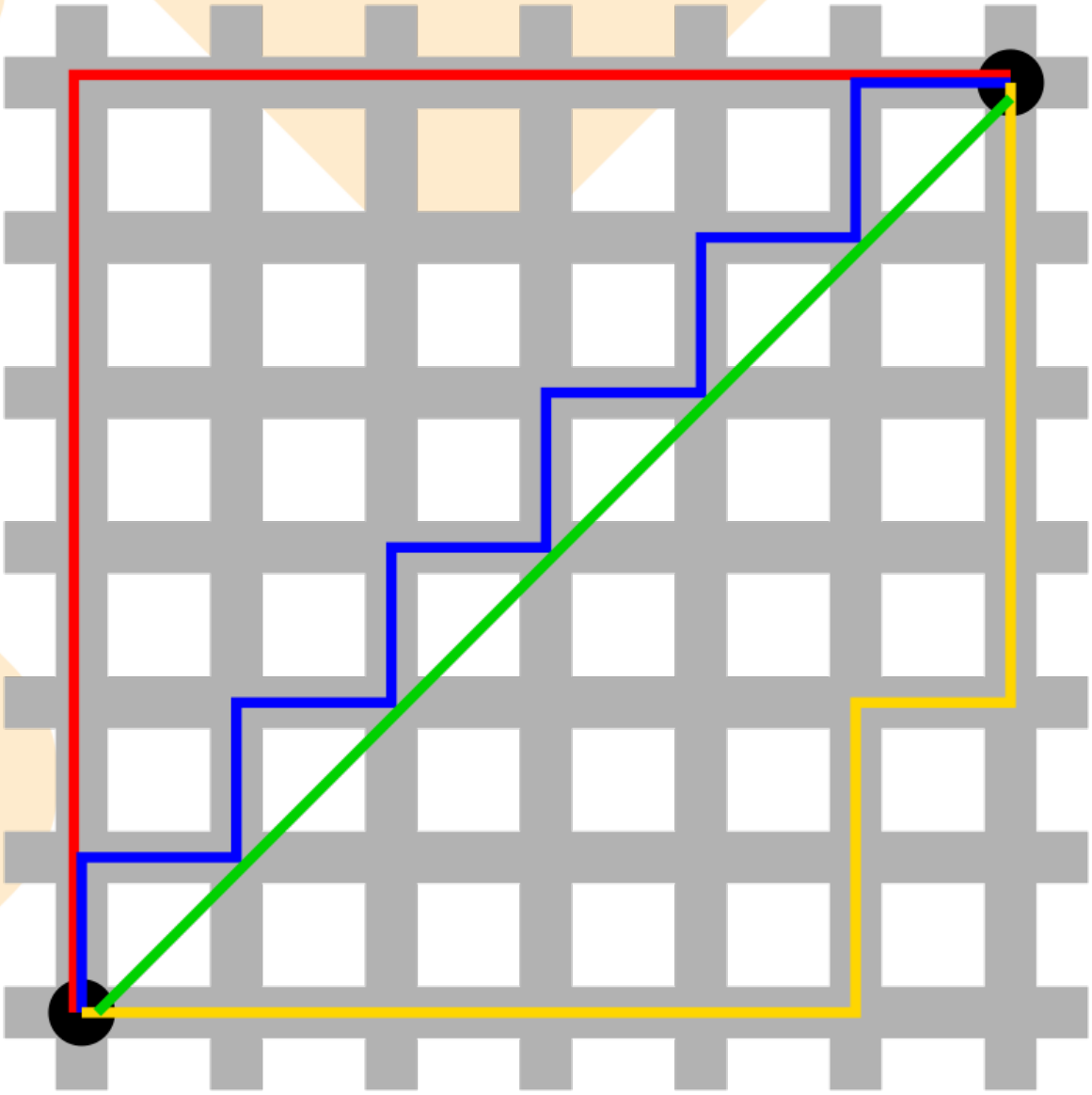

Figura 1: Distancia manhatan

1.6 Conclusió

Algorisme	Aplicació	Complexitat Temporal	Avantatges	Inconvenients
BFS (Breadth-First Search)	- Cerca en amplària per grafos no ponderats	O(V + E)	- Troba el camí més curt en grafos no ponderats	- No pot manejar grafos ponderats
	- Detecció de cicles en grafos no dirigits		- Simple d'implementar	- Pot ser costós en termes de memòria
	- Cerca de camins mínims en laberints		- Garanteix trobar la solució més curta en grafos no ponderats	- Requereix més memòria que DFS
	- Generació d'arbre mínim en grafos no ponderats			
DFS (Depth-First Search)	- Cerca en profunditat per detectar cicles o components connectats	O(V + E)	- Menys memòria que BFS	- No garanteix camí més curt
	- Ordenació topològica		- Apte per a recórrer tot el grafo	- Pot quedar-se en bucles infinits sense mecanismes de prevenció
	- Cerca en puzzles com el Sudoku o problemes de coloració		- Funciona bé amb profunditats limitades	- Pot trobar solucions subòptimes en grafos grans
	- Detecció de connexions fortament connectades			

Algorisme	Aplicació	Complexitat Temporal	Avantatges	Inconvenients
Profunditat Iterativa	- Cerca combinant els avantatges de DFS i BFS	O(bˆd), on b és el factor de ramificació i d la profunditat del node	- Eficient en espai com DFS, però més complet com BFS	- Pot explorar nodes repetidament
	- Resolució de problemes on no es coneix la profunditat de la solució		- No requereix estimar la profunditat màxima	- Més lent que BFS per a grafos grans
	- Cerca de solucions en problemes de jocs (com damuntes o escacs)		- Troba solucions en jocs on la profunditat òptima no es coneix	- Exploració repetida de nodes
A*	- Cerca del camí òptim utilitzant una funció heurística	O(bˆd), on b és el factor de ramificació i d la profunditat del node	- Eficient si es tria una bona heurística	- Difícil de definir una heurística adequada
	- Cerca de camí més curt en problemes de navegació		- Troba el camí més curt si la heurística és admissible	- Pitjor rendiment amb heurístiques mal dissenyades
	- Cerca de camins en mapes amb obstacles (per exemple, en robots)		- Molt utilitzat en IA i videojocs per trobar el camí òptim	- Més costós en termes de càlcul que BFS o DFS
	- Resolució de problemes de rutes òptimes en logística			

1.6.1 Llegend:

- **V**: Nombre de nodes (vèrtexs).
- **E**: Nombre d'arestes (arestes).
- **b**: Factor de ramificació.
- **d**: Profunditat del node objectiu.