UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Boca Alina-Maria
Grupa: 30234

Facultatea de Automatica
si Calculatoare

17 Noiembrie 2025

Cuprins

1 Tutorial

1.1 Subtitlu 1

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

1.1.1 Subsubtitlu

Born into the noble Crownguard family, along with his younger sister Lux, Garen knew from an early age that he would be expected to defend the throne of Demacia with his life. His father, Pieter, was a decorated military officer, while his aunt Tianna was Sword-Captain of the elite Dauntless Vanguard—and both were recognized and greatly respected by King Jarvan III. It was assumed that Garen would eventually come to serve the king's son in the same manner.

1.2 Tutorial elemente de baza

O propozitie normala. Daca vrem sa adaugam text in **bold**, *italic* sau ***bold italic***.
Astfel se poate utiliza o **lista numerotata**:

1. primul element
2. al doilea element
3. al treilea element

Astfel se poate utiliza o **lista neumerotata**:

- primul element
- al doilea element
- al treilea element

Asa se adauga o **imagine**.

Figura 1: Baba Voss

Daca vrem sa referim imaginea 1 in text se procedeaza astfel.

Asa se adauga **cod**:

```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

2 Uninformed search

2.1 Question 1 - Depth-first search

Căutarea în adâncime este un algoritm de căutare, cum sugerează si numele, care caută cât mai ân adâncime. DFS explorează arcele care ies din nodul cel mai recent adăugat care incă are arce neexplorate.

In cazul unei probleme de cautare, daca folosim DFS, frontiera va fi reprezentata ca si o stiva, pentru a putea implementa comportamentul algoritmului DFS. Stiva ne permite sa luam mereu cel mai nou element adaugat, care creeaza astfel efectul de adancime.

2.2 Question 2 - Breadth-first search

Cautarea in latime, la fel cum sugereaza numele, va exploata in intregime vecinii de acelasi ordin ai unui nod. BFS are garantia distantei cele mai scurte, deoarece mereu va explora toti descendetii unui nod, inainte sa treaca la nivelul urmator.

In cazul unei de probleme de cautare folosinf BFS, frontiera va fi reprezentata ca si o coada care va facilita comportamentul algoritmului BFS. Coada ne permite sa folosim nodurile in ordinea in care au fost adaugate.

3 Informed search

3.1 Question 3 - Varying the Cost Function

Căutarea cu Cost Uniform (Uniform Cost Search) este un algoritm care generalizează BFS pentru situațiile în care acțiunile au costuri diferite (grafuri ponderate). Spre deosebire de BFS, care extinde cel mai scurt drum ca număr de pași, acest algoritm extinde nodul cu cel mai mic cost total acumulat de la start ($g(n)$). Garantează găsirea soluției cu costul minim, nu doar a celei cu cei mai puțini pași.

În cazul unei probleme cu funcție de cost variabil, frontiera va fi reprezentată ca o coadă cu priorități, ordonată în funcție de costul drumului parcurs (g). Această structură ne permite să extragem mereu nodul cel mai "ieftin" de atins până în acel moment, ignorând numărul de noduri vizitate și concentrându-se strict pe efortul total consumat.

3.2 Question 4 - A* search algorithm

A* (A-Star) este un algoritm de căutare informată care, spre deosebire de BFS sau DFS, folosește o euristică pentru a estima distanța rămasă până la țintă. A* garantează găsirea celui mai scurt drum (similar cu BFS), dar este mult mai rapid deoarece nu explorează uniform în toate direcțiile, ci se concentrează pe direcția destinației. Decizia se bazează pe formula $f(n) = g(n) + h(n)$, unde combinăm costul real parcurs deja (g) cu costul estimat până la final (h).

În cazul unei probleme de căutare folosind A*, frontiera va fi reprezentată ca o coadă cu priorități (Priority Queue). Aceasta nu servește nodurile în ordinea sosirii (ca la BFS), ci le reordonează dinamic astfel încât în vârf să fie mereu nodul cu cel mai mic cost total estimat (f). Coada cu priorități ne permite să expandăm mereu "cea mai promițătoare" cale, evitând zonele care ne îndepărtează de soluție.

3.3 Question 5 - Finding All the Corners

Corners Problem este o problemă de căutare în care obiectivul nu este o coordonată fixă, ci satisfacerea unei condiții cumulative: vizitarea a 4 puncte distincte. Spre deosebire de o problemă simplă de navigare (A la B), aici definiția "țintei" este dinamică și depinde de istoricul drumului parcurs.

În cazul acestei probleme, starea (State) trebuie redefinită: nu mai este suficientă poziția (x, y), ci avem nevoie de o structură compusă: ((x, y), (c1, c2, c3, c4)). Această reprezentare extinsă permite algoritmului să distingă între a fi într-un colț pentru prima dată (progres) și a fi în același colț a doua oară (redundant).

Frontiera va conține aceste stări complexe. Funcția getSuccessors are rolul critic de a genera noi stări prin actualizarea listei de colțuri vizitate doar atunci când Pacman pășește pe unul dintre ele. Astfel, isGoalState va returna True doar atunci când componenta de "memorie" a stării indică faptul că toate cele 4 colțuri au fost bifate (True), indiferent de poziția finală a agentului.

3.4 Question 6 - Corners Problem: Heuristic

Pentru a găsi o euristică admisibilă, am relaxat problema ignorând pereții labirintului. Deoarece obiectivul este vizitarea tuturor colțurilor, euristica estimează costul total simulând un traseu simplificat: calculează distanța Manhattan de la poziția curentă la cel mai apropiat colț

nevizitat, apoi de la acela la următorul cel mai apropiat, și tot așa până la epuizarea colțurilor, însumând distanțele.

3.5 Question 7 - Eating All The Dots

Pentru a reusi sa manance toate punctele, am folosit o functie predefinita in fisierul searchAgents.py care ruleaza BFS pentru fiecare pereche de 2 puncte si returneaza distanta drumului dintre cele doua.

Am observat ca Pacman "se gandeste" foarte mult (mai mult de 10s) daca am apela functia mazeDistance pentru fiecare pereche de puncte, astfel am ajuns la varianta in care ma folosesc de $heuristicInfo$ pentru a stoca informatii despre distantele deja calculate. Sa nu fie necesara recalcularea lor.

pacman mereu va lua in considerare distanta maxima

3.6 Question 8 - Suboptimal Search

Pentru a gasi distanta optima intre doua puncte date, vom folosi BFS

4 Adversarial search

4.1 Question 9 - Improve the ReflexAgent

Pacman trebuie sa stie consecintele actiunilor sale, astfel imi propun sa iterez peste fantome, in cazul in care distanta este mai mica de 2, Pacman v-a muri, asfel ii spun ca va pierde 500 de puncte, daca mananca toata mancarea va primi 500. Altfel voi calcula scorul total in functie

4.2 Question 10 - Minimax

Minimax este un algoritm decizional recursiv utilizat in teoria jocurilor si inteligenta artificiala pentru a minimiza posibilele pierderi in cel mai nefavorabil scenariu (cand adversarul joaca optim) si pentru a maximiza potentialele castiguri.

Algoritmul presupune un joc cu suma nula (sau adversar direct), unde fiecare jucator incearca sa ia decizia cea mai buna pentru el. In cazul nostru:

- **Pacman (MAX):** Incearca sa maximizeze scorul.
- **Fantomele (MIN):** Incearca sa minimizeze scorul lui Pacman (sa il prinda).

Implementarea se bazeaza pe trei functii principale:

1. **maxValue:** Apelata pentru agentul 0 (Pacman). Itereaza prin toate actiunile legale si returneaza valoarea maxima posibila ($v = \max(v, successor)$).
2. **minValue:** Apelata pentru agentii $index > 0$ (Fantome). Itereaza prin actiuni si returneaza valoarea minima. Deoarece pot exista mai multe fantome, functia determina cine urmeaza la mutare: o alta fantoma (tot MIN) sau Pacman (MAX).
3. **value:** O functie dispecer care decide ce tip de nod urmeaza in arbore in functie de indexul agentului.

O particularitate a implementarii este incrementarea adancimii ($depth$). Un nivel complet de adancime se considera parcurs doar dupa ce toti agentii (Pacman + toate fantomele) au mutat. Astfel, adancimea creste doar cand `nextAgent` revine la 0.

Conditia de oprire a recursivitatii apare cand starea este terminala (Castig/Pierdere) sau cand s-a atins adancimea maxima definita, moment in care se apeleaza `evaluationFunction` pentru a estima utilitatea starii curente.

4.3 Question 11 - Alpha-Beta Pruning

Alpha-Beta pruning optimizează algoritmul minimax prin eliminarea ramurilor care nu pot influența decizia finală.

Parametrii:

- α (alfa) - Valoarea minimă garantată pentru MAX (inițial $-\infty$)
- β (beta) - Valoarea maximă garantată pentru MIN (inițial $+\infty$)

Regula de tăiere: $\alpha \geq \beta$

Când apare această situație:

- **La un nod MIN**: MAX a descoperit deja o opțiune mai bună pe altă ramură, deci MIN nu va permite accesul pe acest drum
- **La un nod MAX**: MIN poate deja forța un rezultat mai favorabil pe o altă cale, deci această ramură devine irelevantă

4.4 Question 12 - Expectimax

Context: Nu întotdeauna jucăm împotriva unui adversar optim. Un jucător imperfect poate alege din mai multe variante cu probabilități diferite. La Pacman, fantomele aleg aleatoriu din lista legală de acțiuni, fiecare acțiune având aceeași probabilitate.

Diferența față de Minimax:

- **Minimax**: Presupune că adversarul joacă optim (alege cea mai bună mutare)
- **Expectimax**: Modelează adversari care aleg probabilistic sau suboptimal
- În loc de noduri MIN (care aleg minimul), avem **noduri de șansă** care calculează *valoarea așteptată* (media ponderată)

Structura arborelui:

- **Noduri MAX**: Jucătorul nostru - alege acțiunea cu valoarea maximă
- **Noduri CHANCE**: Adversarul - calculează $\sum P(a) \cdot valoare(a)$ pentru toate acțiunile
- La nodurile de șansă, fiecare ramură contribuie proporțional cu probabilitatea sa

Exemplu concret: Dacă o fantomă poate merge în 4 direcții cu probabilitate egală ($P = 0.25$):

$$Valoarenodsansă = 0.25 \times v_{sus} + 0.25 \times v_{jos} + 0.25 \times v_{stânga} + 0.25 \times v_{dreapta}$$

Caracteristici importante:

- Toate ramurile din nodurile de șansă trebuie explorate (nu putem folosi Alpha-Beta pruning complet)
- Util când adversarul nu joacă optim sau când există incertitudine în mișcări