Tema 1
Inteligenta Artificiala

Grigore Lucian-Florin 343C4
Facultatea de Automatica si Calculatoare
Universitatea Politehnica, Bucuresti
Noiembrie 2021

Descrierea problemei

Nonogramele sunt puzzle-uri logice care in urma rezolvarii dezvaluie diferite imagini. Solutia este data de catre colorarea sau nu a unor cellule dintr-o grila pe baza interpretarii logice a numerelor prezente le inceputul fiecarui rand si coloane.

Se da o grila formata din patrate care trebuie innegrite sau lasate libere. Langa fiecare linie se afla lungimile sirurilor de patrate negre de pe linia respectiva. Deasupra fiecarei coloane se afla lungimile sirurilor de patrate negre de pe acea coloana.

Reprezentarea problemei in stari. Spatiul starilor

Pentru prima parte a rezolvarii, cerinta este formularea problemei ca o problema pe spatiul starilor si definirea notiunii de stare. In implementarea proprie, o stare este un dictionar in Python care contine informatii despre:

- numele testului care ruleaza in momentul curent
- dimensiunile tabelei: inaltime si latime
- conditiile de pe linii si coloane date in input
- grila de la pasul curent
- calea: o lista de miscari ce au avut loc pana la starea curenta
- parinte: pointer catre starea de la care a avut loc ultima miscare

Ca functii care ajuta la gestionarea starilor am considerat:

- is_state_final: verifica daca starea curenta este finala
- check_rows si check_columns: functii care verifica daca conditiile de pe linii si respectiv coloane sunt indeplinite

Algoritmi pe spatiul starilor

Pentru a putea realiza o comparatie eficienta intre diferitii algoritmi folositi, spatiul starilor este, in implementare, semi-informat. O miscare de la o stare la alta este reprezentata de adaugarea unui nou rand grilei formate pana in starea curenta. Acest pool de randuri disponibile este generat pe baza conditiilor de pe linii oferite in input. In acest fel, la orice pas, starea curenta poate adauga un nou rand care respecta conditia aferenta, dar care poate sau nu poate face parte din solutia finala, in functie de validarea conditiilor de pe coloane.

Starea initiala are un grid gol (care contine doar celule de 0) si pozitia de start pe randul cel mai de sus din imagine.

Toti algorimii folosesc aceasta logica, existand diferente doar intre alegerea urmatoarei miscari si a ordinii de parcurgere a miscarilor disponibile in fiecare stare.

BFS

Algoritmul Breadth-First Search tine starile generate intr-o coada. La fiecare pas din algoritm, el alege prima stare din coada, verifica daca este stare de final si, daca nu, genereaza posibilele miscari. Acestea sunt adaugate la finalul cozii.

Observatii:

- fiind neinformat, algoritmul isi pierde din eficienta cu cat spatiul alegerilor creste. De exemplu, de la testul 3 in sus, rularea dureaza cel putin 30 de minute. Pentru primele 2 teste dureaza sub 0.02 secunde.
- de asemenea, generaza un numar mare de noduri care de multe ori nu sunt folosite in solutia finala, avand deci nevoie de mai multa memorie decat ceilalti algoritmi

DFS

Algoritmul Depth-First Search tine starile generate pe o stiva. La fiecare pas din algoritm, el alege prima stare de pe stiva, verifica daca este stare de final si, daca nu, genereaza posibilele miscare pe care le pune la inceputul stivei. Dat fiind faptul ca la fiecare miscare dintr-o stare in alta, adaug un rand la grila construita pana atunci, adancimea maxima pentru DFS este egala cu inaltimea grilei.

Observatii:

- la fel ca la BFS, dureaza exponential mai mult cu cat creste dimensiunea spatiului starilor. Daca pentru primele 2 teste dureaza sub 0.015 secunde, pentru testele urmatoare dureaza cel putin 20 de minute
- o diferenta fata de BFS este faptul ca ocupa mai putina memorie. Acest lucru se datoreaza faptului ca el incearca continuu sa construiasca cai de lungime egala cu inaltimea grilei si astfel nu memoreaza atat de multe stari generate pe parcurs.

IDS

Asemanator cu DFS prezentat mai sus, singura diferenta fiind limitarea lungimii cailor pentru fiecare cautare a solutiei. Considerand spatiul starilor prezentat mai sus, intotdeauna IDS va gasi o solutie la pas-ul height din iteratie, unde height este inaltimea grilei.

Observatii:

- la fel ca cei doua algoritmi prezentati mai sus, timpul de rulare creste exponential de la testul 3 mai departe. Totusi, este un algoritm mai lent decat BFS si DFS pentru ca

trebuie sa parcurga un numar de pasi pana sa ajunga la height. Astfel, el ajunge, in rulare sa regenereze stari si cai generate anterior, la pasi precedenti ai iteratiei.
- de asemenea, consumul de memorie este mai mare decat la ceilalti algoritmi
- la rularea pasului height, numarul de noduri expandate si generate este identic cu DFS, fiind echivalent cu acesta

A*

Fiind un algoritm informat, A* foloseste un heapqueue pentru memorarea starilor generate si pentru alegere mereu ia starea cu costul cel mai mic. De asemenea, pastreaza intr-un dictionar costul curent al unei stari generate anterior, pentru a vedea daca descopera o cale mai eficienta pana la aceasta.

Functiile de cost folosite:

1. Am considerat un analog al functiei logice NXOR: cost += 1 acolo unde grid_stare_curenta[i][j] != grid_solutie[i][j]. Astfel, algoritmul va fi „condus" catre reprezentarea solutiei la fiecare pas.
2. Am considerat un cost care este egal cu numarul liniilor ramas necompletate. Astfel, „indemn" algoritmul sa incerce sa completeze toata grila si sa nu se intoarca sa revizuiasca randuri completate anterior.

Observatii:

- acest algoritm se termina pentru toate testele propuse si rezultate sunt considerabil mai bune comparativ cu cei trei algoritmi de mai sus, acolo unde se poate face comparatia
- numarul de noduri generate si expandate este semnificativ mai mic
- memoria utilizata este mai mare datorita structurilor specifice folosite
- timpul de rulare este mai mic

Reprezentarea problemei ca o problema de satisfacere a restrictiilor

Am considerat in continuare folosirea structurilor definite anterior pentru spatiul starilor, simuland pasii algoritmului prin variabile auxiliare si nu prin multimi care sa semnifice domeniul variabilelor initializate si neinitializate sau domeniul de valori. Astfel, am considerat ca starea curenta (care e modificata la fiecare iteratie din algoritm) si all_rows sunt necesare ca parametri. Variabila all_rows contine tot spatiul starilor folosit la algoritmii anteriori. In esenta, am simulat rularea MAC pe un graf in care o miscare dintr-un nod in altul simbolizeaza adaugarea unui nou rand in grila.

MAC

Fiind foarte similar cu backtracking, singura diferenta notabila fiind pastrarea consistentei la fiecare pas, ma asteptam ca MAC sa nu se comporte la fel de bine ca A*. Pentru procesul de select, am considerat alegerea primului rand necompletat din grila (nu neaparat necompletat, intrucat pot exista si randuri goale in solutie, ci randul la care am ajuns in iteratia curenta a

algoritmului). Iar pentru procesul de AC-3, am folosit urmatoarea logica: O trecere dintr-un nod in altul pastreaza consistenta arcelor daca, prin adaugarea randului in grila partial construita, nu ajungem la o grila care sa nu fie solutia problemei. Adica in implementare se folosesc doar randuri care respecta conditia de pe randuri, dar adaugarea unui nou rand poate strica una sau mai multe din conditiile de pe coloane. Astfel, consideram o pastrare a consistentei doar acele treceri (adaugari de randuri) care nu strica niciuna din conditiile de pe coloane.

Observatii:
- algoritmul MAC este mai lent decat cea mai buna solutie pe spatiul starilor odata cu cresterea numarului de miscari posibile la fiecare nod
- pe de alta parte, este mai rapid atunci cand numarul de miscari este mai mic
- de asemenea, consumul de memorie este mult mai mic decat la A* si numarul de noduri expandate si generate este mai mare

Observatii finale

Algoritmul A* pare a fi cel mai potrivit pentru rezolvarea unor astfel de probleme.

Din pacate, calea pana la solutie nu este afisata in notebook acolo unde numarul de stari generate este mare.

Generarea spatiului starilor dureaza mult pentru ultimul test ('train'), adica aproximativ 55-60 de minute.

Daca doriti testarea notebook-ului, mutati fisierele din '/tests' in acelasi folder cu 'tema1.ipynb'.