TUDOR Costin-Cristian
Grupa 342B2

TEMA 1 - Strategii de căutare

Inteligență artificială (Anul universitar 2021-2022)

Introducerea și prezentarea funcțiilor utilizate pentru prelucrarea datelor

Pentru implementarea strategiilor de căutare din cadrul temei, am ales să generez la începutul programului toate posibilitățile corecte de completare ale fiecărei linii/coloane. Apoi, pornind de la prima linie a unei matrice fără niciun element completat, am realizat tranziția între stări completând pe linii cu permutările obținute anterior. Astfel, se vor verifica doar restricțiile impuse pentru coloane și nu se vor expanda nodurile care încalcă aceste restricții, chiar dacă acestea nu au încă toate liniile completate. Am considerat că, în acest fel, voi economisi memorie și voi obține un timp de execuție mai bun, deoarece nu mai există posibilitatea redescoperirii nodurilor și nu mai sunt expandate noduri care nu pot conduce către rezolvarea problemei.

Așadar, în cadrul primei cerințe, am implementat următoarele funcții ajutătoare:

→ ***CreateInitialMatrix( ):*** Returnează o matrice cu ”height” linii și ”width” coloane, fiecare element având valoarea –1;

→ ***CheckSolution(mat):*** Returnează ”True” dacă matricea ”mat” respectă toate restricțiile impuse atât pentru linii, cât și pentru coloane. Altfel, returnează ”False”;

→ ***CheckPartialSolution(mat, ln):*** Primește o matrice completată până la linia ”ln” și returnează ”True” dacă aceasta respect (până la linia ”ln”, inclusiv) restricțiile impuse pentru coloane. Altfel, returnează ”False”;

→ ***GetLeftMostPerm(num_vals, length):*** Primește o listă cu valori numerice reprezentând lungimile șirurilor de 1 pentru o linie/coloană și returnează o listă cu ”length” elemente în care toate șirurile de 1 sunt grupate la stânga, cu un singur spațiu între ele (permutarea cea mai din stânga);

→ ***AddZero(line, pos):*** Elimină ultimul element dintr-o linie/coloană (funcția este apelată doar dacă ultimul element este 0) și adaugă un 0 înaintea șirului de 1 cu indicele ”pos”;

→ ***GetLinePerms(ln):*** Primește indicele unei linii din matrice și generează, folosind BFS, toate permutările care respectă restricțiile impuse pentru acea linie, pornind de la permutarea cea mai din stânga. Pentru a face trecerea de la o permutare la alta, se folosește funcția ***AddZero***;

→ ***GetColumnPerms(cl):*** similar, dar pentru o coloană.

Strategii implementate

Iterative Deepening Search

Deoarece adâncimea la care se află potențiale soluții ale problemei este o constantă egală cu înălțimea imaginii, pentru fiecare AdMax < înălțimea, algoritmul pierde foarte mult timp generând noduri care nu vor conduce către o stare ce poate fi testată.

În această situație, IDS nu aduce nicio îmbunătățire față de un simplu DFS, ci doar realizează foarte multe operații în plus.

Breadth First Search

Având în vedere aceeași observație, și anume că adâncimea este egală cu înălțimea imaginii, BFS va procesa nodurile ce pot reprezenta soluții ale problemei abia în momentul în care va ajunge la cea mai mare adâncime, ceea ce înseamnă că trebuie să genereze toate nodurile din arborele de căutare pentru a găsi soluția problemei, consumând astfel foarte mult timp și resurse.

Totuși, BFS va procesa fiecare nod o singură dată, spre deosebire de IDS, care va procesa aceleași noduri de mai multe ori, până la atingerea adâncimii maximii la care se află posibile soluții.

Depth First Search

Consider că, dintre strategiile de căutare neinformată implementate, DFS este cea mai bună alegere în această situație, deoarece generează fiecare nod o singură dată și ajunge în mod uniform la posibile soluții ale problemei. În medie, pentru găsirea soluției, DFS va genera de două ori mai puține noduri decât BFS, acest lucru fiind realizat fără a mai memora coada cu nodurile care urmează să fie vizitate.

A* Search – Heur1_ColumnPermsSum

Pentru implementarea primei funcții euristice, am calculat înainte de rularea algoritmului **A*** suma valorilor din permutările pe coloană pentru fiecare element. Am obținut o matrice în care fiecare valoare reprezintă numărul de permutări posibile ale coloanei sale, în care elementul de pe poziția respectivă este 1. Având numărul total de permutări pentru fiecare coloană, pot calcula cu ușurință și numărul de permutări în care un anumit element este 0.

În funcția euristică, pentru a departaja nodurile care urmează să fie procesate, am considerat că liniile care respectă un număr cât mai mare de permutări ale coloanelor au șanse mai mari de a conduce către soluția finală. Astfel, am calculat un cost care reprezintă suma numărului de permutări pe coloană încălcate de fiecare element de pe linia respectivă. De asemenea, am împățit acest cost la indicele liniei pentru a oferi prioritate nodurilor mai apropiate de adâncimea unei posibile soluții.

Folosind această funcție, am obținut o optimizare față de strategiile de căutare neinformată BFS și IDS, însă timpul de execuție este foarte apropiat de cel obținut în cazul rezolvării DFS.

A* Search – Heur2_NeighborsAbove

Analizând testele propuse și nu numai, am observat faptul că, pentru a oferi continuitate imaginii generate, șirurile de 1 sunt de cele mai multe ori grupate unele sub altele. Astfel, am realizat o funcție euristică ce numără câte elemente cu valoarea 1 de pe linia L au vecinul de pe linia L-1 cu valoarea 0. De asemenea, am împărțit valoarea astfel obținută la raportul dintre numărul de elemente de pe linia L-1 cu valoarea 1 și numărul de elemente de pe linia L cu valoarea 1, pentru a realiza un echilibru în cazul în care o linie conține mai multe sau mai puține căsuțe completate decât vecina sa.

Utilizând această funcție euristică, am obținut un timp de execuție de două ori mai bun și am generat semnificativ mai puține noduri față de prima euristică pentru toate testele încercate.

Problema satisfacerii restricțiilor

Am modelat problema satisfacerii restricțiilor în felul următor:

→ ***Variabile:*** Liniile și coloanele din matrice;

→ ***Domenii:*** Permutările liniilor/coloanelor;

→ ***Constrângeri:*** Între fiecare linie și coloană, elementul comun trebuie să fie egal.

Am realizat algoritmul MAC pentru a stabili arc-consistența la nivel de graf, iar pentru majoritatea testelor, stabilirea arc-consistenței a fost suficientă pentru a reduce domeniul fiecărei variabile la o singură valoare. Am utilizat apoi algoritmul backtracking nemodificat pentru a obține soluția finală.

Modelând puzzle-ul logic ca pe o problemă de satisfacere a restricțiilor, am obținut rezultate mult mai bune pentru testele propuse decât în cazul celorlalte strategii, impunerea arc-consistenței reducând semnificativ dimensiunea domeniilor variabilelor.

Rezultate obținute în urma rulării testelor propuse

Tabel timp de execuție (în secunde)

	DFS	BFS	IDS	A*(HEUR1)	A*(HEUR2)	MAC & BKT
Test 1	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	0.005
Test 2	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
Test 3	0.094	0.329	0.728	0.104	0.058	0.016
Test 4	61.984	> 600	> 600	60.093	20.362	1.436
Test 5	> 600	> 600	> 600	> 600	> 600	1.202
Test 6	> 600	> 600	> 600	> 600	> 600	16.411

Tabel noduri expandate (nodurile pentru care s-au descoperit toți copiii)

	DFS	BFS	IDS	A*(HEUR1)	A*(HEUR2)	MAC & BKT
Test 1	5	26	15	18	27	56
Test 2	0	13	10	5	6	10
Test 3	1332	25540	9723	9960	3891	20
Test 4	102873	-	-	2350100	772515	30
Test 5	-	-	-	-	-	35
Test 6	-	-	-	-	-	50

Tabel noduri generate (noduri descoperite în căutarea soluției)

	DFS	BFS	IDS	A*(HEUR1)	A*(HEUR2)	MAC & BKT
Test 1	26	27	41	21	27	76
Test 2	8	14	27	14	14	11
Test 3	18635	25547	137139	10073	5822	21
Test 4	8279194	-	-	2442270	1078614	31
Test 5	-	-	-	-	-	36
Test 6	-	-	-	-	-	51

Analiza rezultatelor

După cum se poate observa în tabelele prezentate, modelarea puzzle-ului ca pe o problemă de satisfacere a restricțiilor utilizând MAC și Backtracking a obținut, de departe, cele mai bune rezultate, fiind singura strategie care a calculat soluția corectă pentru fiecare test cu o viteză acceptabilă.

Totuși, în cazul primului test, această metodă a obținut cel mai slab timp de execuție și a generat cel mai mare număr de noduri în căutarea soluției. Acest timp de execuție este rezultatul faptului că domeniile variabilelor nu au putut fi reduse la o singură valoare în cadrul algoritmului MAC, ca în cazul altor teste, iar algoritmul clasic Backtracking nu este suficient pentru a genera rapid soluția pentru domenii de lungime mare.

Strategiile de căutare neinformată IDS și BFS au fost dezavantajate de modul în care am ales să prelucrez datele problemei, acestea generând un număr foarte mare de noduri în căutarea soluției, lucru care se reflectă și în timpul de execuție (exemplu: testul 3). De asemenea, acestea au fost singurele strategii de căutare care nu au găsit o soluție pentru testul 4 în mai puțin de 10 minute.

Din aceste date se mai poate observa faptul că a doua funcție euristică este mai bună decât prima, timpul de execuție și numărul de noduri generate/expandate pentru descoperirea soluției fiind de 2-3 ori mai mici. De asemenea, deși numărul de noduri generate de prima euristică este mai mic decât numărul de noduri generate de DFS, timpii de execuție obținuți sunt asemănători pentru toate testele deoarece prelucrarea unui nod în algoritmul A* este mai costisitoare decât în DFS, fiind necesare calcularea funcției euristice, introducerea în heap (care poate influența timpul de execuție pentru un număr foarte mare de noduri) și reconstrucția matricei pentru procesarea nodului la fiecare pas al algoritmului.

Așadar, consider că algoritmii implementați ar fi putut obține rezultate mai bune sau mai slabe, în funcție de metoda prin care erau prelucrate și reprezentate datele problemei.