Tema 1 Inteligența Artificială
Orar

Mîrza Ana-Maria, 331CA
Facultatea de Automatică și Calculatoare
Calculatoare și Tehnologia Informației

Keywords: A*, Memory Bound A* (MA*/SMA*), Stochastic Hill Climbing, Random Restart Hill Climbing

1 Descrierea Problemei Rezolvate

Context Generarea de orare este o problemă bine cunoscută, având aplicații în viața de zi cu zi a elevilor, studenților și a profesorilor. Problema care apare la generarea orarelor este dată de constrângerile fizice dată de numărul săli disponibile, capacitățile limitate ale sălilor și numărul de profesori disponibili pentru a preda o materie, dar și de constrângerile de preferințe a profesorilor. Pentru rezolvarea problemei s-au folosit variații ale algorimtilor de căutare A* și Hill Climbing.

Constrângeri Hard Aceste contrângeri sunt cele care nu pot fi încălcate de un orar valid. Dacă una din aceste constrângeri este încălcată, soluția nu va fi considerată finală.

1. într-un interval orar și într-o sală se poate susține o singură materie de către un singur profesor.
2. într-un interval orar, un profesor poate ține o singură materie, într-o singură sală
3. un profesor poate ține ore în maxim 7 intervale pe săptămână.
4. o sală permite într-un interval orar prezența unui număr de studenți mai mic sau egal decât capacitatea ei maximă specificată.
5. toți studenții de la o materie trebuie să aibă alocate ore la acea materie. Concret, suma capacităților sălilor peste toate intervalele în care se țin ore la materia respectivă trebuie să fie mai mare sau egală decât numărul de studenți la materia respectivă.
6. toți profesorii predau doar materiile pe care sunt specializați.
7. în toate sălile se țin ore doar la materiile pentru care este repartizată sala.

Constrângeri Soft Aceste constrângeri au asociate un cost foarte mare pentru a evita pe cât posibil încălcarea uneia dintre ele, dar în același timp lăsând posibilitatea găsirii unei soluții ce nu respectă toate contrângerile soft.

1. Preferă anumite zile sau nu doresc să predea într-o zi anume.
2. Preferă sau nu doresc anumite intervale orare, în oricare din zile

2 Reprezentarea Stărilor și a Restricțiilor

Stări Pentru o reprezentare eficientă a stărilor am creat clasa *State* ce conține un orar sub formă de dicționar ce are chei zilele, cu valori dicționare de intervale reprezentate ca tupluri de int-uri, cu valori dicționare de săli, cu valori tupluri (profesor, materie). Suplimentar, o stare mai conține și numărul de conflicte soft încălcate și materiile rămase de alocat sub formă de dicționar ce are chei materiile și ca valori numărul de studenți rămași de alocat la aceea materie. De asemenea, este reținut un dicționar cu profesorii primit din fișierul de intrare, conținând materiile la care sunt specializați și preferințele de orar, sau contrângerile soft. Aceste informații sunt

folosite la generarea următoarelor stări valide, cât și a numărului de conflicte din starea curentă. Dicționarul cu săli primit la intrare ce are valori capacitatea maximă și materiile care se pot ține în ele, este salvat într-o stare pentru a putea genera corect următoarele stări ce nu încalcă contrângerile hard. Informațiile ce determină o stare unic este orarul.

Restricții Restricțiile soft sunt reținute în stări așa cum sunt primite din fișierul de intrare, în dicționarul profesorilor, și pentru fiecare stare, la inițializarea acesteia, se iterează prin ele pentru a determina câte sunt încălcate de orarul din starea respectivă.

Pentru a îndeplini restricțiile hard, am ținut cont de acestea la generarea unei stări succesor. Generarea de stări succesor se realizează prin iterarea în orarul curent și verificarea fiecărui slot liber din orar ce poate fi cuplat cu un tuplu (materie - profesor) din materiile rămase de alocat, ce îndeplinește condițiile hard. O stare ce încalcă restricțiile hard nu este una validă, drept urmare nu va fi generată ca succesor și se va încerca următoarea configurație de stare. La final, funcția de generare de stări succesoare returnează garantat numai stări valide ce îndeplinesc condițiile hard, având asociat un cost dat de numărul de condiții soft încălcate.

3 Evaluarea Soluțiilor

Algoritmul A* Primul algoritm folosit pentru generarea unei soluții valide a unui orar este A*. Funcția de cost folosită pentru g(x) a fost inițial numărul de constrângeri soft încălcate de starea respectivă, iar f(x), funcția euristică era calculată în funcție de numărul de studenți rămași de alocat înmulțit cu un factor de pondere, adunat cu numărul de profesori disponibili pentru fiecare materie rămasă nealocată, proporțional cu numărul de studenți rămași la materia respectivă, înmulțit cu un factor de pondere. Astfel, ne asigurăm că materiile cu profesori mai puțini ce o pot ține vor fi prioritizare la alocare pentru a nu rămâne la final neacoperite.

Una din schimbările aduse algoritmului față de cel din laborator sunt eliminarea memorării stărilor parțiale deoarece la final nu ne era utilă o reconstrucție a stărilor, ci ne interesa doar răspunsul final. De asemenea, nu reținem părintele unei stări în dicționarul de stări explorate pentru că nu ar avea utilitate. Costul g(x) este de asemenea calculat diferit pentru adaptarea la problema orarelor.

Deoarece acest algoritm generează un număr foarte mare de stări, iar sistemul de calcul nu facea față la un calcul de asemenea intensitate (în cazul de față era omorât procesul de sistemul de operare când se umplea memoria), am recurs la niște variațiuni ale algoritmului ce sunt mai memory-friendly.

Memory Bound A* (MA*/SDA*) Ca să limităm memoria folosită de algoritm, am început prin a limita numărul de stări introduse în frontieră la fiecare pas, astfel asigurând faptul că se putea ajunge la o stare finală înainte de a se umple memoria sistemului de calcul și a fi omorât. Implementarea este asemănătoare cu Beam Search, alegând primii n cei mai buni succesori și adăugându-i în coada de explorare. Această abordare a făcut posibilă găsirea unei stări finale optime, dar nu este garantată aceasta dat fiind pierederea soluțiilor parțiale eliminate.

Dezavantajele acestei abordări sunt, așa cum am menționat mai sus, că nu este garantată optimalitatea fiind pierdute soluții pe drum. Deși acestea sunt pierdute, totuși o funcție euristică bună poate face posibilă găsirea unei soluții optime, așa cum este în cazul de față. Euristica aleasă reușește să ghideze căutarea asatfel încât să fie găsită o soluție optimă în timp util și să nu fie eliminată pe drum.

Complexitățile obținute la rularea acestui algoritm pe testele date sunt următoarele:

Test fișier	Timp Rulare	Stări Construite	Stări Memorate	Cost
dummy.yaml	0.0544s	407	407	0
orar_mic_exact.yaml	4.5233s	13453	3396	0
orar_mediu_relaxat.yaml	75.8900s	129037	6034	0
orar_mare_relaxat.yaml	350.8296s	449105	7601	0
orar_contrans_incalcat.yaml	-	-	-	-

Testul pentru orarul constrând încălcat nu a fost inclus pentru că timpul de rulare este foarte mare și timpul nu a mai permis și analiza lui. Dat fiind faptul că un număr de stări se pierd la fiecare iterație, nu avem garantat faptul că este găsită soluția optimă pentru el. Putem observa o limitare evidentă a memoriei folosită, dată de o creștere ținută sub control a numărului de stări păstrate în memorie.

Iterative Deepening A* (IDA*) Pentru a furniza o soluție optimă și completă, am ales să implementez și algoritmul IDA*, ce caută în spațiul stărilor limitând frontiera după un cost maxim al funcției f(x) = g(x) + h(x).

Acest algoritm asigură optimalitatea, ajungând să exploreze toate nodurile succesor, dar limitând memoria folosită, mergând din frontieră în frontieră.

Am ales o implementare a acestui algoritm schimbând funcția de cost g(x) pentru a fi egală cu costul dat de contrângerile soft încălcate înmulțit cu un factor de pondere, w, ce asigură faptul că sunt păstrate în frontieră doar acele noduri succesor ce au inițial nu au nicio constrângere soft încălcată, apoi la a doua iterație este permisă una, la a treia fiind permisă doua.. și așa mai departe.

Valoarea ponderii w este calculată ca fiind costul stării inițiale plus o unitate pentru că aceasta este prima limită de h pentru IDA*. Astfel, garantăm că fiecare iterație de IDA* va avea întâi doar nodurile cu 0 constrângeri soft încălcate, apoi cele cu 1, și așa mai departe, deoarece g(x) este egal cu numărul de constrângeri încălcate înmulțit cu această pondere.

Din nou, cu funcția euristică aleasă inteligent, algoritmul nu are nevoie de o a doua iterație prin nodurile de cost 1 (constrângeri soft încălcate), ci soluția este găsită relativ rapid, iar problema memoriei limitate este rezolvată.

Astfel, avem asigurat faptul că dacă există o soluție de cost 0, aceasta va fi găsită, fără a reține în memorie acele soluții de cost 1. Dacă în schimb, nu există o soluție de cost 0, algoritmul IDA* garantează că va găsi soluția de cost minim, deoarece după ce explorează toate soluțiile parțiale de cost 0, va începe o a doua iterație unde se vor explora și acele stări cu cost 1.

Dezavantajele acestei abordări sunt de timp. Deoarece algoritmul parcurge nodurile pe frontiera dată de funcția h(x) maximă permisă, aceasta poate dura mai mult decât la algoritmul clasic.

Performanțele obținute pentru acest algoritm sunt următoarele:

Test fișier	Timp Rulare	Stări Construite	Stări Memorate	Cost
dummy.yaml	0.0465s	406	144	0
orar_mic_exact.yaml	5.2466s	13453	2956	0
orar_mediu_relaxat.yaml	49.1036s	91070	36468	0
orar_mare_relaxat.yaml	320.1045s	314592	70016	0
orar_contrans_incalcat.yaml	-	-	-	-

Similar ca la MBA*, nu am dispus de suficient timp pentru a rula până la final testul pentru orarul contrâns încărcat, dar avem garantată găsirea soluției optime de către algoritm, chiar dacă durează mai mult.

O observație pe care o putem face este faptul că acest algoritm poate fi mai eficient din punct de vedere al memoriei decât MBA* pentru anumite teste mai mici și chiar mai rapid pentru cazuri mai complexe.

Algoritmul Hill Climbing Al doilea algoritm folosit pentru a genera o soluție validă de acoperire a orarului este Hill Climbing. Pentru a asigura completitudinea algoritmului, am recurs la abordarea Random Restart cu Stochastic Hill Climbing. Astfel, avem garantat că soluția optimă va fi găsită în câteva iterații sau restart-uri.

Pentru filtrarea succesorilor am folosit costul minim de constrângeri găsit în lista de succesori, ajungând să fie ales un succesor dintre aceia cu cost minim.

Random Restart Stochastic Hill Climbing Această variantă de Hill Climbing produce soluții foarte bune și într-un timp foarte scurt.

O eficientizare pe care am aplicat-o algoritmului pentru a folosi mai puțină memorie și timp este returnarea următoarelor stări sub formă de acțiuni posibile asociate cu numărul de conflicte soft pe care le-ar cauza, în loc de stări complete. Astfel, după ce este aleasă cea mai bună acțiune următoare, aceasta este aplicată stării curente.

În urma testelor, am observat că atunci când există o soluție de cost 0, aceasta este găsită rapid, iar atunci când nu avem o soluție exactă, ea poate fi găsită cu o probabilitate mai mare proporțional cu numărul de restart-uri permise. Dezavantajul este că, cu cât se permit mai multe restarturi, cu atât va dura mai mult căutarea, deoarece aceasta continuă cât timp nu găsește o soluție de cost 0, reținând pe parcurs cea mai bună soluție.

În următorul tabel este prezentat timpul mediu de rulare al algoritmului pe fișierele de test, numărul de stări construite și costul stării finale (mediu pentru orarul constrâns încălcat cu 30 de restart-uri posibile).

Fișier test	Timp Rulare	Stări Construite	Cost
dummy.yaml	0.0017s	457	0
orar_mic_exact.yaml	0.0838s	24627	0
orar_mediu_relaxat.yaml	0.6563s	246185	0
orar_mare_relaxat.yaml	1.4757s	337549	0
orar_contrans_incalcat.yaml	4.4880s	1728620	3

În acest tabel se poate observa foarte bine diferența de performanță a timpului de rulare față de cele ale algoritmului A*, dar și un număr mai mare de stări construite. Totuși, aceste stări nu sunt păstrate în memorie, ci după ce este ales nodul succesor, se continuă mai departare fără acestea.

Concluzii

În concluzie, algoritmul A* nu este cea mai bună alegere pentru problema generării orarelor ce trebuie să respecte constrângeri, deoarece spațiul stărilor este extrem de mare, ocupă foarte multă memorie și funcția euristică nu poate fi ușor aleasă pentru a garanta optimalitatea, iar admisibilitatea ei este greu de demonstrat.

Hill CLimbing-ul, pe de altă parte, este mai facil de folosit pentru această problemă și produce soluții bune având complexități de timp și spațiu relativ bune.

Soluțiile date de rularea acestor algoritmi pe testele date sunt salvate în directorul de output cu numele algoritmului aferent.