Inteligenta Artificiala: Tema1 - Orar Seria CC
Niculae Karla 333CC

---

1. Reprezentarea stărilor și a restricțiilor

Clasa Scheduler creează starea un obiect ce retine toate informațiile necesare pentru orar cât și orarul:

- days: lista de zile a orarului
- intervals: lista de intervale orare
- materials: dictionar ce conține materiile care trebuie predate și numarul de studenți care au materia respectiva
- professors: dictionar cu profesorii, materiile pe care le pot preda ai constrangerile fiecărui profesor.
- rooms: dictionar ce include salile disponibile pentru cursuri, capacitatea pentru fiecare sala si materiile care pot fi predate in acestea
- professor_constraints: dictionar cu constrangerile fiecarui profesor
- professor_subjects: dictionar ce asociaza fiecarui profesor materiile pe care le poate preda
- room_capacity: dictionar ce retine citi numărul de studenți care pot încăpea în fiecare sala
- room_subjects: dictionar ce retine materiile care pot fi predate in fiecare sala
- preferred_days, preferred_intervals: împărțirea constrângerilor fiecărui profesor, în zile preferate și respectiv intervale preferate, folosite pentru restricțiile soft
- scheduler_state: întreg orarul
- professor_assignments: retine pentru fiecare profesor materia la care poate preda și câte clase preda în săptămână respectiva în total

  În cazul în care nu primesc un orar, când încerc sa creez obiectul e tip scheduler, înseamnă ca trebuie sa creez orarul inițial:
- create_initial_schedule: creeaza orarul inițial care respecta toate constrângerile hard, și îmi creeaza și professor_assignments pt orarul inițial
  - creearea orarului inițial ca sa poată sa respecte constrângerile hard se face prin trecerea prin toate materiile ordonate descrescător(pentru a repartiza clasele libere mai întâi sa acopere materiile cu cei mai multi studenti)
  - îmi aleg cea mai buna camera (camerele ordonate tot descrescător ca sa iau camerele cele mai mari prima data pentru materia respectiva) în care se poate preda materia respectiva astfel încât sa o pot umple cu cât mai multi studenți
  - pentru fiecare zi interval verific pt fiecare profesor de la materia respectiva si sa nu aibă mai mult de 7 intervale, dacă nu nimeni în camera respectiva sa predea deja, îmi pot adaugă profesorul sa predea materia respectiva acolo

Constrângerile hard sunt verificate în crearea orarului intial, după descrierea de cum creez orarul inițial. Totuși la următoarele orare generate, ele sunt verificate prin funcția: def calculate_constrains(schedule, preferred_days, preferred_intervals, materials, score, room_capacity):

Aceasta calculează un scor pentru orarul dat ca parametru, pentru constrângerile hard am adunat un scor foarte mare(3000), astfel încât sa se creeze un total de scor final ridicat și astfel incat sa nu fie luat în considerare la algoritmi mtcs și hill climbing, din cauza costului ridicat și astfel evitându-se crearea unor orare care ar încalcă hard_constraints.
Constrângerile soft sunt găsite prin funcția def compute_conflicts(schedule, preferred_days, preferred_intervals):

Aceasta funcție verifica constrângerile încălcate de profesori, dacă o zi sau un interval nu se afla în preferred_days sau preferred_intervals al profesorului, atunci se va Adaugă în prof_with_constraints, ora,intervalul,materia,clasa,profesorul . Astfel funcția va întoarce toate constragerile soft încălcate în orarul respectiv.

După generarea unui orar nou, lui i se atribuie prin funcția calculate_constraints menționată mai sus un scor în funcție de regulile hard încărcate(3000), dar și de cele soft încălcate, costul fiind mai mic comparativ cu cele hard încălcate, deci pentru cele soft am adunat doar 10. Și astfel se

calculează scorul final al orarului respectiv, Apoi în funcție de algoritm, mts sau hc se ia în considerare cel care are cel mai bun scor, în cazul meu cel mai mic (înseamnă ca are cele mai putine constrângeri incalcate).
Pentru algoritmul Monte Carlo Tree Search node reprezintă o stare în arborele de căutare. Fiecare node conține informații despre starea actuală a orarului (obiect scheduler cu toate informațiile orarului și orarul curent), scorurile asociate acelei stări (Q pentru valoarea totală reward și N pentru numărul de vizite), precum și alte stări denumite actions, stări derivate din orarul curent (acțiuni posibile care duc la alte noduri).

2.Optimizări realizate pentru cei doi algoritmi, fată de varianta de la laborator, specifice acestei probleme:

a.**Hill Climbing**

- varianta din laborator este mult mai simplificata, fata de cea legată de tema orar. A trebuit sa modelez codul pe baza rezolvării problemei orarului.
- am introdus restricții mult mai complexe:
    - verificarea preferințelor profesorilor, adică verificarea constrângerilor soft, pentru ca în funcție de constrângerile soft încălcate încerc sa îmi creez un vecin nou care sa nu mai încalce respectiva constrângere
    - verificarea constrângerilor hard: sa nu predea un profesor mai mult de 7 ori pe săptămână, sa se repartizeze toți elevii de la o materie la cursuri, sa nu predea un profesor în accelasi interval de doua ori, toti studenții de la o materie sa fie alocați la materia respectiva
    - datorită faptului ca am plecat de la un orar care deja respecta constrângerile hard, anumite constrângeri hard nu mai aveau cum sa fie încălcate:profesorii sa predea doar materiile pe care le cunosc, repartizarea studenților în funcție de numărul de locuri dintr o clasa, într-un interval orar sa se predea o singura materie de către un singur profesor
- metoda de generare a vecinilor este mult mai complexa, iau orarul curent și toate constrângerile soft încălcate în acel orar, pentru fiecare constrângere soft încerc sa o permut în fiecare interval orar și sa creez un nou orar cu mutarea respectiva, dacă clasa e goală și se poate preda materia respectiva acolo atunci mut doar profesorul in sala goală, dacă sala nu este goală, atunci interschimbabile profesorul și materia din intervalul din care vreau sa mut profesorul care are încălca constrângerea, cu cel unde vreau sa mut, asta doar dacă materia din intervalul cu care vreau sa mut se poate preda în clasa unde vreau sa mut. Și a doua metoda de generare a vecinilor este de a înlocui profesorul cu constrângeri încălcate cu un alt profesor cre nu a predat încă de 7 ori în săptămână respectiva și preda materia respectiva, astfel având mai multe tipuri de vecini, pot explora mai multe posibilități, deci se va ajunge la un rezultat mai bun
- nu mai trebuie sa generez cum este, în cazul problemei reginelor, “tabla de joc” pentru ca eu deja pornesc de la orarul completat care îndeplinește hard constraints
- nu mă bazez pe numărul de conflicte încălcate, ci eu am o funcție care pentru fiecare orar creat îmi asignez un scor, cu cât scorul este mai mic, înseamnă ca orarul este mai bun, îndeplinește mai multe constrângeri. Pentru constrângerile hard adun 3000 ca sa știu ca atunci algoritmul nu va lua în considerare orarul care încalcă constrângeri hard, iar pentru soft am adunat doar 10, pentru ca constrângerile soft pot fi încălcate fata de cele hard
- dacă nu se găsește un vecin cu un scor mai bun, algoritmul meu se oprește, ceea ce previne rulări inutile ale algoritmului și crește eficiența, dacă de exemplu am ajuns într-un maxim local.

b. MCTS

- am folosit un dicționar de stari posibile in care se poate trece din starea curenta in loc de o lista de actiuni pe care le pot face din starea curenta, Adic din loc sa salvez doar acțiunea pe care aș fi putut sa fac, de exemplu sa interschimbabile doi profesori din 2 intervale, eu am făcut schimbarea și am și generat orarul, și orarul respectiv reprezint starea mea în care pot trece
- metoda de generare a stărilor este mai complexa decât cea din laborator, și este accesai pe care am folosit-o la hill climbing

- evaluez stările în funcție de scor, doar ca scorul este bazat pe câte constrângeri sunt încălcate de către orarul respectiv, acceasi idee ca la hill climbing
- am pornit de la orarul care respecta hard constraints, astfel arborele meu de start nu va fi niciodată gol, și încep direct cu orarul meu deja configurat ca root din arbore și stările posibile în care pot sa mă duc sunt orarele care se pot genera din acesta
- când ajung în acc stare de mai multe ori, deci nu mai găsesc îmbunătățiri, opresc programul

---

3.Comparatia între cei doi algoritmi din punct de vedere al:

a.Timpului de executie:

	HC	MCTS
Dummy.yaml	0.06098580360412598 secunde	12.825492143630981 secunde
Orar_mic_exact.yaml	4.900326251983643 secunde	404.2400360107422 secunde
Orar_mediu_relaxat.yaml	21.09016442298889 secunde	1558.233397245407 secunde
Orar_mare_relaxat.yaml	103.27811431884766 secunde	
Orar_constrans_incalcat.yaml	36.758718967437744 secunde	2716.453418970108 secunde

- Diferenta timpului de execuție intr HC și MCTS se datorează modului în care se explorează spațiul stărilor.
- Hill Climbing este un agloritm de cautare locala care încearcă sa caute o solutie optima modificând iterativ soluția curenta prin alegerea celei cu cel mai bun scor, pe când monte Carlo explorează în adâncime , și folosește o selecție bazată pe select_action. Hill Climbing explorează numai în vecinătatea imediata a stării curente. MCTS evaluează fiecare posibilitate de mai multe ori.
- La MCTS dureaza mai mult și din cauza ca eu îmi generez de fiecare data toate orarele întregi ca acțiuni, în loc sa generez doar acțiunile pe care aș fi putut sa le fac, eu îmi generez stări de orare.(în loc sa rețin doar mut profesorul a în intervalul x al profesorului b și profesorul b îl mut în locul profesorului a).

b.Număr de stări construite:

- În HC, numărul de stări construite este legat de numărul de iterații executate până când algoritmul nu mai poate găsi îmbunătățiri.La fiecare iterație, se generează un număr de noi stări vecine (prin get_neighbors), dintre care se alege una dacă este mai bună decât starea curentă.
- MCTS construiește un arbore de căutare unde fiecare nod reprezintă o stare posibilă. Numărul de stări generate este influențat de adâncimea arborelui și de numărul de pași făcuți în simulate.Fiecare poate explora o nouă cale în arbore, adăugând mai multe noduri.

c.Calitate a solutiei (număr de restrictii încălcate)

	HC	MCTS
Dummy.yaml	HARD=0 SOFT=1	HARD=0 SOFT=6
Orar_mic_exact.yaml	HARD=0 SOFT=6	HARD=0 SOFT=32
Orar_mediu_relaxat.yaml	HARD=0 SOFT=1	HARD=0 SOFT=27
Orar_mare_relaxat.yaml	HARD=0 SOFT=23	HARD=0 SOFT=
Orar_constrans_incalcat.yaml	HARD=0 SOFT=20	HARD=0 SOFT=70

- Hill Climbing nu reușește sa rezolve toate softurile deoarece am folosit varianta standard a algoritmului Hill Climbing și acesta rămâne blocat în maximul local, neputând sa încerce sa mai ia și alta stare mai putin eficienta pentru a avea șansă de a explora stări noi și a găsi optimul global. (Aș fi putut folosi stochastic hill climbing, unde aș fi avut luată următoarea stare random,

astfel încercându-se-se o diversitate de stări noi, și aș fi putut face și cu temperatura -> cum este Simulated Annealing).

- De asemenea am pornit și de la un orar cu toate constrângerile hard satisfăcute, deci de exemplu eu atunci când generez noi vecini, pornesc deja de la un schelete de orar, și de exemplu aș fi putut avea diferită tabla de orar, poate în orarul meu care respecta hard_constraints nu am folosit niște profesori, și eu chiar dacă atunci când generez noi vecini, ii iau în considerare, mie cumva orarul îmi impune cumva niște restricții de baza de cum sa generez următorii vecini bazat pe orarul cu hard_constraints făcut, și astfel nu toate variantele de orare sunt generate. Pentru ca de la sine și orarul inițial cu hard_constrains poate fi respectat, făcut prin mai multe metode, depinde cum iei for-urile.
- Algoritmul MCTS poate alege o soluție mai putin buna pentru a explora stări mai bune pe viitor, de aceea este posibil ca eu sa am încălcate atâtea constrângeri soft, pentru ca eu opresc algoritmul prematur după 30 de iteratii dacă ajung sa am rezultatul respectiv de mai multe ori la rând.

Pentru orar_mare_relaxat, procesul este omorât pentru ca am consum excesiv de resurse. MCTS construiește un arbore de căutare în memorie, care poate deveni foarte mare. Dacă fiecare nod al arborelui ocupă multă memorie, este ocupată foarte multa memorie, ceea ce duce la terminarea forțată a procesului de către sistemul de operare.
Cum se poate rezolva asta?Sa nu mai creez atât de multe alte stări, cu get_neighbours, get_neighbours ar trebui sa fie optimizat. Aș putea utiliza un dicționar în care sa mi tin stare din neighbours și apoi sa pui în vectorul neighbours doar stările neduplicate, ca sa evit sa nu am stări duplicate.