Tema1 IA – Orar

Preda Marius-Cristian, grupa 334CC

Introducere

În acest document este prezentată încercarea rezolvării problemei orarului folosind 2 algoritmi. Algoritmii folositi (Hill climbing, Monte carlo tree search) nu ajung mereu la cea mai buna soluție, iar costul soluției obținute variază datorită faptului ca algoritmii folosiți nu sunt optimali și deterministi. Problema este modelata ca o căutare în spațiul stărilor pentru a minimiza numărul de restricții încălcate.

Reprezentarea stărilor și a restricțiilor

O stare e reprezentată astfel:

- timetable : `{day: {interval: {classroom: (professor, subject)}}}`
  Acesta reprezinta orarul propriu-zis la un moment de timp în format corespunzător pentru pretty_print

- fitness: `{c_intervals: int, c_stud_left: int, c_mult: int, c_soft: int, c_pause: int}`
  Acesta reprezinta numarul de conflicte pentru o stare, dar ponderat în funcție de cat de grava e încălcarea constrangerii

- profs: `{professor: list(day, interval)}`
  Variabila ajutătoare pentru actualizarea mai eficienta a fitness-ului la trecerea între stari. Memorează intervalele în care un profesor preda

- students: `{subject: int}`
  Variabila ajutătoare pentru actualizarea mai eficienta a fitness-ului la trecerea între stari. Memorează numărul de studenți asignati pentru fiecare materie

- depth: `int`
  Variabila ce memoreaza adâncimea stării într-un arbore (folosita doar pt mcts)

Mutarile între stări se executa prin aceasta functie:

`apply_move(day, interval, classroom, prof, subject, depth)`

Aceasta pune în orar în ziua `day`, in intervalul `interval`, sala `classroom`, materia `subject`, predată de profesorul `prof` și actualizează fitness-ul stării, structurile corespunzătoare calculării acesteia `(students, profs)`. O mutare este practic de 2 tipuri de bază (add si remove): adaugarea `(prof, subject)` intr-un slot gol, ștergerea unui slot ce avea deja o pereche `(old_prof, old_subject)`, iar schimbarea unui tuplu este o mutare compusă (remove `old_pair` + add `new_pair`)

Motivare calcul fitness

Deoarece este mai important să respectăm constrângerile hard, acestea au o pondere mai mare. Constrangeri:

- c_intervals: pentru fiecare professor, pentru fiecare oră pe care o tine peste 7 pe saptamana, se aduna 200

```
for each prof
   fitness[c_intervals] += 200 * max(0, (nr_ore_prof - 7))
```

- c_stud_left: pentru fiecare materie, la fiecare min_stud studenți neasignati se aduna 40. min_stud reprezinta capacitatea minima a unei sali din cele disponibile

```
for each subject
    dif = num_stud_for_subject - num_stud_assigned_for_subject
    fitness[c_stud_left] += 40 * max(0, ceil(dif /
min_cap_of_classroom))
```

- c_mult: pentru fiecare interval în care un profesor se afla în mai multe săli în același timp, se aduna 100

```
for each interval
    for each prof_that_teaches
        if prof_in_multiple_classrooms
            fitness[c_mult] += 100
```

- c_soft: numărul de constrangeri legate de preferințe de zi, interval pe care un profesor le prefera

```
for each prof
    for each constraint
        if not_satisfied(constraint)
            fitness[c_soft] += 1
```

- c_pause: pentru fiecare profesor, cand are o fereastra in program, se aduna 1 pentru fiecare 2 ore pe care le are in plus in fereastra fata de ce prefera acesta

```
for each day
    for each prof
        fitness[c_pause] += real_window - max_window_wanted
```

Calculand astfel fitness-ul unei stari, deși 2 stări ar avea conform enuntului același număr de conflicte, programatic vor fi favorizate stările ce sunt mai apropiate de o stare cu un număr mai mic de conflicte. In acelasi timp, vor fi favorizate stările cu mai puține constrangeri hard încălcate.

Hill climbing

Soluția ce folosește algoritmul clasic hill climbing pleacă de la un orar gol și încearcă la fiecare pas sa caute cea mai buna mutare viitoare (prezentata anterior) pentru completarea orarului până în momentul în care nu mai exista mutări ce obțin o stare mai buna decat cea curentă. Algoritmul clasic ajunge de obicei la un rezultat valid, însă explorează foarte multe stări și pentru anumite teste (orar_constrans_incalcat) poate ramane blocat într-un minim local cu multe constrângeri încălcate.

Optimizări aduse algoritmului clasic:

1.1 Generarea stărilor ce succed starea curentă. Taieri deterministe

Pentru a micșora spațiul de căutare nu se genereaza stările ce știm cu siguranță că vor încalcă constrangeri hard (sala nu suporta o anumită materie, mutarea nu provoacă nicio modificare în orar, profesorul preda deja in alta sala într-un interval orar) sau mutări ce știm sigur ca nu vor aduce un improvement costului (dacă o materie este deja acoperita nu mai adaugam intervale pentru acea materie)

1.2 Generarea stărilor ce succed starea curentă. Taieri nedeterministe

Din cauza faptului că hill climbing este un algoritm ce nu revine la o alegere / stare precedenta, nu este nevoie ca o stare să aibă mereu aceiași succesori. Prin urmare am observat ca este rară alegerea schimbării unui interval (mutare compusa), deci putem sa nu generam mereu aceste mutări. Astfel o mutare de acest tip este generata mereu cu o probabilitate de 50%.

> Observatie: O posibilă îmbunătățirea viitoare a algoritmului ar fi ajustarea probabilității pe parcursul algoritmului (la început mai puține schimbări, la final mai multe). Acest lucru ar rezulta într-un timp de execuție mai scurt fără a obține un rezultat cu cost mai mare (la început oricum nu sunt alese mutarile de schimbare).

1.3 Generarea stărilor ce succed starea curentă. Distribuirea uniforma a mutarilor

Pentru a evita minime locale, mutarile sunt mereu amestecate. Din teste repetate reiese că acest lucru ajută la obținerea unor soluții mai bune din punct de vedere al costului.

```
for day in shuffle_dict(self.timetable).keys()
     for interval in shuffle_dict(self.timetable[day]).keys()
         for classroom in shuffle_dict(self.timetable[day][interval]).keys()
             ...generate move further
```

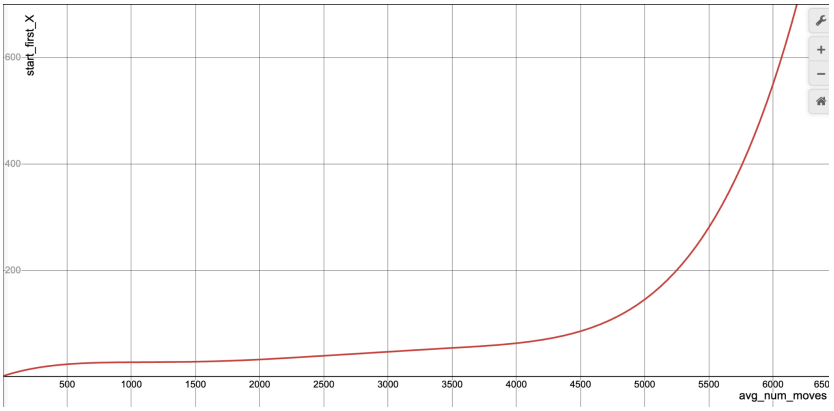
1.4 Generarea stărilor ce succed starea curentă. Ordinea mutarilor

După ce s-a ales un interval și o sală pentru o mutare, trebuie aleasă materia. Materia corespunzătoare mutării este aleasă în următoarea ordine: Mai întâi este aleasă materia ce are mai puține săli în care se poate preda.

2.1 Alegerea stării următoare. first_x states

Prin rulari succesive se observa ca este suficient sa alegem cea mai buna stare viitoare din primele x, unde x variază în funcție de dimensiunea orarului (branching factor). Am găsit o valoare optimă pentru fiecare test ce are o posibilă soluție de cost 0 (valoarea minima pentru care success rate-ul > 90%). Pentru a generaliza aceasta valoare am căutat cel mai apropiat polinom ce trece suficient de aproape de punctele găsite.

x in functie de branching_factor (avg_num_moves)



2.2 Alegerea stării următoare. Alte posibile imbunatatiri încercate

Ar părea ca exista și alte îmbunătățiri posibile în alegerea stării următoare:

- alegerea probabilistica a stării următoare (fiecare stare viitoare are atribuit un weight invers proportional cu fitness-ul acesteia. Acest lucru adauga mai mult overhead temporal decat ar parea și nu aduce cu adevărat un improvement în rezultate.

  Observatie: Este posibil ca aceasta metoda sa fie viabilă cu funcția de calcul a weight-ului potrivită, însă eu nu am reușit sa găsesc o funcție suficient de buna

- alegerea random a stării următoare din cele mai bune y dintre cele x. Acest lucru nu aduce nicio îmbunătățire din niciun punct de vedere

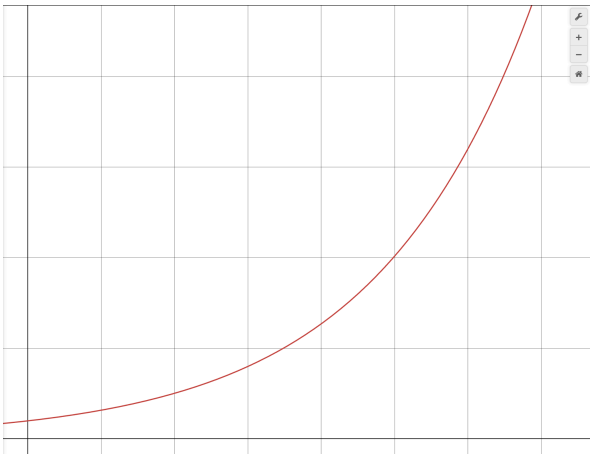
  Observatie: Am încercat metodic diverse valori pentru y, dar nu se poate observa vreun improvement din vreun punct de vedere

3.1 Random restart

Pentru a mari rata de succes a algoritmului (varianta first x nefiind determinista, folosim random restart cu maxim 10 iterații și se alege soluția cea mai buna dpdv a costului.

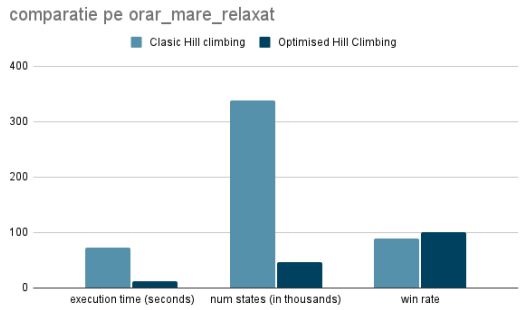
3.2 Random restart. Modificare dinamica x

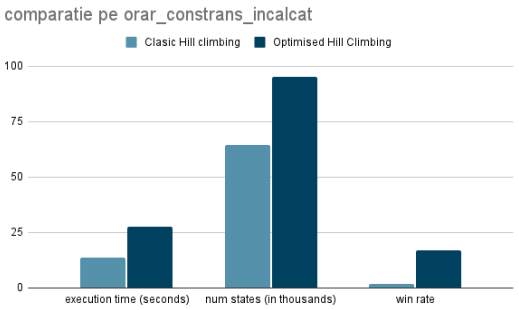
Cresterea x-ului relativ la nr de restart-uri



Pentru a ne asigura ca nu am făcut overfit pe testele oferite prin folosirea polinomului pentru calcularea x-ului (mentionat mai sus), gradual crestem x-ul pe parcursul iteratiilor din random restart. Practic se merge pe logica: dacă nu au fost suficiente x stări, incercam mai multe, poate acum merge. Din testele efectuate pare ca, cu cat testul este mai complicat avem nevoie de un x mai mare pentru a avea o rata de succes decentă.

Comparatie variante Hill Climbing (clasic cu optimizări de generare a stărilor vs varianta finală - random restart optimizat)





Pe teste mai ușor de satisfăcut (orare relaxate + mic), timpul de execuție și numărul de stări scade drastic prin optimizări, iar win rate-ul este putin mai bun.

Pe teste foarte greu de satisfăcut (orar_constrans_incalcat), timpul de execuție este mai mare, datorită numărului mare de restarturi, dar rata de succes este dramatic mai buna decat alternativa (de cel putin 15 ori mai mare).

Monte Carlo Tree Search

Solutia ce foloseste MCTS, la fel ca cea ce folosește Hill Climbing, pleaca de la un orar gol si la fiecare pas calculeaza cea mai buna mutare posibila prin metode statistice (scorul fiecărei stări viitoare este calculat prin scorul obtinut de niște simulări aleatoare din starea respectivă). Arborele creat este păstrat între alegerea mutarilor succesive.

Rezultatele obținute prin algoritmul clasic sunt groaznice: sunt încălcate constant multe constrângeri soft și uneori și unele hard (varianza de la test la test), iar timpul de rulare este foarte mare. O posibilă soluție pentru algoritmul clasic ar fi mărirea bugetului, dar asta ar duce la un timp de rulare impracticabil.

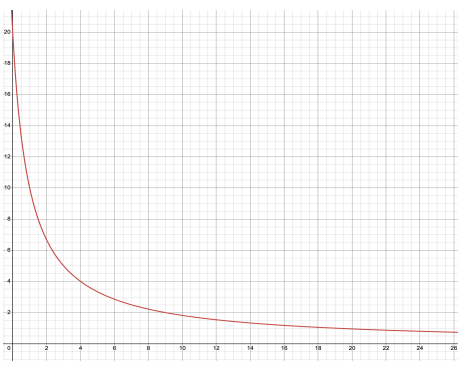
Optimizări aduse algoritmului clasic

1. Optimizare functie de reward

   Ajustarea funcției de reward astfel incat sa favorizeze foarte mult stările perfecte (0 hard, 0 soft incalcate)

```
f(x) = {
        50.0, if hard, soft = 0,
        0.0 if hard > 0,
        20 / (1 + soft), elsewhere
       }
```

reward function (num_soft_conflicts)



2. Optimizare adancime maxima

Ajustarea adancimii maxime (o stare e considerata finala daca num_conflicts = 0 or max_depth is reached). Prin multiple teste se observa ca o adancime maximă potrivită este chiar numărul de slot-uri din orar.

3. Găsire succesor în faza de simulare

In faza de simulare, se pot amesteca mutarile (ca la hill climb, explicat mai sus) și se poate genera prima mutare. Acest lucru este echivalent cu generarea tuturor succesorilor și alegerea uneia, dar este mai rapid dpdv computational

4. Tăieri deterministe

Aceasta este cea mai importantă optimizare (taieri), fără de care algoritmul nu ajunge la soluții bune și are un timp de rulare foarte mare. Se taie mutarile viitoare ce produc cu siguranța orare ce nu respecta constrangeri hard (la fel ca la hill climb). Se taie toate mutarile ce modifica un slot deja asignat. Se taie majoritatea mutărilor ce încalcă constrangeri soft (se pastreaza doar 3 - nr ales empiric). Prin ultima tăiere algoritmul ajunge și la soluții cu un număr mic de constrangeri soft încălcate. (inainte erau încălcate un număr mare de constrangeri soft constant)

5. Tăieri nedeterministe

În faza de simulare sunt tăiate 90% din mutarile ce încalcă o nouă constrângere soft. Astfel se crește artificial reward-ul produs de anumite ramuri.

6. Alte posibile optimizări

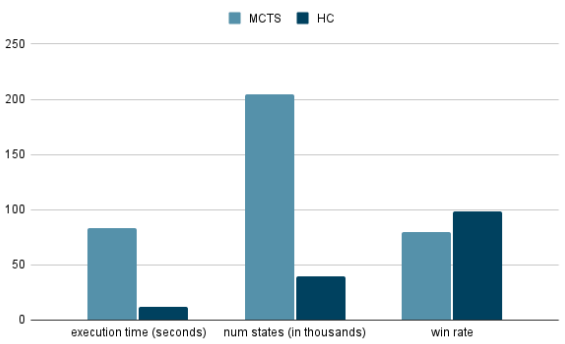
MCTS nu are rezultate extraordinare prin comparație cu Hill climbing, dar ar mai putea fi facute anumite optimizari pentru a îmbunătăți rezultatele obtinute / timpii de rulare:

a. ajustare CP (balansarea explorare - exploatare)
b. ajustare dinamica a bugetului pe masura ce se avansează în crearea orarului
c. ajustarea dinamica a procentului de taieri ce nu respecta constrangeri soft
d. O funcție de reward mai buna

Comparatie rezultate

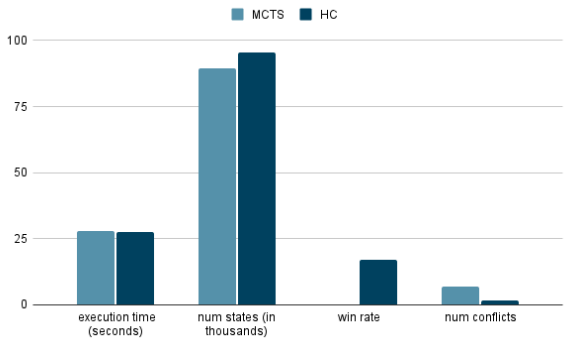
1. Test mai ușor de îndeplinit (orar_mare_relaxat)

Pe teste mai ușor de îndeplinit, mcts are rezultate comparative ca și cost, însă explorează mult mai multe stari si are un timp de execuție mult mai mare (de cateva ori mai mare, acest lucru poate fi rezolvat scazand bugetul, insa va exista un tradeoff in rata de succes)



2. Test mai greu de îndeplinit (orar_constrans_incalcat)

   Pe teste mai greu de îndeplinit, mcts este foarte apropiat ca timp de execuție, însă acest lucru vine cu un tradeoff in rata de succes destul de mare. MCTS obține rezultate cu mai multe constrângeri încălcate în medie, iar de multe ori încalcă chiar și constrangeri hard.



Ghid rulare și fișiere

1. Rulare

   `python3 orar.py <algorithm> <input_file> [n_trials]`

2. Ghid fișiere

   `orar.py =>` rulare teste algoritmi

   `state.py =>` reprezentarea unei stari, metode de manipulare a starilor

   `hill_climb.py =>` implementari algoritmi hill climbing

   `mcts.py =>` implementare MCTS, reprezentare nod arbore mcts

   `my_utils.py =>` utilitare extra fata de cele oferite in schelet

Concluzii

Algoritmul de tip hill climbing ar părea sa fie mult mai potrivit pentru problema orarului atat dpdv al timpului de execuție, cat și al costului obtinut, însă niciunul din cei 2 algoritmi nu este total potrivit pentru aceasta problema. In plus, consider ca testele oferite (numărul scăzut de teste) nu sunt suficiente incat sa poate fi trasa o concluzie în sensul de cat de bine funcționează algoritmi HC și MCTS pe problema orarului (dpdv al obținerii unei soluții cu cost minim).

Mentiune: A fost folosit Copilot pentru generarea de cod asemanator logic cu ce era deja scris de mine, code refactoring, docstrings. Orice înseamnă idei, optimizari, analiza a codului, structurare a codului, etc inclusiv acest document au fost facut in totalitate de mine. În plus îmi pare rău ca utilizarea diacriticelor nu este consistentă în acest document.