University Timetable Generator

Cuprins

0 – Organizare arhiva si cod

Arhiva contine urmatoarele fisiere si directoare:

- Director „inputs”: fisierele .yaml de intrare, date din schelet
- Director „refs”: output-urile propuse in schelet
- utils.py – necesar pentru ca este importat si utilizat in notebook
- check_constraints.py – la fel ca mai sus, dar este modificat astfel incat sa verifice si pauzele pentru cerinta bonus si sa le afiseze la stdout, la fel ca pe niste cerinte soft normale
- orar.ipynb – rezolvarea propriu-zisa
- Director „outputs_hill” cu orarele pretty-printed, iar detaliile de rulare sunt afisate in notebook
- Director „outputs_mcts” cu orarele pretty-printed, iar detaliile de rulare sunt afisate in notebook
- Director „custom_outputs”, care contine atat orarele, cat si detaliile de rulare, pentru niste teste suplimentare („neoficiale”) de mcts

Structura **orar.ipynb**:

Resurse generale de parsare, apoi tot ce inseamna Hill Climbing (stare, running environment, teste), apoi tot ce inseamna MCTS. Dupa testele normale de MCTS exista si niste teste suplimentare pentru a calcula un average.

- Clasa „Conditions” – parseaza un fisier de input, **ordoneaza datele in functie de importanta** si le adauga in cond.dict, folosit de algoritmi
- Clasele „State” si „AlterState” – folosite de HC si MCTS drept o incapsulare a starii curente
- Metodele „run_test” si „run_mcts” – running environments pentru cei 2 algoritmi, adaugand si statistici
- Metodele „hill_climbing” si „mcts” – algoritmii efectivi, call-ati din metodele de mai sus

1 – Hill Climbing

a) Reprezentarea starilor

Clasa State (incapsulare a elementelor ce definesc o stare) cuprinde:

- **Board**, orarul efectiv din starea respectiva, definit sub forma:
  { „zi” : { (ora_start, ora_fin) : { „sala” : („profesor”, „Materie”) } } }.
  Initial board-ul este gol, adica None in loc de (prof, materie)
- Timetable_specs: detele din fisierul initial, dupa ce le-am parsat si ordonat la inceputul algoritmului
- Conflicts – soft conflicts neindeplinite
- Cost – cati studenti nu sunt asignati la momentul respectiv
- Unassigned – lista de forma [(materie, nr. studenti neasignati)], actualizat constant
- Teacher_week_count – dictionar de forma { „prof” : nr. ore tinute }, pentru a vedea usor daca un profesor depaseste numarul de 7 ore

b) Gestionare restrictii

- Implementarea se bazeaza pe ideea ca niciodata sa nu avem alte restrictii hard neindeplinite in afara de studenti neasignati
- Pornim de la un orar gol, deci initial nicio restrictie incalcata
- Forma board-ului este aleasa astfel incat majoritatea restrictiilor hard sa nu se poata intampla in niciun moment: nu se pot preda mai multe materii in aceeasi sala concomitent
- Cand alegem actiunile urmatoare, facem in asa fel incat nici macar sa nu existe posibilitatea sa se „strice” vreo restrictie hard
- Restrictiile soft sunt verificate doar prin intermediul costului, nefiind vitale
- O stare este considerata finala cand toti studentii de la toate materiile sunt asignati, indiferent de restrictiile soft incalcate. Astfel ne asiguram telul: profesorii pot avea preferinte incalcate in cazul unui numar absurd de mare, dar, atat timp cat numarul de studenti insumeaza mai putin decat totalul de locuri in sali si se incepe alocarea cu materiile mai constranse de sali, se poate ajunge la o stare finala

c) Tranzitii si cost

- O tranzitie este definita ca una din 2 mutari posibile: alocarea unui slot gol in orar sau facut switch cu un slot deja ocupat
- Conflictele sunt calculate pe baza celor din starea anterioara, astfel nefiind nevoie sa parcurgem tot orarul

- Pentru fiecare stare vecina este calculat costul asociat, dupa formula: 2*nr_total_de_studenti_neasignati + cost_soft (zile, intervale si pauze)
- Starea vecina este cea care are costul de mai sus cel mai mic
- Daca nu se gaseste un vecin cu cost mai mic, algoritmul se incheie
- **Here's the catch**: Functia min din Python, in caz de egalitate (ceea ce se intampla des), returneaza prima intrare. Aici intra in joc ordonarea la parsare si greedyness-ul algoritmului. Cand am generat starile candidate, am tinut cont de urmatoarea ordine: materii cu putine Sali, zile nepreferate, intervale nepreferate, profesori cu putine intervale pentru materia aceea

```
for (new_subject, no_stud) in self.unassigned:
    if no_stud > 0:
        for day in self.timetable_specs[ZILE]:
            for interval in self.timetable_specs[INTERVALE]:
                for teacher in self.timetable_specs[MATERII][new_subject][PROFESORI]:
                    if self.teacher_week_count[teacher] >= 7:
                        continue
                    for room in self.timetable_specs[MATERII][new_subject][SALI]:
```

- Practic ce se intampla: se vor umple salile mari primele. Se iau materiile pe rand pana cand raman cu studenti mai putini decat capacitatea unei Sali mari. Apoi se trece la urmatoarea materie putin, pana se vor umple salile mari complet. Apoi chiar se vor lua materiile pe rand. Ciclul continua.

d) <u>Optimizare fata de laborator</u>

- Practic ordonarea beforehand reprezinta optimizarea in sine. Si abuzarea de mecanismul din functia min. De asemenea, fata de laborator, exista 2 „costuri" – unul care scade (nr de studenti neasignati) si unul care poate creste (costuri soft)
- Am folosit o strategie **best-first**. Iar pentru ca starea initiala este aceeasi, atunci algoritmul este determinist 😊

e) <u>Outputs</u>

-

	DUMMY	MIC	MEDIU	MARE	INCALCAT	BONUS
HARD	0	0	0	0	0	0
SOFT DEFAULT	0	0	0	0	0	0
PAUZE	N/A	N/A	N/A	N/A	N/A	5

- Modul de calcul al pauzelor nu este optim: costul este calculat in momentul in care s-ar adauga un slot nou si s-ar crea o pauza. Nu se tine cont daca, de exemplu, pauza aceea s-ar umple ulterior. Asta pe langa faptul ca este prioritizata punerea studentilor in Sali. Problema este si mai putin convexa decat inainte, iar abordarea mea cam prefera o problema convexa.
- Probabil daca nu as fi schimbat ordinea intervalelor orare, ar fi fost mai putine pauze neindeplinite, dar as fi riscat sa fie overfit, deci am ramas asa.

2 – MCTS

Multe elemente (in special reprezentarea starilor si a restrictiilor) sunt comune cu cele de la HC, asa ca mai mult le voi expune pe cele importante sau diferite.

a) Reprezentarea starilor

Clasa AlterState (incapsulare a elementelor ce definesc o stare) cuprinde aceleasi elemente primordiale ca cea de la HC:

- Doar ca sa fie notat, elementul central, **board**, este de aceeasi forma:
  { „zi" : { (ora_start, ora_fin) : { „sala" : („profesor", „Materie") } } }.
  Initial board-ul este gol, adica None in loc de (prof, materie)
- **Nodul de arbore** este format din:
  { N: 0, Q: 0, STATE: instanta_a_clasei_AlterState, ACTIONS :{} }
  ACTIONS : actiuni explorate, sub forma: {index_lista_completa_actiuni: Nod copil}

b) Gestionare restrictii

- Implementarea se bazeaza pe ideea ca niciodata sa nu avem alte restrictii hard neindeplinite in afara de studenti neasignati, pornind tot de la un orar gol
- Cand alegem actiunile urmatoare, facem in asa fel incat nici macar sa nu existe posibilitatea sa se „strice" vreo restrictie hard
- Restrictiile soft sunt verificate doar prin intermediul costului, nefiind vitale
- O stare este considerata finala cand toti studentii de la toate materiile sunt asignati, indiferent de restrictiile soft incalcate

c) Tranzitii, cost si reward

- Se porneste tot de la un orar gol
- O tranzitie poate fi doar alocarea unui slot gol in orar, iar costul pentru o stare se calculeaza pe baza starii anterioare, cu aceleasi incrementari ca la HC
- Pentru a afla starile urmatoare ale unui nod, facem uz de 2 hinturi: luam cele mai promitatoare stari (am pus cap la 15 stari care imbunatatesc cel mai mult scorul din starea respectiva) si le asociem probabilitati de explorare descrescatoare (o distributie exponentiala/progresie geometrica facuta de mine de mana. Sunt normalizate, este ok)

```python
@staticmethod
def generate_exponential_distribution(n, ratio=1.5):
    ''' generate probabilities, stating the first elements should occur more '''
    lambdas = np.array([1 / (ratio ** i) for i in range(n)], dtype=float)

    # Normalize lambdas to ensure the sum equals 1
    lambdas /= np.sum(lambdas)
    return lambdas
```

-

- Daca vom considera o iteratie ca fiind alegerea unui singur succesor, atunci in cadrul unei iteratii simularea ajunge pana la o stare finala, indicata astfel: nu mai exista succesori sau s-au asignat toti studentii (sunt cam acelasi lucru in definintiv, dar este un safety measure)
- Calcul reward:
  daca avem restrictii soft incalcate => reward = (-10) * nr_restrictii_soft;
  Altfel orarul este quasi-perfect => reward = 300.
  Nu bagam in calcul numarul de restrictii hard pt ca: din constructia starilor nu se poate genera ceva ce „strica hard”, iar o stare finala ar avea cam toti studentii asignati
- Pentru alegere succesor, folosim UCT ca la laborator

d) Optimizare fata de laborator

- Aici avem un singur jucator care joaca pentru el, deci nu mai avem nimic legat de next player sau ceva de proportionalitate
- Fiind un spatiu de stari foarte mare, bagam in seama numai primii maxim 15 potentiali vecini, calculati dupa cat de mult se imbunatateste starea. Si, pe langa aceasta, le asociem si probabilitati de explorare. Teoretic puteam sa nu luam doar „primii 15”, ci sa ii luam pe toti cu o probabilitate (oricum ar fi fost absurd de mici pentru numerele de mai incolo), dar ar fi insemnat, in implementarea mea, sa ridic un nr la o putere foarte mare si sa am eroare de Overflow (s-a intamplat)

e) Outputs

	DUMMY	MIC	MEDIU	MARE	INCALCAT	BONUS
HARD	0	0	0	0	0	0
SOFT DEFAULT	0	0	0	0	0.8	0
PAUZE	N/A	N/A	N/A	N/A	N/A	4.7

Pentru orarele dummy, mic, incalcat si bonus, am facut o medie dupa vreo 10-11 rulari. MCTS nu o ia intotdeauna pe calea optima, dar are rezultate foarte bune, multumita probabilitatilor asociate. Daca era un choice complet random, atunci foarte posibil sa nu fi gasit nici macar acoperire completa.
Cea mai buna rulare de bonus a fost de o pauza neindeplinita. Bineinteles, cu buget mai mare ar fi fost sanse si mai mari de a gasi mai putine restrictii

3 – Analiza individuala si comparativa

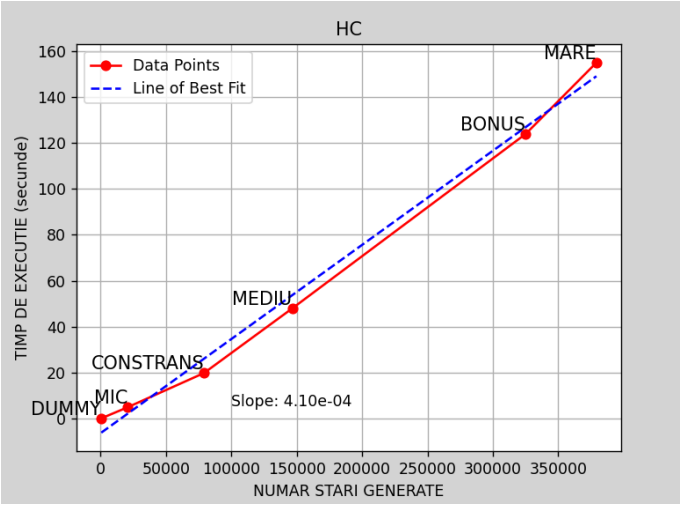
Exista o diferenta fundamentala intre cei 2 algoritmi (in implementarea mea) din punctul de vedere al generarii starilor urmatoare:

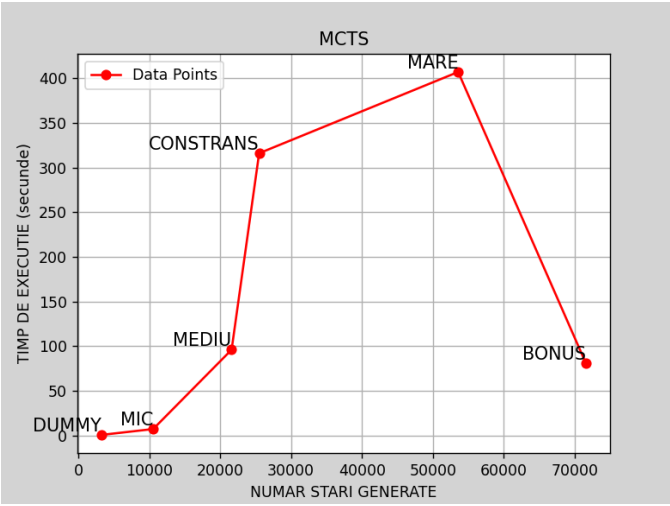
- Pentru HC -> si vecinii care nu vor fi vizitati sunt generati (instante ale clasei State, cu creare si actualizare completa a structurilor interne)
- Pentru MCTS -> vecinii care sunt doar luati in considerare NU sunt generati, ci doar li se calculeaza costul in prealabil. Bineinteles, cand se viziteaza, sunt generati.

De ce o diferenta in abordare?

- Initial era aceeasi abordare si la MCTS, insa, pentru testele mari (Bonus mai ales), o rulare dura 17 min, in conditiile in care nu simulam pana la final, ci pana cand se completa urmatoarea materie. Dupa schimbarea abordarii (ceea ce implica virtually doar eliminarea a 3 deepcopy-uri pt fiecare stare luata in considerare), timpul, pe **aceiasi parametri si aceeasi conditie de oprire la simulare** s-a redus la ~40s. Deci un **speedup de ~25x**. M-a surprins si pe mine ca atat de mult afectau deepcopy-urile. Acum aveam room for trial, asa ca simularea se duce acum pana cand se umple orarul.
- Cum la HC timpii erau acceptabili, n-am mai schimbat abordarea si acolo
  ⇨ Nu are prea mult sens compararea inter-algoritm a timpilor de executie standalone.

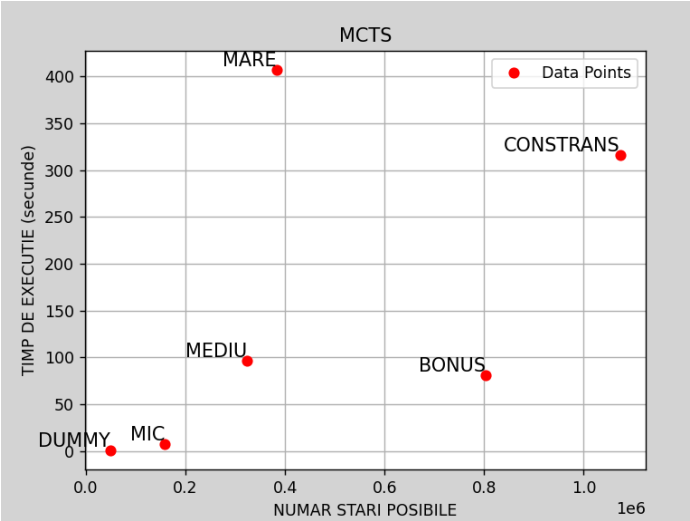
TIMP functie de NR STARI GENERATE (pe care am aplicat deepcopy)





Putem observa ca timpul de executie in cadrul MCTS este mai mare si nu depinde aproape liniar de numarul de stari pe care le generam (adica doar de overhead-ul de deepcopy-uri). Insa, la MCTS timpul creste mai rapid in functie de dimensiunea problemei.

Daca analizam, insa, <u>numarul de stari posibile (si vizitate si nevizitate) si ratio-ul (pe care il detaliez ulterior)</u>. Bugetele sunt, in mare parte, similare



	DUMMY	MIC	MEDIU	MARE	INCALCAT	BONUS
RATIO	2.3	2	2	1.5	2	2.3

Putem observa 2 lucruri:

1. Acum se poate spune ca majoritatea punctelor, pot fi aproximate cu o linie
2. Mai mult, daca tinem cont ca cele de pe linia imaginara au acelasi ratio, iar celelalte sunt cu atat mai departate cu cat diferenta de ratio este mai mare, chiar putem afirma ca dependenta este aproape liniara

RATIO-ul la MCTS: reprezinta cu ce rata scad probabilitatile de explorare cu cat avansam in lista de optiuni de vecini. Cu cat ratio-ul este mai mare, fiind exponentiere, cu atat sunt sanse mai mari sa aleaga primele elemente. Daca ratio-ul ar fi 1 (minimul logic), atunci toate posibilitatile ar avea probabilitate normala, echivalentul unui choice default.

Am incercat mai multe posibilitati si am ajuns la concluzia ca 2 este ok. La testele lejere (cu mult wiggle room), cum ar fi MARE, merge si un 1.5. La CONSTRANS-INCALCAT si BONUS a trebuit serios testat cu mai multe ratio-uri, preferabil mai mari. Altfel o lua pe cai mult prea nefavorabile in timpul simularii.

CALITATE A SOLUTIEI:

Reiau tabelele puse anterior (MCTS agregare a cate 10 rulari):

HC:

	DUMMY	MIC	MEDIU	MARE	INCALCAT	BONUS
HARD	0	0	0	0	0	0
SOFT DEFAULT	0	0	0	0	0	0
PAUZE	N/A	N/A	N/A	N/A	N/A	5

MCTS:

	DUMMY	MIC	MEDIU	MARE	INCALCAT	BONUS
HARD	0	0	0	0	0	0
SOFT DEFAULT	0	0	0	0	0.8	0
PAUZE	N/A	N/A	N/A	N/A	N/A	4.7

- Ambii algoritmi duc spre **solutii bune, spre foarte bune**. Intr-adevar MCTS ia mai mult timp, depinzand atat de buget, cat si daca nimereste un drum bun spre final (din experienta, daca o lua pe o cale nefavorabila dura mai mult).

- Se poate observa ca in cazurile foarte stranse, de genul **CONSTRANS**, mcts nu gaseste intotdeauna calea optima, variind intre 0 si 3-4 constrangeri incalcate. In functie de buget si de ramura pe care o alege la choice, poate gresi. Nu exista o singura solutie de cost 0, in timpul testelor am gasit vreo 5-6 si unele sunt si salvate.

- Si in cazul **MIC** s-a mai intamplat sa dea cate 1 soft incalcat, dar rar. Chiar niciodata in agregarile facute aici.

- Pentru **BONUS**: avand in vedere ca trebuia sa am o abordare mai buna in calculul costurilor pentru pauze si, cum am spus anterior, pentru abordarea mea de la hill climbing „functia” ar trebui sa fie cat de aproape de convexa se poate, se poate observa ca MCTS poate obtine rezultate chiar mai bune, nemergand mereu pe prima optiune. Nu tin minte sa fi scos 0 complet, dar o singura pauza sau un singur interval nefavorabil am obtinut sigue, fiind un exemplu si in „custom_outputs”.