Inteligenta Artificiala

Tema 1: Orar

Dan Alin-Constantin 331CC

Introducere:

Timp : 35h

Dificultatea cea mai mare pe care am intalnit-o la aceasta tema este cel de a face overfitting pe exemplele de testare si de a reduce din stari la algoritmul MCST.

Am ales sa folosesc multe structuri de date pentru a retine datele din fisierul YAML, dar in special dictionare (hash table) pentru ca timpul de cautare al unui element este O(1).

Am facut un compormis in cee ace consta rularea mai rapida si cu incalcari soft ale orarului decat una care sa intoarca un rezultat fara constrangeri de nici un fel nici hard nici soft dupa o perioada mai lunga de timp.

Descriere reprezentare stare initiala:

```
def generate_first_state():
    state = {zi: {eval(interval): {sala: None for sala in sali} for interval in intervale} for zi
in zile}
    return state
```

Am generat pentru fiecare zi , interval si sala o intrare None (adica niciun curs nu a fost asignat pentru acea pozitie din orar)
Zile = reprezinta zile din yaml => yaml_data[‘Zile’]
Intervale = reprezinta intervale din yaml => yaml_data[‘Intervale’]
Sali = reprezinta salile din yaml => yaml_data[‘Sali’]

In concluzie o stare este de tipul [zi][interval][sala] = (profesor, materie)

timetable[zi][interval][sala] = (profesor, materie)

Cum am tinut cont de restrictii soft si hard?

Pentru inceput o sa va prezint ce date contine clasa mea Stare

```
def __init__(
        self,
        timetable : dict[str, dict[tuple[int, int], dict[str, tuple[str, str]]]] | None = None,
        conflicts: int | None = None,
        ore_profesori: dict[str, int] | None = None,
        studenti_materie: dict[str, int] | None = None,
        profesor_interval: dict[str, dict[str, list[str]]] | None = None,
    ) -> None:

        self.timetable = timetable if timetable is not None \
            else generate_first_state()
        self.nconflicts = conflicts if conflicts is not None \
            else (100 * total_unassigned_students)
        self.ore_profesori = ore_profesori if ore_profesori is not None \
            else {profesor : 0 for profesor in profesori}
        self.studenti_materie = studenti_materie if studenti_materie is not None \
            else {subject : 0 for subject in studenti_materii}
        self.profesor_interval = profesor_interval if profesor_interval is not None \
            else {zi : {interval : [] for interval in intervale} for zi in zile}
```

Timetable este generate_first_state() daca nu ii dau ceva ca parametru, adica orarul gol
Nconflicts este numarul total de student neasignati la cursuri * 100
Ore_profesori este este un dictionar de forma {“nume_profesor”: ore_predate}
Studenti_materie este un dictionar de forma {“materie”: numar_studenti_asignati}

Euristicile pe care am creat-o pentru a nu avea constrangeri hard in realizarea tabelului sunt am sortat in array-ul profesori, profesorii in functie de numarul de amterii pe care il predau crescator si materiile din Sali le-am sortat crescator in functie de numarul de aparitii a materiei respective in sali.

def get_next_states(self):

In functia aceasta am generat toate starile posibile in ordinea euristicii de mai sus pe care ati dat-o si voi in exemplul din tema care nu incalca nici o restrictive hard inafara de cea cu numarul de student neasignati toate miscarile care sunt valabile le dau apply_move()

available_moves.append(self.apply_move((zi, interval, sala, profesor, materie)))

apply_move imi calculeaza conflictele si intoarce o stare pe care o bag in lista available_move[]

def apply_move(self):

Metoda pe care am utilizat-o eu este mult mai eficienta, pentru ca nu calculez conflictele parsand tot tabelul ci pur si simplu adun sau scad de la starea precedenta.

Cum functioneaza: pentru inceput numarul de conflicte este numarul de studenti neasignati * 100 si apoi incep sa dau update prin orar sa maresc numarul de ore la profesor sa bag profesorul pe intervalul respective si sa adun numarul de studenti din sala respective in dictionarul care tine cont de asta. Si formula finala este practice numarul de conflicte care erau inainte plus cele soft – 100 * numarul de studenti adaugati. Numarul 100 este pur orientativ pentru ca eu consider ca cele soft pot fi inculcate si le-am pus weight 1 si pentru fiecare student care trebuie sa fie assignat fiind mai important decat o dorinta a unui profesor i-am dat weigh-ul 100.

Functia is_final() care returneaza o valoare bool daca starea e finala sau nu ce reprezinta ca o stare e finala . Ce reprezinta ca stare e finala? Nu mai exista get_next_states() sau ca numarul de student neasignati este egal cu 0.

Functia display() printeaza orarul cu functia din utils

Functia clone() face deepcopy la o stare in alta stare aceasta functie o folosesc in apply_move() pentru a face deepcopy la o stare noua pentru a testa in HillClimbing sau in MCTS care este starea mai buna.

Hill Climbing

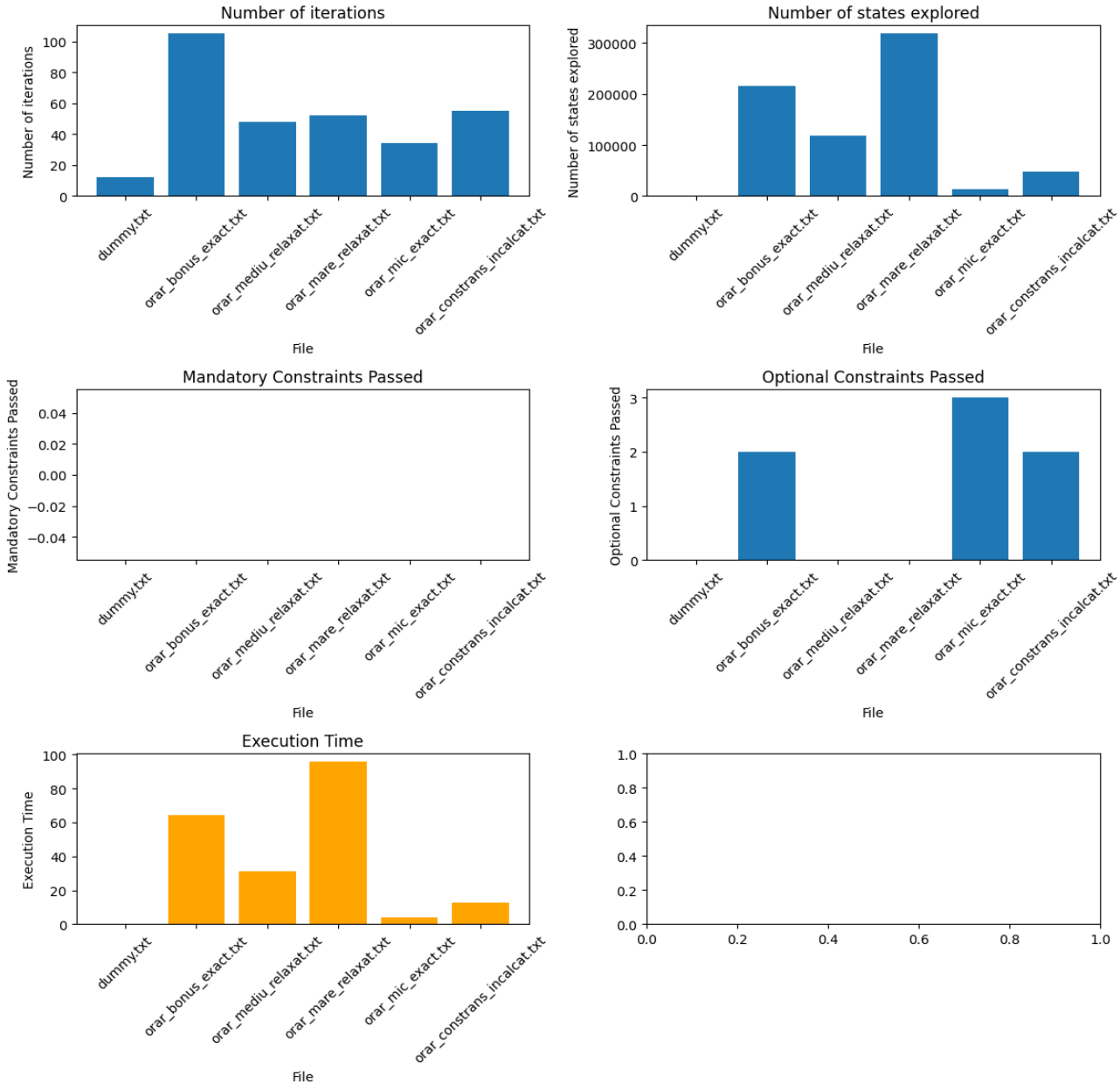
Este o metoda simpla de optimizare folosita în problemele de cautare locala. Scopul sau este de a gasi o solutie cat mai buna (de obicei, o solutie optima locala) intr-un spatiu de cautare prin explorarea vecinilor. Metodele de optimizare pe care le-am folosit eu sunt de a nu baga stari care nu indeplinesc conditiile hard si aceea de a calcula conflictele in pasul respective de a nu parcurge tot tabelul pentru a calcula conflictele. Cautarea conflictelor este in O(1). Acesta este un algoritm determinist cum se observa si la mine daca rulati de mai multe ori o sa va dea acelasi raspuns.

Monte Carlo Tree Search

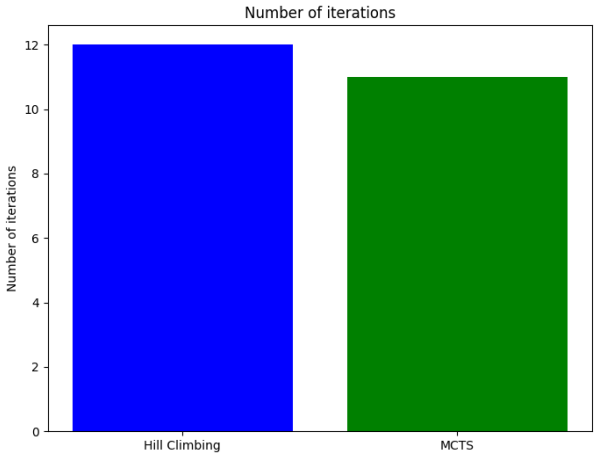
Monte Carlo Tree Search (MCTS) este o metoda de cautare si luare a deciziilor utilizata in spatii de decizie complexe, cu informatii incomplete sau decizii secventiale. Acesta se bazeaza pe ideea de a efectua simulari aleatoare si evaluarea lor pentru a ghida procesul de luare a deciziilor catre solutii bune. Algoritmul nu e determinist.

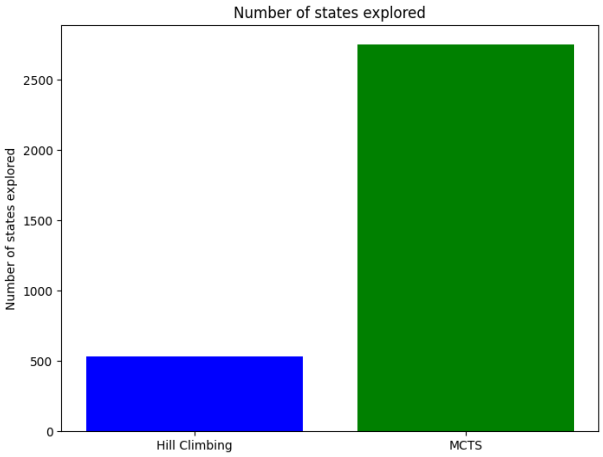
Acestea sunt 5 grafice care compara rezultatele celor 6 teste pentru fiecare output care se gaseste in fisierul cu numele fiecarui algoritm, respective (HillClimbing, MCTS).

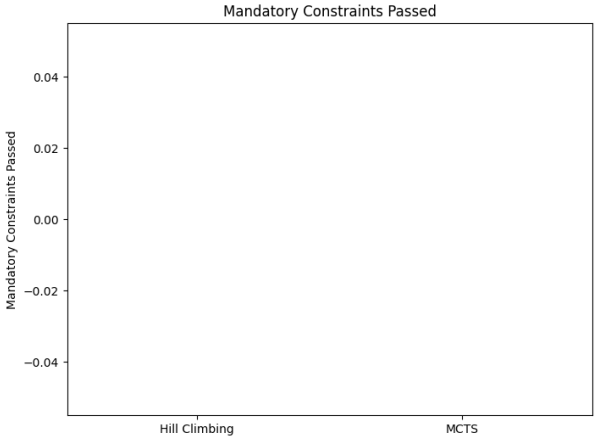
Un exemplu de continut de fisier este:
Hill Climbing Algorithm
Test File: ./inputs/dummy.yaml
Number of iterations: 12
Number of states explored: 530
Mandatory Constraints Passed: 0
Optional Constraints Passed: 0
Execution Time: 0.03631401062011719 seconds
+ Orarul final doar ca se vede urat in PDF

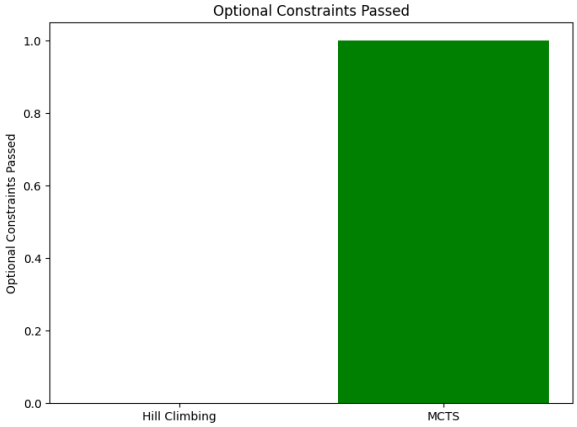


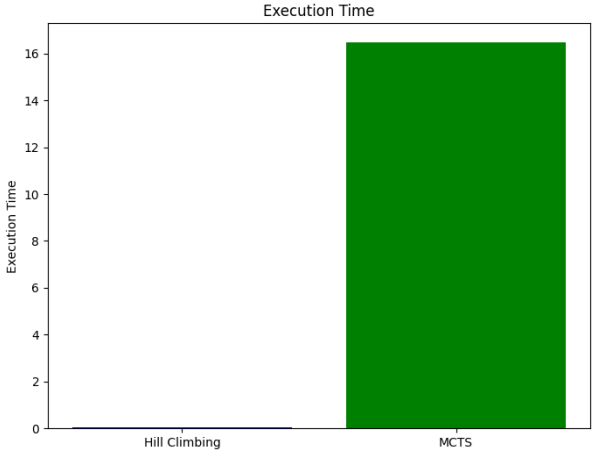
Comparatie HillClimbing si MCTS











Dupa cum se observa in grafice si si in folder-ul de output pentru fiecare algoritm. Algoritmul MCTS este rulat cu mult mai greu pentru ca el trece in mai multe stari decat Hill Climbing. Cu un buget de 50 a reusit sa imi scoata un orar cu 0 constrangeri hard si 2 soft doar ca acesta a rulat in 16 sec in timp ce hill climbing-ul a rulat in 0.03 sec. In schimb pentru testele mai mari el vca gasi o solutie mult mai buna decat Hill Climbing doar ca va dura mult mai mult. Ca acest algoritm sa functioneze mai bine trebuia sa facem reducere de stari doar ca aici nu am stiut exact ce stari nu ne ajuta.