Tema 1 IA

Drăgușin Daniela-Alexandra
Facultatea de Automatică și Calculatoare
Grupa 331CA
daniela.dragusin@stud.acs.upb.ro

30 aprilie 2024

1 Introducere

Tema presupune generarea unui orar in care să nu existe conflicte, sau în cazuri excepționale, în care să existe cât mai puține conflicte.

Prin conflicte ne referim la două tipuri de constrângeri încălcate:

- constrângeri obligatorii (un orar valid trebuie să le respecte)
- constrângeri opționale (este de preferat să nu fie încălcate, însă încălcarea acestora nu afectează validitatea orarului)

În cele ce urmează voi descrie modalitatea mea de rezolvare a problemei.

2 Rezolvarea problemei cu un algoritm de tip Hill-Climbing

2.1 Structura unei stări

Am reprezentat starea ca o clasă cu proprietățile: input_data, schedule, conflicts_number și teacher_assignments care reprezintă în ordine: datele citite din fișierul de intrare, orarul ce reprezintă starea, numărul de conflicte din orar, modul în care sunt asignați profesorii in funcție de zi și interval.

Numărul de conflicte este definit ca suma dintre numărul de constrângeri obligatorii încălcate și numărul de constrângeri opționale încălcate.

```
INTERVALS = 'Intervale'
DAYS = 'Zile'
SUBJECTS = 'Materii'
TEACHERS = 'Profesori'
CLASSROOMS = 'Sali'
CAPACITY = 'Capacitate'

class State:
    def __init__(self,
        input_data,
        schedule: dict | None = None,
        conflicts: int | None = None,
        teacher_assignments: dict | None = None) -> None:

        self.input_data = input_data

        self.schedule = schedule if schedule is not None else\
                        self.generate_schedule(self.input_data[INTERVALS],
                        self.input_data[SUBJECTS], self.input_data[TEACHERS],
                        self.input_data[CLASSROOMS], self.input_data[DAYS])

        self.conflicts_number = conflicts if conflicts is not None else self.
                                                    compute_conflicts()

        self.teacher_assignments = teacher_assignments if teacher_assignments\
                                   is not None else self.compute_teacher_assignments()
```

Pentru calculul conflictelor obligatorii am folosit funcția check_mandatory_constraints din fișierul check_constraints.py. Pentru calculul conflictelor opționale am implementat o funcție ce parcurge clasele din orar în care sunt alocate ore și verifică dacă profesorul ce ține cursul preferă/ nu preferă să predea în acea zi/interval.

Algoritmul Hill-Climbing pornește de la o stare inițială în care sunt satisfăcute toate constrângerile obligatorii. Astfel, pentru a construi starea inițială se va apela constructorul in următoarea forma:

```
# Creez starea ini ial
init_state = State(input_data)
```

Am ales această variantă, întrucat pornirea de la o stare inițială în care orarul era gol și umplerea ulterioară a acestuia cu toate posibilele combinații de zile, intervale, materii și profesori ar fi generat un număr extraordinar de mare de stări, ceea ce ar fi dus la timpi de execuție foarte mari.

2.2 Generarea stării inițiale

Pentru a genera starea inițială mă folosesc de funcția generate_schedule() din clasa State.

În timp ce generez starea inițială mă asigur de respectarea constrângerilor obligatorii.

Generarea presupune parcurgerea materiilor și încercarea de a aloca toți studenții ale acelei materii în săli. Acest lucru presupune alegerea aleatoare a unei zile, a unui interval, a unei săli și a unui profesor. Bineînțeles, alegerea oricăror dintre cele menționate va reduce numărul de posibilități pentru următoarele. Spre exemplu, sala va fi aleasă random dintre sălile disponibile în ziua și intervalul alese, profesorul va fi ales dintre cei care sunt disponibili să predea în acea zi/interval și care pot preda această materie.

2.3 Implementarea algoritmului Hill-Climbing

Pentru o rată de succes de 100% am folosit algoritmul Random Restart Hill-Climbing din laborator cu număr maxim de restart-uri = 50 și număr maxim de iterații per rulare = 100.

Am folosit algoritmul Hill-Climbing din laborator care pentru o stare curentă, generează stările vecine ale acesteia și alege să continue cu cea care încalcă cele mai puține conflicte. O modificare adusă algoritmului este că am limitat numărul de stări vecine generate la:

```
MAX_NUMBER_GENERATED_STATES = 10
```

O altă modificare este faptul că funcția de generare a stărilor vecine get_next_states() generează doar stări vecine favorabile, pentru a reduce numărul de stări create și implicit pentru a reduce timpul de execuție. Stările vecine sunt create prin interschimbarea sălilor alese random fie cu săli goale, fie cu săli în care se țin deja cursuri. Interschimbarea are loc doar dacă nu se încalcă constrângerile obligatorii și dacă noua structură a orarului rezolvă mai multe conflicte opționale decât cea curentă.

Algoritmul presupune alegerea celei mai bune stări vecine din cele MAX_NUMBER_GENERATED_STATES stări vecine generate și oprirea în cazul în care nu se găsește o stare mai bună.

2.4 Rezultate Hill-Climbing

Rezolvarea mea nu produce soluții de cost 0 pe toate testele. Am ales ca rezolvarea mea să producă un număr minim de conflicte, pentru a avea un timp de execuție rezonabil, în detrimentul unei soluții de cost 0 care ar fi avut un overhead temporal. Consider următoarele cauze:

- Faptul că nu dau posibilitatea algoritmului să meargă pe căi mai "proaste", pentru că eu aleg de fiecare dată drumul mai bun.
- Faptul că generez un număr limitat de stări următoare.

Am rulat de 20 de ori algoritmul pentru fiecare test de intrare și am creat statistici pentru numărul de stări generate, timpul total de execuție, numărul total de iterații efectuate de algoritm și pentru numărul de conflicte ale stării inițiale/numărul final de conflicte ale soluției. În toate testele se observă cum timpul de execuție este direct proporțional cu numărul de stări generate și numărul de iterații.

2.4.1 Rezultate dummy.yaml

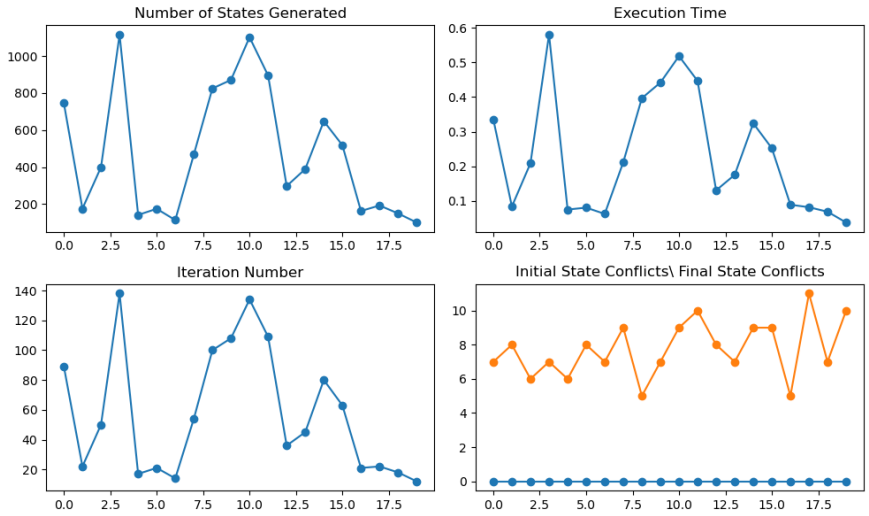

Figura 1: Rezulatate rulare algoritm Random Restart Hill-Climbing cu input dummy.yaml.

Se poate observa că se ajunge la o soluție de cost 0 în 20/20 rulări, cu un timp mediu de execuție de 0,23 secunde și un număr mediu de aproximativ 474 stări generate.

De asemenea, se poate observa că soluția de cost 0 se obține în urma unei stări inițiale ce are o valoare a costului cuprinsă între 5 și 11.

2.4.2 Rezultate orar_mic_exact.yaml

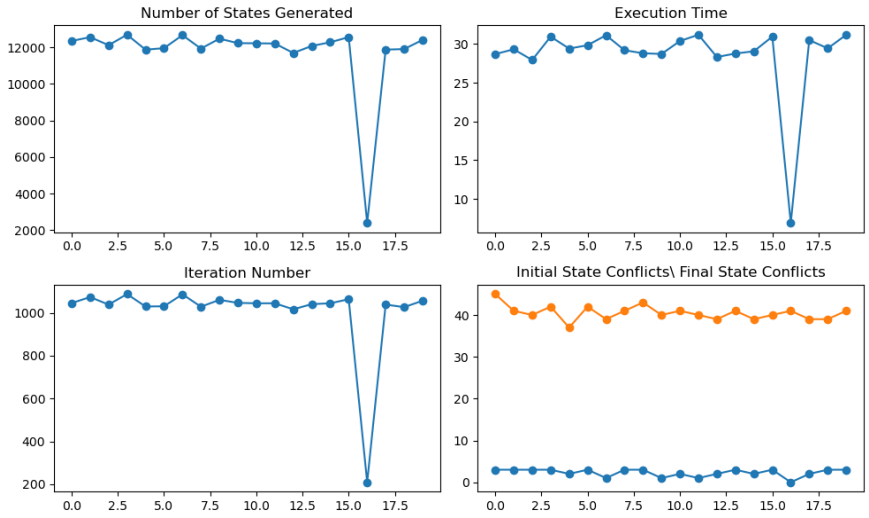

Figura 2: Rezulatate rulare algoritm Random Restart Hill-Climbing cu input orar_mic_exact.yaml.

Se poate observa că se ajunge la o soluție de cost 0 în 1/20 rulări, la o soluție de cost 1 în 3/20, iar la o soluție de cost 2 în doar 5/20 rulări. Costul mediu obținut pe cele 20 de teste este 2,3. Se remarcă un comportament bun de rezolvare al algoritmului, deoarece stările inițiale au un cost cuprins între 39 și 45 (cu o medie de 40,5).

În ceea ce privește stările generate, observăm o medie de nu mai puțin de 11,718 stări, iar timpul de execuție este unul pe măsură..

2.4.3 Rezultate orar_mediu_relaxat.yaml

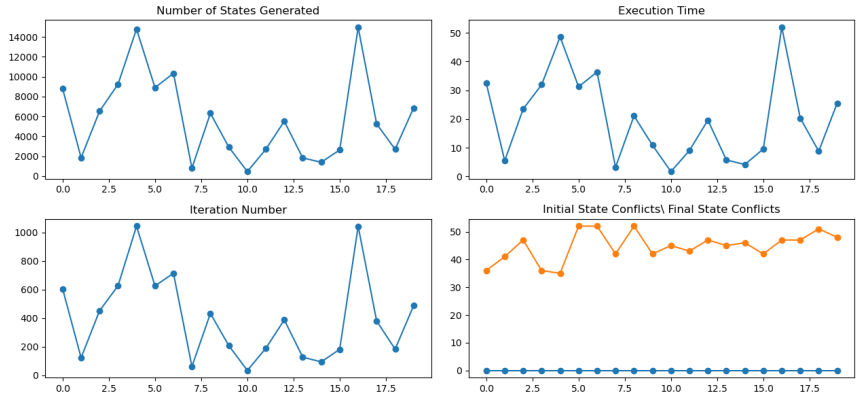

Figura 3: Rezulatate rulare algoritm Random Restart Hill-Climbing cu input orar_mediu_relaxat.yaml.

Se observă obținerea unei soluții de cost 0 în toate cele 20 de rulări, cu un timp mediu de execuție de 20,5 secunde și un număr mediu de 5735 stări generate.

Corectitudinea algoritmului se observă prin faptul că se pornește de la stări inițiale cu orare ce încalcă un număr de constrângeri cuprins între 35 și 52, și se ajunge la stări cu orare ce nu introduc niciun conflict..

2.4.4 Rezultate orar_mare_relaxat.yaml

Figura 4: Rezulatate rulare algoritm Random Restart Hill-Climbing cu input orar_mare_relaxat.yaml.

Orarul mare produce în medie soluții de cost 6, pornind din stări inițiale cu costuri cuprinse între 70 și 102 (în medie 83). Timpul mediu de execuție este de aproximativ 182 de secunde, iar numărul mediu de stări generate este aproximativ egal cu 32,400.

2.4.5 Rezultate orar_constrans_incalcat.yaml

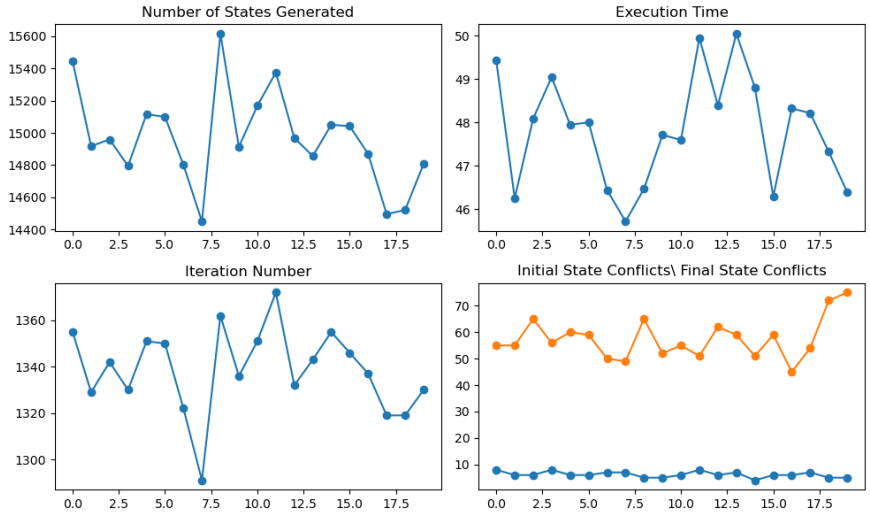

Figura 5: Rezulatate algoritm Random Restart Hill-Climbing cu input orar_constrans_incalcat.yaml.

Orarul cu constrângeri încălcate produce în medie soluții de cost 6, pornind din stări inițiale cu costuri cuprinse între 45 și 175 (în medie 58). Timpul mediu de execuție este de aproximativ 47,8 secunde, iar numărul mediu de stări generate este aproximativ egal cu 14,960

3 Rezolvarea problemei cu un algoritm de tip A*

3.1 Structura unei stări

Starea este reprezentată asemănător celei folosite la rezolvarea cu ajutorul algoritmului Hill-Climbing, dar are două proprietăți în plus: teacher_assignments_number și subjects_assignments. Prima este un dicționar care reține pentru fiecare profesor numărul total de cursuri pentru care este asignat. A doua proprietate este un dicționar în care cheile sunt materiile și valorile sunt acoperirea fiecărei materii în orarul corespunzător stării curente.

Calculul numarului de conflicte din orar se realizeaza la fel ca la Hill-Climbing.

```
class State:
    def __init__(
        self,
        input_data,
        schedule: dict | None = None,
        teacher_assignments: dict | None = None,
        teacher_assignments_number: dict | None = None,
        subjects_assignments: dict | None = None,
        conflicts: int | None = None ) -> None:

# initializare input_data, schedule, conflicts_number la fel ca la Hill-Climbing

self.teacher_assignments = teacher_assignments if teacher_assignments is not None\
                        else {teacher : {day : {interval : False\
                        for interval in input_data[INTERVALS]}\
                        for day in input_data[DAYS]} for teacher\
                         in input_data[TEACHERS].keys()}

self.teacher_assignments_number = teacher_assignments_number\
                    if teacher_assignments_number is not None\
                    else {teacher : 0  for teacher in input_data[TEACHERS].keys()}

self.subjects_assignments = subjects_assignments if subjects_assignments is not None\
                            else {subject : 0 for subject in input_data[SUBJECTS]}
```

3.2 Starea inițială și generarea stărilor vecine

Am ales ca în cadrul acestei rezolvări starea inițială să fie un orar gol.

Pentru a genera vecinii unei stări am implementat o funcție care ia toate sălile libere din orar și realizează o asignare de tipul (profesor, materie).

```
def get_next_states(self):
    next_states = []
    schedule = self.schedule

    # Iau toate salile libere si le asignez materii si profesori
    for day in schedule.keys():
        for interval in schedule[day].keys():
            for classroom in schedule[day][interval].keys():
                if not schedule[day][interval][classroom]:
                    new_states = self.assign_teacher_subject(day, interval, classroom)
                    next_states = list(next_states + new_states)
    return next_states
```

În funcția assign_teacher_subject(day, interval, classroom) se parcurg toate materiile care nu au fost acoperite încă și care pot fi predate în classroom. Pentru fiecare materie, se aleg profesorii care pot preda acea materie, care nu au deja 7 cursuri asignate și care nu predau deja în day și interval.

Pentru a reduce numărul de stări create, am ales să țin cont în acest pas și de preferințele profesorilor, astfel încât să se creeze stări vecine care respectă și intervalul și ziua preferate de profesori.

3.3 Implementarea algoritmului A*

Am folosit implementarea algoritmului A* din laborator. Modificările aduse sunt:

- Oprirea algoritmului cu timeout dacă trec 240 de secunde și nu este găsită soluția.
- Adaug în dicționarul discovered chei de tipul State și valoare ca fiind costul g de la starea inițială până la starea curentă. Nu am adăugat și părintele, întrucât nu am scopul de a reface drumul parcurs.
- Mi-am creat o euristică proprie care ține cont de numărul de conflicte existente în orarul curent și de produsul dintre numărul de materii neacoperite și clasele goale. Am realizat produsul celor două din urmă deoarece un număr mare de clase goale indică o distanță mare față de soluție, însă odată ce toate materiile sunt acoperite, această valoare devine irelevantă pentru a defini distanța până la soluție. În acest caz, distanța până la un orar fără conflicte este dată doar de numărul curent de conflicte din orar. Odată ce numărul de conflicte ajunge la 0, am ajuns într-o stare finală, iar valoarea euristicii pentru acea stare este 0.

  ```
  def heuristic(state):
      return state.conflicts_number +
          len(state.get_uncovered_subjects()) * state.get_empty_classrooms()
  ```

În implementarea mea, am ales ca vecinii unei stări să fie generați ținând cont atât de constrângerile obligatorii ce trebuie îndeplinite, cât și de preferințele profesorilor, pentru a obține un timp de execuție mai bun, eliminând stările care nu m-ar duce către soluție (un orar fără conflicte).

În cazul în care se generează stări vecine ținând cont doar de constrângerile obligatorii, timpul de execuție și numărul de stări generate sunt mult mai mari.

Algoritmul pornește automat cu constrângeri obligatorii încălcate, deoarece nicio materie nu este acoperită la început, având un orar gol. Aceste conflicte se rezolvă pe măsură ce este populat orarul cu cursuri, dar constrângerile opționale ce au fost încălcate prin așezarea cursurilor în săli nu vor mai putea fi rezolvate exploatând aceeași stare.

Comparație între cele două variante de rezolvare privind timpul de execuție.

	Generare vecini ținând cont doar de constrangeri obligatorii	Generare vecini ținând cont si de constrangerile optionale
dummy.yaml	0,0297 s	0,0092 s
orar_mic_exact.yaml	6,5969 s	1,1082 s
orar_mediu_relaxat.yaml	52,6297 s	23,8795 s
orar_mare_relaxat.yaml	253,6946 s	69,6492 s

Comparatie între cele două variante de rezolvare privind numărul de stări generate.

	Generare vecini ținând cont doar de constrangeri obligatorii	Generare vecini ținând cont si de constrangerile optionale
dummy.yaml	534 stari	159 stari
orar_mic_exact.yaml	22014 stari	4058 stari
orar_mediu_relaxat.yaml	146893 stari	61419 stari
orar_mare_relaxat.yaml	390902 stari	102686 stari

4 Comparația celor doi algoritmi

Valorile pentru rezultatele algoritmului Hill-Climbing reprezintă valorile medii ale rezultatelor obținute în urma celor 20 de rulări ale fișierelor de intrare.

4.1 Comparația timpilor de execuție ai celor doi algoritmi

	Hill-Climbing	A*
dummy.yaml	0,2299 s	0,0092 s
orar_mic_exact.yaml	28,5371 s	1,1082 s
orar_mediu_relaxat.yaml	20,0566 s	23,8795 s
orar_mare_relaxat.yaml	182,3825 s	69,6492 s

4.2 Comparația numărului de stări construite de cei doi algoritmi

	Hill-Climbing	A*
dummy.yaml	474 stări	159 stări
orar_mic_exact.yaml	11718 stări	4058 stări
orar_mediu_relaxat.yaml	5735 stări	61419 stări
orar_mare_relaxat.yaml	32433 stări	102686 stări

4.3 Comparația numărului de restricții încălcate de cei doi algoritmi

	Hill-Climbing	A*
dummy.yaml	0 restricții	0 restricții
orar_mic_exact.yaml	2,3 restricții	0 restricții
orar_mediu_relaxat.yaml	0 restricții	0 restricții
orar_mare_relaxat.yaml	5.95 restricții	0 restricții

Se poate observa cum algoritmul A* ajunge la soluții ce încalcă 0 restricții pe toate cele 4 teste de intrare, în timp ce Hill-Climbing ajunge la stare finală doar pe testele dummy.yaml și orar_mediu_relaxat.yaml.

5 Structura arhivei

Arhiva conține următoarele fișiere:

- **orar.py** fișier ce primește ca argumente in linia de coamanda algoritmul astar sau hc si numele fișierului de intrare
- **hill_climbing.py** implementează logica pentru algoritmul Hill-Climbing
- **astar.py** implementează logica pentru algoritmul A*
- **check_constraints.py**
- **utils.py**
- **inputs** folder cu fișierele de intrare
- **outputs** folder ce conține fișierele de ieșire pentru algoritmii astar si hc
- **refs** folder cu output-urile de referință
- **plot_results_hc** folder ce conține pentru fiecare test imaginea cu plot-urile obținute în urma rulării de cele 20 de ori a algoritmului Hill-Climbing și un fișier cu rezultatele obținute. Aceste fișiere au fost folosite la calcularea valorilor medii pentru timpii de executie, numărul de stări generate, numărul de constrângeri încălcate de starea rezultată, numărul de constrângeri încălcate de starea inițială.