Tema 1 IA

Andra Vlad - 331CC

30.04.2024

1 Descrierea temei

Tema a constat în aplicarea algoritmilor Hill Climbing și Monte-Carlo Tree Search în contextul generării unui orar care respectă atât constrângeri de tip hard, cât și constrângeri de tip soft. Provocarea realizării acestei teme a fost găsirea unui echilibru între o soluție generată în timp util și o soluție care produce costuri soft minime. De asemenea, găsirea unei eursitici potrivite a fost o parte importantă în minimizarea costurilor. Am înțeles mult mai bine, după folosirea celor doi algoritmi, avantajele și dezavantajele folosirii unui algoritm determinist vs. unui algoritm nedeterminist.

1.1 Reprezentarea stărilor

Inițial, am plecat de la generarea unei clase care reținea toate datele despre un orar, însă aceasta nu a fost o reprezentare eficientă, deoarece informațiile despre orar nu se schimbau odată cu fiecare stare. De aceea, am separat reprezentarea în 2 clase: `InformatiiOrar`, în care rețin datele din yaml și `StareOrar`, unde îmi rețin dicționarul cu starea curentă a orarului, în formatul de pretty print folosit în utils.py:

```
{'Luni': {
    (8, 10):
        {'EG324': ('Roxana Gheorghe', 'MS'), 'EG390': ('Elena Gheorghe', 'DS')},
    (10, 12):
        {'EG324': ('Andreea Dinu', 'IA'), 'EG390': None},
    (12, 14):
        {'EG324': ('Pavel Filipescu', 'IA'), 'EG390': ('Roxana Gheorghe', 'DS')}},
'Miercuri':
    {(8, 10):
        {'EG324': ('Roxana Gheorghe', 'MS'), 'EG390': ('Elena Gheorghe', 'DS')},
    (10, 12):
        {'EG324': None, 'EG390': None},
    (12, 14):
        {'EG324': None, 'EG390': None}}}
```

2 Hill Climbing

2.1 Algoritm

Implementăm algoritmul Hill Climbing pentru **optimizarea unei stări inițiale reprezentată de un orar gol**. Algoritmul evaluează costul stării inițiale și caută iterativ să îmbunătățească soluția prin explorarea vecinilor acesteia, selectând starea vecină cu cel mai mic cost ca potențială nouă stare curentă. Procesul se repetă până când se găsește un vecin cu cost 0. Algoritmul are, așadar, o abordare de tip greedy în care încercăm să **îmbunătățim o stare mai proastă**.

Inițial, am gândit problema să plece de la un orar deja construit, care respectă constrângerile hard și să generez stări astfel încât să rezolve constrângerile soft. Problema la această rezolvare a fost faptul că într-o situație cu un cost soft, spre ex. un interval nepreferat de un profesor, algoritmul va genera multe mișcări până o va găsi pe cea care va rezolva conflictul. Deși soluția mă ducea într-un final mereu la un

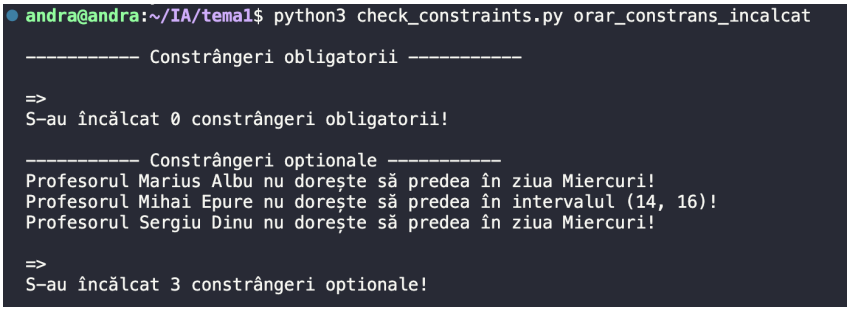

Figure 1: Constrângeri soft încălcate de Hill Climbing pentru orar constrans incalcat

cost hard 0 cost soft 0, nu era deloc favorabilă ca și timp de rulare. Așadar, am ales **plecarea de la un orar gol pe care îl "umplu" cât mai eficient**.

2.2 Generarea stărilor vecine

Funcția de generare vecini pentru Hill Climbing încearcă să încarce un orar gol astfel încât să respecte toate constrângerile hard. Deoarece nu randomizez alegerea variabilelor, stările obținute sunt **deterministe**. Fiind aceleași rezultat la fiecare rulare, nu voi face o comparație între rezultatele obținute.

Abordarea este următoarea:

1. Calculăm numărul de **studenți repartizați** deja în orarul curent, cât și numărul de profi alocați unui curs și care profi sunt alocați în fiecare pereche (zi, interval). Aceste date ne vor ajuta să verificăm respectarea constrângerilor hard din start.
2. Sortăm cursurile care nu au fost acoperite încă dupa minimul de elevi înscriși la curs. Dacă două cursuri au același număr de elevi înscriși la curs, îl alegem pe cel care are mai puțini elevi repartizați. Am ales această **abordare de tip greedy**, în care aleg stările cele mai mici pentru a alege soluția optimă la momentul respectiv.
3. Pentru fiecare curs, alegem dintre sălile în care se pot preda acel curs, sala cu numărul de studenți minim.
4. Alegem profii care pot preda acel curs și alegem cel cu numărul minim de ore deja predate. Eliminăm acele soluții care deja încalcă regula hard ca fiecare prof să predea maxim 7 ore.
5. Pentru fiecare zi și interval, dacă profu respectă regula de a nu preda deja în acel interval, actualizez noua stare generată și o adaug în vectorul în care îmi salvez stările vecine.

Găsirea unei parcurgeri eficiente a necesitat observarea output-urilor de referință, pentru a înțelege în ce ordine ar trebui alese variabilele. Singurul test la care nu se rezolvă costurile soft este `orar_constrans_incalcat`, după cum se poate observa în imagine.

2.3 Euristică

Singura constrângere hard neverificată în timpul calculării stărilor vecine este cea daca toate materiile sunt acoperite sau nu, deoarece este singura care poate fi verificată doar atunci când ajungem la o stare finală. După mai multe încercări de găsire a unei euristici eficiente, am ajuns la următorul calcul:

```
cost = sum(studenti nerepartizati) * 10 + sum(materii nerepartizate) * 150 + cost soft
```

Costul soft este mai mic pentru consrângerile soft pentru a avantaja alegerea unor stări ce duc spre o soluție completă, nu neapărat favorabilă din punct de vedere al preferințelor.

2.4 Concluzii

Algoritmul are timp de rulare buni (output-urile sunt în `outputs_hc`, alături de constrângerile încălcate și timpii de rulare), însă ar putea fi optimizat prin a elimina calcularea dicționarelor auxiliare atunci când ajungem în starea curentă, reducând astfel spațiul temporal al acestor căutări în O(1). Cu toate acestea, nu am mai schimbat codul, deoarece am considerat că ajunge la un trade-off potrivit între găsirea soluției optime și a timpului de rulare. Hill Climbing ne asigură că este un algoritm care **evoluează la fiecare pas**, însă generarea și parcurgerea stărilor în contextul unor orare mai mari influențează mult timpul la inputuri mai mari. De asemenea, euristica a fost necesară să fie una ce diversifică mult spațiul de căutare, deoarece comparativ cu alți algoritmi, Hill Climbing **poate rămâne blocat în minime locale**.

---

3 Monte-Carlo Tree Search (UCT)

3.1 Algoritm

Comparativ cu Hill Climbing care evoluează mereu într-o stare mai bună, MCTS **explorează spațiul de soluții pe baza unui arbore de decizie**, unde selectarea următoarei acțiuni se bazează pe calculul UCT, care echilibrează explorarea arborului în funcție de numărul de vizite al unei stări și cost, pentru a nu rămâne blocat în minime locale. MCTS selectează din ce în ce mai bine stările odată ce crește complexitatea problemei. Așadar, pentru soluții mari ca și input (însă nu cu mai multe preferințe), MCTS este o alegere bună.

3.2 Generarea stărilor vecine

Funcția `selectare_vecini_mcts` din imaginea de pe următoarea pagină eficientizează selectarea vecinilor pentru algoritmul MCTS. Deoarece generarea tuturor vecinilor ar dura mult prea mult în cadru explorării și a simulării, selectăm câțiva din aceștia astfel încât să fim siguri că vom găsi o soluție ce respectă constrângerile hard.

O idee inițială și eficientă a acestui algoritm a fost să selectez **doar acei vecini care au cost soft 0**, deoarece suntem siguri că pentru toate testele noastre există soluții de cost total 0. Însă, deoarece implementarea mea a funcției `generare_vecini` nu reușește să găsească acea soluție de cost soft 0 în Hill Climbing pentru `orar_constrans_incalcat`, nu va reuși nici în MCTS și astfel găsea pentru acest test soluție de cost hard 1 și cost soft 0.

Așadar, pentru obținerea unor rezultate corecte pentru toate testele, am ales să elimin această implementare și să selectez doar primii 10 vecini cu cost minim, care au costul mai mic decât starea curentă. Alegerea doar a primilor 10 vecini eficientizează timpul de căutare și, cu un buget mai mare, reușește să ajungă la soluții decente.

3.3 Pași

Selecția Coborâm în arbore până când ajungem la un orar cu toate materiile acoperite sau la un nod cu acțiuni neexplorate.

Explorarea Construim un nou nod copil dacă găsim unul care nu are toate materiile acoperite.

Simularea Simulăm jocul până când ajungem într-o stare finală sau până când nu mai avem stări vecine.

Propagarea inversă: Propagăm înapoi în arbore rezultatele, incrementând numărul de vizite și decrementând reward-ul cu noul cost, deoarece dorim soluția de cost minim.

3.4 Optimizări și rezultate

Pentru selectarea acțiunilor favorabile, fiind o situație stocastică, folosim o **distribuție de probabilitate softmax**, folosită și în algoritmul de Simmulated Annealing din laborator. Am observat că această optimizare ne aduce mult mai repede la soluția finală. De asemenea, am ales un buget de 30 pentru a avea un echilibru între găsirea unei soluții bune și un timp de rulare decent pe testele mari. Un

```python
# Selectam vecinii care au cost soft 0 si care au un numar mare de materii acoperit.
def selectare_vecini_mcts(orar, initial_state):
    neighbours = generare_vecini(orar, initial_state)
    neighbours.sort(key=lambda neighbor: calculare_cost(orar, neighbor.stare_orar))
    chosen_neighbours = []
    for neigh in neighbours:
        if calculare_cost(orar, neigh.stare_orar) < calculare_cost(orar, initial_state.stare_orar):
            chosen_neighbours.append(neigh)
    return chosen_neighbours[:10]
```

Figure 2: Selectare vecini pentru algoritmul MCTS

improvement care ar putea fi adăugat ar fi să selectăm bugetul în funcție de mărimea testului pe care rulăm, oferind un buget mai mare pe testele mai constrânse și mici și un buget mai mic pe testele mai relaxate și mari. Un alt improvement ar fi simularea până într-un punct mai devreme decât cel final. Am încercat să simulez până când orarul meu acoperă minim 3 materii, însă improvement-ul nu a fost unul semnificativ și am ales să extind simularea până la găsirea unui orar final.

De asemenea, deoarece este un algoritm nedeterminist, am rulat de mai multe ori testele și acestea sunt **rezultatele obținute în format [cost hard, cost soft] și un timp aproximativ**:

- [0, 0] în 50 de rulări `dummy` în timp de rulare de 0.5 sec
- [0, 0] x 8, [0,3] și [0,4] în 10 rulări `orar_mic_exact` în timp de rulare 3 min 43 sec
- [0, 0] în 10 rulări `orar_mediu_relaxat` în timp de rulare de 1 minut
- [0, 0] în 10 rulări `orar_mare_relaxat` în timp de rulare de 8 min 42 sec
- [0, 6], [0,4], [0,6], [0, 5], [0,6] în 5 rulări `orar_constrans_incalcat` în timp de rulare de 13 min 16 sec
- [0, 0] în 10 rulări `orar_bonus_exact` în timp de rulare de 3 min 7 sec.

Menționez că acești timpi sunt estimativi, diferind de la rulare la rulare semnificativ în funcție de nodul selectat cu choice.

---

4 Compararea celor doi algoritmi

Pentru compararea rezultatelor și a timpilor, ne putem uita în `output_hc` și `output_mcts`. Cea mai mare diferență o putem observa la testul `orar_constrans_incalcat`, unde MCTS dă rezultate de cost 4-6 în aproximativ 13 minute, în timp ce Hill Climbing găsește o soluție de cost 3 în doar 15 secunde! Pe testele mai relaxate obținem timpi mai buni la MCTS, deoarece ajunge la soluția finală mult mai repede și nu mai trece prin foarte multe stări ca și Hill Climbing. Cu toate acestea, pe orarele cu constrângeri soft timpul este mult mai crescut. Așadar, MCTS este mai bun pentru situații unde avem posibilități multe de alegeri și puține constrângeri soft. Pentru mai multe constrângeri soft, nefiind niște constrângeri rigide, Hill Climbing poate fi mai adecvat pentru acest tip de orare și putem observa și faptul că găsește soluții de cost mai bun mai rapid.

Din punct de vedere al numărului de iterații, Hill Climbing trece prin mult mai multe decât MCTS. MCTS se îndreaptă mai rapid spre o soluție bună, dar acesta durează mult timp din cauza simulărilor.

References

- **Search-HillClimbing.ipynb** - funcția de Hill Climbing din laborator
- **Search-HillClimbing.ipynb** - Simmulated Annealing pentru distribuția de probabilitate softmax
- **Search-MCTS.ipynb** - Scheletul de laborator
- **check_constraints.py** - Funcțiile de calcul pentru constrângerile încălcate