Pocket Cube

Andrei Dugăeșescu, Andrei Olaru, Mihai Nan

ultima actualizare: 4 decembrie 2023
deadline: 10 decembrie 2023

1 Introducere

Ne propunem să investigăm și să comparăm algoritmi pentru rezolvarea cubului lui Rubik[1]. Cum cubul obișnuit (3×3×3) implică un număr de stări prea mare, vom lucra cu varianta de 2×2×2, sau "Pocket Cube"[2].

Cubul lui Rubik de dimensiune 2×2×2 – pe care îl vom numi pe scurt "Cubul" – are 6 fețe, fiecare formată din 4 *stickere* (sau "mini-fețe"). Vom numi o mutare o rotire cu 90° a uneia dintre cele 3 jumătăți a cubului dintre jumătățile din față, din dreapta, și de deasupra. Rotirile celorlalte 3 jumătăți este echivalentă cu rotirea jumătății opuse. Astfel, fiecare stare a cubului are 6 stări vecine. O stare finală este o stare în care cubul este rezolvat – oricare stare în care pentru fiecare față, toate mini-fețele de pe acea față au aceeași culoare.

2 Pocket cube

Pentru mecanica cubului vom folosi pachetul `pocket_cube`. Vom folosi pentru reprezentarea cubului clasa `Cube` din acest pachet și pentru mutări clasa `Move`. Mișcările disponibile pentru amestecarea/rezolvarea cubului (urmărind notațiile oficiale [3]) sunt:

- În sensul acelor de ceasornic, 90 de grade - R (Right Face - fața din dreapta), F (Front Face - fața frontală), U (Upper Face - fața superioară)
- În sens invers acelor de ceasornic, 90 grade - R' (opusul lui R), F' (opusul lui F), U' (opusul lui U)

Logica asociată reprezentării și parsării acestor mișcări fie ca string-uri, numere sau instanțe ale clasei `Move` se află în fișierul `moves.py`.

În cadrul implementării, principalele funcționalități de interes sunt următoarele:

- `move` - Aplică o mutare asupra stării curente a cubului și întoarce un nou cub aflat în starea indusă de mutarea respectivă.
- `clone` - Întoarce o nouă instanță a cubului păstrând aceeași configurație a stării.
- `hash` - Returnează un ID unic pentru starea curentă a cubului (aceeași stare va produce același ID).
- `render` - Afișează o reprezentare grafică 2D a stării curente a cubului.

[1] Cubul lui Rubik: https://en.wikipedia.org/wiki/Rubik%27s_Cube
[2] Pocket Cube: https://en.wikipedia.org/wiki/Pocket_Cube
[3] WCA notații: https://www.worldcubeassociation.org/regulations/#article-12-notation

(a) Configurația inițială a cubului (b) Efectul aplicării unei mișcări de tip R

Figura 1: Exemplu de aplicare a unei acțiuni asupra cubului

- `render3D` - Afișează o reprezentare grafică 3D a stării curente a cubului.
  Notă: pentru vizualizarea *interactivă* a cubului în forma 3D trebuie să folosiți directiva `%matplotlib notebook`.
- `render3D_moves` - Creează o animație (GIF) pentru afișarea unei serii de mișcări (date ca argument) pornind de la o stare a cubului Rubik (de asemenea dată ca argument).

În implementarea cubului de 2×2×2, starea curentă a cubului (`state`) este reținută drept un `numpy array` de lungime 24 în care fiecare grupare de 4 valori consecutive corespunde unei fețe a cubului (mai multe detalii sunt incluse în `constants.py`). Astfel, starea unui cub rezolvat (imaginea (a) din Fig. 1) va avea reprezentarea:

```
array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2,
       3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5])
```

care corespunde la:

```
array([0, 0, 0, 0,    -> fata albastra
       1, 1, 1, 1,    -> fata rosie
       2, 2, 2, 2,    -> fata verde
       3, 3, 3, 3,    -> fata portocalie
       4, 4, 4, 4,    -> fata galbena
       5, 5, 5, 5]) -> fata alba
```

iar în urma aplicării unei mișcări pentru fața din dreapta a cubului - R (imaginea (b) din Fig. 1), starea cubului va fi:

```
array([0, 5, 0, 5, 1, 1, 1, 1, 4, 2, 4, 2,
       3, 3, 3, 3, 4, 0, 4, 0, 5, 2, 5, 2])
```

2.1 Cazuri de test

Vom folosi următoarele 4 cazuri de test (disponibile în fișierul `tests.py`), de complexitate crescătoare. Fiecare dintre cele 4 stringuri poate fi pasat direct constructorului `Cube`:

```
case1 = "R U' R' F' U"
case2 = "F' R U R U F' U'"
case3 = "F U U F' U' R R F' R"
case4 = "U' R U' F' R F F U' F U U"
```

3 Cerințe

Se cer diverse implementări de strategii de căutare și comparații între rezultatele acestora.

3.1 Cerința 1: A* și Bidirectional BFS

Pocket Cube are aproape 3.7×10^6 stări posibile. Pentru a putea gestiona numărul mare de stări, avem două variante: fie folosim un algoritm de căutare informat, fie realizăm o căutare care construiește doar o parte din arborele de căutare.

1. (1p) implementați o funcție euristică h_1 (admisibilă!) pe care să o folosiți în algoritmul A*, care să permită rezolvarea cazurilor de test într-un timp decent ~~(10s pentru un caz de test)~~. (3-4 minute pentru ultimul caz de test, 15s pentru testele mai scurte) .
2. (2p) implementați o căutare de tip BFS bidirecțional, după principiul:
   - pornesc două căutări simultane, de la starea inițială și de la starea finală
   - fiecare căutare este o căutare pe lățime
   - în momentul în care există o stare pe care au descoperit-o ambele căutări, se poate construi un drum de la starea inițială la cea finală.
3. (2p) realizați o comparație folosind grafice elocvente (e.g. bar charts) între algoritmul A* și bidirectional BFS.
   - Comparați timpul de execuție, numărul de stări descoperite, și lungimea căii până la soluție.
   - Comparați folosind cazurile de test date.
   - Adăugați **comentarii personale** despre rezultatul comparației.

3.2 Cerința 2: Monte Carlo Tree Search

Putem folosi MCTS pentru rezolvarea cubului. Vom folosi algoritmul Upper Confidence Bound (UCB), ca și la laborator. Astfel:

- cum nu este vorba de un joc cu mai mulți jucători, vom construi arborele, pornind de la starea inițială, până când ajungem la limita de buget, fără a ne întoarce mai târziu la construcția arborelui pornind de la o altă stare.
- faza de selecție rămâne aceeași ca și la laborator, folosind aceeași metodă de a alege nodurile de explorat.
- după selectarea unui nod de explorat, construim un nod nou pentru o acțiune aleatoare din cele neexplorate.
- la faza de *playout* (sau *roll-out*), este posibil ca lanțul de stări să se desfășoare foarte mult, astfel că:
  - vom limita numărul de stări explorate la 14, pentru că Pocket Cube poate fi rezolvat întotdeauna în cel mult 14 mutări.
  - vom alege ca recompensă asociată cu un playout valoarea maximă peste stările explorate pentru o anumită funcție euristică.

Cu aceste modificări:

- (2p) implementați MCTS așa cum este descris mai sus, folosind euristica de la Cerința 1. Notați că la MCTS este nevoie ca recompensa să fie mai mare mai aproape de starea finală, iar euristicile estimează *costul* până la starea finală.
- (1p) implementați o a doua funcție euristică h_2, diferită de cea de la Cerința 1. Aceasta nu trebuie neapărat să fie admisibilă.
- (2p) realizați o comparație (timp, stări, lungime cale) între diverse valori pentru constanta C și între diverse variante de buget pentru MCTS, pentru cele 2 euristici. Pentru cea mai bună dintre euristici, și pentru cele mai bune valori pentru C și pentru buget, comparați cu rezultatele de la A* și bidirectional BFS de la Cerința 1. Folosiți cele 4 cazuri de test. Adăugați comentarii personale despre rezultatul comparației.

  Pentru C, folosiți valorile 0.5 și 0.1.

  Pentru buget, folosiți valorile 1000, 5000, 10000 și 20000.

Notă:

- Cum o parte a algoritmului MCTS se bazează pe alegeri aleatoare, pentru fiecare combinație de parametri realizați un număr mai mare de rulări cu aceiași parametri (e.g. 20 de rulări) și mediați rezultatele.
- Pentru reducerea timpului de execuție în cazul testelor asociate cu algoritmul MCTS, procesul de căutare poate fi oprit odată ce a fost adăugat în arbore un nod corespunzător stării finale (odată ce a fost găsită prima soluție).

3.3 Cerința 3: Pattern database

Eficiența rezolvării pentru cubul lui Rubik poate fi crescută considerabil prin folosirea unei baze de date – un catalog – de stări în care să pre-calculăm distanța de la o stare la soluție. Folosind un astfel de catalog, putem evalua corect stările, cu costul lor real.

Cum, însă, stările sunt destul de multe, chiar și pentru Pocket Cube, putem să nu construim catalogul pentru toate stările, ci doar pentru cele la o anumită distanță față de soluție, folosind o căutare BFS pornind de la starea finală, și reținând distanța stării față de starea finală.

- (2p) construiți catalogul de stări pentru toate stările aflate la distanță de cel mult 7 față de soluție. Măsurați timpul necesar construcției catalogului.
- (1p) construiți o funcție euristică h_3 care:
  - pentru stările aflate în catalog, întoarce exact costul acestora.
  - pentru alte stări, întoarce valoarea pentru una dintre euristicile deja implementate (dacă ambele sunt admisibile, altfel întoarce h_1).
- (2p) realizați o comparație (timp, stări, lungime cale) între A* și MCTS, ambii algoritmi folosind euristica h_3, pe cele 4 cazuri de test. Vedeți **nota** de la cerința anterioară. Adăugați comentarii personale despre rezultatul comparației.

4 Observații

- aveți nevoie de bibliotecile `numpy` și `matplotlib` instalate. Pentru integrarea cu Jupyter, folosiți directivele `%matplotlib notebook` și/sau `%matplotlib inline`
- puteți folosiți directiva `%%time` pentru a măsura timpul de execuție în jupyter notebook.

5 Trimiterea temei

Trimiteți 2 fișiere: 1 fișier `.zip` care conține implementarea și un fișier `.pdf` care conține graficele și comentariile asupra comparațiilor.

6 Changelog

- 15.11 Publicare enunț și cod.
- 4.12 Actualizare timpi de execuție. Adăugare notă MCTS.