Vasiliu Diana-Elena

Problema Klotski

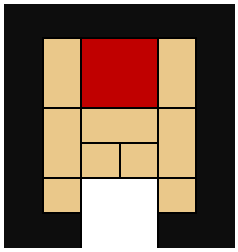
I. Introducere

Klotski este un puzzle bazat pe blocuri glisante, de diferite forme, în care scopul este să se aducă un bloc special într-o poziție predefinită.

La fel ca în cazul altor puzzle-uri de acest fel, mai multe piese sunt așezate pe o tablă (într-o cutie), care, de obicei, are dimensiunea de 4x5 blocuri unitare. Printre aceste blocuri, există o piesă specială, în general mai mare, care trebuie mutată într-un loc predefinit în cadrul tablei de joc, pe unde poate fi scoasă de pe tablă. **Există anumite restricții:** doar piesa specială poate fi scoasă, nicio altă piesă nu poate depăși spațiul destinat ieșirii, iar piesele pot fi mutate doar orizontal sau vertical, evident, doar dacă există spațiu pentru a efectua mutarea.

În general, scopul jocului este să se elimine piesa specială în cât mai puține mutări sau într-un timp cât mai scurt.

Jocul clasic are următoarea configurație:



Piesa specială este cea roșie, care trebuie adusă în partea de jos a tablei, unde se află ieșirea.

Problema de analizat

Problema noastră este similară cu jocul clasic Klotski, dar aduce câteva modificări. Astfel:

1. piesele pot avea diverse forme, nu neapărat dreptunghiulare
2. se dorește eliminarea completă a piesei speciale, nu doar aducerea ei într-un anumit loc
3. toate piesele pot fi mutate inclusiv în spațiul din bordură destinat ieșirii, dar nu pot depăși deloc acest spațiu

Scopul nostru este să comparăm mai multe tehnici de căutare informată, plecând de la problema adaptată a jocului Klotski.

II. Definirea formală a problemei

Stările și tranzițiile

O **stare** reprezintă o configurație a tablei (o așezare a pieselor pe tablă) la un moment dat.

O **tranziție** reprezintă mutarea unei singure piese. Mutările pot fi realizate doar pe orizontală sau pe verticală, doar dacă există spațiu pentru mutare.

Costul

Costul mutării unei piese îl reprezintă dimensiunea acestei piese, adică numărul de spații / celule pe care îl ocupă în cutie). Excepție face piesa specială, pentru care costul unei mutări este egal cu 1.

Piesele

Piesele sunt reprezentate prin niște simboluri, astfel:

- '#' – pentru rama cutiei
- litere sau cifre – pentru piese
- '.' – pentru spațiile libere
- '*' – pentru piesa specială

În cazul pieselor, un grup de simboluri identice formează o singură piesă. Nu este permis să existe mai multe piese cu același simbol.

Datele de intrare

Configurația inițială a tablei va fi scrisă într-un fișier, sub forma unei matrici, folosind simbolurile definite anterior.

Datele de ieșire

Fișierul de ieșire va conține configurațiile succesive realizate pe parcursul drumului de la starea inițială la starea finală. Ultima configurație conține piesa specială eliminată de pe tablă.

Pentru fiecare configurație, se vor afișa:

1. ce piesă s-a mutat și în ce direcție
2. lungimea drumului de la starea inițială la cea finală
3. costul acestui drum
4. timpul de găsire a unei soluții (de la a doua soluție, timpul se consideră de la începutul execuției algoritmului, nu de la ultima soluție găsită)
5. numărul maxim de noduri existente la un moment dat în memorie
6. numărul total de noduri calculate (în cazul IDA*, se adună toate nodurile generate, chiar dacă se repetă generarea arborelui)

Algoritmi

Vom implementa algoritmii **UCS** (Uniform Cost Search), **A***, **A* optimizat** și **IDA*** (Iterative Deepening A*).

Toți algoritmii vor fi rulați în limita unui timp de timeout, necesar în special pentru configurații fără soluție sau configurații foarte mari, unde se ajunge la generarea unui arbore foarte mare.

III. Descrierea programului

Clasa SearchNode

Problema se rezumă la crearea unui arbore de parcurgere, în care nodurile sunt configurațiile în care s-a ajuns pe parcursul căutării. Aceste noduri sunt reprezentate folosind clasa SeachNode, care conține date despre:

- configurația tablei la un moment dat
- nodul părinte (configurația de la care s-a ajuns, printr-o singură mutare, la configurația curentă)
- costul necesar ajungerii din starea inițială în starea curentă (suma costurilor mutărilor de pe drum)
- factorul euristic (o valoare estimată a costului necesar ajungerii în starea finală din starea curentă)
- valoarea funcției utilizate de algoritmul A* (suma dintre cost și factorul euristic)
- un text reprezentând mutarea realizată imediat anterior prin care s-a ajuns în configurația curentă

Folosind această clasă, putem verifica dacă un nod oarecare se află deja pe drumul obținut până la configurația curentă, pentru a evita, de exemplu, întoarcerile în locurile în care am mai fost pe acest drum.

Clasa Graph

Problema este reprezentată printr-un graf de tip arbore, în care nodurile sunt stările în care am ajuns pe parcursul căutării. Acest graf conține noduri de tip SearchNode.

Despre acest graf se știe, de la început (din datele de intrare), doar configurația stării inițiale a jocului. În cadrul constructorului acestei clase, se precalculează diferite date necesare pe parcursul algoritmului, cum ar fi dimensiunile tablei și coordonatele ieșirii. De asemenea, de la instanțierea grafului problemei, sunt verificate și datele de intrare. În cazul în care sunt greșite, tot programul se oprește, generând o eroare.

În această clasă sunt implementate mai multe metode, care se ocupă de diferite funcții ce trebuie îndeplinite pe parcursul algoritmului.

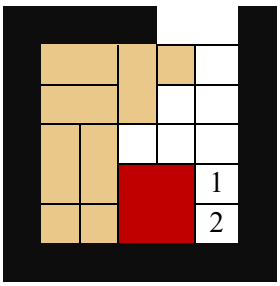
Analiza metodelor din clasa Graph

Vom analiza în continuare cele mai importante metode din clasa Graph.

1. ***goalTest()*** – testează dacă starea curentă este o stare scop, verificând dacă mai există, pe tablă, vreun fragment din piesa specială. Returnează *True* numai dacă piesa a fost eliminată complet.

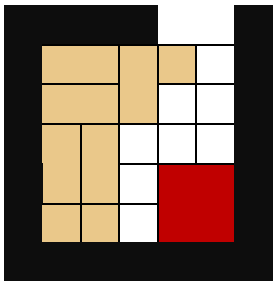
2. ***generateSuccessors()*** – generează succesorii undei anumite configurații a tablei. Un succesor este aflat prin mutarea unei piese, cu o singură poziție, în limita spațiilor libere, ținând cont de restricția conform căreia nu se poate scoate, nici măcar cu un fragment, o piesă diferită de cea specială.

   Metoda caută fiecare spațiu liber din matrice, inclusiv cele din afara tablei, pentru a verifica dacă se poate scoate piesa specială, în dreptul ieșirii. Când este găsit un spațiu liber, sunt verificate toate piesele vecine acestui spațiu și se încearcă mutarea fiecărei piese. Dacă mutarea a fost realizată cu succes (verificare realizată cu o funcție locală acestei metode), atunci s-a găsit un succesor al stării curente. Se verifică, de asemenea, să nu se genereze același succesor de mai multe ori, ca în cazul ilustrat mai jos.



Prima dată se găsește spațiul de coordonate (5, 6) – celula marcată cu 1. Se generează succesorul, mutând piesa specială la dreapta.

La următorul pas, se găsește spațiul de coordonate (6, 6) – celula marcată cu 2. Se generează același succesor, mutând piesa specială la dreapta.

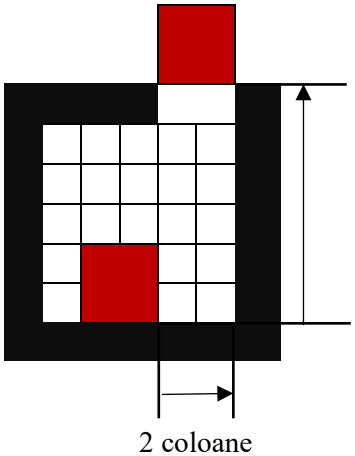


3. ***calculate_h()*** – calculează valoarea factorului euristic pentru o configurație a tablei. Această valoare este calculată în funcție de anumite euristici. Factorul euristic este folosit în funcția folosită de algoritmul A*, conform căreia alege nodurile din graf (stările tablei ce trebuie analizate) în ordinea dată de această funcție.

IV. Euristicile implementate

1. Cea mai simplă euristică implementată, **euristica trivială / banală**, se bazează pe faptul că, dacă starea curentă nu este o stare scop (finală), atunci este necesar să realizăm **cel puțin** încă **o** mutare pentru a ajunge în stare finală. Deci, dacă starea curentă nu este finală, atunci factorul euristic ***h*** este 1, altfel ***h*** este 0.

2. O euristică mai bună decât cea trivială, **euristica admisibilă**, se bazează pe *distanța Manhattan*. Această distanță este calculată adăugând numărul de linii cu numărul de coloane ce trebuie parcurse pentru a scoate piesa specială de pe tablă. Acest calcul este exemplificat în figura de mai jos.



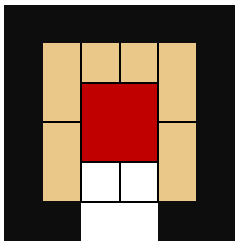
$\Rightarrow$ distanța Manhattan este

$2 + 6 = 8$

Euristica este admisibilă, deoarece este nevoie de cel puțin aceste mutări pentru a scoate piesa, **în cazul în care tabla ar conține doar piesa specială**. Dacă am încerca să scoatem piesa în mai puține mutări, nu am ajunge la ieșire. Distanța Manhattan oferă distanța minimă (numărul minim de mutări) de la locația curentă a piesei speciale și poziția ei în afara tablei.

3. O **euristică neadmisibilă** ia în considerare numărul de blocuri vecine piesei speciale. Putem demonstra printr-un exemplu că această euristică nu este admisibilă.

   Considerăm următoarea stare inițială:



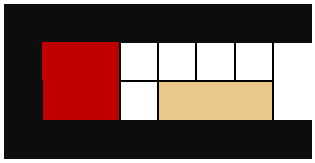
Numărul de piese din jurul piesei speciale: 6

Numărul de mutări necesare pentru a elimina piesa specială: 4

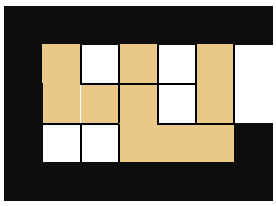
V. Fișierele de input

Am creat patru fișiere de input, pentru a exemplifica diferitele cazuri tratate de algoritm. Astfel, cele patru fișiere conțin următoarele inputuri:

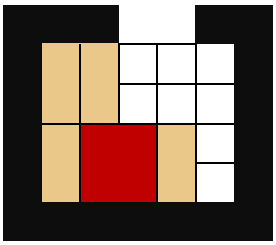
1. **Configurație fără soluție:**



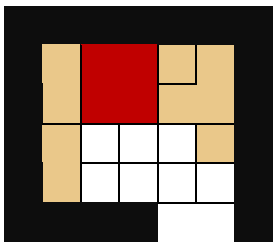
2. **Configurația inițială este și stare finală (scop):**



3. **Configurație de la care se ajunge la o soluție folosind toți algoritmii:**



4. **Configurație care ajunge la timeout pe algoritmul IDA* și dă o soluție care nu e de cost minim pe algoritmul A***



VI. Compararea algoritmilor

Vom nota unele date:

- Numărul maxim de noduri existente la un moment dat în memorie – memMax
- Numărul total de noduri – nMax

Am rulat algoritmii pentru a obține, la fiecare dintre ei, câte două soluții, cu un timeout de 10 secunde.

Algoritmul UCS						
Fișierul de input	**Lungimea**	**Costul**	**Timp**	**memMax**	**nMax**	**Indicele soluției**
3	8	8	0.069	135	448	1
3	9	10	0.4859	482	2066	2
4	11	12	4.2418	1974	11440	1
4	12	13	7.8504	2751	16227	2

Algoritmul A*							
Fișierul de input	**Lungimea**	**Costul**	**Timp**	**memMax**	**nMax**	**Euristica**	**Indicele soluției**
3	8	8	0.031	87	240	trivială	1
3	9	10	0.184	319	1189	trivială	2
3	8	8	0.0	27	34	admisibilă	1
3	9	10	0.016	62	91	admisibilă	2
3	8	8	0.015	92	160	neadmisibilă	1
3	9	10	0.132	262	819	neadmisibilă	2
4	11	12	2.137	1291	7101	trivială	1
4	12	13	4.457	1979	11446	trivială	2
4	11	12	0.185	289	1205	admisibilă	1
4	11	12	0.185	302	1250	admisibilă	2
4	12	13	2.039	1258	7301	neadmisibilă	1
4	13	14	4.448	2037	11842	neadmisibilă	2

Algoritmul A* optimizat							
Fișierul de input	**Lungimea**	**Costul**	**Timp**	**memMax**	**nMax**	**Euristica**	**Indicele soluției**
3	8	8	0.03	124	216	trivială	1
3	8	8	0.005	34	34	admisibilă	1
3	8	8	0.02	114	154	neadmisibilă	1
4	11	12	1.118	1611	4347	trivială	1
4	11	12	0.107	352	739	admisibilă	1
4	11	12	0.536	1030	2505	neadmisibilă	1

Algoritmul IDA*							
Fișierul de input	**Lungimea**	**Costul**	**Timp**	**memMax**	**nMax**	**Euristica**	**Indicele soluției**
3	8	8	0.188	846	1516	trivială	1
3	9	10	1.733	4123	8701	trivială	2
3	8	8	0.005	42	45	admisibilă	1
3	9	10	0.018	107	156	admisibilă	2
3	8	8	0.094	456	801	neadmisibilă	1
3	9	10	0.659	2754	5342	neadmisibilă	2
4	-	-	Timeout	-	-	trivială	1
4	-	-	Timeout	-	-	trivială	2
4	11	12	6.051	10629	48401	admisibilă	1
4	11	12	6.057	10678	48450	admisibilă	2
4	-	-	Timeout	-	-	neadmisibilă	1
4	-	-	Timeout	-	-	neadmisibilă	2

Se poate observa, în primul rând, că algoritmul IDA* nu are o performanță foarte bună în cazul inputului din fișierul 4. Pentru euristica trivială, care furnizează o valoare *f* destul de generală, și pentru euristica neadmisibilă, algoritmul nu găsește o soluție într-o limită de timp de 10 secunde. În același timp, pentru euristica admisibilă, algoritmul IDA* găsește o soluție, dar într-un timp destul de mare (6.051s), comparativ cu ceilalți algoritmi. Din punct de vedere al memoriei, algoritmul generează foarte multe noduri în arborele de parcurgere (conform tabelului, 48401 noduri pentru prima soluție), lucru care este un dezavantaj foarte mare pentru acest algoritm.

Față de algoritmul IDA*, algoritmul UCS furnizează, pe inputul fișierului 4, două soluții. Trebuie menționat că algoritmul UCS nu se folosește de nicio euristică, ci doar de costul drumului de la configurația inițială, până la nodul curent. Algoritmul UCS găsește o primă soluție în 4.2418s, care este chiar mai rapid decât algoritmul IDA* care folosește euristica admisibilă. De asemenea, memoria maximă folosită de UCS (1974 de noduri) este semnificativ mai mică decât cea folosită de IDA* (10629 de noduri). În schimb, ambii algoritmi furnizează soluția optimă.

Comparând algoritmii IDA* și UCS pe fișierul de input 3, observăm că se comportă diferit, în funcție de euristica folosită de IDA*. În cazul euristicii triviale și a celei neadmisibile, memoria maximă folosită de IDA* la un moment dat, pentru prima soluție (846, respectiv 456 de noduri), este destul de mare față de memoria folosită de UCS (135 de noduri). De asemenea, timpul de găsire a soluției în cazul algoritmului IDA* este mai mare decât cel al algoritmului UCS (0.188s pentru euristica trivială, respectiv 0.094s pentru euristica neadmisibilă, față de 0.069s la UCS). În schimb, dacă algoritmul IDA* folosește euristica admisibilă, se poate observa o îmbunătățire semnificativă a dimensiunii resurselor folosite (0.005s și 42 de noduri pentru IDA* față de 0.069s și 135 de noduri pentru UCS).

În cazul algoritmului A*, memoria folosită este clar mai scăzută decât în cazul UCS. Chiar și în cazul euristicii triviale, în care A* generează cele mai multe noduri dintre toate cele 3 euristici, memoria lui A* este mai bună decât cea folosită de UCS, pe ambele inputuri. Acest lucru se poate observa chiar și la a doua soluție, unde, pe fișierul 3, diferența este de la 1189 de noduri generate în total de A* la 2066 de noduri generate în total de UCS. De asemenea, timpul necesar pentru găsirea a două soluții este clar mai mic pentru algoritmul A* (0.184s) față de UCS (0.4859s) și față de IDA* (1.733s) pe euristica trivială, pe fișierul de input 3. Se observă că, în oricare din cazuri (orice fișier de input, orice indice al soluției, respectiv orice euristică folosită), algoritmul A* este mai bun și din punct de vedere al timpului, și din punct de vedere al memoriei față de algoritmul UCS.

Problema apare în cazul algoritmului A* rulat cu inputul din fișierul 4, folosind euristica neadmisibilă. Observăm că lungimea drumului de la starea inițială, până la soluție, respectiv costul lui, care au fost găsite în acest caz sunt mai mari decât lungimea și costul drumului optim de la starea inițială la cea finală (găsit, de altfel, în cazul celorlalte euristici). Deci, euristica se dovedește a fi neadmisibilă. Cu toate acestea, în continuare algoritmul rămâne mai bun din punct de vedere al resurselor folosite față de ceilalți 2 algoritmi (UCS și IDA*).

Despre algoritmul A* optimizat putem spune, în primul rând, că este implementat să furnizeze doar drumul minim de la nodul inițial la un nod scop. Comparându-l cu algoritmul A* în funcție de euristică, observăm că, pe fișierul de input 3, deși este aproximativ la fel de rapid ca A*, algoritmul A* optimizat generează, în total, mai puține noduri. În schimb, A* optimizat are, la un moment dat, un număr mai mare de noduri decât are A* (lucru observat în coloana memMax).

Concluzia observată este că, dintre cei 4 algoritmi implementați, în funcție de resursele disponibile, A* și A* optimizat sunt, într-adevăr, cei mai buni. Dacă memoria nu este foarte limitată, atunci A* optimizat găsește cel mai rapid o soluție optimă.

VII. Validări și optimizări

Pentru a asigura că algoritmii funcționează corespunzător, am realizat o serie de validări a datelor de intrare și optimizări în ceea ce privește implementarea. Astfel:

1. informația unui nod am reținut-o în memorie ca o matrice. Nu există o metodă mai bună de atât, întrucât configurația tablei de joc încape într-o matrice.
2. după citirea datelor de intrare, este realizată o serie de validări. Astfel, se verifică următoarele:
    - matricea să aibă spațiu în interior, pentru a putea avea piese
    - dimensiunea matricei trebuie să fie regulată (toate liniile trebuie să aibă aceeași lățime)
    - caracterele folosite trebuie să fie cifre sau litere (pentru piese), '.' (pentru spațiu), '*' (pentru piesa specială) sau '#' (pentru bordură). Se verifică, de asemenea, că '#' se regăsește doar pe bordură și nu și în interiorul tablei.
    - tabla trebuie să aibă exact o ieșire
    - o piesă are toate fragmentele unite. Nu pot exista mai multe piese reprezentate prin același caracter

VIII. Rularea programului

Pentru a rula programul, trebuie folosită o sintaxă specială pentru o comandă în terminal:

```
$ python main.py -if $folder_input -of $folder_output -t $timeout -nsol $nr_solutii
```

unde:

- *$folder_input* este calea către folderul în care se găsesc fișierele de input
- *$folder_output* este calea către folderul în care se dorește crearea/găsirea fișierelor de output
- *$timeout* este numărul maxim de secunde în care dorim ca algoritmii să găsească soluțiile
- *$nr_solutii* este numărul de soluții pe care dorim să le găsească fiecare algoritm (cu excepția A* optimizat)

Pentru a colecta datele din tabelul de la VI, am rulat algoritmul folosind comanda:

```
$ python main.py -if input_folder -of output_folder -t 10 -nsol 2
```

Atenție! Dacă fișierul main.py se află în alt folder decât cel curent, atunci fie trebuie specificată întreaga cale până la el, fie trebuie schimbată calea din terminal.

În cazul unei erori conform căreia nu se poate găsi comanda **python**, atunci comanda trebuie înlocuită cu **python3**.

O altă metodă de a apela programul este dată de comanda:

```
$ python3 main.py --input_folder input_folder --output_folder output_folder --timeout 15 -nsol 5
```

Orice combinație de parametri funcționează. De exemplu:

```
$ python main.py -if input_folder -of output_folder -nsol 5 --timeout 15
```

De asemenea, folderul de input și fișierele din el trebuie să existe! Dacă nu există, programul va genera o eroare. Folderul de output și fișierele din el sunt generate în mod automat.