Blocks world as a benchmark

Studiu al problemei joc *Lumea Blocurilor*

Crețu Rodica-Elena (C114D)

Mihaiu Mihaela (C114D)

Vaman Adina-Maria (C114E)

Cuprins

[1. Introducere 2](#_Toc90245064)

[2. Definiții și concepte folosite în BW 3](#_Toc90245065)

[3. Algoritmi folosiți în rezolvarea BW 5](#_Toc90245066)

[3.1. Unstack – Stack 5](#_Toc90245067)

[3.2. GN1 6](#_Toc90245068)

[3.3. GN2 8](#_Toc90245069)

[3.4. PERFECT 9](#_Toc90245070)

[3.5. Depth First Search 9](#_Toc90245071)

[3.6. Breadth First Search 10](#_Toc90245072)

[3.7. Best First Search 10](#_Toc90245073)

[3.8. A\* 10](#_Toc90245074)

[4. Rezultate experimentale 10](#_Toc90245075)

[Referințe 12](#_Toc90245076)

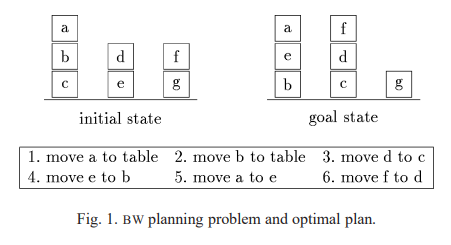
# Introducere

Problemele joc sunt probleme ce nu prezintă o valoare științifică imediată, dar servesc la ilustrarea unor trăsături ale unor probleme mai complicate din viața de zi cu zi și la demonstrarea unor tehnici generale de rezolvare a acestora. O astfel de problemă joc este ***Blocks World*** (Lumea blocurilor): aceasta constă într-un număr finit de blocuri așezate unul peste celălalt, formând turnulețe pe o masă suficient de încăpătoare, poziționarea turnulețelor fiind irelevantă.

Problema de planificare BW constă în transformarea unei stări inițiale a blocurilor într-o stare scop, mutând pe rând câte un bloc cu următoarele constrângeri:

* se pot muta doar blocurile care sunt pe masă sau în vârful unui turnuleț
* un bloc se poate muta doar pe masă sau în vârful unui turnuleț

Problema de planificare optimă BW este aceea care se rezolvă într-un număr minim de mutări.

 (1)

Planificarea BW se poate rezolva cu ajutorul algoritmilor ce rulează în timp polinomial, iar planificarea optimă BW face parte din categoria problemelor NP-hard, mai exact Max-SNP-hard, adică există o anumită proporție fixă sub care aproximările soluțiilor nu se mai pot găsi în timp polinomial. În cazul BW, se pot aproxima soluții în timp polinomial în limita raportului de 2 între lungimea soluției aproximate și celei optime (se pot ajunge la soluții mutând de cel mult două ori un bloc). (1)

Domeniile artificiale precum Blocks World, Damele, Comis-voiajorul nu reprezintă în sine interes practic, însă folosesc la crearea experimentelor sistematice, relevante și ușor de realizat. Cu toate acestea, pentru a folosit BW ca benchmark pentru compararea diverșilor algoritmi, trebuie să luăm în considerare următorul aspect: avem nevoie să știm cum să construim instanțe de probleme potrivite pentru experimente sistematice, de ex. fie unele distribuite aleatoriu uniform, fie unele dificile, pentru a evita testarea algoritmilor doar pe probleme simple, fără relevanță sau compararea rezultatelor algoritmilor atunci când unul a primit probleme simple, iar unul doar dificile.

# Definiții și concepte folosite în BW

Fie B o mulțime finită de blocuri, cu TABLE aparținând B, dar cu proprietatea specială că acesta nu se află pe nimic.

Fie funcția , *S(x)* = blocul pe care se află blocul *x*. Această funcție are proprietatea că este injectivă peste tot, mai puțin în elementul *TABLE*, iar închiderea sa tranzitivă este antireflexivă.

Fie perechea o stare intermediară; vom identifica o stare BW cu o stare intermediară în care *S* este o funcție complet definită (specificând astfel dispunerea blocurilor).

Pentru o stare intermediară și oricare blocuri și *b* definim:

* (dacă a se află pe b, atunci )
* (dacă *a* este *TABLE* sau nu există niciun bloc *b* care să se afle pe *a*, atunci )
* (*ABOVE* închiderea tranzitivă alui *ON*)
* (poziția unui bloc este reprezentată de secvența de blocuri de sub el la care se adaugă blocul, dacă această secvență există, sau blocul în sine, în caz contrar)

Un turnuleț de blocuri va fi reprezentat de poziția unui bloc liber a (). Dacă un turnuleț se termină cu masa, acesta va fi *așezat*. Într-o stare BW, fiecare turnuleț este așezat, însă în cele intermediare, nu.

O instanță de problemă BW este reprezentată de o pereche de stări , unde I este starea inițială, iar G este starea scop. O mișcare în starea este perechea de blocuri , cu și , astfel încât și (blocurile *a* și *b* sunt libere, adică nu au un alt bloc pe ele, și *a* nu se află deja pe *b* - dacă *b* este TABLE, acesta este mereu liber, chiar dacă este deja *a* pe el, pentru că putem avea oricâte blocuri pe masă). Rezultatul mișcării în starea esre o nouă stare , unde (toate celelalte blocuri în afară de *a* sunt definite la fel de funcția suport, dar acum *a* se află pe *b*).

Un plan (soluție) pentru problema BW reprezintă o secvență finită de mișcări astfel încât fie *I = G* și *p* = 0, fie este o mișcare în I și secvența este un plan pentru . Un plan pentru problema BW este optimal dacă nu există un alt plan cu mai puține mișcări pentru problema respectivă.

Un bloc *b* a cărui poziție în I (secvența de blocuri de sub acesta, inclusiv acesta) este diferită de poziția sa în G este *așezat greșit*, altfel este *așezat corect*. Doar blocurile așezate greșite trebuie mutate pentru rezolvarea unei probleme. O mișcare este *constructivă* dacă și numai dacă blocul *a* va fi așezat corect după mutarea acestuia pe *b*. După ce un bloc a fost mutat constructiv, acesta nu trebuie mutat din nou.

Dacă nu există nicio mișcare constructivă ce poate fi realizată în starea curentă, spunem că problema este *blocată* (deadlock). Atunci, pentru orice bloc așezat greșit *a*, există un bloc *b* care trebuie mutat înainte ca o mutare constructivă cu a să fie posibilă. Pentru că numărul de blocuri este unul finit, secvența blocurilor din blocaj este ciclică. Un *blocaj* este așadar o submulțime a mulțimii de blocuri *B* care poate fi ordonată astfel încât între oricare două blocuri succesive din submulțime (inclusiv blocul și blocul ) există relația:

(blocurile *a* și *b* nu sunt așezate corect și există un bloc *x* care este deasupra blocului *b* în starea inițială *I* și deasupra blocului *a* în starea scop *G* și care nu va putea fi mutat fără o mutare neconstructivă în prealabil)

Așadar, pentru a rezolva o problemă blocată, cel puțin un bloc din această submulțime trebuie mutat neconstructiv – pe masă, pentru a nu introduce un alt blocaj. Pentru a ajunge la soluție, mulțimea blocurilor mutate neconstructiv trebuie să conțină cel puțin un bloc din fiecare blocaj. Pentru ca planul rezultat să fie optim, această mulțime trebuie să aibă cardinalul minim – problema NP-hard. (2)

# Algoritmi folosiți în rezolvarea BW

Pentru a găsi o soluție a problemei BW care să nu fie optimă, putem produce în timp liniar planuri de cel mult de două ori mai lungi decât cel optim, cât timp avem grijă să nu mutăm blocuri care sunt deja așezate corect sau să introducem voit noi blocaje.

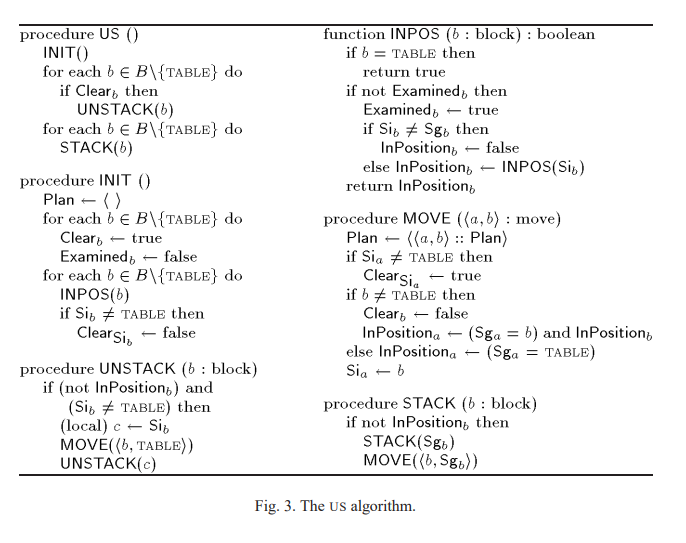
## Unstack – Stack

După cum sugerează și numele, strategia acestui algoritm este următoarea: pune pe masă orice bloc din vârful unui turnuleț care nu este în poziție; când nu mai sunt blocuri care nu sunt în poziție în afară de cele de pe masă, pune în ordine blocurile în turnulețe prin mișcări constructive. Pentru că orice bloc care nu este în poziție trebuie mutat cel puțin o dată și în planurile optime, iar în algoritmul US acestea sunt mutate de două ori (o dată pe masă și o dată pe turnuleț), lungimea planului rezultat va fi de cel mult două ori mai lung decât cel optim.

Pentru a implementa algoritmul US într-un timp linear, trebuie calculat pentru fiecare bloc dacă este în poziție în O(n), o singură dată în execuția programului, astfel:

* Inițial, fiecare bloc este marcat ca fiind liber și neexaminat
* Pentru fiecare bloc se apelează funcția INPOS(b) care verifică dacă este în poziția corectă (în mod recursiv, verificând și actualizând variabila *inPosition* pentru toate blocurile de sub el); blocul de sub blocul curent este marcat ca nefiind liber
* Fiecare bloc marcat liber este pus pe masă dacă nu este în poziție
* Fiecare bloc de pe masă este pus în turnulețul corespunzător dacă nu este în poziție (nu trebuie să fie pe masă)

Cum fiecare dintre funcții este apelată de cel mult două ori pentru un bloc oarecare, complexitatea va fi cel mult O(2n) – liniară.

 (1)

## GN1

Propus de N. Gupta și D. Nau, acest algoritm îmbunătățește algoritmul US prin recunoașterea mișcărilor constructive și executarea preferențială a acestora față de cele nonconstructive (un bloc nu va fi pus niciodată pe masă dacă ar putea fi așezat corect în mod direct). Pentru îndeplinirea acestui obiectiv, va fi luat în considerare și statusul unui bloc:

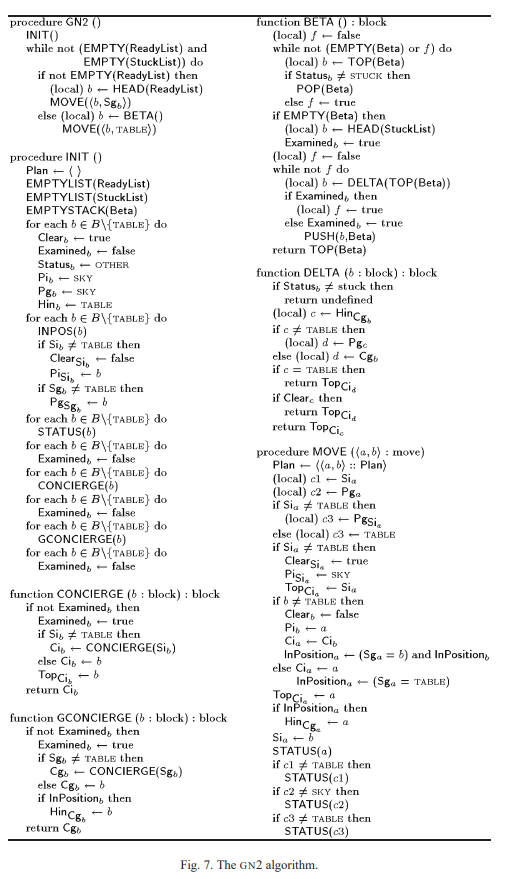
* BLOCAT – nu poate fi mutat pentru că încă nu este așezat corect blocul care ar trebui să fie sub el în starea scop
* GATA – poate fi așezat corect în mod direct
* ALTUL – se află pe masă, dar nu este în poziție

Actualizarea acestui status va fi realizată tot în timp liniar prin procesarea și informațiilor legate de blocul ce se află deasupra blocului curent. Blocurile aflate în statusul BLOCAT și cele aflate în statusul GATA vor fi reținute în două liste separate, dublu înlănțuite. Cât timp există elemente într-una din cele două, întâi vor fi mutate blocurile GATA (dacă există), apoi primul bloc BLOCAT va fi mutat pe masă (devine OTHER); la fiecare mutare, statusurile sunt actualizate în O(1).

 (1)

## GN2

Îmbunătățirea adusă de GN2 față de GN1 constă în alegerea blocurilor ce vor fi mutate neconstructiv: acestea trebuie să fie incluse în cel puțin un blocaj. Pentru implementarea acestui aspect, este introdus un nou concept, acela de *portar*: blocul ce se află la baza unui turn și care reține printr-o variabilă proprie care este blocul aflat pe poziția cea mai înaltă care se află în poziție. Prin accesarea acestei variabile, schimbările sunt propagate în timp constant și putem ține evidența submulțimii ce reprezintă în mod curent un blocaj.



(1)

## PERFECT

Algoritmul PERFECT se folosește de o versiune modificată a GN1 (denumită în continuare GN1H) pentru găsirea unui plan optim în timp exponențial.

Pașii urmați sunt următorii:

* Inițializează K cu toate blocajele singleton (formate dintr-un singur bloc)
* Până când se găsește un plan:
  + Generează H a.î. toate blocajele din K au cel puțin un element comun cu H
  + Testează dacă H este soluție
    - Dacă da, întoarce planul găsit
    - Dacă nu, întoarce un blocaj D disjunct cu H și adaugă-l în K

Singurul pas care nu poate fi rezolvat în timp liniar cu ajutorul GN1H este cel de generare al H: acesta trebuie generat recursiv, în timp exponențial.

Modificarea adusă lui GN1 este: dacă blocul care se va muta neconstructiv nu aparține mulțimii de blocuri dată ca parametru H, problema nu are soluție. Astfel putem testa și dacă {b} este blocaj singleton (dând ca parametru , dacă nu se ajunge la un plan fără să se miște neconstructiv blocul b, înseamnă că – blocaj), și dacă H sparge toate blocajele (dând ca parametru H generat recursiv). De asemenea, putem găsi și blocaje disjuncte cu H adăugând pe rând fiecare alt bloc în H și testând cu GN1H – dacă nu întoarce soluția, am găsit un nou blocaj.

## Depth First Search

Algoritmul DFS este un tip de algoritm de *blind search* care va parcurge în adâncime graful stărilor până va ajunge în nodul corespunzător soluției.

Pornim din starea inițială reprezentată printr-un șir de blocuri (structuri cu datele: nume, numele blocului pe care se află, liber/acoperit), adăugăm copiii nodului curent în coada de căutare (dacă nu au fost deja marcați ca descoperiți) și continuăm cu primul nod din stivă până ajungem la nodul soluție. Generarea nodurilor copil se realizează prin iterarea tuturor blocurilor, alegerea celor libere și mutarea acestora pe un alt bloc sau masă.

## Breadth First Search

Algoritmul BFS este un tip de algoritm de *blind search* care va parcurge în lățime graful stărilor până va ajunge în nodul corespunzător soluției.

Diferența față de DFS este aceea că în locul folosirii unei stive, se va folosi o coadă; astfel întâi se vor parcurge toți copiii nodului curent și de abia apoi copiii unui nod copil.

## Best First Search

Algoritmul Best First Search îmbunătățește timpul parcurgerii BFS a grafului prin adăugarea unei euristici în alegerea următorului nod din plan. Euristica folosită calculează numărul de blocuri care nu sunt în poziție în starea curentă (astfel, dintre două stări, va fi aleasă cea cu mai puține blocuri așezate greșit).

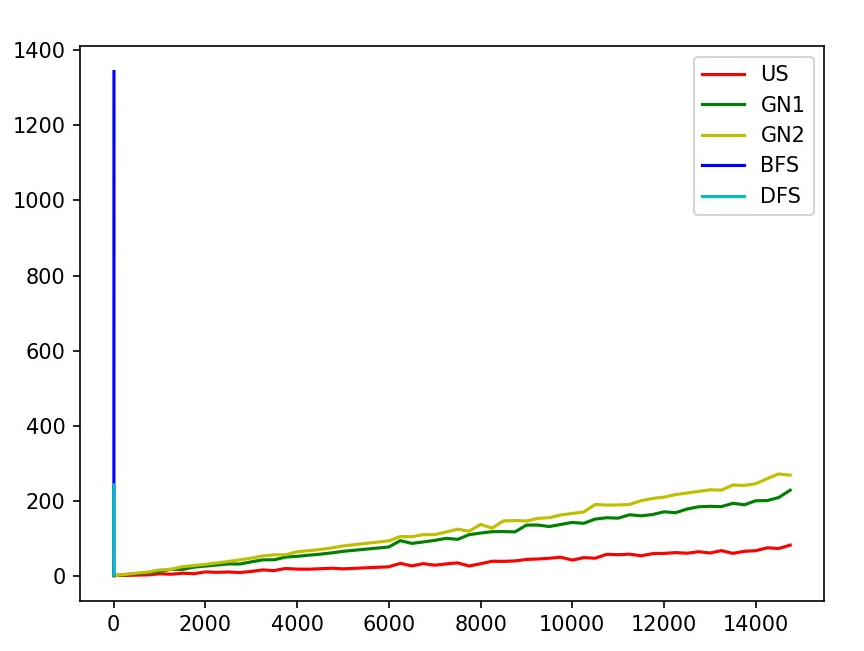
## A\*

Algoritmul A\* îmbunătățește timpul parcurgerii BFS a grafului prin adăugarea unei euristici în alegerea următorului nod din plan. Euristica folosită calculează numărul de blocuri care nu sunt în poziție în starea curentă și adaugă distanța de la nodul inițial (astfel, dintre două stări cu același număr de blocuri așezate greșit, va fi aleasă cea la care se ajunge mai repede).

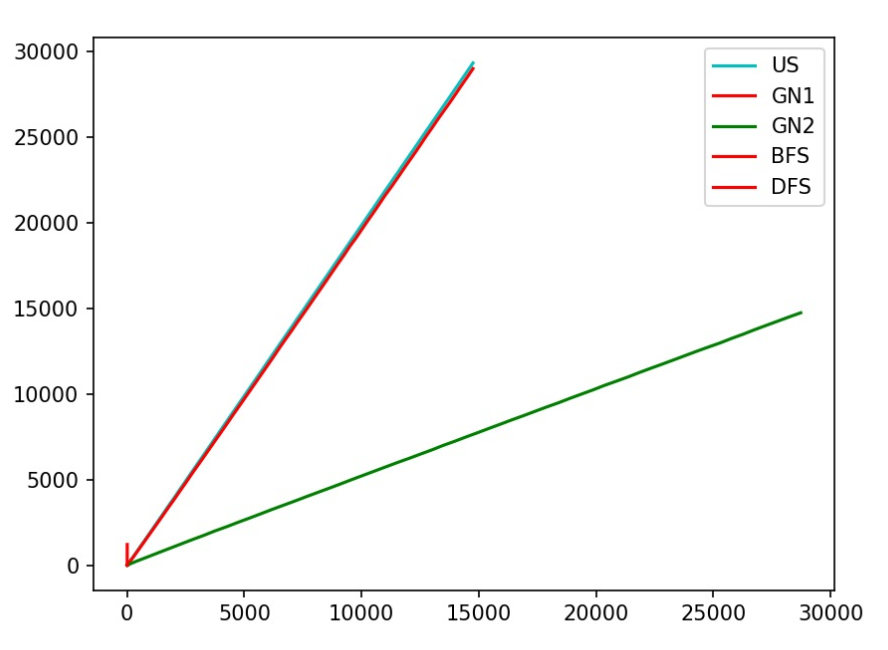
# Rezultate experimentale

Din experimentele realizate, am obținut următoarele rezultate:

* Dintre algoritmii cu timp de rulare liniar, US este cel mai rapid, iar GN1 și GN2 au timpi de rulare similari; în schimb, planurile obținute de US sunt în general mai lungi, cele obținute de GN2 fiind, de obicei, cele mai scurte
* Algoritmul optim este fezabil doar pentru probleme de cel mult 100 de blocuri – are timpi de rulare similari cu cei liniari pentru 100.000 de blocuri
* Algoritmii bazați pe parcurgerea grafului prin *blind search* nu sunt optimi nici ca timpi de rulare, nici ca lungime a planului rezultat; Best First Search și A\* au rezultate mai bune, dar nu sunt comparabili cu GN1/2



Figură Grafic timp de rulare (în ms) în funcție de numărul de blocuri



Figură Grafic al lungimii planului (soluției) în funcție de numărul de blocuri

# Referințe

1. **John Slaney, Sylvie Thiebaux.** Blocks World revisited. *Artificial Intelligence.* 2001, Vol. 125, 1-2.

2. **N. Gupta, D. Nau.** On the complexity of Blocks World planning. *Artificial Intelligence.* 1992, Vol. 56.