

Haskell: Bloxorz

- Responsabili:
 - Mihnea Muraru [mailto:mmihnea@gmail.com]
 - Vlad Neculae [mailto:neculae.vlad@gmail.com]
 - Șerban Ciofu [mailto:sciofu@gmail.com]
- Deadline soft: **22.04.2019**
- Deadline hard: **27.04.2019**
- Data publicării: 03.04.2019
- Data ultimei modificări: 03.04.2019
- Tema se va încărca pe **vmchecker**
- Data tester-ului: 03.04.2019
- Forum temă [https://acs.curs.pub.ro/2018/mod/forum/view.php?id=12366]

Obiective

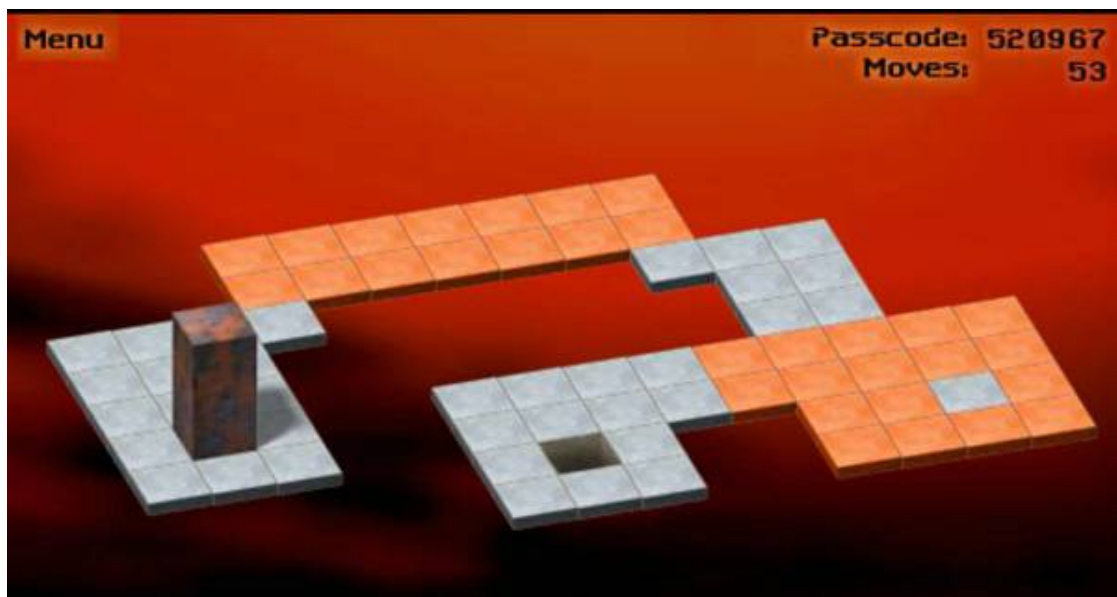
- Utilizarea mecanismelor **funcționale**, de **tipuri** și de **evaluare leneșă** din limbajul Haskell pentru rezolvarea unei probleme de **căutare** în spațiul stărilor.
- Exploatarea evaluării leneșe pentru **decuplarea conceptuală** a etapelor de construcție și de explorare a acestui spațiu.

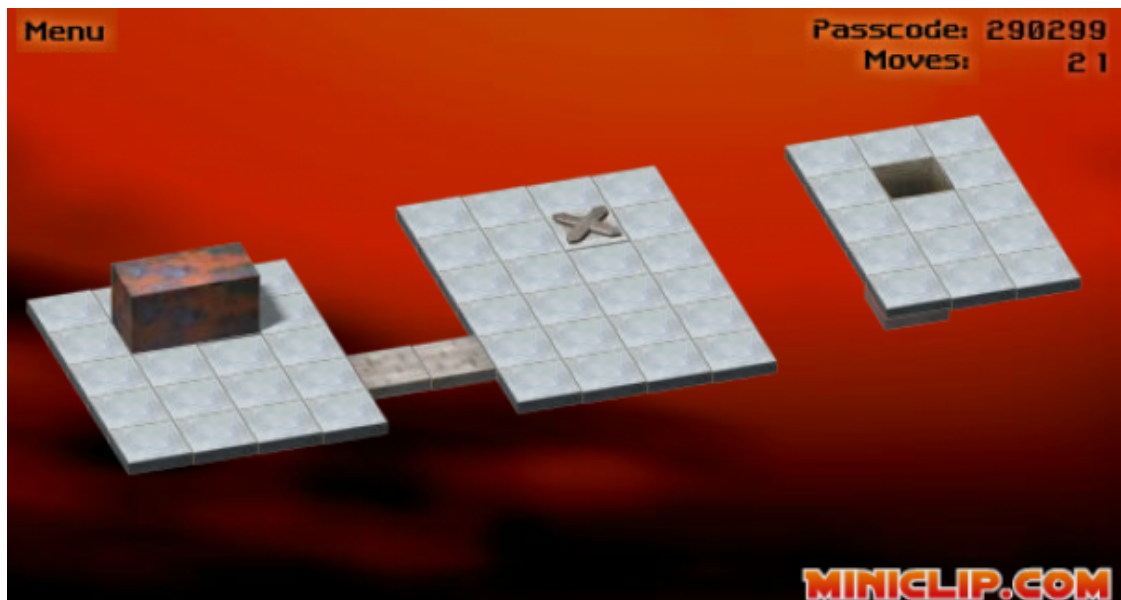
Descriere

Tema urmărește implementarea jocului *Bloxorz* [https://www.miniclip.com/games/bloxorz-classic/en/], și a unui mecanism de rezolvare a oricărui nivel, utilizând **căutare leneșă** în spațiul stărilor. În acest sens, se va întrebuința o versiune specială a **parcurgerii în adâncime**, numită **adâncire iterativă** (*iterative deepening depth-first search* [https://en.m.wikipedia.org/wiki/Iterative_deepening_depth-first_search]).

Bloxorz

Jocul presupune deplasarea unui paralelipiped pe o tablă până când acesta va ajunge la poziția de finish. De exemplu:





Putem observa următoarele elemente:

1. **Block:** Paralelipipedul care este mutat în cele 4 direcții și poate ocupa fie 1, fie 2 celule de pe tabla de joc; se garantează că se află pe tabla de joc la începutul nivelului;
2. **Cell:** Plăcile pe care se sprijină **block-ul**, acestea pot fi de mai multe tipuri: **hard** - gri deschis pe tablă (susțin block-ul în orice poziție), **soft** - portocaliu pe tablă (nu pot susține block-ul în poziție verticală), **winning** (unic cell pe tabla de joc, block-ul trebuie să ajungă pe acest cell în poziție verticală pentru a termina nivelul) sau **switch** - „x”-ul de pe Hard Cell (plăci speciale care, pe lângă susținerea block-ului, pot fi activate/dezactivate, pentru a adăuga/elimina un număr predefinit de cell-uri la tablă, cu poziție specificată de la început - gri închis pe tablă).

Dacă **block-ul** va ajunge în poziție verticală pe un **cell soft** sau va cădea de pe tablă, nivelul va fi considerat pierdut.

Va încurajăm să încercați jocul [<https://www.miniclip.com/games/bloxorz-classic/en/>] înainte de implementarea temei 😊.

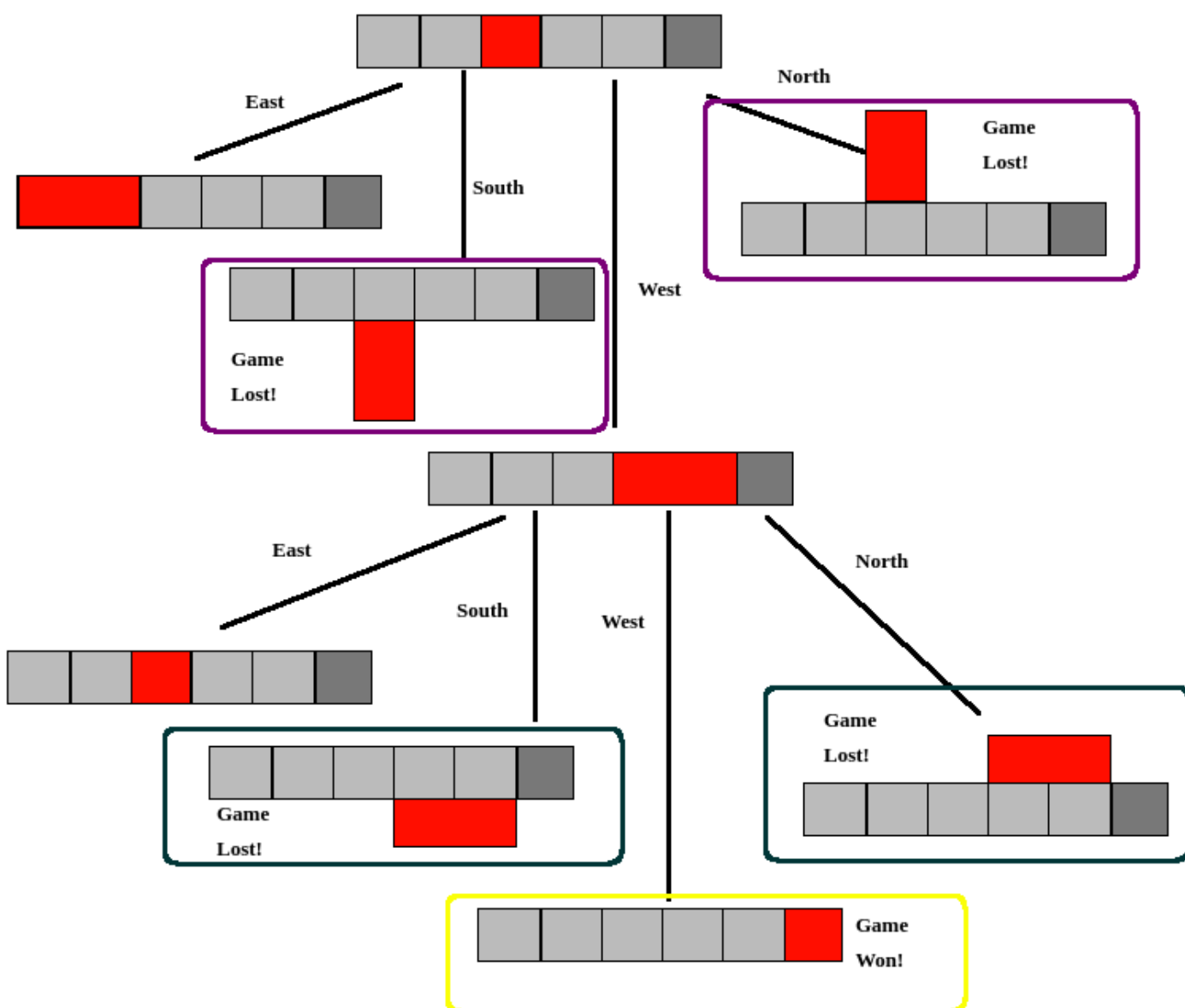
Spațiul stărilor problemei

Căutarea soluției se va desfășura în **spațiul stărilor** problemei. Acesta este reprezentat de un graf, în care nodurile sunt **configurațiile** tablei de joc, iar muchiile, **acțiunile** care asigură tranzițiile între stări.

Fiecare nod va avea maxim 4 copii proveniți din cele 4 mutări care duc la modificarea stării actuale.

Având această reprezentare vom putea enumera stările utilizând parcurgerea în adâncime (deși ar putea fi utilizată în alte situații și parcurgerea în lățime).

Mai jos aveți reprezentarea spațiului stărilor pentru **level0** din Levels.hs.



Muchiile sunt adnotate cu **mişcarea** care va altera starea curentă și va genera un nou nod copil. Putem observa marcată cu galben starea finală pe care o căutăm, iar cu mov stările pentru care nu mai are sens să continuăm generarea spațiului, deoarece jocul este pierdut.

Parcurgerea în adâncime cu adanciri iterative

Spațiul stărilor poate fi foarte mare, astfel că o parcurgere în adâncime standard va fi **ineficientă** pentru găsirea stării finale, întrucât căutarea se poate canaliza pe căi care nu conduc către aceasta. Pentru a rezolva această problemă ne vom seta o **adâncime maximă** până la care să căutăm, pe care o vom **incrementa** treptat.

Concret, în prima faza adâncimea va fi **0** și vom verifica dacă nodurile de adâncime 0 (adică rădăcina) reprezintă o soluție. În caz contrar, vom incrementa adâncimea maximă la **1**, și vom verifica și nodurile provenite din rădăcină (cei 4 copii). Acest algoritm va continua până când vom găsi starea finală, deci un mod de a rezolva nivelul actual.

Cerințe

Rezolvarea temei este structurată pe etapele de mai jos. Începeți prin a vă familiariza cu structura **arhivei** de resurse. Va fi de ajutor să parcurgeți indicațiile din enunț **în paralel** cu comentariile din surse. În rezolvare, exploatați testele drept **cazuri de utilizare** a funcțiilor pe care le implementați.



Implementarea jocului și afișarea (35p)

Elementele care compun jocul, **mecanica** și **afișarea** jocului se vor realiza în fișierul `Bloxorz.hs`. Va trebui să completați propriile definiții pentru tipurile de date din fișier, urmărind **TODO-urile**.

Pentru reprezentarea unui nivel recomandăm folosirea modului `Data.Array` [<http://hackage.haskell.org/package/array-0.5.3.0/docs/Data-Array.html>]. Recomandăm să urmăriți de asemenea și testele pentru a vă asigura că implementarea voastră este corectă.

Rezolvarea **afișării nivelului** se va realiza prin implementarea instanței de **Show** atât pentru **nivel** cât și pentru fiecare dintre obiectele care îl vor compune: tipurile **Cell** și **Block**.

Vom folosi următoarea convenție de afișare:

- *Hard Cell* - 
- *Soft Cell* - '='
- *Block Cell* - 
- *Switch Cell* - '±'
- *Winning Cell* - '✱'
- *Spațiu gol* - ''

Astfel, reprezentarea textuală a nivelului va deveni:



După ce fiecare obiect va avea o instanță de `Show`, vom trece la construirea nivelului. Veți implementa funcțiile `addCell` și `addSwitch`, care vor adăuga tipurile de celule la nivelul actual. Se garantează că toate pozițiile primite ca parametru pentru aceste funcții sunt valide.

În final, în vederea deplasării blocului, veți implementa funcția **activate**. Aceasta va trebui să determine comportamentul celulelor în momentul în care block-ul ajunge pe ele. După aceasta, veți putea implementa funcția **move**, care va deplasa block-ul în cadrul nivelului actual.

Vizualizare

Având implementate toate cele de mai sus, veți putea juca în linia de comandă. Pentru a realiza acest lucru rulați fișierul `Interactive.hs` și apelați funcția `play`, pasând level-ul ca argument. De exemplu, `play level3`. Toate level-urile sunt predefinite în fișierul `Levels.hs`.

Înainte de rulare setați valoarea `working0s` la Windows sau Linux în funcție de sistemul pe care rezolvați tema.

Rezolvarea jocului (65p)

În continuare, pentru a ajunge la starea finală va trebui să reprezentăm spațiul stărilor și să îl parcurgem. În fișierul `ProblemState.hs` veți găsi clasa care va interfața în mod generic funcțiile pentru generarea spațiului stărilor. În fișierul `Bloxorz.hs` veți crea o instanță a clasei **ProblemState** pentru jocul din enunț cu tipurile **Level** și **Direction**.

Apoi, în fișierul `Search.hs` va trebui să va construiți tipul de date pentru a reprezenta arborele stărilor și să implementați funcția care va genera „tot” spațiul (`createStateSpace`).

Având spațiul generat vom putea implementa funcțiile **iterativeDeepening** și **limitedDfs**, care vor **explora** și vor căuta starea finală dorită. Pentru a evita reiterarea stărilor deja vizitate și generarea ciclurilor va trebui să reținem o mulțime a stărilor deja vizitate/generate. În acest sens puteți folosi structura `Set` [<http://hackage.haskell.org/package/containers-0.6.0.1/docs/Data-Set.html>] pentru o verificare eficientă.

Având posibilitatea de a găsi o stare finală, vom dori să găsim și **calea** către aceasta (secvență de mutări). Pentru aceasta va trebui să implementați funcția **extractPath**.

Pentru **debugging** puteți folosi funcția `visualize` din `Interactive.hs`, care va simula secvența de mutări găsită de soluția voastră pentru un nivel dat.

Consultați fișierul `README` din arhiva de testare pentru a vedea cum se rulează atât checkerul cât și `play` și `visualize`.

Bonus: Euristica (20p)

Dorim să accelerăm găsirea soluției prin **ordonarea succesorilor** fiecărui nod după un anumit criteriu. Astfel, vom vizita mai întâi noduri cu o probabilitate mai mare de a ne conduce spre starea finală. Implementați funcția **heuristic** din instanța clasei `ProblemState` pentru `Level` și `Direction`. Pe baza acesteia, copiii fiecărui nod din spațiul stărilor vor fi **ordonați**, implementând funcția **orderStateSpace**. Euristica este la alegerea voastră. Punctajul obținut depinde de numărul de soluții îmbunătățite: 1, 2 sau cel puțin 3.

Precizări

- Atenție!** 10 puncte din 100 sunt alocate implementării **fără** recursivitate explicită a funcțiilor, cu **excepția** `createStateSpace`, `orderStateSpace` și `limitedDfs` (3.33p / funcție).
- Toate nivelele descrise în `Levels.hs` și folosite pentru testare au un winning cell unic, **CU EXCEPȚIA** `testLevel0`, care va fi folosit doar pentru testarea mișcărilor simple pe un board, și va fi format **DOAR** din **Hard Cells**.
- Exploatați cu încredere **pattern matching** și **gărzi**, în locul *if*-urilor imbricate.
- Având în vedere că unele funcții din modulele `Data.Array` și `Data.Set` au același nume cu funcțiile pe liste (exemple: `map`, `filter`), **conflictele de nume** se rezolvă prin următorul mecanism:
 - Importarea modulelor se face printr-un **alias**: `import qualified Data.Set as S`.
 - Tipurile și funcțiile din acel modul se menționează întotdeauna **prefixate** de acel alias: `S.Set`, `S.insert` etc.

Resurse

- [Schelet de pornire + checker](#)