Haskell: Mobile target search

- Deadline soft: 16.05.2021
- Deadline hard: 16.05.2021
- Data publicării: 14.04.2021
- Data ultimei modificări: 23.04.2021
- Forum temă [https://curs.upb.ro/mod/forum/view.php?id=267787]
- vmchecker (în curând)

Obiective

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

Descriere

Tema urmărește implementarea unui joc, denumit ***Mobile target search***, și a unui mecanism de rezolvare a oricărui nivel, utilizând **căutare leneșă** în spațiul stărilor. În acest sens, se va întrebuința **algoritmul A* [https://en.wikipedia.org/wiki/A*_search_algorithm]**, descris mai jos.

Mobile target search

Jocul, inspirat din *Among Us [https://en.wikipedia.org/wiki/Among_Us]*, presupune existența un vânător care urmărește mai multe ținte, scopul fiind de a le captura. Jocul se desfășoară pe un teren bidimensional, ca în diagrama de mai jos:

```
@@@@@@@@
@#  *  @
@ *  ! @
@   #  @
@@@@@@@@
```

În aceasta, putem observa următoarele entități. Am ales să utilizăm pe alocuri denumiri englezești, fiind mai naturale din perspectiva codului sursă:

- ***hunter***-ul, reprezentat prin `!`;
- ***target***-uri, reprezentate prin *;
- **obstacole**, reprezentate prin @;
- ***gateway***-uri, reprezentate prin #, între care *hunter*-ul și *target*-urile se pot teleporta.

La fiecare pas, *hunter*-ul și a *target*-urile se deplasează o poziție doar **pe orizontală sau pe verticală**, nu și pe diagonală. Denumim simbolic **mutările** posibile pentru fiecare dintre aceste entități `North`, `South`, `East` și `West`. Un *target* se consideră **capturat** dacă *hunter*-ul se află pe oricare dintre cele patru poziții adiacente.

Fiecare *target* posedă o strategie de deplasare, denumită ***behavior***, prin care alege o mutare în funcție de poziția curentă și eventual de alte informații. *Hunter*-ul poate primi într-un pas mutarea de la **tastatură** sau și-o poate **determina singur** pe baza unui algoritm de căutare. În a doua variantă, presupunem că *target*-urile își **planifică** mai întâi traseul către poziția de moment a unui *target*, dar se pune întrebarea cât din acest traseu ar trebui apoi urmat, din moment ce *target*-urile sunt **în mișcare**. Pentru simplitate,

presupunem că este aleasă **doar prima mutare** din acest traseu planificat, și că traseul este **refăcut** în momentul următor de timp, pentru a reflecta noua poziție a *target*-ului.

Spațiul stărilor problemei

În a doua variantă de mai sus, în care *hunter*-ul își planifică traseul către poziția de moment a unui target, are loc un așa-numit proces de **căutare în spațiul stărilor** problemei. Mai precis, *hunter*-ul își **„imaginează”** traseuri posibile pe teren, urmând să aleagă în final un traseu cât mai bun. Având în vedere că *hunter*-ul **nu cunoaște** *behavior*-urile *target*-urilor, considerăm că traseele imaginate de *hunter* reflectă **doar mutările acestuia, nu și pe ale *target*-urilor**, de ca și cum acestea ar rămâne pe loc, și nu ar fi afectate de mutările *hunter*-ului.

Spațiul stărilor poate fi reprezentat ca un **graf**, în care nodurile sunt **configurațiile** posibile ale jocului, reflectând doar mutările *hunter*-ului (v. paragraful anterior), iar muchiile sunt **mutările (acțiunile)** *hunter*-ului, care asigură tranzițiile dintre stări. Mai jos, este prezentată o parte a spațiului stărilor, începând de la configurația exemplificată mai sus:

Se observă că prima stare succesor, în care se ajunge prin acțiunea `North`, corespunde **capturii** *target*-ului de sus, datorită adiacenței dintre pozițiile celor doi.

Atenție! Acțiunile sunt **reversibile**, ceea ce înseamnă că din starea din dreapta-jos, în care se ajunge prin acțiunea `West`, se poate reveni în starea inițială prin acțiunea `East`. Implicația este că, la realizarea parcurgerii, este necesară reținerea stărilor **vizitate**, pentru **evitarea ciclurilor**.

Algoritmul A*

Algoritmul A* [https://en.wikipedia.org/wiki/A*_search_algorithm] este unul de **căutare informată** în spațiul stărilor. Acest lucru înseamnă că, spre deosbire de căutările în lățime (BFS) sau adâncime (DFS), care vizitează nodurile în ordinea în care acestea apar în graf, A* **prioritizează** expandarea nodurilor mai „promițătoare”, despre care se estimează că sunt mai apropiate de nodul căutat.

BFS utilizează o **coadă** pentru stabilirea ordinii de expandare a nodurilor, și opțional o mulțime de noduri finalizate pentru evitarea ciclurilor. Coada, numită și **frontieră**, conține nodurile atinse în procesul de vizitare, dar pentru care vecinii nu au fost încă enumerați; acest lucru urmează să se întâmple de-abia la înlăturarea nodului din coadă, concomitent cu adăugarea lui la mulțimea de noduri finalizate.

Una dintre proprietățile BFS este că întotdeauna sunt expandate nodurile cu adâncime `n` (calculată față de un nod inițial) din graf, **înainte** de a le expanda pe cele cu adâncime `n + 1`. Astfel, specificul BFS poate fi înțeles (aproximativ) și din perspectiva unei **cozi de priorități** în care prioritatea unui nod este dată de **adâncimea** sa, astfel că în fiecare moment de timp este extras un nod cu prioritate **minimă**.

A* exploatează această idee a cozilor de priorități, pentru a adăuga la prioritatea anterioară dată de adâncimea unui nod o **estimare** a distanței până la nodul căutat, calculată cu o funcție **euristică**. Astfel, într-un pas, este selectat din coadă în vederea expandării nodul cu **suma minimă** dintre adâncimea sa și distanța estimată până la destinație. În plus, spre deosbire de BFS, este posibil ca un nod să fie **reintrodus** în frontieră (coada de priorități), dacă se ajunge din nou la starea respectivă, dar de data aceasta cu o sumă **mai mică**. Dacă estimarea este tot timpul 0, A* **se reduce** la BFS (aproximativ).

Veți avea ocazia să implementați acest algoritm în cadrul temei, urmărind etapizarea propusă în schelet.

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.

Partea 1: Implementarea regulilor jocului și afișarea (40p)

Elementele care compun jocul, regulile de mutare și afișarea, se vor implementa în fișierul `Basics.hs`. Va trebui să completați propriile definiții pentru tipurile de date și funcțiile din fișier, urmărind TODO-urile.

Reprezentarea stării jocului sub formă de șir de caractere, în vederea afișării la consolă și a testării, va fi realizată definind funcția `gameAsString :: Game -> String` (sinonim pentru funcția predefinită `show`), și va lua forma din diagramele de mai sus.

Odată definite tipurile și funcțiile de mai sus, puteți citi indicațiile din secțiunea de **Rulare interactivă**, pentru a da comenzi de la tastatură *hunter*-ului.

Partea 2: Implementarea algoritmului de căutare (60p)

În continuare, pentru a îi permite *hunter*-ului să își **planifice** drumul către o țintă, 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 `Basics.hs`, veți crea o instanță a clasei `ProblemState` pentru jocul din enunț cu tipurile `Game` și `Direction`. Euristica pe care o veți folosi pentru estimarea distanței rămase până la starea dorită (în cadrul funcției `h` a clasei) este **distanța euclidiană**, predefinită în fișier.

Apoi, în fișierul `Search.hs` va trebui să va construiți tipul de date `Node s a` pentru a reprezenta **spațiul stărilor** și să implementați funcția care va genera „tot” spațiul (`createStateSpace`). Aici merită evidențiată distincția dintre o **stare**, care reprezintă o configurație a jocului, în care se poate ajunge prin **diferite** secvențe de mutări, și un **nod**, care desemnează o stare, dar în plus surprinde o secvență **particulară** de mutări care a condus la acea stare.

Ulterior, veți defini în același fișier funcțiile care, pas cu pas, conduc la implementarea algoritmului **A*** (funcția `astar`), cu posibilitate de determinare a **căii** de la nodul inițial la cel final, prin funcția `extractPath`.

Bonus (20p)

Punctajul de bonus este **împărțit** pe cele două părți ale temei, după cum urmează.

Pentru **Partea 1** (10p), se cere implementarea unui ***behavior*** mai complex în fișierul `Basics.hs`, care să simuleze o deplasare **circulară**, în jurul unei poziții a terenului, pe o rază fixată. Pentru aceasta, implementați funcția `circle`. Aceasta **nu este testată automat**, și aveți libertatea să nuanțați comportamentul. Punctajul va fi acordat la **prezentarea temei**, dacă puteți demonstra că *target*-ul poate realiza un **cerc complet** în jurul centrului.

Pentru **Partea 2** (10p), se cere definirea unei **euristici netriviale** (de exemplu, funcția `h s = 0` nu este acceptată), care să îmbunătățească mutările alese de *hunter*. Având în vedere că implementarea noii euristici poate **interfera** cu testele aferente Părții 2 a temei, puteți instanția separat clasa `ProblemState`, de data aceasta cu tipul `BonusGame` din `Basics.hs`, care nu este decât un ***wrapper*** peste tipul `Game`.

Acest artificiu este necesar pentru că nu pot exista două instanțe ale aceleiași clase pentru același tip, `Game`.

Precizări

Această secțiune prezintă detalii suplimentare cu relevanță pentru implementarea voastră.

Dependențe

În cadrul algoritmului A*, pentru reprezentarea **mulțimii de noduri finalizate** (numită `visited` în schelet), se utilizează modulul `Data.Set`. De asemenea, pentru **coada de priorități** (numită `frontier` în schelet), vom folosi modulul `Data.PSQueue`. Acesta din urmă trebuie instalat prin comanda:

```
> stack install PSQueue
```

Găsiți un **tutorial** de utilizare a celor două module în fișierul `DemoSetPSQueue.hs`.

Dacă, totuși, interpretorul generează eroare din cauză că nu găsește ultimul modul, puteți edita fișierul `stack.yaml` din instalarea `stack` și adăuga linia:

```
extra-deps: [PSQueue-1.1.0.1]
```

O alternativă ar fi opțiunea `--package`. Exemplu:

```
> stack exec ghci --package PSQueue TestMTS.hs
```

Detalii și constrângeri de implementare

- **Atenție!** Funcțiile `gameAsString`, `successors`, `suitableSuccs`, `insertSuccs` și `extractPath` trebuie implementate **fără recursivitate explicită**. Nerespectarea acestei cerințe va conduce la **penalizări** de 2p din 100 per funcție.
- Utilizați, pe cât posibil, **funcționale** și ***list comprehensions***.
- Exploatați cu încredere ***pattern matching* și gărzi**, în locul *if*-urilor imbricate.
- Încercați să folosiți la maxim funcții predefinite din modulul Data.List [https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-List.html]. Este foarte posibil ca o funcție de **prelucrare a listelor** de care aveți nevoie să fie deja definită aici.
- Pentru **rularea testelor**, vedeți fișierul `Readme`.
- Arhiva pentru **vmchecker** este suficient să conțină fișierele `Basics.hs` și `Search.hs`.

Rulare interactivă

Pentru a putea vizualiza rezultatele implementării voastre, fișierul `Interactive.hs` vă pune la dispoziție trei funcții:

- `interactive` poate fi folosită după implementarea **Părții 1** a temei, pentru a controla *hunter*-ul cu comenzi de la tastatură;
- `hunt` poate fi folosită după implementarea **Părții 2** a temei, pentru a observa cum *hunter*-ul alege singur mutările;
- `bonusHunt` poate fi folosită pentru **bonus-ul Părții 2**.

Toate acestea utilizează funcția `loadGame` din fișierul `Terrain.hs`.

Timeline

Trimiterea temei **NU** se va face în etape, dar scheletul este foarte **structurat** și vă permite să progresați constant. Sugestia noastră este să finalizați **Partea 1** a temei în jurul zilei **laboratorului 8**.

Resurse

- Schelet și checker [https://ocw.cs.pub.ro/courses/_media/pp/21/teme/haskell/mts.zip]

Changelog

- 23.04 (ora 15:35) - Adăugare constrângere `extractPath`
- 22.04 (ora 18:00) - Corectură la subtestul `advanceGame 6`
- 20.04 (ora 15:28) - Modificare mică pentru testarea `suitableSuccs`
- 16.04 (ora 10:15) - Clarificată distincția dintre funcțiile `gameAsString` și `show`.
- 16.04 (ora 00:12) - Mențiune rulare.
- 15.04 (ora 09:22) - Actualizare schelet.