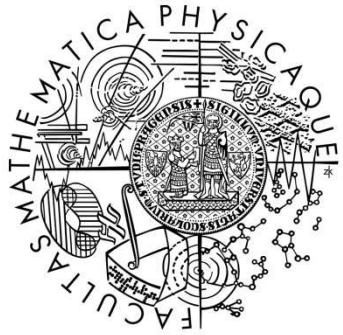
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



István Satmári

Adaptace chování neumírajících agentů

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Otakar Trunda

Studijní program: Informatika, obecná informatika

Praha 2015

Rád by som vyjadril srdečné poďakovanie vedúcemu svojej práce, pánovi Mgr. Otakarovi Trundovi, za cenné pripomienky a všetok čas, ktorý mi venoval.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Prahe dňa 31. 7. 2015 István Satmári

Názov práce: Adaptace chování neumírajících agentů

Autor: István Satmári

Katedra (ústav): Katedra teoretické informatiky a matematické logiky

Vedúci bakalárskej práce: Mgr. Otakar Trunda

Abstrakt: Predmetom bakalárskej práce je návrh a implementácia aplikácie umožňujúcej simuláciu jednoduchých neumierajúcich agentov na dvojrozmernej ploche, ktoré sú riadené svojím vlastným kódom. Rozhodovanie agentov je adaptívne. Aplikácia je zameraná na flexibilitu a umožňuje užívateľovi namodelovať veľké množstvo situácii a popísať veľa scenárov. Súčasťou práce je aj prevedenie s danou aplikáciou niekoľkých testov a popis ich výsledkov.

Kľúčové slová: agentový systém, adaptácia, rozhodovanie, riešenie problémov, neumierajúci agenti

Title: Adaptation of behavior of non-dying agents

Author: István Satmári

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Otakar Trunda

Abstract: The task of the bachelor thesis is to design and implement an application allowing a simulation of simple non-dying agents on a two-dimensional space. The agents are controlled by their own algorithm and their behavior is adaptive. The application's aim is flexibility and it allows a user to model a great amount of situations. Another part of the task was to conduct a few tests with the application and to describe the results.

Keywords: agent system, adaptation, decision making, problem solving, non-dying agents

Obsah

Úvod

Prevádzanie simulácií s agentmi je súčasťou mnohých vedeckých výskumov, ktoré sa venujú vývoju populácie v závislosti od okolitých podmienok. Typickým príkladom je tzv. umelý život, ktorý spája biológiu a informatiku a študuje tento fenomén vytváraním systémov, ktoré simulujú život organizmov v určitom prostredí. Využitie nachádza aj v rozličných štúdiách spoločenstiev ako je napríklad sociológia.

Cieľom tejto práce preto bolo vytvoriť aplikáciu umožňujúcu simuláciu jednoduchých bytostí, ktoré sú riadené vlastným kódom na dvojrozmernej ploche. Táto plocha predstavuje prostredie, v ktorom dané bytosti spolunažívajú. Z tohto dôvodu sú označované v kontexte tejto aplikácie ako organizmy, napriek tomu však tieto bytosti nezomierajú, ani nemôžu samovoľne pribudnúť.

Kód agentov predstavuje správanie, od ktorého v každom kroku simulácie odvodzujú, akú činnosť majú vykonať. Ten sa taktiež v priebehu simulácie vyvíja, čo chápeme tak, že sa daní agenti voči špecifikovanému prostrediu adaptujú.

V prvej kapitole rozoberieme funkcie, ktoré od našej aplikácie budeme požadovať a navrhneme spôsob ich implementácie. Naznačíme ďalšie problémy, ktoré sa cestou objavia a možné riešenia. Nakoniec porovnáme náš návrh s programami podobného zamerania.

V druhej kapitole sa budeme podrobne venovať práci s hotovým programom. Uvedieme niekoľko názorných príkladov použitia aplikácie. Načrtneme, čo by sme mohli dosiahnuť jednotlivými funkciami.

Tretia kapitola bude zľahka popisovať konkrétnu implementáciu. Rozoberieme najvýznamnejšie triedy a ich vzájomnú komunikáciu. Pozrieme sa taktiež na niektoré zaujímavejšie použité algoritmy.

V rámci štvrtej kapitoly sa zameriame na prevedenie niekoľkých experimentov, ktoré ukazujú možnosti a situácie, ktoré sa dajú touto aplikáciou namodelovať. Budú taktiež rozobraté vplyvy vybraných parametrov na vývoj kódu agentov a taktiež rozobraté niektoré správania a stratégie, ku ktorým populácia dospela.

1. Analýza problému

1.1 Úvod

Simulátor by mal umožňovať užívateľovi prevádzanie simulácií s jednoduchými, neumierajúcimi agentmi, ktorých rozhodovanie je adaptívne. Zároveň by sa mala simulácia odohrávať na dvojrozmernej ploche. My sa obmedzíme na agentov, ktorí dokážu vykonávať len nasledujúce činnosti:

- Krok vpred
- Otočenie vľavo o 90°
- Otočenie vpravo o 90°
- Čakať

Tento princíp je prebratý predovšetkým z programu Broučci [1], ktorý sa zaoberá simuláciou agentov riadených vlastným kódom. Umožňuje základný pohyb po prostredí. Je však možné predpokladať, že sa na ploche budú objavovať aj špeciálne objekty, s ktorými budú agenti môcť interagovať. Žiadna z vymenovaných činností podporu pre explicitnú interakciu (t.j. iniciovanú samotným agentom) nepodporuje a činnosť Čakať by mala slúžiť pre ničnerobenie. Z tohto dôvodu zavedieme ešte jednu činnosť práve pre tento účel:

- Akcia

Činností by bolo možné zadefinovať podstatne viac, ale snažíme sa v rámci tejto práce udržať simulátor jednoduchý, avšak stále dostatočne flexibilný.

Každý krok simulácie by mal predstavovať rozhodnutie a vykonanie činnosti každým agentom. Tu sa nám ponúkajú 2 možnosti:

- Každý agent vykoná v jednom kroku práve jednu činnosť
- Každý agent vykoná v jednom kroku niekoľko činností

My sa obrátime na prvú možnosť, pretože umožnenie agentom vykonávať niekoľko činností naraz v každom kroku môže viesť ku skomplikovaniu práce so simulátorom a navyše viesť k názoru, že niektoré činnosti popísané vyššie, ako napríklad Čakať, sú zbytočné. V každom kroku teda vynucujeme, aby sa každý agent rozhodol pre práve jednu z dostupných činností a vykonal ju.

1.2 Prostredie

Ako prostredie, v ktorom budú agenti spolunažívať, budeme uvažovať jednoduchú dvojrozmernú štvorčekovú mapu. Nebudeme rozlišovať veľkosť

agentov a preto prehlásime, že každý zaberá v každom kroku simulácie práve jedno políčko.

Simulátor by mal poskytovať flexibilitu pri navrhovaní prostredia, v ktorom budú agenti spolunažívať. Zároveň by tvorba prostredia nemala byť príliš komplikovaná a zdĺhavá. Pozrime sa na požiadavky, ktoré kladieme na jeho prispôsobovanie:

- Na rozličných políčkach môže stáť rozličný maximálny počet agentov naraz.
- Niektoré políčka predstavujú stenu, cez ktorú sa nedá prejsť.
- Políčka majú rozličný výzor, aby sa v nich užívateľ dokázal orientovať.
- Políčka s agentmi interagujú na základe užívateľom definovaných pravidiel.
- Políčka sa môžu dynamicky meniť, čo by mohlo simulovať napr. striedanie ročných období, prípadne simulovať akúkoľvek inú zmenu prostredia s časom.

Nekladieme požiadavku na vzájomnú interakciu samotných políčok medzi sebou, nakoľko nám v tejto práci ide o simuláciu agentov v prostredí a nie prostredia samotného.

Keďže je reálne očakávať, že mnohé políčka sa na ploche budú správať rovnako, alebo budú aspoň podobného typu, v snahe zjednodušiť prácu s aplikáciou zavedieme iba prispôsobovanie tzv. blokov. Každé políčko tak bude tvorené práve jedným z vopred určených blokov.

Z požiadaviek, ktoré na prostredie kladieme, vyrieši prvé dve jednoduchá vlastnosť bloku, ktorú označíme ako nosnosť. Tá predstavuje maximálne množstvo agentov, ktoré sa na daný blok vojdú. Nastavením nulovej nosnosti dostaneme stenu.

Požiadavku na výzor bloku splníme predpísaním formátu obrázkov, ktorý bude možné blokom prideliť. Túto vlastnosť nazveme ikona.

Interakciu a dynamickú zmenu políčok zabezpečíme vytvorením tzv. udalostí, ktoré sa aktivujú, pokiaľ nadobudnú platnosť užívateľom definované podmienky. Viac sa budeme udalostiam venovať v časti 1.4.

Bloky by mali mať ešte nasledujúce vlastnosti:

- **Názov.** Mal by definovať konkrétny blok, na ktorý sa bude možné cez systém udalostí neskôr odvolávať.
- **Popis.** Mal by predstavovať užívateľský komentár, na čo má daný blok slúžiť. Sprehľadní užívateľovi tvorbu mapy.

Okrem zadefinovania blokov budeme po užívateľovi chcieť ešte ich konkrétne rozmiestnenie. Preto zavedieme editor mapy, v ktorom užívateľ dokáže pohodlne určiť pozície pre každý blok a následne svoje prostredie uložiť. Keďže predpokladáme, že užívateľ využije rovnako zadefinované bloky pri tvorbe viacerých máp, umožníme zvlášť ukladanie súboru nesúceho túto informáciu, ktorý sa s konkrétnym prostredím len neskôr spáruje.

1.3 Agenti

Agenti sú objekty, ktoré sa môžu pohybovať po mape tým, že v každom kroku volia niektorú zo skôr spomínaných činností. Na agentov kladieme nasledujúce požiadavky:

- Podobne ako bloky, sú rozlíšiteľné výzorom.
- Užívateľ vie ľahko určiť ich cieľ.
- Vedia s prostredím interagovať na základe užívateľom definovaných pravidiel.
- Vedia interagovať medzi sebou na základe užívateľom definovaných pravidiel.

Podobne, ako pri blokoch, bude možné navrhnúť iba druhy agentov a samotné konkrétne jedince pri simulácii budú prislúchať práve jednému z týchto druhov.

Interakciu s prostredím a medzi sebou vyriešime takisto pomocou systému udalostí, ktorý prispôsobíme špeciálne pre agentov.

Zamerajme sa teraz na druhý bod. Nemienime priamo určovať správanie agentov, keďže k tomu musia vedieť agenti dospieť sami tým, že sa adaptujú. Je nutné však vedieť určiť ich cieľ, aby bolo možné nejakú adaptáciu vôbec sledovať. Týmto cieľom môže byť napríklad nachádzanie a prijímanie potravy, prípadne plnenie jednoduchých úloh. Cieľ by mal byť splniteľný taktiež ľubovoľne veľakrát, inak by simulácia po jeho prípadnom náhodnom dosiahnutí musela skončiť. Najjednoduchšie je preto zadefinovať pre každého agenta celočíselnú premennú, ktorá odráža jeho spokojnosť, nakoľko cieľ splnil. Po jeho splnení by mala daná premenná postupne klesať, aby mal daný agent motiváciu plniť cieľ znovu. Túto premennú označíme ako satisfakcia.

Systém udalostí teda navrhneme tak, aby bolo možné jednoducho pracovať s touto premennou. Napríklad budeme požadovať, aby agent predstavujúci nejaký organizmus po zjedení potravy mal zvýšenú satisfakciu, ale postupne s ďalšími krokmi simulácie mu daná satisfakcia klesala, čo predstavuje vyhladnutie. Naopak by užívateľ mohol vytvoriť objekty, ktoré by pre organizmus predstavovali nepriaznivú potravu, čo by sa prejavilo na znížení spomínanej satisfakcie po jej zjedení.

Zhrnieme teraz vlastnosti, ktoré by mali byť pre každého agenta editovateľné:

- **Názov.** Podobne, ako pri blokoch, názov bude jednoznačne určovať druh agenta, na ktorý sa bude možné cezeň neskôr odvolávať.
- **Popis.** Bude slúžiť ako užívateľský komentár pre sprehľadnenie, čím je konkrétny druh agenta výrazný a na čo slúži.
- **Ikona.** Bude definovať obrázok, ako majú vyzerať jedince tohto druhu.

- **Rozsah satisfakcie.** Poslúži na určenie maximálnej hodnoty satisfakcie agenta. Minimálnu hodnotu určíme ako nulovú. Dôvod, prečo by sme mohli chcieť odlíšiť tieto rozsahy pri rôznych druhoch agentov je napríklad ten, že pri modelovaní môžeme napríklad uvážiť agenta, ktorý by predstavoval väčší organizmus. Ten by musel zjesť pochopiteľne väčšie množstvo potravy, aby sa nasýtil, a preto by sme mu dali rozsah satisfakcie väčší.

- **Udalosti.** Tie budú rozobraté v nasledujúcej časti.

Okrem toho budeme požadovať, aby bol každý agent riadený svojím vlastným kódom, na základe ktorého v každom kroku simulácie vykoná práve jednu z dostupných činností. Agent sa adaptuje zmenou tohto kódu, čo sa prejaví častejším dosahovaním vysokých úrovní satisfakcie agenta.

1.4 Udalosti

Udalosti by mali predstavovať najdôležitejšiu časť simulátoru. Pomocou nich by malo byť možné zadefinovať cieľ agentov cez vplyv na satisfakciu, umožniť dynamickú zmenu prostredia a zabezpečiť interakciu medzi agentmi a agentmi a prostredím.

Udalosti musia byť ľahko nastaviteľné zo strany užívateľa a nemali by vyžadovať rozsiahle programátorské zručnosti. Preto nie je možné postaviť tento systém na vytvorení framework-u. Taktiež žiadna jednoduchá zmena by nemala vyžadovať prekompilovanie žiadnej časti programu. Z toho dôvodu bol zvolený jazyk XML. Ide o značkovací jazyk vyvinutý a štandardizovaný konzorciom W3C. Predstavuje súbor pravidiel pre kódovanie dokumentov do formátu, ktorý je ľahko čitateľný pre človeka ale taktiež aj dobre strojovo spracovateľný.

Ujasnime si najvýznamnejšie situácie a vlastnosti, ktoré by sme chceli umožniť užívateľovi nastavovať spolu s príkladmi použitia:

- **Reakcia na prítomnosť agenta na políčku, prípadne na jeho neprítomnosť.** Umožní simuláciu jedenia potravy, prípadne rast potravy práve počas neprítomnosti agenta na políčku predstavujúcom potravu.
- **Reakcia na zvolenie činnosti Akcia agentom.** Špeciálny druh políčok, ktoré aktivujú nejaké dianie na ploche. Napríklad políčko, ktoré otvorí dvere.
- **Reakcia agenta na prítomnosť iného agenta na tom istom mieste.** Vzájomná interakcia agentov, ako napríklad agent parazitujúci na iných agentoch.
- **Reakcia na čas.** Umožní striedanie vlastností prostredia a samotných agentov.
- **Zvyšovanie a znižovanie úrovne satisfakcie.** Dôležité pre určovanie cieľa agentov.

- **Zmena políčok na iné políčka.** Ako príklad poslúži zmena neprechodných políčok na prechodné, či dočasná zmena potravových políčok na prázdne políčka po ich využití.
- **Obmedzenie podmienok na agentov určitého druhu.** Agenti živiaci sa výlučne jedným druhom potravových políčok a agenti živiaci sa výlučne druhým druhom.

Priorita simulátoru je flexibilita, čiže užívateľ by mal byť schopný kombinovať navrhnuté vlastnosti. Systém by mal teda pozostávať z užívateľom definovaných podmienok, ktoré pokiaľ sú všetky splnené, vedú k vykonaniu určitých efektov. Zároveň je nutné vedieť určiť poradie vo vyhodnocovaní definovaných udalostí, čo sa vyrieši zavedením priority. Udalosti s rovnakou prioritou tak nebudú zaručovať medzi sebou žiadne poradie.

Vďaka umožneniu kombinovania podmienok a efektov do jednotlivých udalostí nie je možné sa jednoducho prihlásiť k odberu a je nutné explicitne kontrolovať naplnenie každej podmienky po každom kroku simulácie.

Prejdime k návrhu jednotlivých podmienok, ktoré umožníme užívateľovi pri definovaní udalosti voliť:

- **Náhoda.** Šanca vyjadrená v percentách, že v danom kroku bude táto podmienka vyhodnotená ako splnená. To je užitočné pre simulovanie náhodných javov.
- **Čas.** Zabezpečí reakciu na čas.
- **Stav.** Skúsenejším užívateľom umožníme testovať jednoduché celočíselné premenné pre dosiahnutie komplikovanejších schém. Rozlíšime globálny a lokálny stav.
  - **Globálny stav.** Bude zdieľaný všetkými entitami, agentmi aj políčkami.
  - **Lokálny stav.** Bude prislúchať len konkrétnej entite. Špeciálne však zabezpečíme, že lokálny stav sa pri políčkach bude viazať na ich konkrétnu pozíciu na mape a nie na políčko samotné. To môže byť užitočné, ak sa bude potrebovať nejaké políčko zmeniť na iné a odovzdať novému nejakú informáciu. Pri agentoch takéto opatrenie nemá zmysel, nakoľko od nich nepožadujeme, aby sa vedeli meniť na agentov iného druhu.
- **Organizmy.** *(Organizmami označujeme v aplikácii agentov.)* Budeme potrebovať zabezpečiť reakcie na agentov, napríklad na ich prítomnosť či vykonanie určitej činnosti. Udáme ako sadu podmienok, z ktorých všetky bude musieť splniť aspoň užívateľom zadaný minimálny počet agentov a zároveň najviac užívateľom zadaný maximálny počet agentov.

    - **Nachádza sa tu.** Zabezpečíme reakciu na prítomnosť agenta. Buď sa bude požadovať zhodná pozícia s iným agentom, alebo s konkrétnym políčkom na mape.
    - **Nenachádza sa tu.**
    - **Vykonal činnosť** [činnosť]**.** Agenti, ktorí v danom kroku vykonajú činnosť [činnosť]. Umožníme tým ešte väčšiu flexibilitu, než len reakciou na činnosť Akcia.
    - **Nevykonal činnosť** [činnosť]**.**
    - **Je nasmerovaný na** [smer]**.** Bude slúžiť pre neštandardnejšie efekty, ktoré budú závisieť na tom, v akom smere bude agent natočený.
    - **Nie je nasmerovaný na** [smer]**.**
    - **Organizmus je druhu** [názov]**.** Obmedzíme týmto podmienky iba na agentov druhu s názvom [názov].
    - **Organizmus nie je druhu** [názov]**.**
    - **Stav.** Budeme týmto testovať lokálny stav agenta.

- **Tento organizmus.** V prípade definovania udalostí pre konkrétny druh agentov budeme môcť chcieť testovať niektoré špeciálne vlastnosti samotného agenta. Z tohto dôvodu táto podmienka nemá zmysel pre bloky. Obmedzíme sa len na niektoré podmienky z predchádzajúceho bodu.

    - **Vykonal činnosť** [činnosť]**.**
    - **Nevykonal činnosť** [činnosť]**.**
    - **Je nasmerovaný na** [smer]**.**
    - **Nie je nasmerovaný na** [smer]**.**

Teraz vyriešime efekty, ktoré umožníme užívateľovi pri definovaní udalostí voliť:

- **Zmena blokov z** [názov1] **na** [názov2]**.** Týmto zabezpečíme zmenu všetkých políčok jedného druhu na políčka iného druhu. Tento efekt by mohol byť definovaný napríklad pre políčko predstavujúce kľúč, ktoré otvorí dvere tým, že zmení nejaké neprechodné políčka na prechodné.

- **Zmena tohto bloku na** [názov]**.** Umožní zmenu konkrétneho políčka na políčko iného druhu. Táto podmienka nemá zmysel pre agentov.

- **Zmena stavu.** Pokročilejším užívateľom umožníme zápis do premennej.

- **Zmena organizmov.** Budeme potrebovať umožniť zmenu niektorých vlastností určitých agentov. Najprv však bude nutné špecifikovať, ktorých agentov sa má zmena týkať. To vyriešime rozdelením tohto efektu na dve časti: na vymedzenie agentov a na určenie zmeny. Vymedzenie agentov môže pozostávať z rovnakej schémy, ako

v prípade podmienky Organizmy. Pridáme však ešte atribút maximum, pokiaľ by užívateľ nechcel pripustiť zmenu úplne všetkých agentov, ktorí vyhoveli podmienkam. Vyriešime ešte možné zmeny:

  - **Zmena stavu.** Zápis do lokálneho stavu agenta.
  - **Presun organizmov na** [názov bloku]**.** Zabezpečíme presun agentov na náhodné políčko, ktorého druh je [názov bloku]. Týmto by sme mohli napríklad simulovať výťah, alebo akýkoľvek automatizovaný pohyb v priestore.
  - **Otočiť o** [stupňov]**.** Umožníme otáčať agentov vo zvolenom smere o požadovaný uhol.
  - **Nasmerovať na** [uhol]**.**
  - **Pridať / Odobrať satisfakciu o** [hodnota]**.** Dôležité pre určovanie cieľa agentov.
  - **Nastaviť satisfakciu na** [hodnota]**.**

- **Zmena tohto organizmu.** V prípade definovania udalostí pre konkrétny druh agentov umožníme zmenu viažucu sa len na konkrétneho jedinca tohto druhu. Možné zmeny budú pozostávať z predchádzajúceho bodu.

V snahe zachovať simulátor čo najflexibilnejší sme sa pokúšali umožniť testovať a meniť čo najširší výber rozličných vlastností rozličných entít. Na mnohé situácie si však užívateľ vystačí len s niektorými základnými funkciami, ktorých požiadavky sme naznačili v úvode tejto časti.

Ako príklad použitia tohto systému môžeme uviesť klasický model hľadania potravy, ktorý rozšírime o predátorské organizmy. Zhrnieme vlastnosti, ktoré by mal tento model zahŕňať:

- Na ploche sa pohybujú bylinožravé organizmy, ktoré hľadajú potravu v podobe potravového políčka.
- Potravové políčko je po prejdení organizmom (simuluje zjedenie) vyčerpané a mení sa dočasne na prázdnu zem. Satisfakcia bylinožravých organizmov stúpa.
- Políčko zeme sa môže zmeniť po nejakej náhodnej dobe naspäť na potravu.
- Predátorské organizmy sa nemôžu živiť potravovými políčkami.
- Bylinožravé organizmy sú po prejdení predátorskými (simuluje ulovenie) „zjedené“. Ich satisfakcia klesá a naopak satisfakcia predátorských organizmov stúpa.
- Všetky organizmy s postupujúcim časom hladnú, čo sa prejavuje na znižovaní ich satisfakcie. Hľadať potravu tak musia neustále.

Tento model by napríklad obsahoval nasledujúce bloky a druhy agentov:

- Bylinožravý organizmus.

- Prázdny blok, bez udalostí. Predstavuje miesta, kde nikdy nebude rásť potrava.
- Zem, blok s jednou udalosťou. Podmienka by zahŕňala 10% náhodu (v priemere každý desiaty krok) a efekt by pozostával zo zmeny tohto bloku na potravu.
- Potrava, blok s jednou udalosťou. Za predpokladu, že sa na tomto políčku nachádza aspoň jeden bylinožravý organizmus, zabezpečíme zmenu najviac jedného takéhoto organizmu, ktorý sa tu nachádza, a pridáme mu 10 do satisfakcie. Zároveň zmeníme toto políčko na zem.
- Predátorský organizmus s jednou udalosťou. Pokiaľ sa na jeho mieste nachádza aspoň jeden bylinožravý organizmus, zabezpečíme zmenu najviac jedného takéhoto organizmu a to tak, že mu odoberieme 10 satisfakcie. Zároveň zmeníme tento organizmus a pridáme mu 10 satisfakcie.

Zostala ešte jedna nevyriešená vec a to klesanie satisfakcie všetkých organizmov s každým krokom. Vzhľadom na to, že ide o veľmi častú vlastnosť pri rozličných modeloch, túto funkcionalitu zabudujeme priamo do simulácie, odkiaľ ju bude môcť užívateľ pohodlne ovládať. Ide totiž o to, že pokiaľ by táto vlastnosť nebola v nejakom modeli zabudovaná, dostali by sme agentov, ktorí by sa nemuseli adaptovať, nakoľko by ich satisfakcia zostala nemenná a ich kód by mohol zdegenerovať na jednoduché státie na mieste.

Problémy tohto návrhu systému udalostí môžu zahŕňať o čosi vyššiu réžiu vo výpočte pri simulácii. Podmienky, ako už bolo vysvetlené, je nutné kontrolovať v každom kroku simulácie a pre každú entitu, ktorá ich obsahuje. Ak uvážime blok s nejakou udalosťou, tak sa daná udalosť musí vyhodnotiť pre každé políčko umiestnené na mape pozostávajúce z tohto bloku. Napriek tomu je jeho voľba opodstatnená vysokou flexibilitou simulátoru. Navyše nekladie veľké požiadavky na schopnosti užívateľa.

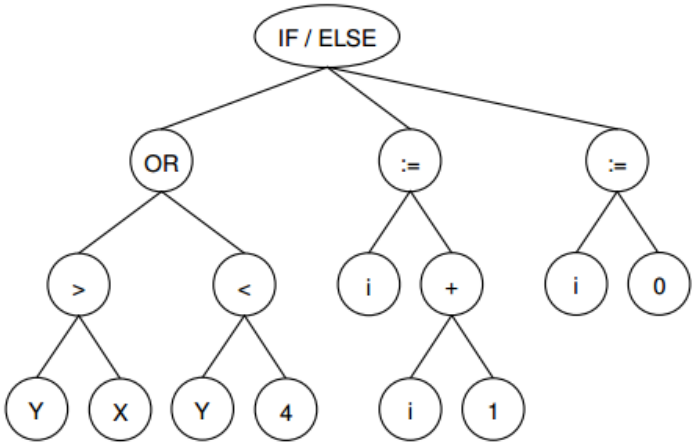
1.5 Kód agentov

Rozhodovanie agentov, akú činnosť majú v danom kroku vykonať, je dané ich vlastných kódom, ktorý má byť adaptovateľný.

Môžeme uvážiť dve hlavné reprezentácie, ktoré sa používajú:

- Stromová reprezentácia, o ktorej píše zakladateľ genetického programovania John Koza a nájdeme ju aj v literatúre o evolučných výpočtoch [2]. Ide o syntaktický strom, často podobný tomu, ktorý vznikne pri syntaktickej analýze zdrojového kódu programu. Typická reprezentácia pozostáva z dvoch typov vrcholov a to funkcií a zakončení. Zakončenia predstavujú premenné alebo konštanty, prinášajú do stromu hodnoty a sú umiestnené na konci vetiev stromových štruktúr. Funkcie, ktoré sú umiestnené vo vnútri stromu, spracovávajú premenné a konštanty.
- Lineárna reprezentácia [3] predstavuje kód uložený a vyhodnocovaný sekvenčne v podobe strojových inštrukcií.

Obidva varianty majú svoje výhody a nevýhody. Stromová reprezentácia je obdobná štruktúram, ktoré vznikajú pri syntaktickej analýze a teda môže uľahčiť načítavanie a ukladanie programu, čo budeme pre simulátor požadovať. Na druhej strane vyžaduje pomerne netriviálnu reprezentáciu v pamäti. Lineárna reprezentácia má na druhej strane síce pomerne jednoduchú implementáciu, ale je pre ňu nutné voliť inštrukcie charakteristické skôr pre zásobníkový či registrový stroj. Takéto inštrukcie sú pomerne ťažko čitateľné a užívateľ by sa v nich horšie orientoval pri posudzovaní kódu jednotlivých agentov. Zároveň je výrazne ťažšie zabezpečiť ochranu proti neplatným inštrukciám skoku používaných pri podmienkach. Mohlo by sa stať, že by vznikol nekonečný cyklus, čo by viedlo k narušeniu práce so simulátorom.



Obrázok 1.1: Príklad stromovej reprezentácie kódu [3]

My sa rozhodneme pre stromovú reprezentáciu predovšetkým kvôli sprehľadneniu práce s aplikáciou.

Problém implementácie vyriešime použitím návrhového vzoru Composite [4].

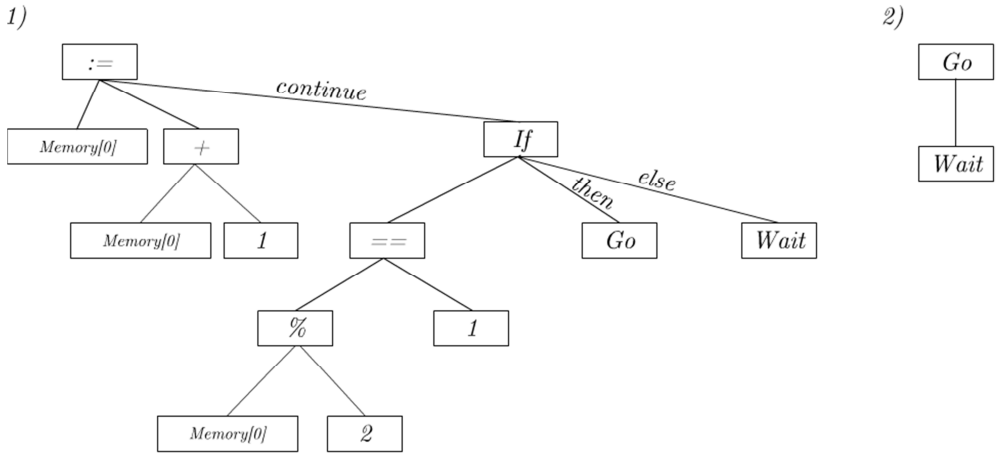
Mienime mať pomerne jednoduchých agentov a podľa toho zanalyzujeme potrebné typy vrcholov:

- Musí existovať vrchol pre určenie činnosti agenta a to bez ohľadu na to, ktorým smerom sa v strome vydáme. Agent sa totiž vždy musí rozhodnúť pre práve jednu činnosť.
- Musí existovať podmienkový vrchol If/Else pre to, aby sa agent pri voľbe činnosti mohol vôbec podľa niečoho rozhodovať.
- Agenti by mali vedieť reagovať aspoň na typ okolitých políčok a na prítomnosť ostatných agentov na okolitých políčkach. Pre to potrebujeme vrcholy zakončenia, ktoré navrátia typ políčka a počet agentov na zvolenom políčku.
- Potrebujeme vrcholy pre porovnanie s hodnotami z predchádzajúceho bodu. Pôjde aspoň o relačné vrcholy typu rovnosť a nerovnosť a konštanty.
- Pre vznik komplikovanejších algoritmov zavedieme ešte pamäťové vrcholy, ktoré budú predstavovať jednoduché premenné pre zápis

a čítanie. Pomocou nich budú môcť agenti voliť svoju činnosť nielen v závislosti od okolitých podmienok ale aj od doterajších javov a rozhodnutí.

- Kvôli predchádzajúcej podmienke zavedieme ešte funkčné uzly pre základné aritmetické operácie, ako je sčítanie a rozdiel. To umožní napríklad inkrementáciu premennej s každým krokom simulácie.

Požiadavka na vrcholy činnosti núti agenta prijímať iba stromy, kde sa tieto vrcholy vyskytujú na konci vetiev. Pokiaľ by to bolo inak, mohlo by sa stať, že by sa vyhodnocovanie stromu zastavilo na vrchole, ktoré neprináša pre agenta žiadne rozhodnutie, čo by odporovalo našej hlavnej požiadavke. Napriek tomu je vhodné túto štruktúru mierne upraviť. Uvažujme totiž prípad, kedy by sme chceli od agenta striedanie dvoch rozličných činností s každým krokom simulácie. Pri tvare stromu, ako bol popísaný doteraz, by musel byť program z pohľadu agenta netriviálne komplikovaný napriek tomu, že ide o pomerne primitívne správanie. Jednoduchým riešením je povoliť vrcholom činnosti mať ďalších synov s tou podmienkou, že ich vyhodnocovanie musí opäť končiť na nejakom vrchole činnosti. V takom prípade sa vyhodnocovanie stromu v ďalšom kroku simulácie jednoducho začne od daného vrcholu, na ktorom v minulom kroku skončil za predpokladu, že ešte má nejakého syna.



Obrázok 1.2: Porovnanie stromov, ktoré striedavo v jednom kroku vracajú činnosť Krok vpred a v ďalšom Čakať

Teraz už môžeme prejsť k návrhu samotných vrcholov.

- **Uzly činnosti.** Budú predstavovať vrcholy, ktoré nesú informáciu o tom, akú činnosť sa agent rozhodol vykonať. Budeme ich označovať ako listy pre vlastnosť, že vyhodnocovanie stromu agenta končí vždy v okamihu po narazení na tento typ vrcholov.

    - **List.** Pôjde o uzol vracajúci činnosť pre agenta.
        - **Krok vpred**
        - **Otočenie vľavo o 90°**
        - **Otočenie vpravo o 90°**
        - **Čakať**

        - **Akcia**
    - **Pokročilý list.** Pôjde o „chytrejší“ list, ktorý vráti činnosť v závislosti od aktuálneho smeru agenta. Umožní agentom lepšiu orientáciu po mape.
        - **Choď nahor.**
        - **Choď nadol.**
        - **Choď doľava.**
        - **Choď doprava.**

- **Uzly vracajúce hodnotu.** Pôjde o uzly vracajúce celočíselnú hodnotu.

    - **Konštanta.** Vráti konkrétnu celočíselnú hodnotu.
    - **Vlastná špeciálna hodnota.** Vráti hodnotu, ktorá sa bude týkať samotného agenta.
        - **Smer.** Vráti hodnoty 0, 1, 2, alebo 3 podľa natočenia agenta v zmysle pohybu hodinových ručičiek.
        - **Čas.** Pôjde o poradie simulačného kroku.
        - **Poloha na súradnici X.** Vráti poradie políčka braného vodorovne.
        - **Poloha na súradnici Y.** Vráti poradie políčka braného zvislo.
    - **Mapová špeciálna hodnota.** Vráti hodnotu, ktorá sa bude týkať situácie na mape.
        - **Druh bloku.** Vráti číslo reprezentujúce typ políčka v okolí agenta.
        - **Voľné miesto na políčku.** Vráti množstvo dostupného miesta na políčku v okolí agenta.
        - **Počet organizmov.** Vráti počet agentov nachádzajúcich sa na políčku v okolí agenta.
    - **Pamäť.** Jednoduchá celočíselná premenná pre čítanie a zápis.
    - **Výsledok operácie.** Bude predstavovať funkčný vrchol pre základné aritmetické operácie z dvoch synov.
        - **Súčet.**
        - **Rozdiel.**
        - **Súčin.**
        - **Celočíselný podiel.**
        - **Modulo.** (Zvyšok po celočíselnom delení.) Aby sme uľahčili situáciu agentom, uvážime len základnú modulárnu aritmetiku, kde napr. $-7 \% 3 = 2$.

- **Priraďovací uzol.** Bude slúžiť na priradenie konkrétnej celočíselnej hodnoty do pamäťového vrcholu. Keďže tento typ uzlu z princípu nevracia žiadnu hodnotu a ani nepredstavuje uzol činnosti, musí pokračovať vyhodnocovanie kódového stromu ďalej. Z tohto dôvodu mu pridelíme okrem dvoch potrebných synov pre zápis do premennej ešte tretieho, ktorý bude predstavovať akýsi pokračovací uzol.

- **Relácia.** Bude slúžiť pre vyhodnocovanie podmienok, či naberajú alebo nenaberajú platnosť.
    - **Rovnosť.**
    - **Nerovnosť.**
    - **Menšie, než...**
    - **Menšie alebo rovné, než...**
    - **Väčšie, než...**
    - **Väčšie alebo rovné, než...**

- **Podmienka.** Bude obsahovať troch synov, z toho jeden pre samotný test a ďalšie dva pre pokračovanie vyhodnocovania stromu v prípade, že test (ne)uspel.

1.6 Adaptácia

Aby bolo možné sledovať nejaké zmeny v populácií a vývin stratégií na rozličných mapách, je potrebné zabezpečiť adaptáciu agentov.

Cieľom tejto práce nie je navrhnúť žiaden komplikovaný algoritmus pre umelú inteligenciu agentov, a preto sa obmedzíme len na algoritmy prehľadávania. Mnohé takéto algoritmy sú však nepoužiteľné kvôli exponenciálnemu nárastu počtu rôznych kódových stromov. Z použiteľných preto uvážime tieto:

- Best-first Search
- Beam Search
- Hill Climbing
- Evolučný algoritmus

Prvé dva však kladú o niečo vyššie pamäťové nároky a sú taktiež o čosi náročnejšie na implementáciu. My sa zameriame na evolučný algoritmus, ktorý má na rozdiel od Hill Climbing algoritmu ešte tú výhodu, že má menšiu tendenciu uviaznuť v lokálnom extréme [2]. Zároveň spomedzi spomínaných metód nekladie veľké nároky na pamäť, jeho implementácia nie je nijak náročná a nepotrebuje poznať presného následníka či predchodcu pri prehľadávaní kódových stromov.

Pre adaptáciu je najprv nutné vedieť určiť, aký kód agenta je vlastne dobrý. Satisfakcia predstavuje len akési naplnenie cieľa agentov, ale jej vyššia hodnota nutne neznamená, že agent s ňou je úspešnejší, než agent s nižšou hodnotou. To je z toho dôvodu, že táto hodnota zohľadňuje iba okamžitú situáciu. Napríklad uvažujme situáciu, kde sa organizmy pohybujú na mape a hľadajú potravu. Každým krokov mierne vyhladnú, čo sa prejaví na znížení satisfakcie. Potrava im však môže naplniť satisfakciu do maxima. Za 10 krokov sa môže stať, že istý organizmus nájde potravu dvakrát v prvých piatich krokoch. Iný organizmus nájde potravu iba raz, ale až v poslednom kroku. Ak by sme vyhodnocovali úspešnosť agentov iba na základe satisfakcie, vychádzalo by, že druhý agent je úspešnejší, pretože má vyššiu

úroveň satisfakcie, nakoľko ten prvý stihol za ďalších 5 krokov mierne vyhladnúť. Z tohto dôvodu budeme vyhodnocovať úspešnosť jednotlivých agentov ako priemernú hodnotu satisfakcie vždy raz za určitý počet simulačných krokov nastaviteľných užívateľom. Toto číslo budeme označovať ako fitness.

Evolučný algoritmus teda môžeme postaviť na základe vyhodnocovania fitness. V danom bode simulácie prebehne mutácia, najlepšie kódy sa môžu zachovať bezo zmeny a niektoré kódy agentov sa prekopírujú na iných agentov. Kríženie v tomto prípade nie je nutné uvažovať, nakoľko v prípade genetického programovania, čo naša situácia určite predstavuje, vznikajú krížením skôr úplne nové kódy, než kódy, ktoré by obsahovali vlastnosti rodičovských agentov [5].

Evolučné algoritmy majú rozsiahly počet parametrov, ktoré je nutné správne nastaviť, aby vznikali rozumné stratégie. V tomto prípade je lepšie prenechať slobodu rozhodovania na užívateľovi, ktorý vie najlepšie, čo sa mu hodí pri vykonávaní konkrétnych experimentov. Z literatúry o evolučných výpočtoch [2] môžeme vyvodiť techniky používané pri generovaní novej populácie kódov:

- **Elitarizmus.** Určité percento najúspešnejších kódov sa zachová a bezo zmeny ich agenti môžu používať naďalej.

- **Náhodný výber.** Časti agentov sa pridelia kódy od iných náhodných agentov. Tieto môžu byť ďalej mutované. Tento spôsob sa snaží zachovať diverzitu kódov.

- **Turnajový výber.** Zvyšnej časti agentov sa pridelia kódy takto: Zvolí sa náhodná dvojica agentov. Vyberie sa kód úspešnejšieho agenta z tejto dvojice. Ten môže byť ďalej mutovaný.

Užívateľovi prenecháme slobodu pri rozhodovaní, aké percento agentov má získať akým z vymenovaných spôsobov nový kód. Zároveň umožníme užívateľovi nastaviť pravdepodobnosť mutácie kódu v závislosti od hodnoty fitness. To by mohlo byť užitočné pre zníženie rizika zahodenia dobrej stratégie, ktorá funguje, pre nejakú inú náhodnú stratégiu, ktorá fungovala lepšie len vďaka výnimočnej dočasnej udalosti. Zároveň by to mohlo umožniť stabilizovať dobré stratégie prípadne zlepšiť samotný výber v okamihu vyhodnocovania.

Na záver ešte poznamenajme, že tento, na prvý pohľad neštandardný, prístup skombinovania evolučného algoritmu a neumierajúcich agentov, zabezpečuje vývin stratégií spôsobom, akoby sa agenti od seba navzájom učili. To má pôvod v istých spoločenstvách a tak môže aplikácia poslúžiť na študovanie vzťahov v takýchto populáciách.

1.7 Mutácie

Pomerne netriviálna úloha je zabezpečenie mutácií kódu agentov. Bez tej by sme nemohli sledovať v populácii žiadnu reálnu adaptáciu, preto sa jej budeme venovať bližšie.

Viaceré literatúry odporúčajú zabezpečiť mutácie pozostávajúce z týchto operácií:

- **Zmena uzlu**
- **Vytvorenie náhodného podstromu**
- **Odstránenie podstromu**
- **Nahradenie podstromu náhodným podstromom**

Zmena uzlu by mala byť taktiež častejšia, ako rozsiahle pretvárania stromu [2]. Preto umožníme konkrétne pravdepodobnosti jednotlivých druhov mutácií užívateľovi nastavovať. Zároveň je vhodné obmedziť veľkosť podstromu, ktorý môže naraz v jednej mutácii vzniknúť či zaniknúť. To je z toho dôvodu, že veľkých stromoch je príliš veľa, väčšina však neprináša agentom žiadne zlepšenie, skôr naopak. Preto je lepší nápad uskutočňovať výber z menších podstromov.

Dôležité je taktiež obmedziť aj maximálnu veľkosť samotného stromu, ktorý môže vzniknúť. Tá je totiž v prípade evolučných algoritmov pomerne častý problém, pretože má tendenciu neustále rásť, pričom sa reálne uplatňujú iba niektoré vetvy pri jeho vyhodnocovaní. V niektorých literatúrach sa nepoužívajúce sa vetvy stromu označujú ako bloat [5]. Proti tomu je vhodné pridať do simulátoru aj akýsi poplatok, daň za vrchol v strome, čo by sa mohlo prejaviť na znižovaní satisfakcie daného agenta. V takom prípade si agenti zachovajú komplikovaný kód iba v prípade, že má skutočne navrch oproti tým jednoduchším, ktoré sú spoplatňované menej.

Aby sme zabezpečili flexibilitu simulátoru, pridáme možnosť upravovať jednotlivé spomínané parametre. To je podstatné, nakoľko niektoré situácie, ktoré by užívateľ mohol chcieť simulátorom namodelovať, vyžadujú iný prístup k spomínaným parametrom. Napríklad veľkosť poplatku za vrchol bude úzko súvisieť s očakávaním užívateľa, aký druh kódu sa má u agentov v danej situácii vyvinúť.

Rozoberme teraz jednotlivé operácie a problémy, na ktoré u nich môžeme naraziť.

- **Zmena uzlu.** Nepredstavuje významnejšie komplikácie, akurát si musíme dať pozor na to, aby sme nahradili vrchol iným, kompatibilným vrcholom. Napríklad je možné nahradiť konštantu nejakou špeciálnou hodnotou[1], ale nie vrcholom činnosti. Naopak vrcholy činnosti musíme nahradzovať iba inými vrcholmi činnosti,

[1] Medzi špeciálnu hodnotu spadá napríklad smer, druh políčka a i. Žiadna mutácia však nikdy nevytvorí uzol, ktorý by vracal polohu na súradnici X alebo Y. Vedomie o tejto hodnote sa dá považovať za podvod, vzhľadom na to, že agenti by mali byť považovaní za jednoduché bytosti. Tento uzol bol pridaný iba pre testovacie účely.

keďže by sa mohla porušiť podmienka, ktorá žiada vyhodnocovanie stromu vždy na niektorom ukončiť.

- **Vytvorenie náhodného podstromu.** Vygenerovanie stromu nie je náročná operácia, akurát sa musí ošetriť, aby každý vrchol, ktorý potrebuje potomka, aj potomka mal. Zároveň sa podstrom musí zmestiť do maximálnej veľkosti obmedzenej užívateľom. Je však ešte otázka, kam novo vygenerovaný podstrom umiestniť. Nie je ho možné jednoducho pripojiť pod vrchol, ktorý normálne žiadnych synov neobsahuje. Riešenie sa naskytá náhradou jediného vrcholu kompatibilným podstromom.

- **Odstránenie podstromu.** Podobne, ako v predchádzajúcom bode nie je možné jednoducho podstrom odstrániť, nakoľko by sa mohlo stať, že zanecháme vrchol na zakončení, ktorý nejakých potomkov očakáva. Aj tu vyriešime problém tým, že v skutočnosti iba nahradíme hlbší podstrom jediným uzlom.
- **Nahradenie podstromu náhodným podstromom.** Vyriešime pomocou predchádzajúcich dvoch bodov.

1.8 Simulačný krok

Je nutné spresniť, z čoho bude pozostávať simulačný krok a ako bude prebiehať.

Navrhnutý spôsob kontroly udalostí vyžaduje, aby prebiehal nezávisle od vykonávania činností agentmi. Keďže navrhnuté udalosti môžu pozostávať aj z testov, akú činnosť agent v danom kroku vykonal, je nutné ich vykonávanie zabezpečiť až po vykonaní činností všetkých agentov. U každého agenta si preto budeme udržiavať naposledy zvolenú činnosť.

Na začiatku vykonávania kroku simulácie sa teda každý agent rozhodne na základe svojho kódu, akú činnosť plánuje vykonať. Pre jednoduchosť simulátoru budeme uvažovať, že sa agenti rozhodujú postupne a nebudeme brať ohľad na to, pokiaľ sa jeden agent rozhodne z preplneného políčka odísť, na ktorom by sa tým pádom sprístupnilo miesto pre iného agenta, ktorý sa na toto políčko snaží dostať. Zároveň si tým ušetríme časť výpočtovej kapacity, ktorá je v prípade nášho simulátoru dosť cenná, nakoľko predpokladáme, že užívateľ bude potrebovať simulovať veľké počty krokov pri vykonávaní experimentov.

Vykonanie činnosti agenta však môže predstavovať problém v prípade voľby kroku vpred, pokiaľ narazí agent na okraj mapy, prípadne stojí pred políčkom, ktoré už nemá dostupné miesto. V takom prípade je najjednoduchšie preťažiť jeho zvolenú činnosť na Čakať. Musíme totiž dodržať pravidlo, že sa v každom kroku každý agent rozhodne pre práve jednu činnosť. Nešpecifikovanie činnosti odporuje logike simulátoru a predstavuje istý problém pre vyhodnocovanie udalostí.

Po vykonaní činností môže prebehnúť kontrola, či ide o krok, v ktorom sa má vyhodnotiť fitness pre každého agenta. Ak áno, prebehne zároveň aj

adaptácia jednotlivých agentov algoritmami spomenutým v predchádzajúcich dvoch častiach.

Nakoniec je nutné zabezpečiť vyhodnotenie udalostí. Tie je nutné zavolať v správnom poradí, t.j. podľa priorít definovaných užívateľom. Navyše sa daná udalosť môže vyskytnúť viackrát, nakoľko sa udalosti viažu na konkrétne entity. Preto je nutné previesť ich korektnú separáciu ešte pred začiatkom simulácie a následne počas nej dôsledne udržiavať konkrétne zoznamy udalostí, ktoré sa môžu meniť. Napríklad potravové políčko sa po využití agentom môže zmeniť na iné políčko s novou udalosťou.

1.9 Porovnanie s existujúcimi riešeniami

Existuje viacero softvérových riešení, ktoré sa venujú podobnej téme, ako táto práca. My sa pozrieme na troch vybraných zástupcov, Avida, Broučci a NetLogo.

Avida [6] študuje evolúciu programov. Jednotlivé organizmy predstavujú akési krátke aplikácie, ktoré majú schopnosť vybudovať nové aplikácie. Organizmy sú zodpovedné za vystavanie programov, ktorými sa budú riadiť ich potomkovia. Za úspešné organizmy sú považované tie, ktoré sa pokúšajú stavať identické kópie seba samých.

Program Broučci [1] sa venuje simulácii evolúcie jednoduchých bytostí, ktoré sa pohybujú po štvorčekovej ploche, na ktorej hľadajú potravu. Jeho autorom je RNDr. Tomáš Holan, PhD. a mal slúžiť ako ukážka rôznych prístupov k riešeniu komplikovaných problémov. Bytosti môžu vykonávať štyri základné činnosti a to pohyb vpred, otočenie vľavo, otočenie vpravo a ničnerobenie. Svoju činnosť odvodzujú od algoritmu, ktorý sa dá popísať ako funkcia troch premenných.

NetLogo [7] je programovateľné prostredie pre modelovanie prírodných a spoločenských javov. Jeho autorom je Uri Wilensky, pochádza z roku 1999 a je naďalej vyvíjaný v centre e-learningu a počítačového modelovania na Northwestern University v Chicagu v USA. NetLogo slúži predovšetkým na modelovanie komplexných systémov, ktoré sa menia s postupujúcim časom. Užívatelia môžu namodelovať správanie niekoľkých stoviek či tisícok jednoduchých agentov a skúmať dopad na celý systém na makro úrovni.

Náš simulátor by sa dal považovať za kombináciu niektorých myšlienok, ktoré prinášajú popísané softvérové riešenia. Inšpiruje sa programom Broučci a prepožičiava si od neho činnosti, ktoré môžu bytosti vykonávať. Zároveň sa snaží umožniť užívateľovi namodelovať podobné situácie. Podobne, ako v programe Avida, sú aj v tomto simulátore agenti riadení pomerne komplikovaným kódom, ktorý nie je jednoducho opísateľný stavovým automatom. Nakoniec umožňuje pomerne flexibilne nastaviť a popísať scenáre, ktoré by užívateľ mohol chcieť modelovať. Neprináša až také možnosti, ako program NetLogo, avšak jazyk pre popis prostredí je výrazne jednoduchší a vyžaduje menej programátorských zručností.

2. Užívateľská dokumentácia

V tejto časti popíšeme prácu s hotovým simulátorom navrhnutým tak, ako sme uviedli v prvej kapitole. Názov simulátoru je Organisms. O agentoch sa tu budeme vyjadrovať ako o organizmoch.

2.1 Spustenie aplikácie

Aplikácia Organisms bola vyvinutá pre operačné systémy MS Windows XP SP3, MS Windows Vista SP1 alebo vyšší, MS Windows 7 a MS Windows 8 s architektúrou x86 alebo x64. Predpokladá sa aspoň 512 MB pamäte RAM a procesor Pentium 1 GHz alebo vyšší.

Vyžaduje sa, aby na počítači bola podpora .NET Framework 4 alebo vyšší. Ten je možné nainštalovať aj z priloženého CD v umiestnení Prerequisites/dotNetFx40_Full_x86_x64.exe.

Samotnú aplikáciu Organisms nie je nutné inštalovať. Je ju možné priamo spustiť z priloženého CD v umiestnení Application/Organisms.exe, prípadne skopírovať priečinok Application/ do požadovaného umiestnenia na pevnom disku.

Obrázok 2.1: Hlavné okno aplikácie

Aplikácia je rozdelená na tri hlavné časti:

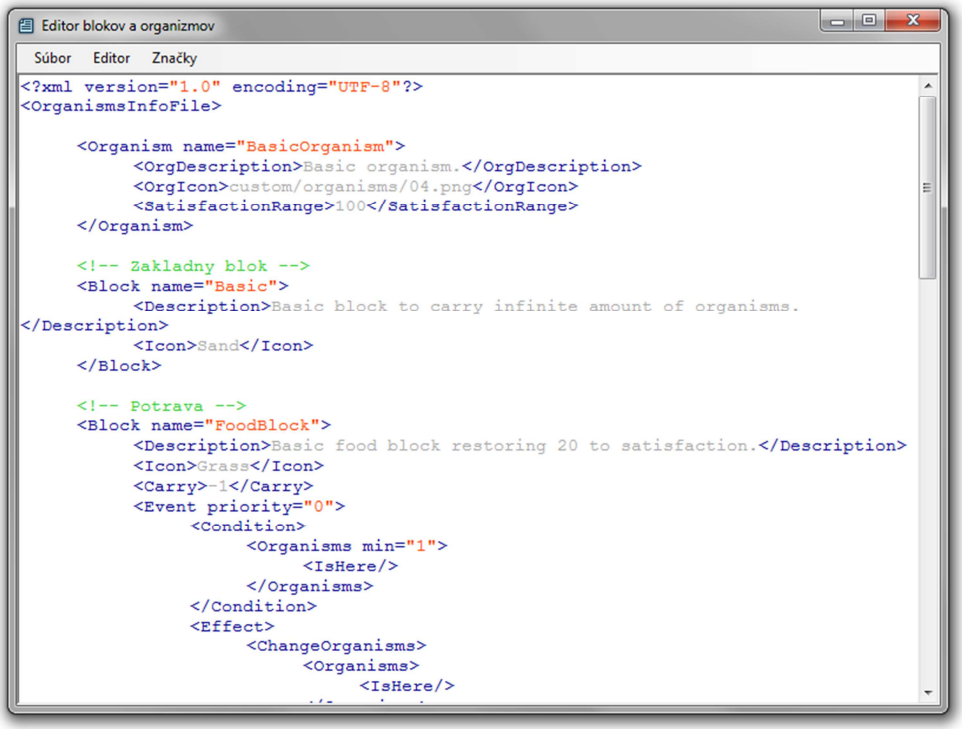
- **Editor blokov a organizmov.**
- **Editor mapy.**
- **Okno simulácie.**

2.2 Editor blokov a organizmov

Editor blokov a organizmov predstavuje okno, ktoré slúži na tvorbu a úpravu definičného súboru. Ide o súbor, ktorý zahŕňa informácie o blokoch a druhoch agentov, ktoré sa na mape budú môcť vyskytovať. Nejde o konkrétnu hotovú mapu, ktorá pozostáva z rozmiestnenia týchto blokov na konkrétne pozície. Tej sa budeme venovať v časti 2.3.

Jeho vstupom aj výstupom je súbor vo formáte XML. Editor je možné otvoriť výberom Mapa/Otvoriť editor blokov a organizmov z hlavného menu hlavného okna aplikácie.

Ide o jednoduchý textový editor so špeciálnymi funkciami uľahčujúcimi tvorbu definičného súboru. Ten má pevne danú štruktúru odvíjajúcu sa od štandardu XML. Pozostáva z nasledujúcich častí:



Obrázok 2.2: Editor blokov a organizmov

- **Hlavička.** Ľubovoľný definičný súbor musí obsahovať hlavičku, ktorá pozostáva z nasledujúcich konštruktov:

```
<?xml version="1.0" encoding="UTF-8"?>
<OrganismsInfoFile>

</OrganismsInfoFile>
```

Všetky ďalšie konštrukty musia byť vpisované medzi štítky `<OrganismsInfoFile>` a `</OrganismsInfoFile>`. Hlavičku je možné vložiť výberom Značky/Hlavička.

- **Komentár.** Správa sa ako štítok, ktorý je pri čítaní definičného súboru odignorovaný. Má nasledujúci tvar:

```
<!-- Ľubovoľný text -->
```

- **Blok.**

```
<Block name="NázovBloku">
   <Description>Popis zobrazovaný v aplikácii.</Description>
   <Icon>Sand</Icon>
   <Carry>7</Carry>
   <!-- Sem umiestniť udalosti (pokiaľ sa vyžadujú) -->
</Block>";
```

Definuje nový blok. Medzi `<Carry>` a `</Carry>` sa uvádza nosnosť bloku, medzi `<Icon>` a `</Icon>` sa uvádza ikona. Tento konštrukt je možné vložiť výberom Značky/Blok/Nový blok.

V rámci uvedenia ikony je možné využiť niektorý zo vstavaných obrázkov:

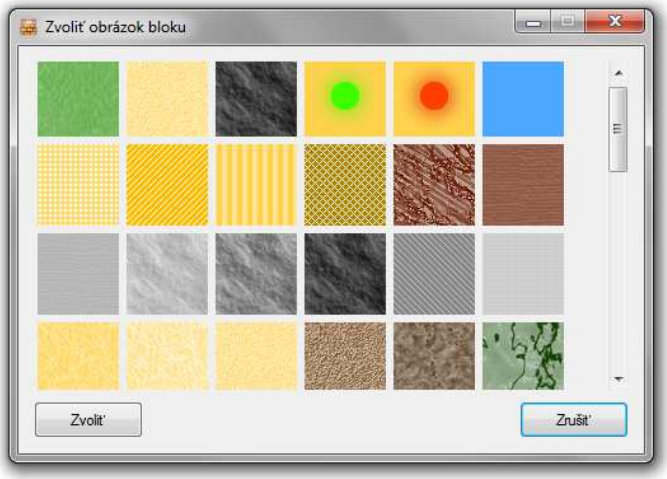
```
Grass
Sand
Stone
SwitchGreen
SwitchRed
Water
```

Prípadne je možné nahrať vlastné obrázky do priečinka custom/mapblocks/ (cesta je relatívna vzhľadom k umiestneniu aplikácie). Obrázky musia byť vo formáte PNG, BMP, JPEG, GIF alebo TIF o veľkosti 64x64 pixlov. Ostatné súbory sú ignorované. Z uložených obrázkov je potom možné použiť ikonu odkázaním:

```
custom/mapblocks/NázovSúboru.prípona
```

Ak sa daný obrázok nenájde, je na jeho miesto použitý štandardný obrázok prázdneho bloku.

Inak je možné použiť aj okno pre voľbu ikony výberom Značky/Blok/Obrázok bloku.



Obrázok 2.3: Okno pre voľbu ikony bloku

Na požadovaný obrázok je nutné kliknúť a potvrdiť tlačidlom Zvoliť.

- **Organizmus.**

```
<Organism name="NázovOrganizmu">
   <OrgDescription>Popis zobrazovaný v aplikácii.</OrgDescription>
   <OrgIcon>default</OrgIcon>
   <SatisfactionRange>1000</SatisfactionRange>
   <!-- Sem umiestniť udalosti (pokiaľ sa vyžadujú) -->
</Organism>
```

Definuje nový druh organizmu (agenta). Medzi `<OrgIcon>` a `</OrgIcon>` sa uvádza ikona organizmu, medzi `<SatisfactionRange>` a `</SatisfactionRange>` sa uvádza rozsah satisfakcie.

Je možné vložiť výberom Značky/Organizmy/Nový organizmus.

Ikona sa uvádza podobne, ako v prípade bloku, avšak aplikácia obsahuje iba jeden vstavaný obrázok pre organizmy. Ten sa použije aj v prípade, že nebude nájdený externý obrázok.

Externé obrázky je nutné nahrať pre zmenu do priečinka custom\organisms\, avšak musia spĺňať rovnaké podmienky, ako v prípade obrázkov blokov. Je ich potom možné použiť odkázaním:

```
custom/organisms/NázovSúboru.prípona
```

Okno pre voľbu ikony je možné otvoriť výberom Značky/Organizmy/Obrázok organizmu. Ovláda sa rovnako, ako v prípade blokov.

- **Udalosť.**

```
<Event priority="0">
   <Condition>
      <!-- Sem umiestniť podmienky -->
   </Condition>
   <Effect>
      <!-- Sem umiestniť efekty -->
   </Effect>
</Event>
```

Definuje novú udalosť. Priorita udalosti sa uvádza za slovom `priority` v úvodzovkách.

- **Podmienky.**

Je možné skombinovať nasledujúce konštrukty pre vytvorenie podmienky pre danú udalosť:

```
<Chance rate="70"/>
```

Predstavuje náhodu. Šanca sa uvádza za kľúčovým slovom `rate`. Musí ísť o celé číslo.

```
<Time method="GlobalExactly" value="45" offset="7"/>
```

Predstavuje podmienku ohľadom poradia simulačného kroku. Za `value` a `offset` je nutné uviesť celé čísla. Za `method` je nutné uviesť jeden z nasledujúcich spôsobov porovnania:

`GlobalExactly` pre presnú zhodu s `value + offset`

`GlobalAtLeast` pre minimálnu hodnotu kroku ako `value + offset`

`GlobalLessThan` pre menšiu hodnotu kroku než `value + offset`

`GlobalMod`
pre modulo kroku znížený o `offset` hodnotou `value` rovnajúce sa `0`
`(krok - offset) % value == 0`
bude splnené každých `value` krokov, počnúc krokom `offset`
(za predpokladu, že je `offset` menší, než `value`)

`GlobalNoMod` pre doplnok k predchádzajúcemu spôsobu
`(krok - offset) % value != 0`

```
<State method="Mod" state="NázovStavu" value="25" offset="7"/>
```

Predstavuje testovanie stavu. Za `value` a `offset` je nutné uviesť celé čísla. Za `state` sa uvádza názov stavu. Stavy s rozličným názvom predstavujú rôzne premenné. Za `method` sa uvádza spôsob porovnania:

Pre porovnávanie s globálnym stavom:
`GlobalExactly, GlobalAtLeast, GlobalLessThan, GlobalMod, GlobalNoMod`

Pre porovnávanie s lokálnym stavom:
`Exactly, AtLeast, LessThan, Mod, NoMod`

Pred prvým priradením má akýkoľvek stav hodnotu `0`.

```
<Organisms min="4" max="7">
      <!-- Sem umiestniť podmienky -->
</Organisms>
```

Predstavuje sadu podmienok pre organizmy. Uvedené podmienky musí spĺňať minimálne `min` a maximálne `max` organizmov. Atribút `max` nie je nutné uvádzať. Špeciálne ak je `min` rovné `-1`, potom musia podmienky spĺňať všetky organizmy na ploche.

Podmienky pre organizmy je možné kombinovať z nasledujúcich konštruktov:

```
<IsHere/>
```

Nachádza sa tu.

```
<IsNotHere/>
```

Nenachádza sa tu.

```
<UsedStep value="Go"/>
```

Vykonal činnosť `value`. Možnosti sú nasledujúce:

`Wait` pre Čakať

`Go` pre Krok vpred

`TurnLeft` pre Otočenie vľavo o 90°

`TurnRight` pre Otočenie vpravo o 90°

`Action` pre Akcia

```
<DidNotUseStep value="TurnLeft"/>
```
Nevykonal činnosť `value`. Je možné uviesť viackrát.

```
<Direction value="90"/>
```
Je nasmerovaný na `value` stupňov. Možnosti: `0, 90, 180, 270`. Počíta sa v smere pohybu hodinových ručičiek.

```
<NotDirection value="270"/>
```
Nie je nasmerovaný na `value` stupňov. Je možné uviesť viackrát.

```
<IsOfType value="NázovOrganizmu">
```
Organizmus je druhu `value`.

```
<IsNotOfType value="NázovOrganizmu">
```
Organizmus nie je druhu `value`. Je možné uviesť viackrát.

```
<OrganismState method="AtLeast" state="NázovStavu" value="35"
offset="54"/>
```
Testovanie lokálneho stavu organizmu. Je možné uviesť viackrát. Uvádzanie jednotlivých atribútov je zhodné s tým ako pri samotnej podmienke stavu, avšak nie je možné testovať globálny stav. Špeciálne je tu možné uviesť `state="satisfaction"` pre testovanie hodnoty satisfakcie organizmu.

```
<ThisOrganism>
      <!-- Sem umiestniť podmienky -->
</ThisOrganism>
```
Predstavuje sadu podmienok pre *tento* organizmus. Konštrukty sú analogické tým z podmienok pre organizmy.

Jednotlivé konštrukty je možné uvádzať viackrát. Podmienky je možné vložiť aj výberom Značky/Blok/Podmienka/... či Značky/Organizmy/Podmienka/...

- **Efekty.**

Je možné skombinovať nasledujúce konštrukty pre vytvorenie efektu pre danú udalosť:

```
<ChangeBlocks to="NázovBloku2" from="NázovBloku1"/>
```
Predstavuje zmenu blokov `from` na bloky `to`.

```
<ChangeThisBlock to="NázovBloku"/>
```
Predstavuje zmenu *tohto* bloku na blok `to`.

```
<ChangeState method="GlobalSet" state="NázovStavu" value="35"/>
```
Predstavuje zmena stavu s názvom `state`. Za `method` je nutné uviesť jeden z nasledujúcich spôsobov zmeny:

Zmena globálneho stavu:

`GlobalSet` pre priradenie hodnoty `value`
`GlobalAdd` pre pričítanie hodnoty `value`
`GlobalTake` pre odčítanie hodnoty `value`
`GlobalMultiply` pre vynásobenie hodnotou `value`
`GlobalDivide` pre celočíselný podiel hodnotou `value`

Zmena lokálneho stavu:
`Set, Add, Take, Multiply, Divide`

Pred prvým priradením majú všetky stavy hodnotu `0`.

```
<ChangeOrganisms>
   <Organisms>
      <!--
         Vymedzenie, ktorých organizmov sa má zmena týkať.
         Udáva sa rovnakými podmienkami, ako v prípade podmienky pre
         organizmy.
       -->
   </Organisms>
   <Change max="-1">
      <!-- Sem umiestniť zmeny pre zvolené organizmy -->
   </Change>
</ChangeOrganisms>
```

Predstavuje sadu zmien organizmov. Tie sa prevedú na maximálna `max` organizmoch. Špeciálne, pokiaľ je `max` rovné `-1`, potom sa zmeny týkajú všetkých vybratých organizmov. Inak sa z nich náhodne zvolí `max`.

Zmeny je možné skombinovať z nasledujúcich konštruktov:

```
<OrganismChangeState method="Add" state="NázovStavu" value="18"/>
```

Zmena lokálneho stavu organizmu. Atribúty sa uvádzajú analogický tým z `ChangeState`, avšak nie je možné meniť globálny stav.

```
<MoveOrganisms to="NázovBloku"/>
```

Presun organizmov na náhodný blok s názvom `to`.

```
<TurnLeft value="90"/>
```

Otočenie organizmov proti smeru pohybu hodinových ručičiek o `value` stupňov. Možnosti: `0, 90, 180, 270.`

```
<TurnRight value="180"/>
```

Otočenie v opačnom smere.

```
<SetDirection value="270"/>
```

Nastavenie smeru na `value` stupňov počítané proti smeru pohybu hodinových ručičiek.

```
<AddSatisfaction value="41"/>
```

Zvýšenie satisfakcie o `value`, maximálne po rozsah.

```
<TakeSatisfaction value="18"/>
```

Zníženie satisfakcie o `value`, maximálne po hodnotu `0`.

```
<SetSatisfaction value="180"/>
```

Nastavenie satisfakcie na hodnotu `value`.

```
<ChangeThisOrganism>
      <!-- Sem umiestniť zmeny pre zvolené organizmy -->
</ChangeOrganisms>
```

Predstavuje sadu zmien pre *tento* organizmus. Konštrukty sú analogické tým z efektov pre organizmy.

Jednotlivé konštrukty je možné uvádzať viackrát. Efekty sa vyhodnocujú v takom poradí, v akom sú zadané. Je ich možné vložiť aj výberom Značky/Blok/Efekt/... či Značky/Organizmy/Efekt/...

Existuje niekoľko špecifických situácií, ktoré majú vlastný spôsob vyhodnocovania:

- ```
  <ChangeBlocks from="block1" to="block2" />
  <ChangeBlocks from="block1" to="block3" />
  ```

V prípade viacerých efektov zmeny toho istého druhu blokov platí iba prvý uvedený, zvyšné sú odignorované. Týka sa aj zmeny *tohto* bloku.

- Zmena *tohto* bloku neukončí predčasne vykonávanie ďalších efektov v rámci tej istej udalosti. Avšak ešte neprevedené udalosti (pokiaľ ich daný blok obsahuje) už áno.

- ```
  <ChangeBlocks from="self" to="block2" />
  <ChangeThisBlock to="block3"/>
  ```

`self` je názov bloku, v ktorom je definovaná udalosť s týmito efektmi. Keďže záleží na poradí uvedených efektov, zmena *tohto* bloku je odignorovaná, keďže už sa previedla v rámci skupinovej zmeny v prvom bode. Opačné poradie ale funguje a prevedú sa obidva efekty.

- ```
  <ChangeBlocks from="A" to="B" />
  <ChangeBlocks from="B" to="C" />
  <ChangeBlocks from="C" to="D" />
  <ChangeThisBlock to="B"/>
  <ChangeBlocks from="D" to="self" />
  ```

`self` je názov bloku, v ktorom je definovaná udalosť s týmito efektmi. Toto je príklad cyklu. Napríklad uvažujme, že je na mape iba jedno políčko typu `self`. Vyhodnotí sa táto udalosť a na mape zostane z políčok typu `B` iba jedno. (`A, B, C` sa najprv zmenia na `D` a potom sa jediné `self` zmení na `B`) Pokiaľ predtým na mape bolo aspoň jedno políčko typu `A`, `B`, `C` alebo `D`, po tejto udalosti sa na mape opäť nachádza aspoň jedno políčko typu `self`, ktorého udalosť môže byť znovu vyhodnotená. Po vyhodnotení sa dostáva opäť do rovnakej situácie. Toto spôsobuje nekonečný cyklus a preto je takýto definičný súbor z bezpečnostných dôvodov odmietnutý.

Okrem týchto obmedzení by ďalej definičný súbor nemal obsahovať diakritiku (tá je na príkladoch použitá len kvôli zlepšeniu čitateľnosti), musí obsahovať definíciu aspoň jedného bloku a jedného organizmu, najviac však definície 256 blokov. Jednotlivé položky (bloky, organizmy) musia byť
  ```

pomenované a ich mená sa nesmú opakovať. Všetky organizmy taktiež musia mať kladný rozsah satisfakcie.

Ako konkrétny príklad definičného súboru je uvedený model popísaný v kapitole 1.4:

```
<?xml version="1.0" encoding="UTF-8"?>
<OrganismsInfoFile>

<Block name="PrazdnyBlok">
   <Description>
      Bezne policko bez udalosti schopne uniest nekonecne vela organizmov.
   </Description>
   <Icon>Sand</Icon>
   <Carry>-1</Carry>
</Block>

<Block name="Zem">
   <Description>Prazdne policko, na ktorom moze vyrast potrava.</Description>
   <Icon>custom/mapblocks/01.png</Icon>
   <Carry>-1</Carry>
   <Event priority="0">
      <Condition>
         <Chance rate="10"/>
      </Condition>
      <Effect>
         <ChangeThisBlock to="Potrava"/>
      </Effect>
   </Event>
</Block>

<Block name="Potrava">
   <Description>Predstavuje potravu pre bylinozravcov.</Description>
   <Icon>Grass</Icon>
   <Carry>-1</Carry>
   <Event priority="1">
      <Condition>
         <Organisms min="1">
            <IsHere/>
            <IsOfType value="Bylinozravec" />
         </Organisms>
      </Condition>
      <Effect>
         <ChangeOrganisms>
            <Organisms>
               <IsHere/>
               <IsOfType value="Bylinozravec" />
            </Organisms>
            <Change max="1">
               <AddSatisfaction value="10"/>
            </Change>
         </ChangeOrganisms>
         <ChangeThisBlock to="Zem"/>
      </Effect>
   </Event>
</Block>

<Organism name="Bylinozravec">
   <OrgDescription>
      Organizus, ktory sa zivi potravou na policka.
   </OrgDescription>
   <OrgIcon>custom/organisms/02.png</OrgIcon>
   <SatisfactionRange>100</SatisfactionRange>
</Organism>
```

```
<Organism name="Masozravec">
   <OrgDescription>
      Organizus, ktory sa zivi inymi organizmami.
   </OrgDescription>
   <OrgIcon>custom/organisms/05.png</OrgIcon>
   <SatisfactionRange>100</SatisfactionRange>
   <Event priority="2">
      <Condition>
         <Organisms min="1">
            <IsHere />
            <IsOfType value="Bylinozravec" />
         </Organisms>
      </Condition>
      <Effect>
         <ChangeOrganisms>
            <Organisms>
               <IsHere />
               <IsOfType value="Bylinozravec" />
            </Organisms>
            <Change max="1">
               <TakeSatisfaction value="10" />
            </Change>
         </ChangeOrganisms>
         <ChangeThisOrganism>
            <AddSatisfaction value="10" />
         </ChangeThisOrganism>
      </Effect>
   </Event>
</Organism>

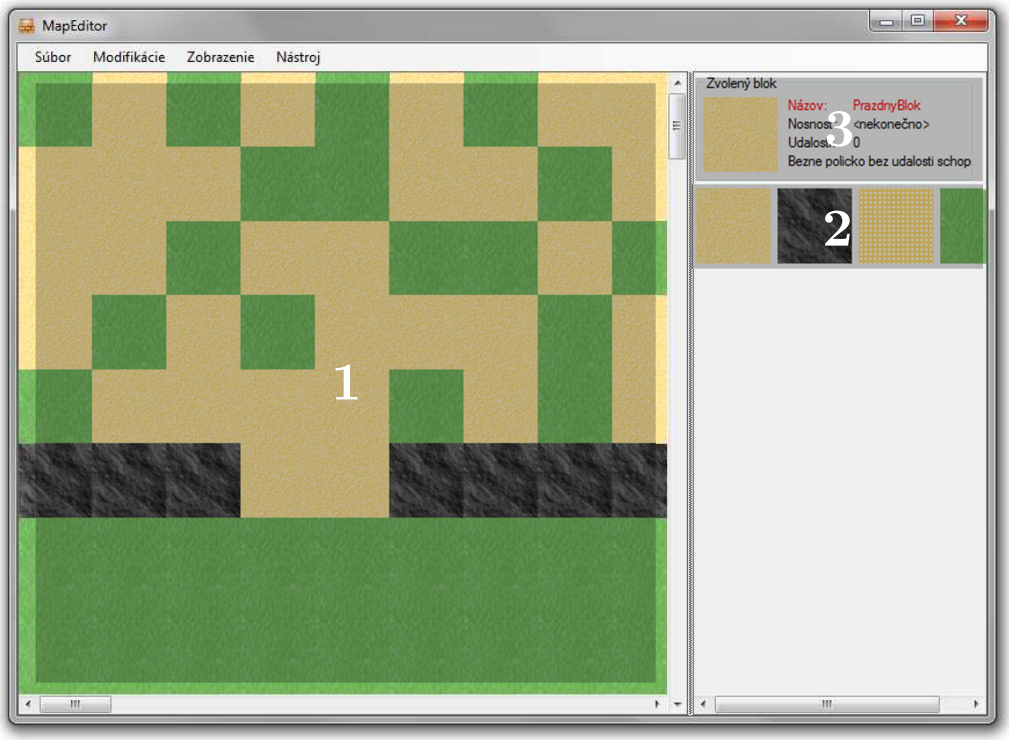
</OrganismsInfoFile>
```

2.3 Editor mapy

Editor mapy sa spúšťa z hlavného okna aplikácie výberom Mapa/Otvoriť editor mapy. Slúži pre vytvorenie konkrétneho rozmiestnenia políčok z nadefinovaných blokov. Pred vytvorením mapy je nutné už mať hotový definičný súbor z predchádzajúcej časti. Po uložení sa s ním mapa striktne spáruje a ďalšie zmeny v definičnom súbore už nie sú povolené.

Okno je rozdelené na 3 logické časti (obrázok 2.4):

- **Plocha.** (1) Určuje výzor mapy.
- **Dostupné bloky.** (2) Kliknutím sa zvolí konkrétny blok, s ktorým je možné ďalej pracovať.
- **Informácie o zvolenom bloku.** (3) Zobrazuje aktuálny výber bloku a jeho hlavné vlastnosti.



Obrázok 2.4: Okno editoru mapy

Výberom Súbor/Nový sa zobrazí dialógové okno pre určenie veľkosti novej mapy a definičného súboru, s ktorým bude mapa spárovaná. Nová mapa nesmie byť príliš malá. Očakáva sa, že bude obsahovať aspoň 9 políčok. Po jej vytvorení je nová mapa inicializovaná prázdnymi (štandardnými) blokmi. Mapa je považovaná za nekompletnú, pokiaľ obsahuje aspoň jeden takýto prázdny blok. Nekompletnú mapu nie je možné uložiť.

Na mape sa zakresľujú vybraté bloky nasledujúcimi nástrojmi:

- **Pero.** (Nástroj/Pero)
- **Rovná čiara.** (Nástroj/Čiara)
- **Obdĺžnik** (Nástroj/Obdĺžnik)
- **Náhodná výplň.** (Nástroj/Náhodná výplň) Slúži pre náhodné rozmiestnenie políčok zvoleného druhu. Udáva sa percentuálne, akú časť plochy má približne pokryť.

Časť Informácie o zvolenom bloku je možné skryť výberom Zobrazenie/Informácie o zvolenom bloku.

Pre väčšie mapy je možné pohľad oddialiť výberom Zobrazenie/Veľkosť mriežky/... Pre extrémne veľké mapy je možné použiť zobrazenie Bitová mapa, kde je každé políčko reprezentované jediným pixlom, ktorého farba je určená ako medián z farieb obrázku bloku. Je dôležité mať však na pamäti, že mapy obsahujúce veľké množstvo políčok s udalosťami sú pri simulácii vyhodnocované značne pomalšie.

Po vytvorení je nutné mapu pred použitím najprv uložiť. (Súbor/Uložiť, Súbor/Uložiť ako...) Existujúcu mapu je naopak možné otvoriť výberom Súbor/Otvoriť a po úprave znovu uložiť.

2.4 Okno simulácie

Zobrazenie okna simulácie je možné vyvolať výberom Simulácia/Otvoriť okno.

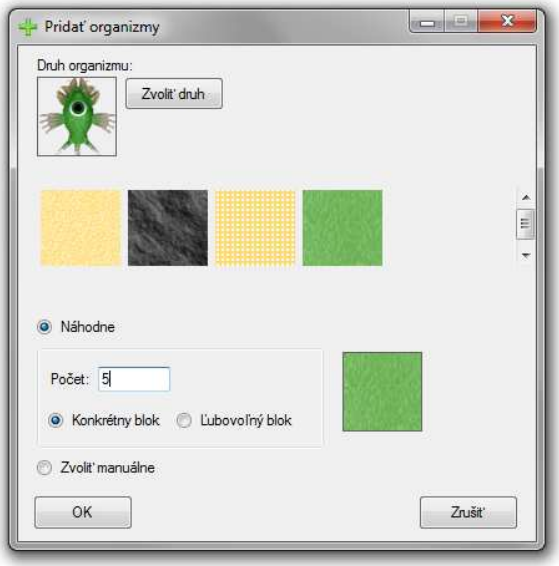
Obrázok 2.5: Okno simulácie

Novú simuláciu je možné spustiť výberom Súbor/Nová simulácia. To otvorí dialógové okno s dotazom na súbor mapy a definičný súbor, s ktorým je mapa spárovaná.

Po úspešnom otvorení mapy je možné pridať organizmy. Výberom Organizmy/Pridať organizmy sa otvorí dialógové okno s detailmi pridania. Kliknutím na tlačidlo Zvoliť druh je možné zvoliť niektorý z druhov organizmov definovaných v definičnom súbore, ktorý bude pridaný. Následne je možné vybrať, či sa majú organizmy rozmiestniť manuálne, alebo automaticky náhodne.

Pri manuálnom spôsobe po potvrdení tlačidlom OK sa dialógové okno zavrie a na mape sa objaví ku každému políčku množstvo voľného miesta. Popisom INF sú označené políčka, ktoré unesú nekonečne veľa organizmov. V tejto chvíli je možné kliknutím na ľubovoľné políčko s dostupným miestom pridať organizmus. Režim pridávania je možné ukončiť výberom Organizmy/Zastaviť.

Pri automatickom režime sa nastavuje, koľko organizmov sa má pridať a na aký druh políčka majú byť umiestnené. Je možné zaškrtnúť Ľubovoľný blok pre výber ľubovoľného políčka s dostupným miestom. Pokiaľ pri pridávaní dôjde políčkam dostupné miesto, aplikácia na to nijak explicitne neupozorňuje a je na užívateľovi, aby si skontroloval pridaný počet organizmov. Automatický režim pridávania sa obzvlášť hodí pri veľkých mapách, kde sa manuálne rozmiestňovanie môže javiť ako nepohodlné.



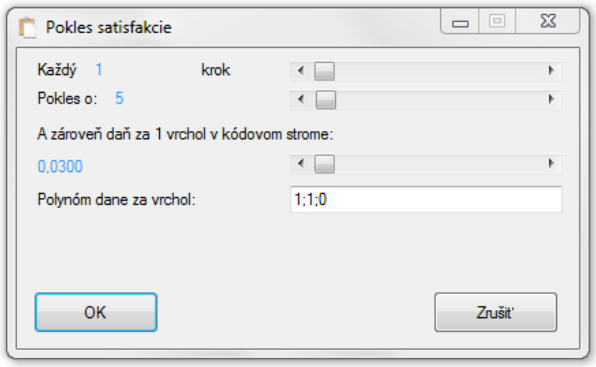
Obrázok 2.6: Dialógové okno pre pridávanie organizmov

V súvislosti s pridávaním organizmov je treba poznamenať, že každý organizmus predstavuje určitú réžiu vo výpočte pri simulácii a to podstatne vyššiu, než sú udalosti jednotlivých políčok. (Organizmy navyše tiež môžu obsahovať udalosti.) Ich množstvo je teda nutné voliť rozumne veľké, odporúča sa zhruba do 50 jedincov.

Po pridaní organizmov by sa mali nastaviť niektoré parametre simulácie. Výberom Organizmy/Preferencie je možné nastaviť maximálnu hĺbku kódového stromu, ktorý môže mutáciami vzniknúť. Príliš malá hĺbka nesie so sebou výrazné obmedzenie pre stratégie, ktoré sa môžu vyvinúť, na druhej strane hlboké stromy výrazne spomalia simuláciu a konvergenciu k „rozumným“ stratégiám. Hĺbku stromu sa odporúča ponechať v rozsahu 5 až 10.

Ďalším významným parametrom je množstvo uchovávaných štatistických údajov. Pre každý organizmus sa sleduje úroveň satisfakcie v každom kroku a čas, v ktorom nastala zmena kódu v dôsledku mutácie. Je možné nastaviť, aby sa štatistické údaje „zhlukovali“. V tom prípade sa bude uchovávať iba priemerná úroveň satisfakcie za užívateľom zvolený väčší počet krokov a čas zmeny kódu sa zapíše až po nadobudnutí tohto počtu zmien. To je možné využiť, ak chceme sledovať vývoj populácie za väčší časový interval.

Dôležitým nastavením je ďalej pokles satisfakcie (Organizmy/Pokles satisfakcie). Zabezpečuje kontinuálne klesanie satisfakcie jednotlivých organizmov. Bez neustáleho klesania satisfakcie organizmy nebudú mať motiváciu niečo robiť a ich kód môže zdegenerovať na jednoduché státie a ničnerobenie na jednom políčku. Pozostáva zo 4 nastavení:



Obrázok 2.7: Nastavenie poklesu satisfakcie

- **O akú hodnotu má satisfakcia poklesnúť.** Hodnota je určená celým číslom.

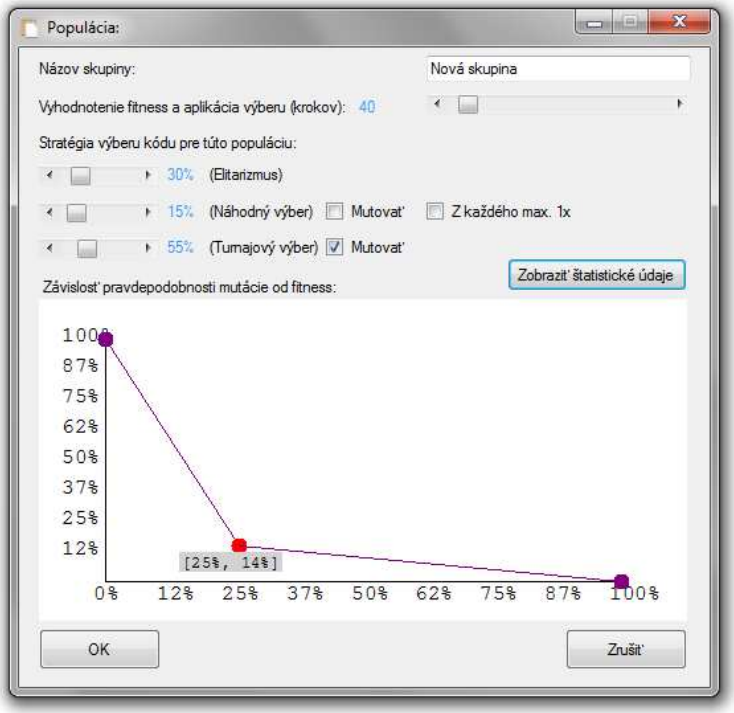
- **Po koľkých krokov má klesať.** Klesá o hodnotu určenú v predchádzajúcom bode.

- **Daň za vrchol.** Táto položka zvýhodňuje organizmy, ktoré majú kód obsahujúci menej (prípadne vôbec) zbytočných vetví. To umožňuje redukovať tzv. bloat. Nastáva raz za počet krokov nastavených v predchádzajúcom bode.

- **Polynóm dane za vrchol.** Určuje sa vo forme celočíselných koeficientov oddelených bodkočiarkou (`;`). Navýši zníženie satisfakcie o polynóm, kde $x = \text{"veľkosť stromu"} \cdot \text{"daň za vrchol"}$ (čiže všetky vrcholy sú pokutované rovnako). Príklad: polynómu $x^2 + 2x + 0$ zodpovedá výraz `1;2;0`. Toto sa hodí v prípade, že chceme výrazne viac pokutovať veľké stromy, pričom menšie stromy majú byť pokutované len málo, prípadne skoro vôbec. Neodporúča sa voliť polynóm väčšieho stupňa, než 2.

Výberom Organizmy/Upraviť je možné získať podrobný prehľad o organizmoch aktuálne na mape. Z tohto dialógového okna je ďalej možné pridať a upraviť populačnú a pamäťovú skupinu.

Populačná skupina predstavuje súbor organizmov, v rámci ktorých platí evolúcia kódu tak, ako je popísaná v kapitole 1.7. Medzi rôznymi populačnými skupinami sa kódy nezdieľajú. To je výhodné, pokiaľ máme na mape viacero druhov organizmov, z ktorých by sa mal každý správať inak. Ako príklad uveďme niekoľkokrát zmieňovaný model bylinožravcov a mäsožravcov, kde je rozumné tieto dve skupiny organizmov oddeliť, inak by sa najúspešnejší kód mäsožravého organizmu mohol preniesť na nejaký bylinožravý organizmus, čo by mohlo viesť k zmäteným výsledkom.



Obrázok 2.9: Úprava populačnej skupiny

Okrem toho je možné pre každú populačnú skupinu zvlášť nastaviť ďalšie parametre evolúcie kódu:

- **Vyhodnotenie fitness a aplikácia výberu...** Určuje počet krokov, za ktorý sa má vyhodnotiť priemerná satisfakcia pre každý organizmus prislúchajúci danej skupine. Po vyhodnotení okamžite nasleduje distribúcia nových kódov. Vyššie číslo typicky znamená lepšie zhodnotenie, ktorý organizmus je skutočne úspešnejší, než ostatné. Na druhej strane je potom väčšinou nutné odsimulovať väčšie množstvo krokov.

- **Stratégia výberu kódu pre danú skupinu.** Umožňuje percentuálne nastaviť spôsob distribúcie nových kódov. V prípade náhodného výberu je možné zaškrtnúť Z každého max. 1x, čo zabezpečí, že žiaden kód sa týmto spôsobom nerozdistribuuje viackrát, než raz. Pokiaľ je zaškrtnuté políčko Mutovať, tak môže prebehnúť mutácia podľa pravdepodobnosti určenej grafom nižšie.

- **Závislosť pravdepodobnosti mutácie od fitness.** Nastavuje, s akou pravdepodobnosťou prebehne mutácia kódu organizmu pri danej fitness, ktorou bol ohodnotený. Umožňuje stabilizovať kód populácie, pokiaľ už nie je veľa priestoru pre zlepšenie. Veľká konštantná pravdepodobnosť mutácie totiž môže spôsobovať rýchle zahadzovanie aj dobrých a osvedčených kódov, ktoré boli vytlačené horšími v dôsledku dočasnej udalosti. Graf sa ovláda myšou. Zvislá os určuje pravdepodobnosť mutácie, vodorovná os určuje fitness vyjadrenú v percentách. (Fitness je inak číslo v intervale $\langle 0,1 \rangle$.)

Kliknutím na vrchol ho je možné presunúť, kliknutím mimo sa pridá nový. Odobrať vrchol je možné pravým tlačidlom myši.

Zmeny je nutné potvrdiť tlačidlom OK, prípadne zamietnuť tlačidlom Zrušiť.

Pamäťová skupina umožňuje prideliť viacerým organizmom spoločnú pamäť. Toto dáva priestor pre vznik stratégií, ktoré by s bežnou pamäťou, vlastnou pre každý organizmus, neboli možné. (Napríklad signalizácia nebezpečenstva, spolupráca organizmov, atď.) Štandardne má však každý organizmus vlastnú pamäť. V rámci pamäte je ešte možné nastaviť, koľko buniek má byť k dispozícii (štandardne 10) a aké majú mať hodnoty.

Priradenie organizmov ku zvoleným skupinám sa prevádza takto:

- Označíme organizmy, ktorým chceme prideliť novú skupinu tým, že klikneme na zaškrtávacie tlačidlo v stĺpci označenom [x].
- Zvolíme skupiny zo zoznamov označených ako Populácia a Pamäť.
- Klikneme na tlačidlo Aplikovať skupiny.

Pokiaľ chceme niektoré organizmy dočasne vylúčiť z mutácií a zabezpečiť, aby sa im nemenil kód, môžeme tak urobiť tým, že ich označíme a zvolíme Zmraziť kód označeným. To však neznamená, že budú vylúčené z populácie. Ich kód bude aj naďalej môcť byť kopírovaný na iné organizmy v dôsledku selekcie.

Konkrétny organizmus je možné upraviť kliknutím na riadok, ktorý ho popisuje a následným zvolením tlačidla Upraviť zvolený. Druhý spôsob je v okne simulácie zvoliť Organizmy/Reagovať na výber. V tomto režime sa po kliknutí na organizmus otvorí dialógové okno pre úpravy. Tento režim je možné vypnúť zvolením Organizmy/Zastaviť. Je dobré mať na pamäti, že pokiaľ stojí viacej organizmov na jednom políčku, je vždy v tomto režime vybratý iba prvý z nich.

V okne úprav konkrétneho organizmu je možné presne nastaviť úroveň satisfakcie, organizmus premiestniť či kopírovať so všetkými stavmi a kódom na iné políčko, zmeniť typ, alebo upraviť jeho kód.

Nakoniec samotnú simuláciu možno spustiť výberom Simulácia/Simulácia. To zobrazí menšie dialógové okno, v ktorom sa zvolí požadovaný počet krokov, ktorý sa má odsimulovať, a následne spustí tlačidlom Spustiť. Simuláciu je možné po spustení kedykoľvek prerušiť tlačidlom Zataviť. Pre krokovanie simulácie by sa mal používať výber Simulácia/Krok vpred (Ctrl+F4).

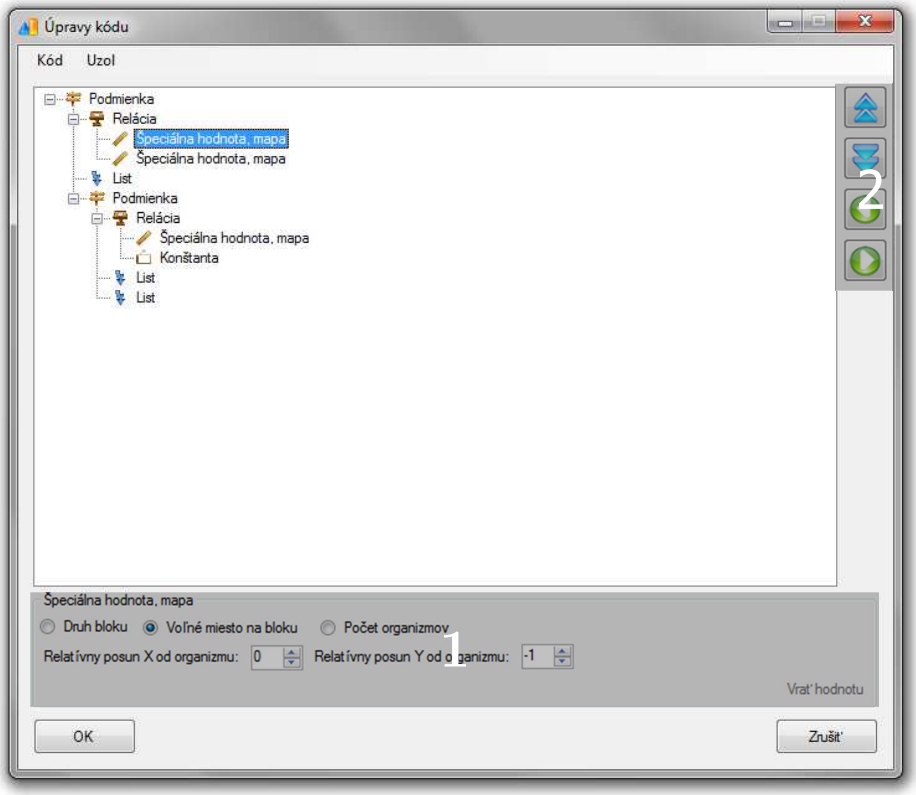
2.5 Úprava kódu organizmu

Okno úprav kódu organizmu umožňuje podrobne upraviť kódový strom, ktorým sa organizmus riadi. Jednotlivé uzly je možné pridať výberom Uzol/Pridať/... či odobrať spolu so synmi výberom Uzol/Odobrať podstrom.

Úprava konkrétneho uzlu spočíva v jeho zvolení kliknutím ľavým tlačidlom myši. V dolnej časti okna (obrázok 2.10, časť 1) sa zobrazia ďalšie možnosti. Špeciálne sa upravuje uzol vracajúci mapovú hodnotu, kde je nutné jednak nastaviť, aký druh hodnoty má uzol vracať (číslo bloku, voľné miesto) a ďalej akého políčka sa má hodnota týkať. To sa určuje zvolením relatívneho posunu súradníc X a Y políčka od organizmu tak, akoby bol organizmus natočený v smere 0° (smerujúci nahor). Príklady:

- **X: 0, Y: -1.** Políčko pred organizmom.
- **X: 0, Y: 1.** Políčko za organizmom.
- **X: -1, Y: 0.** Políčko naľavo od organizmu.
- **X: 1, Y: -1.** Políčko o 1 napravo a o 1 pred organizmom.

Uzol je možné presúvať tlačidlami v pravej časti okna (obrázok 2.10, časť 2), či výberom Uzol/Posunúť nahor, Uzol/Posunúť nadol, Uzol/Vysunúť mimo podstrom, Uzol/Vsunúť do postromu nadol.



Obrázok 2.10: Okno úprav kódu organizmu

Ďalej je možné zobraziť textovú reprezentáciu kódu (Kód/Zobraziť text). V rovnakej podobe sa kód aj ukladá či načíta z externého textového súboru. (Kód/Uložiť ako..., Kód/Otvoriť)

Gramatika jazyka, ktorý dokáže aplikácia rozoznať, je nasledujúca:

- Kód musí byť uvedený medzi kľúčovými slovami `BEGIN` a `END`. Kód tu chápeme ako príkaz.

- Kdekoľvek v kóde je možné umiestniť komentáre, ktoré sú dvoch druhov:

  - Jednoriadkový komentár. Musí začínať znakmi `//`
  - Viacriadkový komentár. Od znakov `/*` až po prvý výskyt znakov `*/`

- Príkaz môže byť týchto druhov:

  - `@`*`List`*`;` *`Príkaz`*
    - predstavuje uzol činnosti (list), ktorý má syna *`Príkaz`*
  - *`List`*`;`
    - predstavuje koncový uzol činnosti (list) bez ďalších potomkov
  - `if (`*`Relačný-výraz`*`)` *`Príkaz`* `Else` *`Príkaz`*
    - predstavuje podmienkový uzol s pokračovacími uzlami *`Príkaz`* a *`Príkaz`*
  - *`Pamäť`* `:=` *`Výraz`*`;` *`Príkaz`*
    - predstavuje priradenie výrazu do pamäťového uzlu, pričom ďalšie vyhodnocovanie pokračuje uzlom *`Príkaz`*

- List môže byť týchto druhov:

  - `Go`
    - Krok vpred
  - `TurnLeft`
    - Otočenie vľavo o 90°
  - `TurnRight`
    - Otočenie vpravo o 90°
  - `Action`
    - Akcia
  - `Wait`
    - Čakať
  - `GoUp`
    - Choď nahor
  - `GoDown`
    - Choď nadol
  - `GoLeft`
    - Choď doľava
  - `GoRight`
    - Choď doprava

- Relačný výraz môže byť týchto druhov:

- *Výraz* `==` *Výraz*
  - predstavuje dotaz na rovnosť dvoch výrazov
- *Výraz* `!=` *Výraz*
  - predstavuje dotaz na nerovnosť dvoch výrazov
- *Výraz* `<=` *Výraz*
  - predstavuje dotaz na to, či je prvý výraz menší alebo rovný druhému výrazu
- *Výraz* `>=` *Výraz*
  - predstavuje dotaz na to, či je prvý výraz väčší alebo rovný druhému výrazu
- *Výraz* `<` *Výraz*
  - predstavuje dotaz na to, či je prvý výraz menší než druhý výraz
- *Výraz* `>` *Výraz*
  - predstavuje dotaz na to, či je prvý výraz väčší než druhý výraz

- Výraz môže byť týchto tvarov:
  - *Výraz* `+` *Výraz*
    - predstavuje podstrom vracajúci výsledok operácie súčtu, kde potomkovia sú *Výraz* a *Výraz*
  - *Výraz* `-` *Výraz*
    - analogicky pre operáciu rozdielu
  - *Výraz* `*` *Výraz*
    - analogicky pre operáciu súčinu
  - *Výraz* `/` *Výraz*
    - analogicky pre operáciu celočíselného podielu
    - po prípadnom delení nulou vracia špeciálne hodnotu `0`
  - *Výraz* `%` *Výraz*
    - analogicky pre operáciu modulo
    - po prípadnom modulení nulou vracia špeciálne hodnotu `0`
  - `(` *Výraz* `)`
  - Celé číslo bez znamienka
    - záporné znamienko je možné určiť iba v prípade, že je číslo v zátvorkách
    - chápe sa ako konštanta
  - *Pamäť*
  - *Hodnota*

Strom sa zostavuje podľa štandardných priorít operátorov, t.j. súčin, podiel a modulo majú prednosť pred súčtom a rozdielom, pričom výraz v zátvorkách má prednosť pred súčinom, podielom a modulom.

- Pamäť má tvar: `Memory[`*index-bunky*`]` kde index bunky je celé číslo v intervale $\langle 0, počet\ buniek - 1 \rangle$.

- Hodnota môže byť týchto druhov:
  - `BlockNumber[`*relatívny-posun-X*`,` *relatívny-posun-Y*`]`
    - predstavuje uzol mapová špeciálna hodnota / druh bloku
    - ide o poradie bloku tak, ako je v definičnom súbore, počítané od `0`; okraj mapy vracia `-1`

- `FreeSpace[`*`relatívny-posun-X`*`, `*`relatívny-posun-Y`*`]`
  - predstavuje uzol mapová špeciálna hodnota / voľné miesto na políčku
  - vráti hodnotu `-1` pre nekonečne veľa voľného miesta
- `OrganismAmount[`*`relatívny-posun-X`*`, `*`relatívny-posun-Y`*`]`
  - predstavuje uzol mapová špeciálna hodnota / počet organizmov
- `Direction`
  - predstavuje uzol vlastná špeciálna hodnota / smer
- `X`
  - predstavuje uzol vlastná špeciálna hodnota / poloha na súradnici X
- `Y`
  - predstavuje uzol vlastná špeciálna hodnota / poloha na súradnici Y
- `Time`
  - predstavuje uzol vlastná špeciálna hodnota / čas

Relatívny posun X a relatívny posun Y sa zadávajú ako celé čísla so znamienkom. Ak znamienko nie je uvedené, chápe sa dané číslo ako nezáporné. Ich význam je rovnaký, ako v prípade grafickej úpravy kódového stromu.

Kód organizmov je teda možné zapísať aj v tomto jazyku s použitím ľubovoľného textového editoru. Výstup je nutné najprv uložiť a až potom otvoriť v tejto aplikácii. Uložený súbor musí mať príponu `*.txt`.

Uvádzame konkrétny príklad kódu zapísaného v tejto gramatike:

```
BEGIN
  @Go;
  @TurnLeft;
  TurnLeft;
END
```

Ide o kód, ktorým by organizmus zotrvával iba na dvoch konkrétnych políčkach umiestnených vedľa seba a neustále by sa medzi nimi premiestňoval.

Iný príklad kódu:

```
BEGIN
 If (Memory[0] > 4)
   Memory[0] := 0;
   Go;
 Else
   If (BlockNumber[0, -1] == -1)
     Memory[0] := Memory[0] + 1;
     TurnLeft;
   Else
     If (BlockNumber[0, -1] == 2)
       Memory[0] := Memory[0] + 1;
       TurnLeft;
     Else
       Memory[0] := 0;
       Go;
END
```

Toto je príklad umelého kódu, ktorým by sa organizmus snažil vyhýbať istému typu bloku, ktorý má číslo 2, a zároveň by sa snažil „nenabúrať“ do okraja mapy. Ak by však bol náhodou obklopený takýmito blokmi, neustále by sa len otáčal a nikdy nevykonal pohyb, čomu sa snažíme v tomto kóde predísť tým, že si ukladáme do pamäte, koľkokrát sa už otočil. Pokiaľ sa už otáčal pridlho, tak vykoná pohyb bez ohľadu na typ políčka pred ním.

3. Implementácia

3.1 Úvod

Program Organisms bol naprogramovaní v jazyku C# s použitím .NET Framework 4. Bolo zvolené vývojové prostredie Microsoft® Visual Studio 2010.

Gramatika jazyka kódu organizmov bola vytvorená v generátoru kompilátorov Coco/R[2]. Jeho výstupom bol opäť kód pre jazyk C#, ktorý bol ďalej využívaný a zakomponovaný do cieľovej aplikácie Organisms.

3.2 Editor blokov a organizmov

Editor blokov a organizmov v rámci užívateľského rozhrania používa komponent `RichTextBox` poskytovaný cez .NET Framework. Je to kvôli lepšej prehľadnosti zadávaného textu, nakoľko tento komponent umožňuje farebne zvýrazňovať a inak vhodne odlišovať text. Tento komponent bol ešte upravený do podoby triedy `SpecialRichTextBox`, ktorý sa snaží minimalizovať efekt „preblikávania“ pri zadávaní textu tým, že sa v správnych okamihoch uzamyká pred vykresľovaním (metódy `LockThis()` a `UnlockThis()`).

Farebné zvýrazňovanie textu je riešené jednoduchým aplikovaním vyhľadávania v texte pomocou regulárnych výrazov. Odlišujú sa iba 4 typy výrazov: XML element, XML atribút, XML komentár a čistý text. Pre rýchlu reakciu bolo zavedené, aby sa vyhľadávalo len v práve upravovanom riadku. Tým pádom výrazy presahujúce jeden riadok (napríklad XML element, ktorý má ukončovací znak na inom riadku, než počiatočný) nie sú správne farebne odlišované. Toto by však nemalo činiť problém, nakoľko všetky legálne[3] konštrukty (s výnimkou komentárov) sa vojdú do jedného riadku.

Zobrazovanie „bublinovej“ nápovede je riešené podobným princípom. Najprv sa určí, či sa kurzor (caret) nachádza mimo, alebo vnútri XML elementu. Potom podľa obsahu najbližšieho ľavého elementu (prípadne slova) sa určí, aká informácia sa má zobraziť. Elementy, ktoré reálne aplikácia rozlišuje, potom museli byť volené s ohľadom na túto implementáciu.

3.3 Editor mapy

Editor mapy využíva služby poskytované triedami `XMLFileProvider` a `MapDrawer`. Používa komponent `PictureBox` pre zobrazenie aktuálneho výzoru mapy. Ten uchováva práve trieda `MapDrawer`, ktorá okrem iného poskytuje aj metódy pre zakresľovanie políčok do mapy.

[2] Dostupné z WWW [7. 5. 2015]: <http://www.ssw.uni-linz.ac.at/Coco/>

[3] Máme na mysli legálne z pohľadu aplikácie Organisms. Chápeme nimi iba elementy, ktoré aplikácia dokáže reálne rozlíšiť a priradiť im význam.

Pri otváraní (nie nutne už existujúcej) mapy sa využije `XMLFileProvider` pre získanie základných informácií o existujúcich blokoch z definičného súboru. Tieto informácie zahŕňajú predovšetkým výzor (ikona), názov a popis bloku. Udalosti sa reálne nenačítavajú, pretože ich editor mapy nedokáže využiť. Výstupom je potom pole štruktúry blokov, `BlockStructure[]`, ktoré uchováva práve tieto získané informácie ku každému bloku. Reprezentácia mapy je potom prevedená jednoduchým odkazovaním sa v dvojrozmernom poli na indexy poľa štruktúry blokov. Toto pole sa označuje v kontexte aplikácie ako `int[,] BlockGrid`.

Ukladanie a načítavanie súboru mapy sa prevádza pomocou triedy `MapFileHandler`, ktorá len vezme pole `BlockGrid` a uloží jednotlivé indexy po stĺpcoch. Vzhľadom na to, že pri zmene definičného súboru by mohlo dôjsť k zmene načítavania poľa štruktúry blokov a tým pádom aj celkovej zmene výzoru mapy, trieda `MapFileHandler` spáruje použitý definičný súbor so súborom mapy tým, že priloží jeho kontrolný súčet ako hlavičku do súboru mapy.

Samotné zobrazovanie mapy umožňuje potom trieda `MapDrawer`. Pri každej zmene veľkosti okna sa musí editor mapy opýtať tejto triedy, či sa je ešte schopný vojsť obrázok mapy pri zvolenej veľkosti na plochu celý. Pokiaľ nie, vytvoria sa príslušné posuvníky (ScrollBars). S ich pomocou trieda `MapDrawer` vždy vráti iba tú časť mapy, ktorá má byť aktuálne viditeľná pri danom posunutí. Tým pádom sa pri zmenách vždy prekresľuje iba tá časť plochy, ktorá je aktívna, čo umožňuje relatívne efektívnu tvorbu rozsiahlych máp.

Pre zakresľovanie políčok do mapy sa najprv získa vektor posunu myši pri stisnutom ľavom tlačidle. Potom sa podľa zvoleného nástroja zavolá príslušná metóda triedy `MapDrawer`, ktorá sa okrem zakreslenia políčok postará aj o zápis informácií do poľa `BlockGrid`. Následne si musí editor mapy sám aktualizovať obrázok v komponente `PictureBox`.

3.4 MapDrawer

Trieda `MapDrawer` uchováva aktuálny výzor mapy. Využíva ju okno simulácie a editor mapy. Tvar mapy je možné získať metódou `GetBitmap()`. Tá však vráti vždy iba jej časť, pokiaľ sa nezmestí celá do veľkosti, ktorá jej bola nanútená metódou `ResizeBitmap(int width, int height)`. Tá sa typicky volá pri každej zmene veľkosti okna (ani malé mapy sa nemusia zmestiť, pokiaľ je okno aplikácie príliš malé). To, či sa následne majú vytvoriť posuvníky, si musia príslušné okná zistiť samy. Posuvníky musia potom každú zmenu hlásiť triede `MapDrawer` volaním metódy `ScrollLargeMap(int xoffset, int yoffset)`, aby zabezpečili, že sa zobrazí vždy požadovaná oblasť mapy.

Ďalej trieda obsahuje metódy `Rescale(int percentage)` pre oddialenie či priblíženie pohľadu a `RescaleToMinimum()` pre oddialenie až na úroveň, kedy je každé políčko zobrazované jediným pixlom. Získanie farby pixlu pre daný blok sa prevádza cez metódu `GetDominantColor(Image sourceimg)`, ktorá získa medián farieb, z ktorých ikona bloku pozostáva. Pre urýchlenie sa pre každý

druh políčka získa táto farba iba raz a následne sa uloží do hashovacej tabuľky, z ktorej sa pri vykresľovaní opätovne získa.

Trieda obsahuje aj metódy pre zakresľovanie políčok do mapy. Metódy ako `SendSingleBlock(int x, int y, int i)` pre zakreslenie jedného políčka, `SendLineOfBlocks(int x1, int y1, int x2, int y2, int i)` využívajúca Bressenhamov algoritmus pre zakreslenie políčok po úsečke, výrazne uľahčujú prácu editoru mapy. Navyše neprekresľujú celú zobrazovanú plochu, ale zachytávajú iba samotnú zmenu, čo prispieva k zvýšeniu plynulosti aplikácie.

3.5 Simulácia

Podobne, ako editor mapy, aj okno simulácie využíva triedu `MapDrawer` pre vykresľovanie plochy a triedu `XMLFileProvider` pre načítanie definičného súboru. V tomto prípade sa však definičný súbor načítava úplne celý zahŕňajúc všetky druhy organizmov (výstup v podobe poľa štruktúry organizmov `OrganismStructure[]`) a udalosti (sú len pridané do spomínaných polí).

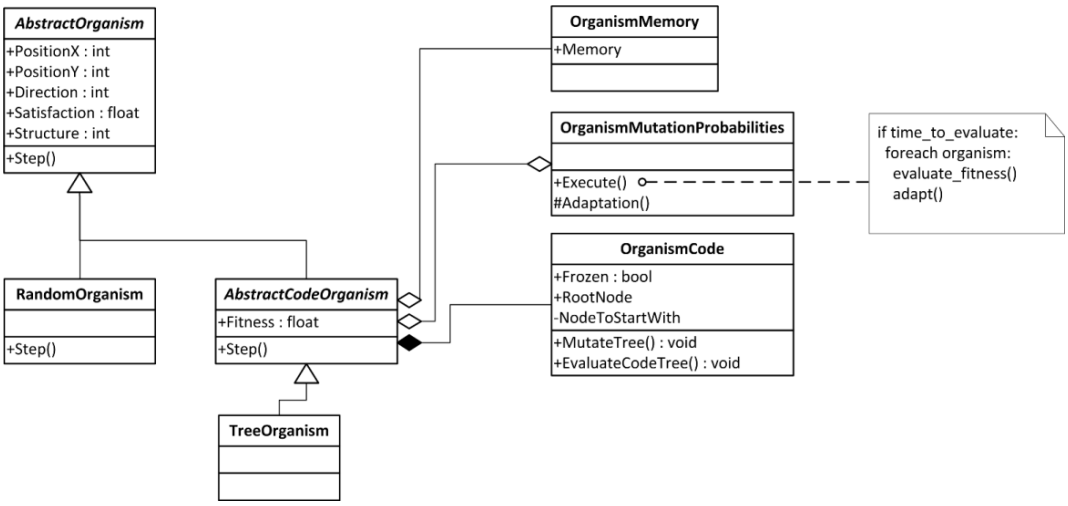
Ďalšia významná trieda, s ktorou okno simulácie spolupracuje, je `SimulationsClass`. Ide o statickú triedu, ktorú musí simulačné okno inicializovať zavolaním vhodných metód. Následným volaním metódy `CallMapSequence()` tejto triedy sa vykoná jeden simulačný krok a zavolaním metódy `RedrawMap()` triedy `MapDrawer` sa plocha prekreslí.

3.6 Organizmy

Organizmy sú asi najkomplexnejšie objekty v celej aplikácii. Sú to potomkovia abstraktnej triedy `AbstractOrganism`, ktorá nesie základné informácie o organizmu, ako sú napríklad jeho poloha, smer, úroveň satisfakcie a príslušnosť k druhu danú indexom do poľa `OrganismStructure[]`. Obsahuje metódu `Step()`, ktorú je povinný každý potomok tejto triedy implementovať. Tá musí vrátiť jenu z dostupných činností, ktorú sa organizmus v danom kroku simulácie rozhodol vykonať.

Trieda `RandomOrganism` je priamy potomok triedy `AbstractOrganism`, ktorá slúžila na testovacie účely. Cez užívateľské rozhranie nie je priamo dostupná. Organizmy tu volili činnosť náhodne.

Abstraktná trieda `AbstractCodeOrganism` poskytuje rozhranie pre organizmy tak, ako s nimi umožňuje pracovať aplikácia, t.j. umožňuje priraďovanie k pamäťovým a populačným skupinám, umožňuje rozhodovanie organizmu na základe vlastného kódu a pridáva atribút fitness pre vyhodnocovanie úspešnosti organizmov. V konečnej aplikácii bol použitý iba jeden potomok tejto triedy, `TreeOrganism`. Táto trieda však bola ponechaná abstraktnou pre potenciálne budúce rozšírenia.

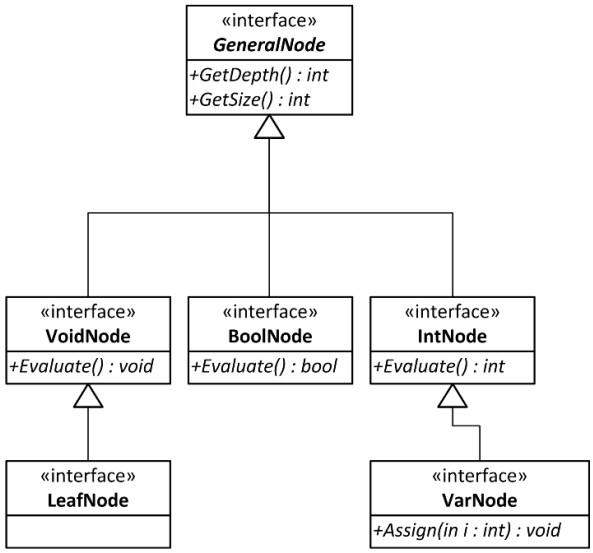


Obrázok 3.1: Zjednodušená schéma tried organizmov

3.7 Kód organizmov

O kód organizmov sa stará trieda `OrganismCode`. Každý uzol kódového stromu je reprezentovaný triedou, ktorá implementuje niektoré z nasledujúcich rozhraní:

- **`VoidNode`.** Predstavuje pokračovací uzol. Tie sa vyhodnocujú bez toho, aby vracali nejakú hodnotu.
- **`LeafNode`.** Je potomok rozhrania `VoidNode`. Predstavuje list, t.j. uzol, ktorým môže končiť vetva kódového stromu[4]. Nedeklaruje žiadne ďalšie metódy. Slúži len na kontrolu, či je strom konzistentný.



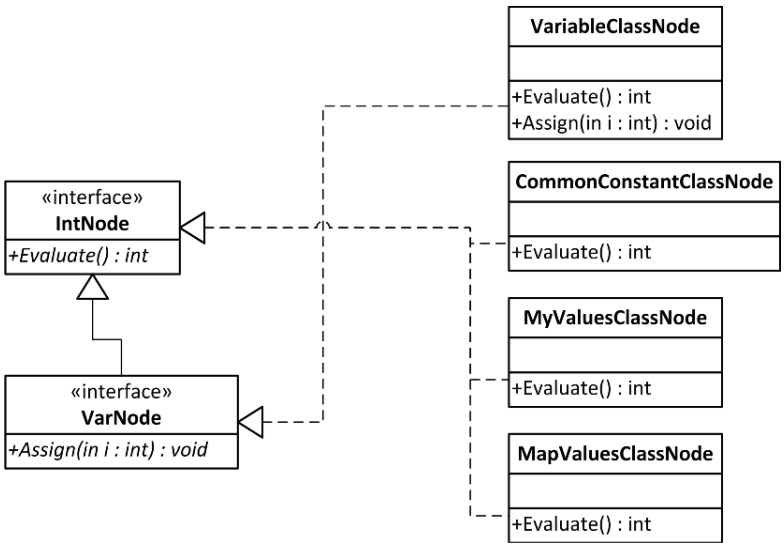
Obrázok 3.2: Schéma rozhraní pre uzly kódového stromu

[4] Triedy implementujúce rozhranie `LeafNode` nie sú jediné, čo môžu vystupovať ako listy (napríklad to spĺňa aj uzol konštanty). Sú však jediné, čo tak môžu vystupovať a zároveň nevracať žiadnu hodnotu.

- **`IntNode`.** Predstavuje uzly vracajúce celočíselnú hodnotu.
- **`VarNode`.** Predstavuje uzly vracajúce celočíselnú hodnotu, do ktorých ide aj zapisovať. V tejto aplikácii ide o premennú - pamäť.
- **`BoolNode`.** Predstavuje uzly, ktoré vracajú logickú hodnotu *true* alebo *false*.

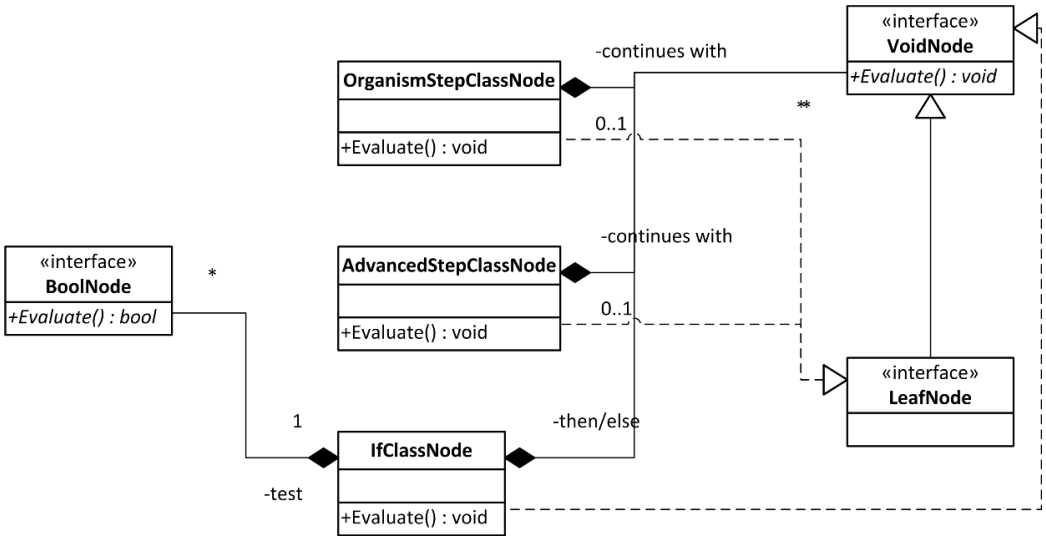
Jednotlivé triedy, ktoré nadobúdajú podobu uzlov v tomto strome sú nasledujúce:

- **`CommonConstantClassNode`**. Ide o konštantu. Implementuje rozhranie `IntNode`.
- **`MyValuesClassNode`**. Ide o vlastnú špeciálnu hodnotu. Implementuje rozhranie `IntNode`.
- **`MapValuesClassNode`**. Ide o mapovú špeciálnu hodnotu. Implementuje rozhranie `IntNode`.
- **`VariableClassNode`**. Ide o pamäť. Implementuje rozhranie `VarNode`.



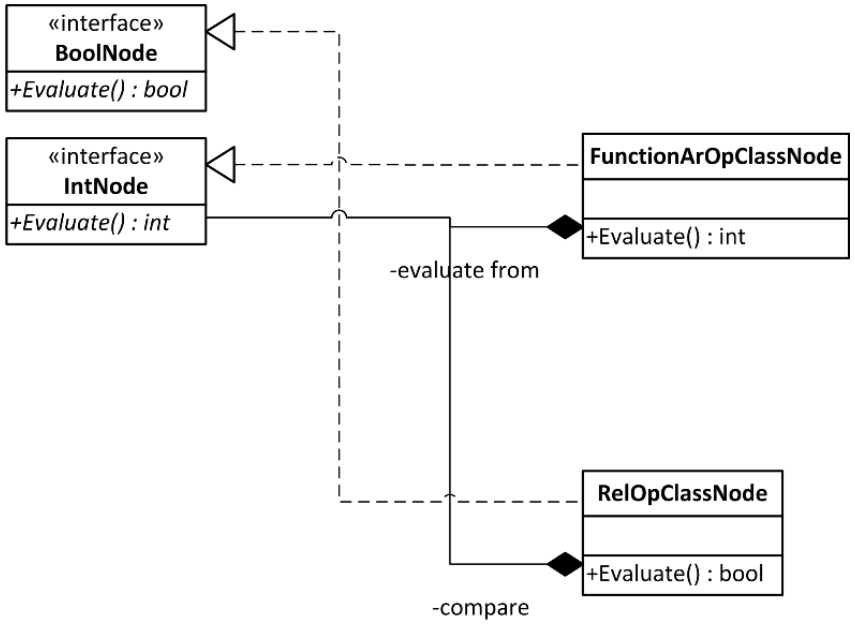
Obrázok 3.3: Schéma prvých štyroch uzlov kódového stromu

- **`OrganismStepClassNode`**. Ide o list. Implementuje rozhranie `LeafNode`.
- **`AdvancedStepClassNode`**. Ide o pokročilý list. Implementuje rozhranie `LeafNode`.
- **`IfClassNode`.** Ide o podmienku. Implementuje rozhranie `VoidNode`.



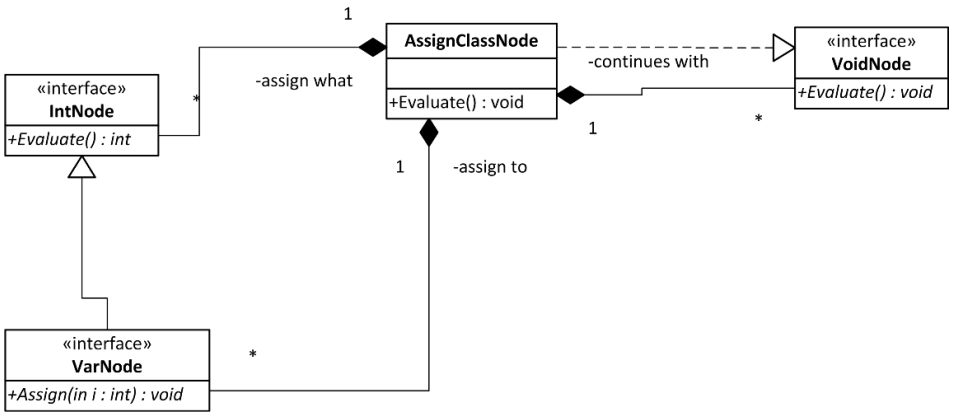
Obrázok 3.4: Schéma ďalších troch uzlov kódového stromu

- **`FunctionArOpClassNode`**. Ide o výsledok operácie. Implementuje rozhranie `IntNode.`
- **`RelOpClassNode`**. Ide o reláciu. Implementuje rozhranie `BoolNode`.



Obrázok 3.5: Schéma predposledných dvoch uzlov

- **`AssignClassNode`**. Ide o priraďovací uzol. Implementuje rozhranie `VoidNode`.



Obrázok 3.6: Schéma posledného uzlu kódového stromu

Okrem toho trieda `OrganismCode` obsahuje množstvo metód pre uskutočnenie mutácií. Zavolaním metódy `MutateTree()` sa začne jej uskutočňovanie. Mutácie sú plne automatické a mimo triedy `OrganismCode` je ich priebeh skrytý.

Mutácie prebiehajú v nasledujúcich krokoch:

- Zvolí sa spôsob mutácie. (Zmena uzlu, vytvorenie náhodného podstromu, ...)
- Metódou `ReturnRandomNode(int minFromBottom, int maxFromBottom)` sa získa náhodný uzol. Musí sa rešpektovať voľba v prvom bode, t.j. napríklad v prípade odstraňovania podstromu nesmie byť vybratý vrchol obsahujúci podstrom väčšej hĺbky, než nastavil užívateľ.
- Prevedie sa mutácia.

Významným atribútom je ešte logická premenná `Frozen`, ktorá určuje, či je kód „zamrznutý“. Vzhľadom na to, že niektoré triedy menia kód organizmu zvonku (teda nie cez mutácie), musia sa najprv ubezpečiť, či to kód organizmu povoľuje práve otestovaním tejto premennej.

3.8 Populačná skupina

Skupiny (týka sa aj pamäťových) sú uložené v statickej triede `OrganismGroupsSubscription` v podobe poľa dvojíc skupiny a jej názvu. O populačnú skupinu sa stará trieda `OrganismMutationProbabilities` (na obrázku 3.1 zjednodušene označená ako PopulationGroup). Obsahuje atribúty, ktoré sa dajú nastavovať z okna Populácia pri úprave danej skupiny. Najdôležitejšou metódou je `Execute()`, ktorá sa stará o evolúciu kódu v danej populácii. Tá volá metódu `Adaptation()`, čo predstavuje konkrétny algoritmus lokálneho prehľadávania, ktorý zabezpečuje adaptáciu správania organizmov. (V tejto práci bol konkrétne použitý evolučný algoritmus.) Metóda `Execute()` sa musí volať v každom kroku simulácie. V istom bode (podľa nastavenia užívateľa, za aký počet krokov sa má vyhodnotiť fitness) sa `Adaptation()` postará o výber kódov, ktoré sa budú distribuovať a zabezpečí príslušné

mutácie. Vzhľadom na to, že kopírovanie kódu organizmov je pomerne náročná operácia, snaží sa šetriť a kopírovať skutočne len tie kódy, ktoré vzniknú navyše. Postup:

- Podľa fitness sa zostaví poradie organizmov. (Napríklad predpokladajme organizmy 1, 2, 3, 4, 5.)
- Podľa percentuálne nastaveného spôsobu výberu nových kódov určí, ktoré sa zachovajú a na ktorých potom prebehne mutácia. (Napríklad v dôsledku elitarizmu sa zachovajú prvé dva a náhodne sa vyberú ďalej kódy organizmov takto: 1, 2, 1, 3, 3.)
- Určí sa, ktoré kódy je reálne nutné prekopírovať. (Napríklad 1, 2, 3 stačí iba zachovať, následne však ďalšie 1, 3 už je nutné prekopírovať z existujúcich a nahradiť nimi kódy organizmov 4 a 5.)
- Prekopíruje kódy a následne prebehne mutácia na niektorých vybraných.

3.9 Udalosti

Jedna udalosť je reprezentovaná triedou `OrganismEvent`, pokiaľ ide o udalosť viazanú na organizmus, prípadne triedou `BlockEvent`, pokiaľ ide o udalosť políčka. V oboch prípadoch obsahujú pole podmienok (`OrganismCondition[]`, `BlockCondition[]`) a pole efektov (`OrganismEffect[]`, `BlockEffect[]`), ktoré sa vyhodnocujú nasledujúcim spôsobom:

- Pre každú podmienku v poli sa zistí, či platí. Ak nie, okamžite sa preruší vykonávanie danej udalosti.
- Vyhodnotia sa postupne všetky efekty.

To samozrejme implikuje, že udalosť bez podmienok vždy vyhodnotí všetky efekty, čo je aj správanie, ktoré sme chceli dosiahnuť.

Každá podmienka a každý efekt je reprezentovaný nejakou triedou, ktorá sa stará o navrátenie výsledku či vykonanie efektu. Podmienky musia implementovať rozhranie `OrganismCondition` (`BlockCondition` v prípade bloku), efekty zase `OrganismEffect` (`BlockEffect` v prípade bloku).

Všetky udalosti pre daný blok sú potom zaznamenané v podobe poľa `BlockEvent[]` v štruktúre daného bloku `BlockStructure[_index]`. (V prípade organizmov je to pole `OrganismEvent[]` v štruktúre daného organizmu `OrganismStructure[_index]`.)

O vyhodnocovanie udalostí sa stará trieda `SimulationsClass`. Tie sa musia najprv vhodne usporiadať, nakoľko má každá udalosť inú prioritu vo vykonávaní. Postup je nasledujúci:

- Zistí sa, aké sú dostupné priority udalostí a usporiadajú sa.

- Pre danú prioritu sa určí, ktoré udalosti jej zodpovedajú. To je dané jednoduchým určením zoznamu indexov do polí `BlockStructure[]` a `OrganismStructure[]` a v rámci nich indexov do polí `BlockEvent[]` prípadne `OrganismEvent[]`.

- Pre danú udalosť sa určí, ktoré jednotky ju obsahujú. Ide o konkrétne organizmy, ktoré sú daného druhu a teda obsahujú danú udalosť. V prípade blokov ide o pozície políčok na mape, ktoré sú daného druhu a teda obsahujú danú udalosť.

Po tejto inicializácii sa v každom volaní metódy `CallMapSequence()` (volanej v každom kroku simulácie) zavolajú všetky udalosti týmto spôsobom:

- Vezme sa najnižšia priorita, pre ktorú ešte neboli vyhodnotené všetky udalosti jej prislúchajúce.

- Pre danú prioritu sa vezme niektorý druh udalosti, ktorý ešte nebol obslúžený. Tento krok sa opakuje, kým nebudú obslúžené všetky udalosti danej priority.

- Pre daný druh udalosti sa vezme niektorá jednotka, ktorá ju obsahuje a obslúži sa. Pri jej volaní metódou `CallEvent(CurrentBlockInfo currentBlock)` (prípadne `CallEvent(AbstractOrganism currentOrganism)` v prípade organizmov) sa musí predať informácia cez parameter, ktorá jednotka ju volá. Tento krok sa opakuje, kým neobslúžia danú udalosť všetky jednotky, ktoré ju obsahujú.

Pripomeňme, že niektoré jednotky sa môžu v dôsledku predošlých udalostí zmeniť, napríklad po vykonaní efektu zmena tohto bloku. Týmto sa samozrejme môžu zneplatniť zoznamy jednotiek, ktoré danú udalosť obsahujú. To však nie je problém a inicializáciu udalostí trieda `SimulationsClass` nemusí vykonávať znova, nakoľko efekty spolupracujú s dátovými štruktúrami tejto triedy a dokážu efektívne upraviť príslušné zoznamy. (Napríklad efekt zmena tohto bloku sa prevedie so zložitosťou *O(1).*)

3.10 SimulationsClass

O dianie na simulačnej ploche sa stará statická trieda `SimulationsClass`. Najdôležitejšou metódou je `CallMapSequence()`, ktorá sa postará o vykonanie jedného simulačného kroku. Pripomeňme, že o prekresľovanie sa stará iná trieda, `MapDrawer`. Simulačný krok spočíva v týchto bodoch:

- Od každého organizmu si vyžiada rozhodnutie pre vykonanie nejakej činnosti zavolaním metódy `Step()`. Túto činnosť nechá organizmus vykonať, prípadne zamietne, ak to mapa nepovoľuje.

- Zavolá metódu `Execute()` pre každú populačnú skupinu. Týmto môže u niektorých organizmov dôjsť ku zmene kódu.

- Vyhodnotia sa všetky udalosti.

Okrem toho obsahuje metódy pre pridávanie, posun a odstraňovanie organizmov, ktoré volá okno simulácie pri výzve užívateľa.

4. Experimenty

4.1 Úvod

S programom Organisms je možné prevádzať veľké množstvo experimentov na sledovanie vývoju populácie. Definičným súborom je možné namodelovať veľké množstvo situácií, či už umelých alebo z reálneho sveta. Aplikácia nám môže dať odpovede na otázky, čo by sa za daných okolností stalo.

V tejto časti boli použité nasledujúce 3 modely:

- **Potrava a jed.** Obsahuje len jeden druh organizmu, ktorý sa môže živiť len potravovým políčkom, ktoré pridáva 10 satisfakcie. Na mape sa však vyskytujú aj jedovaté políčka, ktoré organizmu uberú 25 satisfakcie. Prijímanie potravy či jedu je simulované cez jednoduchým prechodom organizmu cez políčko, ktoré ju obsahuje. Po zjedení ľubovoľného z týchto dvoch sa na danom mieste objaví zem, ktorá sa v každom kroku simulácie môže a nemusí náhodne zmeniť naspäť buď na potravu, alebo na jedovaté políčko. Aby sa predišlo situácií, že na mape zostanú len jedovaté políčka, každé z nich má malú šancu, že sa samovoľne v danom kroku zmení na zem a to aj bez prítomnosti organizmu.

- **Potrava a statický jed.** Je model analogický predchádzajúcemu bodu s tým rozdielom, že jedovaté políčka uberajú iba 10 satisfakcie a nemôžu sa zmeniť na zem (sú nevyčerpateľné). Políčko zeme sa však môže zmeniť naspäť len na potravu.

- **Potrava a kvety.** Obsahuje 2 druhy políčok, ktorými sa môžu organizmy živiť: bežná potrava a tzv. kvety. Tie môžu byť využité len ak na nich organizmus použije činnosť Akcia, na rozdiel od bežnej potravy, cez ktorú stačí organizmu iba prejsť. Po vyčerpaní je zanechané políčko zeme, ktoré sa môže náhodne (šanca 5% za krok) zmeniť buď naspäť na potravu, alebo na kvet s nižšou pravdepodobnosťou (0,5%). Na potravu sa však môže premeniť iba vtedy, pokiaľ na ňom nestojí žiaden organizmus. Potrava pridáva 5 satisfakcie, kvet 15.

Boli prevedené tieto experimenty:

- Vplyv počtu krokov pre vyhodnotenie fitness.

- Vplyv maximálnej hĺbky kódového stromu.

- Vplyv nastavenia závislosti pravdepodobnosti mutácie od fitness.

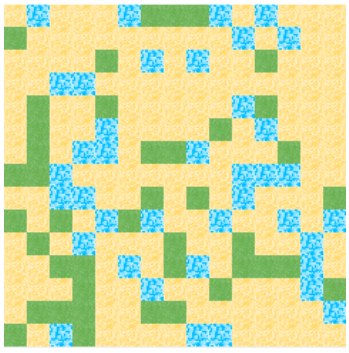
Vo všetkých troch experimentoch sa sleduje vplyv na populáciu a sú rozobraté aj niektoré riadiace algoritmy, ku ktorým niektoré jedince populácie dospeli.

Ako počiatočné nastavenie simulácie sa vždy chápe štandardné nastavenie, ktoré je dané pri spustení aplikácie pred vykonaním akýchkoľvek zmien. V každom experimente sú špecifikované iba dodatočné nastavenia, odlišné od tých štandardných. Štandardné nastavenia zahŕňajú maximálne hĺbku kódového stromu 5 s maximálnou povolenou okamžitou zmenou o hĺbku 3, stratégiu výberu v pomere 6:3:11 (elitarizmus, náhodný výber bez mutácie, turnajový výber s mutáciou), neustály pokles satisfakcie s každým krokom simulácie o 1 a klesajúcou pravdepodobnosťou mutácie rovnou 50% pri nulovej fitness a naopak 0% pri dosiahnutí fitness aspoň 0,5. To je z toho dôvodu, že dosahovanie fitness neustále rovnej 1 je v mnohých mapách nereálne a toto nastavenie predstavuje dobrý kompromis. Viac sa vplyvu nastavenia závislosti pravdepodobnosti mutácie od fitness venuje tretí experiment.

4.2 Experiment 1

Prvý experiment bol zameraný na vplyv počtu krokov simulácie pre vyhodnotenie fitness. Fitness, ako už bolo vysvetlené, je priemer satisfakcií za daný počet krokov. V podstate zhodnocuje kód organizmov. Čím väčšia fitness, tým sa organizmus považuje za úspešnejší, pretože si dokázal dlhšie udržať vysokú úroveň satisfakcie. Je možné predpokladať, že fitness braná za väčší počet krokov bude znamenať lepšie zhodnotenie, ktorý kód je skutočne úspešnejší a teda predpokladáme, že kód organizmov bude smerovať k rozumnejšej stratégií, ako keď sa bude počítať fitness za menší počet krokov.

V prvom experimente sa pracovalo s mapou M1.omf uloženou v adresári Maps/Food&Poison/. Ako definičný súbor bol použitý DF.xml uložený v rovnakom priečinku.



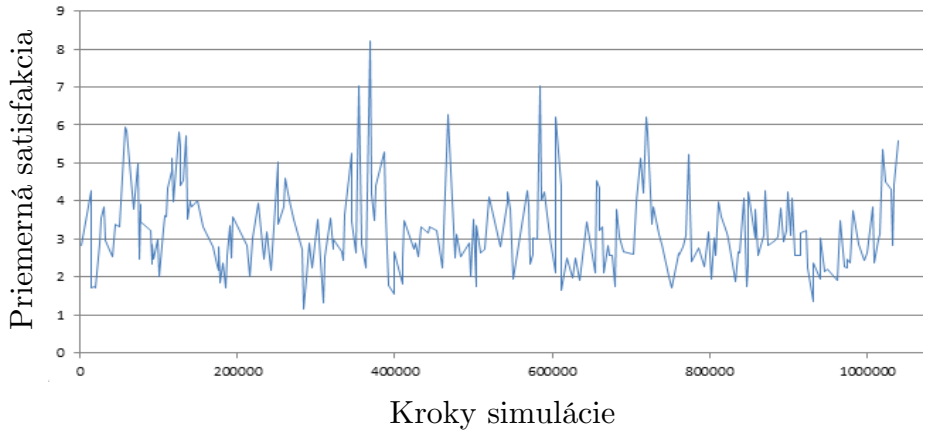
Obrázok 4.1: Mapa Maps/Food&Poison/M1.omf použitá v prvom experimente

Ide o model situácie potrava a jed. Modrými políčkami je znázornený jed, zelené políčka predstavujú zase potravu.

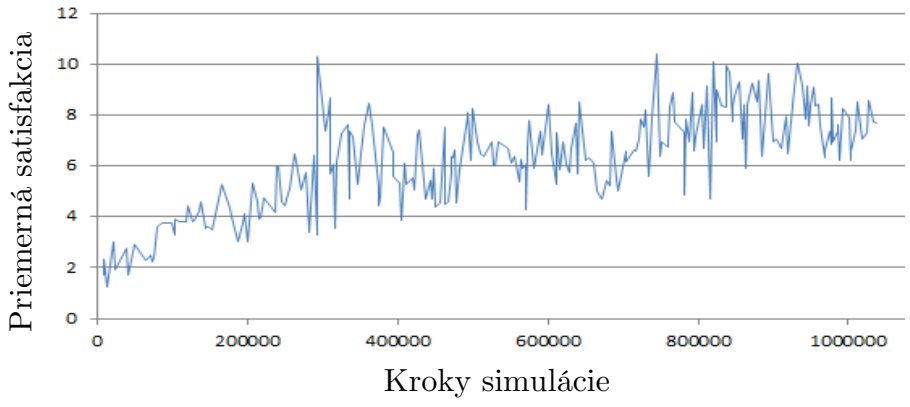
Pred spustením simulácie sme vytvorili 10 jedincov, náhodne rozmiestnených po mape. Sledovali sme rozdiel pri vyhodnocovaní fitness po 30 krokoch a vyhodnocovanie fitness po 200 krokoch. V oboch prípadoch sme odsimulovali 1 041 730 krokov (najbližšie nastaviteľné k 1 000 000 krokom) a pre každý prípad sme spustili simuláciu päťkrát.

Namerané údaje potom boli spriemerované a vyhodnotené za použitia utility naprogramovanej pre tento účel.

Z grafov (obrázky 4.2 a 4.3) vidieť, že priemerná satisfakcia populácie pri použití vyššieho počtu krokov pre vyhodnotenie fitness po určitý bod rastie a následne sa trend spomalí. Naopak, pri vyhodnocovaní fitness iba po 30 krokoch priemerná satisfakcia populácie osciluje a neustále sa vracia aj k nižším hodnotám. Priemer nameraných hodnôt bol v prvom prípade 3,23 (zaokrúhlené na stotiny) pričom medián bol 3,02. Smerodajná odchýlka bola vyhodnotená ako 1,24.



Obrázok 4.2: Priemerná satisfakcia populácie za čas pri prvých piatich meraniach, kde sa vyhodnocovala fitness po 30 krokoch



Obrázok 4.3: Priemerná satisfakcia populácie za čas pri ďalších piatich meraniach, kde sa vyhodnocovala fitness po 200 krokoch

V druhom prípade možno pozorovať rast priemernej satisfakcie populácie, ktorý sa približne od kroku 350 000 spomalí. Celkový priemer hodnôt je 6,23 s mediánom rovným 6,39 a celkovým rozptylom 3,54. Ak sledujeme hodnoty iba od kroku 350 000, dostávame priemer rovný 6,93 s mediánom 6,94 a rozptylom 1,9 čo potvrdzuje odhad spomalenia rastu.

To, že je výhodnejšie nechať vyhodnocovať fitness po väčšom počte krokov naznačujú aj riadiace algoritmy jednotlivých organizmov, ktoré sa vyvinuli. Tie možno posudzovať z mnohých hľadísk. Predovšetkým treba brať do úvahy, kde sa v danom kroku samotné organizmy nachádzajú, nakoľko napríklad stratégia „pohybuj sa medzi dvoma poličkami nahor a nadol“ môže byť výhodná, ak sa organizmus nachádza na potravových políčkach, ale beznádejná pre organizmus, ktorý sa nachádza a je obklopený prázdnymi políčkami.

Jednotlivé algoritmy tu budeme uvádzať vo forme textu, v ktorej je kód možné uložiť či načítať. Stromová štruktúra je síce presnejšia, ale pomerne ťažko čitateľná. V nasledujúcej tabuľke sú algoritmy organizmov s najvyššou fitness z danej populácie po odsimulovaní vyššie spomínaných 1 041 730 krokov z prvej pätice pozorovaní. Fitness je uvedená tiež vo veľkosti z tohto bodu simulácie pre dané organizmy.

Pozorovanie 1	**Pozorovanie 2**	**Pozorovanie 3**

BEGIN
  @TurnLeft;
  @Go;
  @GoUp;
  Go;
END
	BEGIN
	  If (BlockNumber[1, -1] != (-1))
	  Go;
	  Else
	  If (Direction != Direction)
	  Go;
	  Else
	  TurnLeft;
	END
		BEGIN
		  @Go;
		  @Go;
		  @GoUp;
		  TurnRight;
		END

Fitness: 0,216	Fitness: 0,158	Fitness: 0,362
Pozorovanie 4	**Pozorovanie 5**	

BEGIN
  GoRight;
END
	BEGIN
	  Memory[0] := BlockNumber[0, 0];
	  @GoLeft;
	  GoUp;
	END	

Fitness: 0,102	Fitness: 0,147	

Tabuľka 4.1

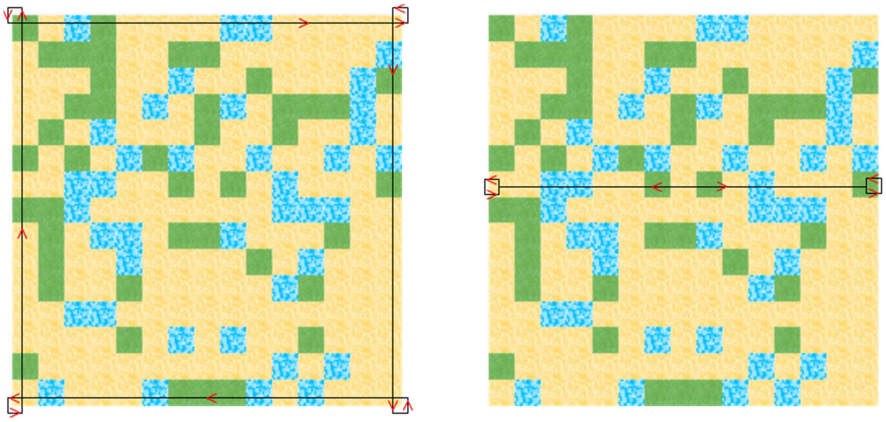
Rozoberme jednotlivé algoritmy:

- **Pozorovanie 1.** Z daného kódu je vidieť, že sa po niekoľkých krokoch organizmus ním riadiaci ocitne v ľavom hornom rohu. To je v dôsledku listu `GoUp` (choď nahor), po ktorom nasleduje v ďalších iteráciach list `TurnLeft` (otočenie vľavo zo smeru nahor - organizmus bude smerovať doľava) a `Go` (vykoná pohyb vľavo). Môže byť prekvapujúce, že tento algoritmus prekonal z hľadiska veľkosti fitness aj nasledujúci kód (fitness rovná 0,12), ktorý sa na konci simulácie vyskytol u iného organizmu:

```
BEGIN
  @TurnLeft;
  @Go;
  @Wait;
  Go;
END
```

Tento kód má oproti minulému aspoň tú výhodu, že organizmy sa ním riadiace sa neustále pohybujú a majú možnosť cestou odchytiť nejakú potravu, pričom ak organizmus zastane iba v ľavom hornom rohu, už sa viac nenasýti.

- **Pozorovanie 2.** V danom kóde sa vyskytla nadbytočná podmienka, ktorá nemôže byť nikdy splnená (`Direction != Direction`). Inak sa podľa tohto kódu organizmus otočí vľavo vždy, pokiaľ políčko napravo pred ním je za okrajom mapy. Vďaka tomu sa organizmus pohybuje po tom istom rovnom páse políčok. Ak dôjde na okraj, jednoducho sa vyberie opačným smerom. V prípade, že sa už nachádzal popri okraji, tak sa bude po dosiahnutí rohu pomerne neprakticky otáčať, ale naďalej bude vykonávať pohyb po obvode mapy.



Obrázok 4.4: Znázornenie pohybov organizmov najúspešnejším algoritmom pri pozorovaní 2 z prvej pätice pozorovaní

- **Pozorovanie 3.** Je kód podobný z pozorovania 1; organizmy sa ním riadiace postupne dôjdu na pravý horný roh. V danom bode sa tiež nachádza iba prázdne políčko. Dokonca podobne, ako v prvom pozorovaní, aj tu sa vyvinul kód u iného organizmu, ktorý je z dlhotrvajúceho hľadiska výhodnejší, než ktorý dosiahol organizmus s najvyššou fitness:

```
BEGIN
  @Go;
  @Go;
  @Go;
  TurnRight;
END
```

- **Pozorovanie 4.** Je jednoduchý pohyb doprava až po dosiahnutie okraju, pričom ďalej sa už organizmus nehýbe.

- **Pozorovanie 5.** Organizmus priradzuje do pamäte číslo políčka na ktorom stojí, ale ďalej túto premennú netestuje. Pohybuje sa smerom nahor a doľava až kým nedosiahne ľavý horný roh mapy.

V ďalšej tabuľke uvádzame algoritmy organizmov z druhej série pozorovaní pri vyhodnocovaní fitness po 200 krokoch:

Pozorovanie 1	**Pozorovanie 2**	**Pozorovanie 3**

BEGIN
  @Go;
  If (FreeSpace[0, -1]
== FreeSpace[-1, 0])
  TurnRight;
  Else
  @Go;
  Go;
END
	BEGIN
	  @Go;
	  @Go;
	  @TurnLeft;
	  Go;
	END
		BEGIN
		  If (FreeSpace[0, -1]
		!= (-1))
		  @TurnRight;
		  @Go;
		  TurnRight;
		  Else
		  Go;
		END

Fitness: 0,365	Fitness: 0,184	Fitness: 0,136
Pozorovanie 4	**Pozorovanie 5**	

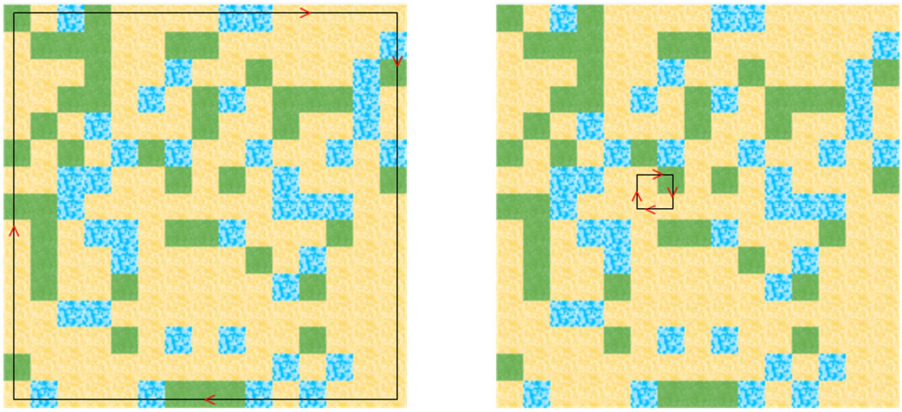
BEGIN
  @Go;
  If (FreeSpace[0, -1]
!= FreeSpace[-1, 0])
  Action;
  Else
  @TurnRight;
  Wait;
END
	BEGIN
	  @TurnLeft;
	  @Go;
	  @Go;
	  Go;
	END	

Fitness: 0,227	Fitness: 0,131	

Tabuľka 4.2

Aj tu rozoberieme jednotlivé algoritmy:

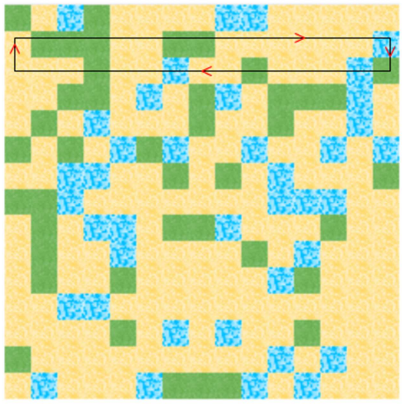
- **Pozorovanie 1.** Predstavuje pohyb organizmu, ktorý sa otočí doprava len v prípade, že voľné miesto na políčku naľavo je rovné voľnému miestu políčku pred ním. Všetky políčka na mape majú nekonečne veľa voľného miesta, čiže táto podmienka nie je splnená iba v prípade, že sa organizmus nachádza pred okrajom mapy, kde políčka pred organizmom majú imaginárne pridelené nulové voľné miesto. Preto tento kód zabezpečuje, aby sa organizmy nachádzajúce ďalej od okrajov pohybovali po ploche 2x2 (neustále vykonajú pohyb vpred a otočenie vpravo). Naopak organizmy pri okraji sa budú pohybovať po obvode celej mapy, pretože pri okraji je neustále porušená daná podmienka. Až dôjde organizmus po ďalší roh, zrazu podmienka splnená je (aj naľavo aj pred organizmom je okraj mapy) a preto sa otočí doprava.



Obrázok 4.5: Znázornenie pohybov organizmov najúspešnejším algoritmom pri pozorovaní 1 z druhej série pozorovaní

- **Pozorovanie 2.** Organizmus sa zrejme bude pohybovať po obvode plochy 4x4.

- **Pozorovanie 3.** Zabezpečuje rovný pohyb organizmu až po okraj mapy, kde sa otočí a prejde do druhého rovného pásu.



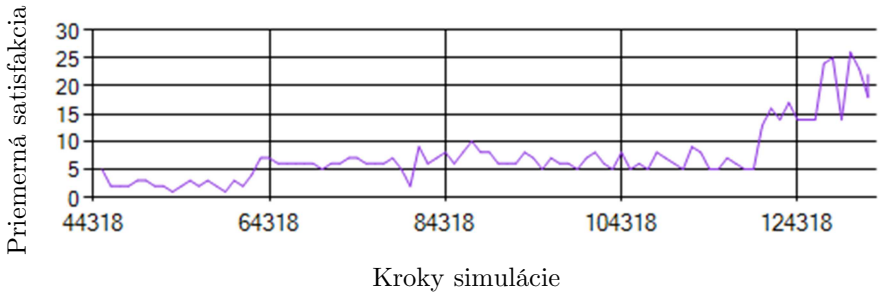
Obrázok 4.6: Znázornenie pohybu organizmu najúspešnejším algoritmom pri pozorovaní 3 z druhej série pozorovaní

- **Pozorovanie 4.** Je analogický algoritmu z pozorovania 1. Organizmus sa však pohybuje „pomalšie“; v niektorých krokoch iba čaká a nič nerobí. Princíp pohybu je však identický.

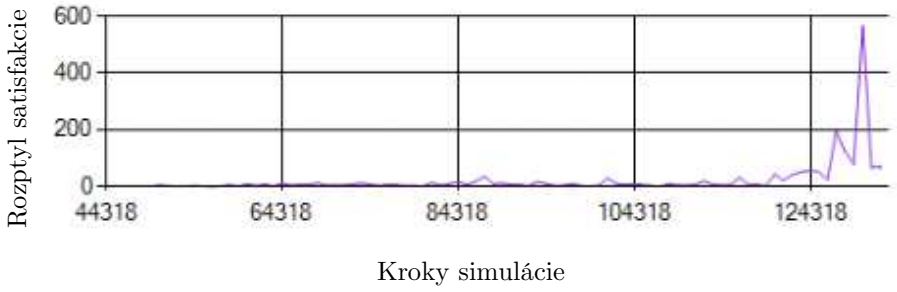
- **Pozorovanie 5.** Princíp pohybu je identický tomu z pozorovania 2.

Zo všetkých rozobratých algoritmov sa zdá, že sa organizmy nepokúšajú vyhýbať sa jedovatým políčkam. Nejde však o to, že by sa kód, ktorý by toto zabezpečoval, nemohol vyvinúť. S popisovanou mapou sa urobilo viacero pokusov s rozličnými nastaveniami, kde sa k takýmto kódom v populácii dospelo.

Bol napríklad vykonaný pokus s rovnakými nastaveniami, ako v prvej sérii piatich pozorovaní (t.j. vyhodnocovanie fitness po 30 krokoch) s tým, že sa simuloval menší počet krokov, konkrétne 132 751. Získali sme nasledujúce štatistické údaje (grafy boli prevzaté priamo z aplikácie):



Obrázok 4.7: Priemerná satisfakcia populácie za čas



Obrázok 4.8: Rozptyl satisfakcie populácie za čas

Najúspešnejší riadiaci algoritmus bol nasledujúci:

```
BEGIN
   If (FreeSpace[0, -1] != 6*FreeSpace[0, -1])
      If (BlockNumber[0, -1] == 3)
         @TurnLeft;
         Go;
      Else
         Go;
   Else
      TurnLeft;
END
```

Pripomeňme, že čísla blokov sú prideľované políčkam v takom poradí, v akom sú zapísané v definičnom súbore. V tomto prípade ide o tieto políčka:

- **0.** Prázdny blok.
- **1.** Stena. (V tejto konkrétnej mape nebolo použité.)
- **2.** Potrava.
- **3.** Jed.
- **4.** Zem. (Ide o vyčerpané políčko potravy alebo jedu.)

Daný algoritmus zabezpečuje, že sa organizmus pred okrajom mapy a pred jedovatým políčkom otočí doľava. (Okraj mapy je jediný prípad, kedy je množstvo voľného miesta pred organizmom rovné šesťnásobku voľného miesta pred organizmom, keďže $6 \cdot 0 = 0$. Zároveň treba poznamenať, že sa

organizmus otočí pred jedovatým políčkom vľavo iba raz; v ďalšom kroku už sa pohne vpred.) Inak sa pohybuje neustále rovno.

Je zaujímavé, že sa tento druh algoritmov v populácii neudrží. Vysvetlením môže byť, že aj vďaka opatreniu otáčania sa pred jedovatými políčkami sa môže organizmus nasýtiť jedovatými blokmi a tým byť menej úspešný, než iné organizmy. V niektorých prípadoch vznikol zase algoritmus, pri ktorom sa organizmus otáčal pred jedovatými políčkami vždy, čím sa mohol dostať do situácie, že bol nimi obklopený a už sa z daného miesta nedostal.

Algoritmy z jednotlivých pozorovaní sa sústredia len na samotný pohyb organizmu. Je možné, že vďaka opatreniu, aby nevznikalo viac jedovatých políčok, než potravových, sa štatisticky organizmom predsa len oplatí „naslepo“ prechádzať a jesť potravu. Ďalším vysvetlením môže byť, že nasýtenie sa jedovatými políčkami nepredstavuje z dlhotrvajúceho hľadiska pre organizmus zásadný problém, pretože jeho satisfakcia nemôže klesnúť pod nulu. Avšak v ľubovoľnom bode môže zjesť bežnú potravu a tým si zvýšiť hodnotu fitness. Algoritmy, ktorými sa organizmus snaží vyhýbať jedovatým políčkam, nemusia organizmus k potrave doviesť vôbec, čo sa v konečnom dôsledku prejaví ako horší spôsob.

V nasledujúcej tabuľke uvádzame ďalšie podobné algoritmy z iných pokusov s mierne odlišnými nastaveniami, avšak pre rovnakú mapu:

```
BEGIN
   If (BlockNumber[0, -1] == 2)
      Go;
   Else
      If (BlockNumber[0, -1] != 0)
        @TurnRight;
        Go;
      Else
        Go;
END
```

```
BEGIN
   If (BlockNumber[0, -1]%Time < 3)
      Go;
   Else
      TurnRight;
END
```

```
BEGIN
   If (BlockNumber[0, -1] != 3)
      If   (BlockNumber[0,   -1]   >=
OrganismAmount[0, -1])
        Go;
      Else
        @Action;
        TurnLeft;
   Else
      If (Direction == Direction)
        TurnLeft;
      Else
        @Go;
        Go;
END
```

Tabuľka 4.3

Druhý algoritmus funguje vďaka modulárnej aritmetike, kde pre okraj mapy táto podmienka splnená nie je a organizmus sa pred ním otočí doprava.

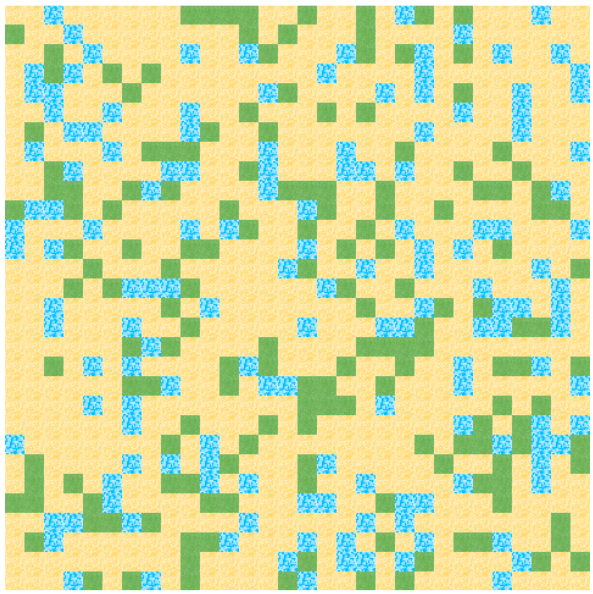
V treťom algoritme podmienka `BlockNumber[0, -1] >=`

`OrganismAmount[0, -1]` *neplatí okrem iného ani pre okraj mapy, kde je číslo bloku rovné -1, ale organizmov je 0, vďaka čomu sa organizmus otočí doľava.*

Záver: Výsledky pokusov ukazujú v prospech vyhodnocovania fitness po väčšom množstve krokov. Ukazuje sa, že vyhodnocovanie po málom počte krokov môže spôsobiť preferovanie algoritmov, ktoré sa chvíľkovo dostali navrch vďaka dočasnej situácii spôsobenej aktuálnym stavom mapy, ale sú nespoľahlivé z dlhodobejšieho hľadiska. Príkladom môže byť algoritmus z pozorovania 4 z prvej série pozorovaní, ktorý len káže organizmu ísť doprava. Takýto algoritmus mohol byť preferovaný pravdepodobne iba kvôli náhodnej situácii, kde sa na istom vodorovnom páse políčok objavilo väčšie množstvo potravy, ktoré organizmus mohol takto zjesť. Keďže sa tento organizmus stal náhodou krátkodobo najsýtejší, bol považovaný za najúspešnejšieho a došlo k rozmnoženiu tohto druhu kódu. K tejto situácii v druhej sérii pozorovaní dochádzalo menej, pretože sa organizmy nestihli stať úspešnejšie za tak krátky čas takýmto nepraktickým algoritmom.

4.3 Experiment 2

Druhý experiment bol zameraný na vplyv obmedzenia hĺbky kódového stromu na vývoj stratégií. Simulácia prebiehala na mape M1.omf uloženej v adresári Maps/Food&StaticPoison/. Definičný súbor bol DF.xml uložený v tom istom adresári.



Obrázok 4.9: Mapa Maps/Food&StaticPoison/M1.omf použitá v druhom experimente

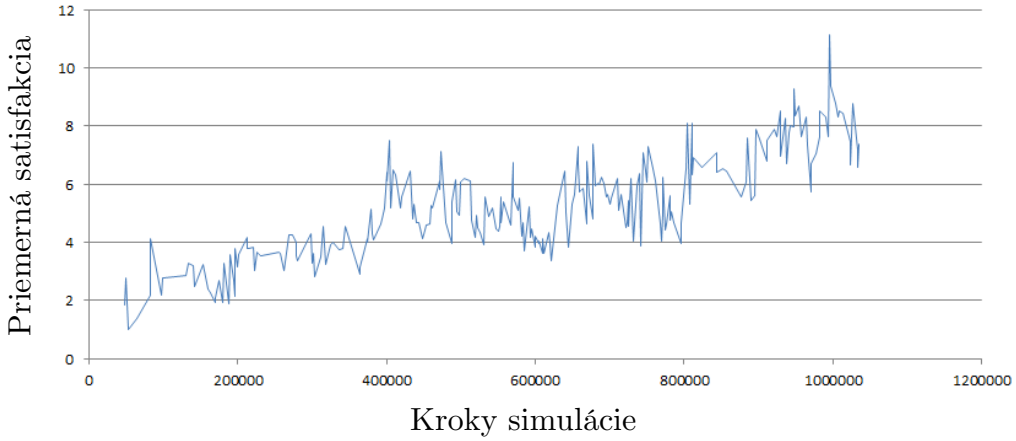
Mapa modeluje situáciu potrava a statický jed. Je veľkosti 30x30 a neobsahuje žiadne steny.

Vykonali sme dve pätice pozorovaní. Prvá pätica sleduje obmedzenie hĺbky kódového stromu na hodnotu 6, druhá pätica na hodnotu 12.

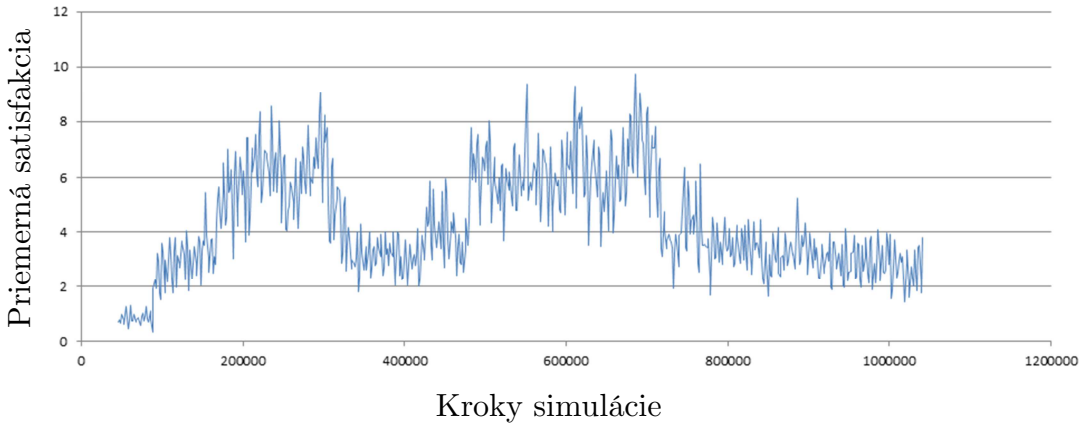
Predpokladáme, že väčšia voľnosť v hĺbke stromu spomalí konvergenciu organizmov k rozumným stratégiám, nakoľko sa exponenciálne zvýši počet možností, ako by mohol strom vyzerať.

Simulovali sme, podobne ako v prvom experimente, 1 041 730 krokov. Fitness bola vyhodnocovaná po 120 krokoch. Organizmov sme na plochu umiestnili 10. Namerané priemerné satisfakcie populácií za čas sme potom z jednotlivých pätíc spriemerovali. Výsledky možno vidieť na grafoch na obrázkoch 4.10 a 4.11.

Ukazuje sa, že pri menšej hĺbke kódového stromu sa priemerná satisfakcia populácie s časom zvyšuje, zatiaľ čo pri väčšej hĺbke táto závislosť nie je natoľko monotónna. Vidieť začiatočný rast satisfakcie, ktorý sa po približne 300 000 krokoch spomalí a následne začne satisfakcia organizmov klesať. Celý tento trend sa však ešte raz v priebehu simulovaných 1 041 730 krokov zopakuje. V prvej pätici pozorovaní sme namerali celkovú priemernú satisfakciu 5,23, medián bol 5,11. V druhej pätici hodnoty poklesli na priemer 4,27 a medián 3,84, čo potvrdzuje domnienku ohľadom veľkosti stromov, ktorú sme na začiatku experimentu naznačili.



Obrázok 4.10: Priemerná satisfakcia populácie pri prvej pätici meraní



Obrázok 4.11: Priemerná satisfakcia populácie pri druhej pätici meraní s väčšou maximálnou hĺbkou kódového stromu

Zároveň sa zľahka pozrieme na niektoré vyvinuté stratégie z jednotlivých pozorovaní. Podobne, ako v prvom experimente, aj tu uvádzame najlepšiu[5] stratégiu z každého pozorovania a fitness, ktorá jej na konci simulácie prislúchala.

Pozorovanie 1	**Pozorovanie 2**	**Pozorovanie 3**

`BEGIN`
`  If (0 != (FreeSpace[-1, -1]+1))`
`  TurnRight;`
`  Else`
`  Go;`
`END`
	`BEGIN`
	`  If (BlockNumber[0, -1] < 7/BlockNumber[-1, 0])`
	`  TurnRight;`
	`  Else`
	`  Go;`
	`END`
		`BEGIN`
		`  If ( BlockNumber[0, -1] == FreeSpace[0, -1]`
		`* BlockNumber[-1, 0])`
		`  @TurnRight;`
		`  Go;`
		`  Else`
		`  Go;`
		`END`

Fitness: 0,720	Fitness: 0,942	Fitness: 0,876

Pozorovanie 4	**Pozorovanie 5**

`BEGIN`
`  If (BlockNumber[0, -1] != (-1))`
`  If (FreeSpace[1, 0] != (-1))`
`  If (FreeSpace[-1, 0] == Time)`
`  If (BlockNumber[1, 0] == BlockNumber[0, 0])`
`  Go;`
`  Else`
`  @TurnLeft; GoDown;`
`  Else`
`  If (BlockNumber[0, 0] != 0)`
`  TurnLeft;`
`  Else`
`  GoUp;`
`  Else`
`  Go;`
`  Else`
`  @TurnRight; @Go; TurnRight;`
`END`
	`BEGIN`
	`  If (FreeSpace[1, 0] == FreeSpace[0, -1])`
	`  @Go;`
	`  If (0 != Direction)`
	`  Go;`
	`  Else`
	`  @GoRight; GoLeft;`
	`  Else`
	`  @TurnLeft;`
	`  If (OrganismAmount[1, 1] == 3)`
	`  Wait;`
	`  Else`
	`  Go;`
	`END`

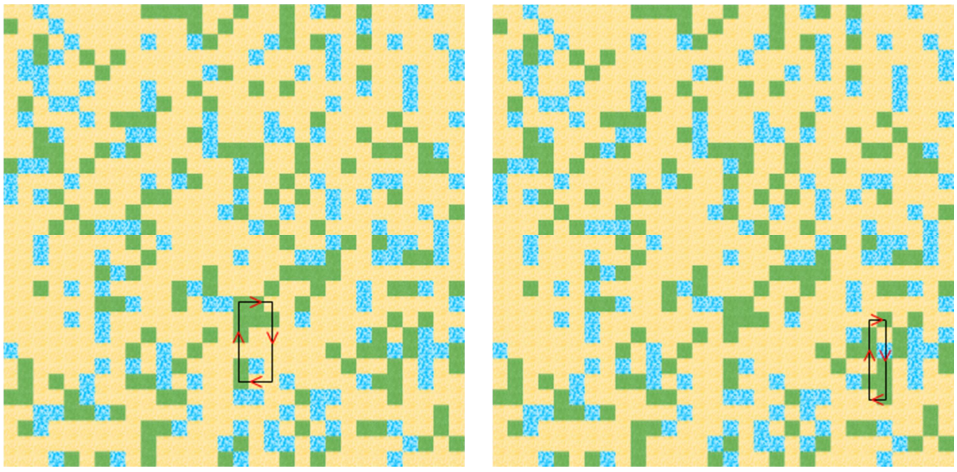
Fitness: 0,110	Fitness: 0,127

Tabuľka 4.4

Kód z pozorovania 1 je podobný situácii znázornenej na obrázku 4.4. Zabezpečuje to jediná podmienka `if (0 != (FreeSpace[-1, -1]+1))`, ktorá je splnená iba pred okrajom mapy. To je v dôsledku toho, že každé políčko hlási množstvo dostupného miesta ako -1 (nekonečno) práve okrem okraja mapy, ktoré hlási 0.

Kód z pozorovaní 2 a 3 je situačný, ale pre najúspešnejšie organizmy fungoval dobre. Ich pohyb znázorňuje obrázok 4.12.

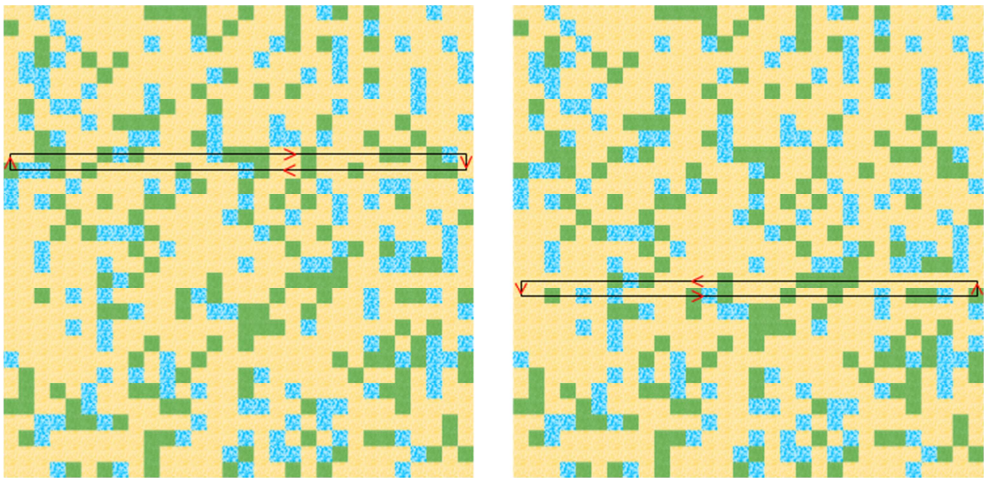
[5] Najlepšiu v zmysle, že organizmus sa ňou riadiaci mal najväčšiu fitness na konci simulácie.



Obrázok 4.12: Znázornenie pohybov najúspešnejších organizmov z pozorovania 2 (vľavo) a 3 (vpravo).

V pozorovaní 3 ide o naplnenie podmienky `If (BlockNumber[0, -1] == FreeSpace[0, -1]*BlockNumber[-1, 0])`, ktorá môže platiť, iba pokiaľ je číslo políčka pred a vľavo od organizmu rovné nule, čiže musí ísť podľa definičného súboru o prázdny blok. V pozorovaní 2 podmienka tiež skúma druh blokov naľavo a pred organizmom, je však oveľa komplikovanejšia a nebudeme ju tu rozoberať.

Kódy z pozorovaní 4 a 5 sú podľa hodnôt fitness pre organizmy menej výhodné. To je možné okrem iného aj kvôli ich veľkosti, za ktorú sú organizmy pokutované. Pohyb najúspešnejších organizmov zachytáva obrázok 4.13.



Obrázok 4.13: Znázornenie pohybov najúspešnejších organizmov z pozorovania 4 (vľavo) a 5 (vpravo).

V nasledujúcej tabuľke sú uvedené najlepšie stratégie z druhej pätice pozorovaní:

Pozorovanie 1

```
BEGIN
   If (0 > OrganismAmount[0, -1])
      GoRight;
   Else
      If ((-1) == FreeSpace[0, -1])
        Go;
      Else
        @Go; @TurnLeft; @Wait;
        @TurnLeft; @Go;
        If  (BlockNumber[0,  0]  ==
Direction)
          GoLeft;
        Else
          Go;
END
```

Fitness: 0,092

Pozorovanie 2

```
BEGIN
   @GoLeft; @GoDown; @TurnLeft;
   If (FreeSpace[-1, 0]
      > OrganismAmount[-1, 0])
      Action;
   Else
      @GoUp;
      If (BlockNumber[0, -1]
        == Time%Time)
        @GoRight; @GoDown; GoRight;
      Else
        @Go;
        If (2%3 != 0)
          If (BlockNumber[0, 0]
             != 0)
             @TurnLeft; TurnLeft;
          Else
             GoRight;
        Else
          @Wait; @GoUp; Action;
END
```

Fitness: 0,015

Pozorovanie 3

```
BEGIN
   If   (Direction   !=
   (Time +
   BlockNumber[1, 0]))
      @Go;
      If
(OrganismAmount[0, -1]
> OrganismAmount[0, 0])
        TurnLeft;
      Else
        @Go; @Go; @Go;
        TurnLeft;
   Else
      GoLeft;
END
```

Fitness: 0,263

Pozorovanie 4

```
BEGIN
   @TurnRight;
   If
(OrganismAmount[0,  -1]
== BlockNumber[-1, 0])
      @Go;
      GoLeft;
   Else
      Go;
END
```

Fitness: 0,108

Pozorovanie 5

```
BEGIN
   Memory[1] := Time;
   @Go;
   @TurnLeft;
   If (
   BlockNumber[-1, 0]
   == 1)
      GoLeft;
   Else
      Go;
END
```

Fitness: 0,102

Tabuľka 4.5

Oproti prvej pätici pozorovaní majú uvedené kódy výrazne viac zbytočných vetví, ktoré by sa dali vynechať. Patria medzi ne napríklad:

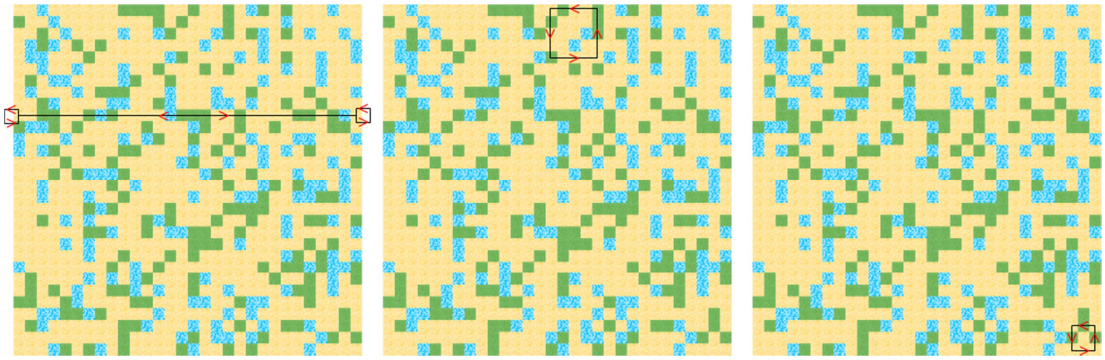
`If (2%3 != 0)` Táto podmienka je splnená vždy.

`If (Direction != (Time + BlockNumber[1, 0]))` Od piateho kroku simulácie už túto podmienku nie je možné nesplniť.

`Time%Time` Dá sa nahradiť konštantou 0.

`If (BlockNumber[-1, 0] == 1)` Číslo bloku rovné 1 sa v danej mape nenachádza a teda podmienka nebude nikdy splnená.

Zároveň riešia pomerne triviálny druh pohybov organizmov, ktorých kódové stromy by mohli byť výrazne menšie. Princíp pohybov zachytáva obrázok 4.14.

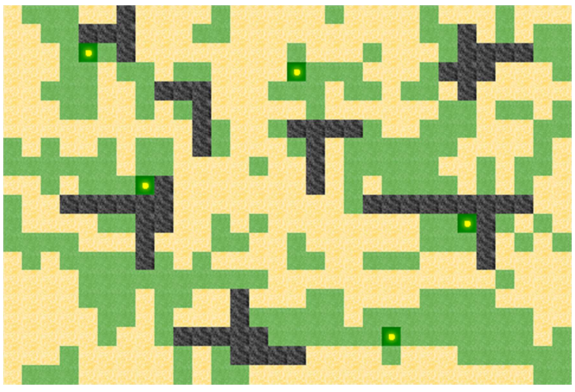


Obrázok 4.14: Znázornenie pohybov najúspešnejších organizmov z pozorovaní 1 (vľavo), 3 (v strede) a 5 (vpravo).

Záver: Druhý experiment ukazuje, že väčšia voľnosť v maximálnej hĺbke kódových stromov spomaľuje konvergenciu k rozumným stratégiám. Priemerná satisfakcia populácie je nižšia a výrazne viac osciluje. Striedanie rastu a klesania, ako zachytáva graf na obrázku 4.11, je možné vysvetliť napríklad tým, že dlhé kódy často obsahujú zbytočné vetvy dané podmienkami, ku ktorým v bežných situáciách nedôjde. Avšak z dlhotrvajúceho hľadiska sa môže prejaviť ich prípadný negatívny efekt a v populácii vďaka tomu uprednostňovať dočasne lepšie, ale inak nevýhodné stratégie, ktoré časom spôsobia celkový pokles satisfakcie. Zároveň vzniknuté stratégie často riešia problémy nepraktickým spôsobom a ich kódové stromy sú väčšie, než je nutné. Celé to je možné vysvetliť tým, že množstvo možných stromov, ktoré môžu za danej hĺbky vzniknúť, je exponenciálne väčšie, než pri hĺbke menšej. Mnohé stromy navyše zabezpečujú podobnú stratégiu a za rovnaký počet simulačných krokov sa tým pádom stihne vybrať iba z menšieho množstva zmysluplných stratégií.

4.4 Experiment 3

Posledný prevedený experiment bol zameraný na vplyv nastavenia závislosti pravdepodobnosti mutácie od fitness. K tomuto experimentu bola využitá mapa Maps/Grass&Flowers/M1.omf s definičným súborom DF.xml v rovnakom adresári. Modeluje situáciu potrava a kvety.

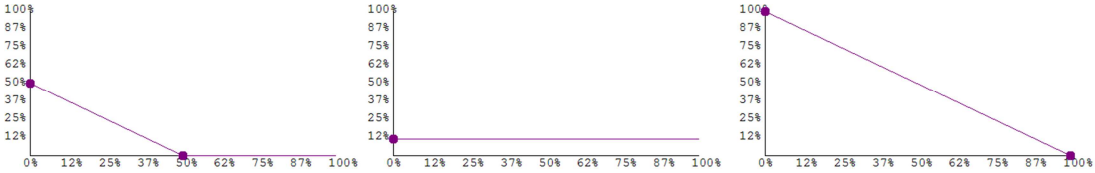


Obrázok 4.15: Mapa Maps/Grass&Flowers/M1.omf

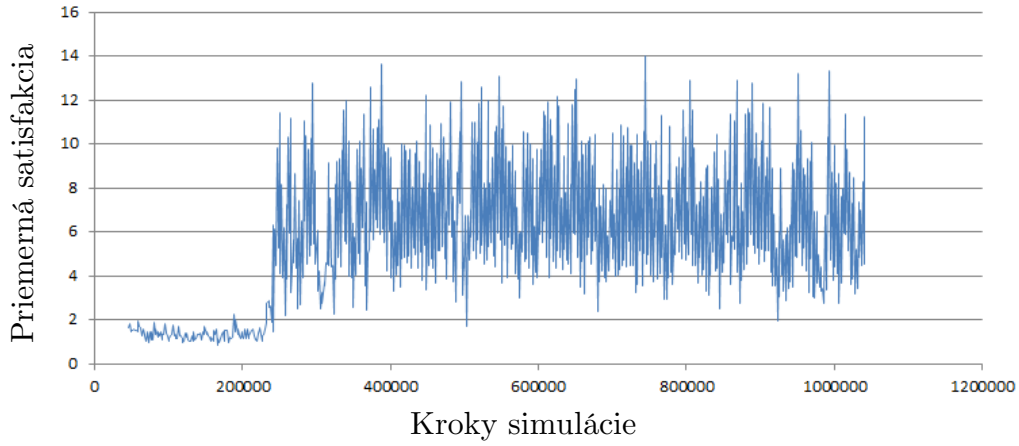
Mapa obsahuje 4 druhy blokov: prázdne políčko, potrava (zelenou), stena (čiernou) a tzv. kvet (zelenou so žltým stredom).

Experiment bol prevedený, podobne, ako v predchádzajúcich prípadoch, s 1 041 730 simulačnými krokmi. Fitness sa vyhodnocovala po 120 krokoch. Organizmov sme tentoraz umiestnili na plochu 15. Urobili sme 3 merania pre každé z troch nastavení. Prvá trojica sledovala štandardné nastavenie pre dvojicu bodov [0, 50%], [50%, 0]. Druhá trojica sledovala konštantnú pravdepodobnosť mutácie nastavenú na 10%. Tretia trojica sledovala nastavenie pre dvojicu bodov [0, 100%], [100%, 0].

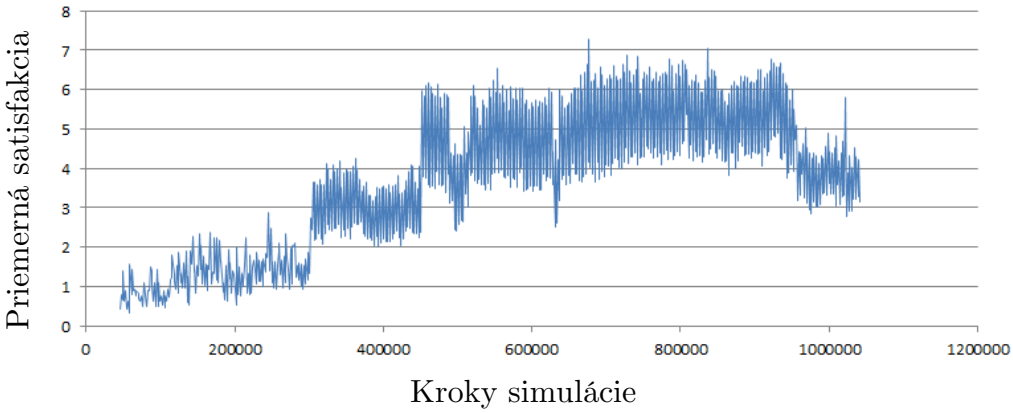
Výsledky pre priemernú satisfakciu populácie možno vidieť na grafoch na obrázkoch 4.17 až 4.19.



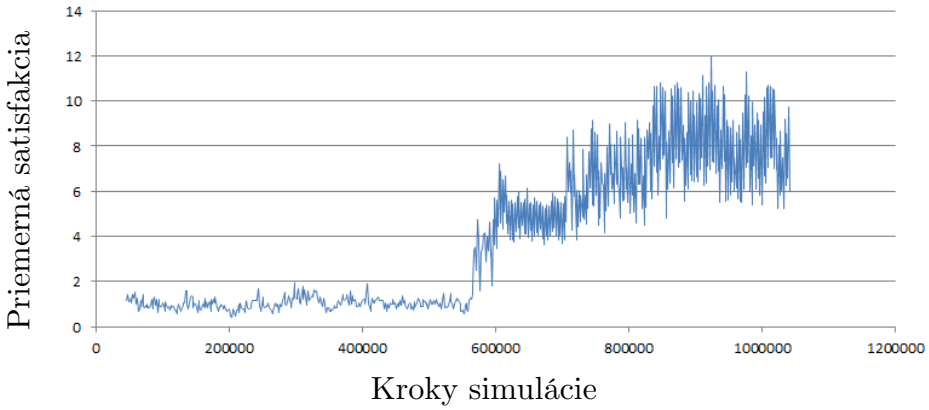
Obrázok 4.16: Nastavenia pravdepodobností mutácií od fitness pri jednotlivých trojiciach pozorovaní



Obrázok 4.17: Priemerná satisfakcia populácie pri meraniach štandardnej závislosti mutácie od fitness



Obrázok 4.18: Priemerná satisfakcia populácie pri meraniach s konštantnou pravdepodobnosťou mutácie



Obrázok 4.19: Priemerná satisfakcia populácie pri meraniach s klesajúcou pravdepodobnosťou mutácie danou bodmi [0, 100%] a [100%, 0]

Kým pre štandardné nastavenie je po dosiahnutí určitej priemernej satisfakcie populácie len snaha udržať tento stav, vidieť, že pre konštantnú a „klesajúcu“ (tak, ako bola nastavená pre tretiu trojicu pozorovaní) pravdepodobnosť mutácie má priemerná satisfakcia tendenciu rásť. Pre prvú trojicu pozorovaní je priemerná satisfakcia meraní 5,71 a medián 5,11. Pre druhú trojicu priemer 3,64 a medián 3,75; pre tretiu trojicu ide o priemer 3,71 a medián 1,52.

Zároveň tu uvádzame najlepšie stratégie z každého pozorovania. Môžeme uvážiť, že vzhľadom na políčka, ktoré sa na mape vyskytujú, dobrá stratégia bude zahŕňať pohyb organizmu, pretože inak na jeho mieste nebude rásť žiadna potrava okrem kvetov, ktoré ale v priemere vyrastú iba raz za 200 krokov. Zároveň aspoň jeden organizmus musí využívať na kvety činnosť Akcia, aby sa nimi nasýtil, pretože inak sa mapa časom pokryje iba kvetmi a pre potravu nezvýši miesto. Z nasledujúcich tabuliek vidieť, že v každom pozorovaní tieto požiadavky najlepší organizmus vždy splnil:

Pozorovanie 1-1	**Pozorovanie 1-2**	**Pozorovanie 1-3**

`BEGIN`
`  If (BlockNumber[0, -1] < 3)`
`  @TurnLeft;`
`  @Go;`
`  Action;`
`  Else`
`  Go;`
`END`
	`BEGIN`
	`  @Go;`
	`  @TurnRight;`
	`  Action;`
	`END`
		`BEGIN`
		`  @Go;`
		`  @Action;`
		`  TurnRight;`
		`END`

Fitness: 0,709	Fitness: 0,131	Fitness: 0,189

Tabuľka 4.6

Pozorovanie 2-1	Pozorovanie 2-2	Pozorovanie 2-3

BEGIN
  @Go;
  Memory[7] :=
OrganismAmount[0, 0]%2;
  @Action;
  TurnRight;
END
	BEGIN
	  If (FreeSpace[0, -1] <
	  OrganismAmount[-1, 0])
	  @Go;
	  Action;
	  Else
	  If (Memory[8] == 1)
	  @Go;
	  GoRight;
	  Else
	  @TurnLeft;
	  TurnLeft;
	END
		BEGIN
		  @TurnLeft;
		  @Action;
		  Go;
		END

Fitness: 0,110	Fitness: 0,865	Fitness: 0,049

Tabuľka 4.7

Pozorovanie 3-1	Pozorovanie 3-2	Pozorovanie 3-3

BEGIN
  @Go;
  Memory[3] := Memory[3];
  @TurnRight;
  Action;
END
	BEGIN
	  @TurnRight;
	  @Action;
	  @Go;
	  Go;
	END
		BEGIN
		  If (BlockNumber[0, -1]
		  < 2)
		  TurnLeft;
		  Else
		  @Action;
		  Go;
		END

Fitness: 0,077	Fitness: 0,019	Fitness: 0,958

Tabuľka 4.8

Je vidieť, že najmenej zbytočných vetví vznikalo práve v prvej trojici pozorovaní. V ďalších dvoch trojiciach sa zbytočne využil zápis do pamäte, čo na správanie organizmu nemalo žiaden vplyv. V kóde z pozorovania 2-2 sa navyše vyvinul test podmienky `If (Memory[8] == 1)`, ktorá ale nebola pre daný organizmus splnená, pretože hodnota ôsmej pamäťovej bunky nebola v danej chvíli 1.

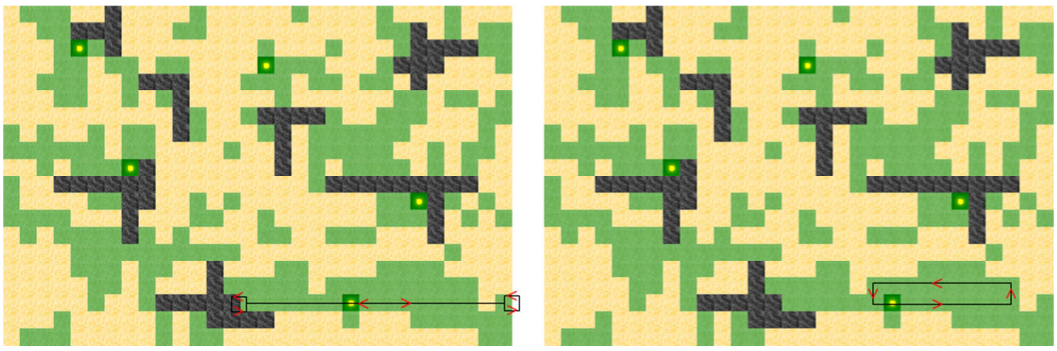
Rozoberieme ešte kódy z pozorovaní 1-1, 2-2 a 3-3, ktorých vplyv nie je na prvý pohľad zrejmý. Predtým ešte uvedieme čísla jednotlivých blokov podľa definičného súboru:

- **0.** Prázdne políčko
- **1.** Stena
- **2.** Zem (vyčerpané políčko kvetu príp. potravy)
- **3.** Potrava
- **4.** Kvet

- **Pozorovanie 1-1.** Organizmus sleduje, či sa pred ním nachádza potrava, alebo kvet. Pokiaľ áno, jednoducho sa pohne napred. Pokiaľ nie, najprv sa otočí doľava, pohne sa o 1 políčko vpred a vykoná činnosť Akcia.

- **Pozorovanie 2-2.** Ako už bolo naznačené, vykoná sa len `else` vetva podmienky `If (Memory[8] == 1)`, ktorá je inak zbytočná. Prvá podmienka je splnená vždy, pokiaľ pred organizmom nie je stena, prípadne okraj mapy. To je dané tým, že všetky ostatné políčka hlásia -1 voľného miesta (nekonečno), ale počet organizmov nemôže byť nikde záporný.

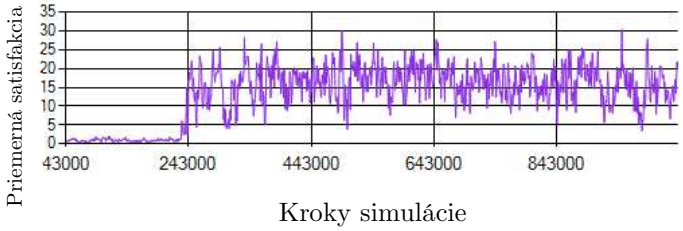
- **Pozorovanie 3-3.** Pokiaľ je pred organizmom stena, prázdne políčko alebo okraj mapy, otočí sa vľavo, inak strieda pohyb vpred s činnosťou Akcia.



Obrázok 4.20: Znázornenie pohybov najúspešnejších organizmov z pozorovaní 2-2 (vľavo) a 3-3 (vpravo).

Okrem prvého pozorovania ohľadom zbytočných vetví však nie je výraznejší rozdiel v kóde organizmov medzi jednotlivými trojicami pozorovaní.

Záver: Z výsledkov experimentu je možné usúdiť, že vplyv nastavenia závislosti pravdepodobnosti mutácie od fitness hrá len menej významnú rolu pri vývoji populácie. Určitá pravdepodobnosť mutácie je, pochopiteľne, nutná a je dobré, ak nie je nastavená na nulu pri príliš nízkych hodnotách fitness. Pri komplikovanejších mapách, kde je neustále dosahovanie nadpolovičnej fitness pomerne ťažké, je však možné ponechať štandardné nastavenie, ktoré zabezpečí rozumný vývoj populácie. Pokiaľ sa už dosiahol, bude snaha populácie si len tento stav udržať. Naopak pri jednoduchších mapách by sa mohol predčasne ukončiť a je lepšie použiť iné nastavenie. Napriek tomu, že výsledky grafov ukazujú na neustály rast satisfakcie pri neustálej mutácii, je nutné poznamenať, že pri štandardnom nastavení dosiahla populácia v priemere zo všetkých meraní najlepšie výsledky. To je však možné vysvetliť tým, že pri pozorovaní 1-1 dosahovala populácia výnimočne veľké hodnoty satisfakcie (obrázok 4.21). Rozdiely v dosiahnutých stratégiách totiž nevykazujú významnejší rozdiel medzi jednotlivými trojicami pozorovaní.



Obrázok 4.21: Priemerná satisfakcia populácie v pozorovaní 1-1

5. Záver

Cieľom tejto práce bolo vytvoriť aplikáciu na simulovanie jednoduchých agentov v rozličných užívateľom nastaviteľných prostrediach, ktorých chovanie je dané ich vlastným kódom. Aplikácia zároveň mala umožňovať rozšírenie na jednoduchú výpočtovú inteligenciu, zabezpečiť nejakú adaptáciu kódu jednotlivých agentov. Jej výsledkom je program Organisms. Ten umožňuje užívateľovi okrem spomínaných vlastností aj riadiť simuláciu prostredníctvom bohatých nastavení, prezerať kódy jednotlivých agentov, tie následne ukladať prípadne načítať z externého súboru. Program taktiež podporuje uchovávanie štatistických údajov v priebehu simulácie, ktoré je schopný zobrazovať pomocou jednoduchých grafov.

Súčasťou práce bolo taktiež previesť s hotovou aplikáciou niekoľko experimentov a ich výsledky vo vhodnej forme prezentovať. Tejto časti sa venuje kapitola 4.

Aplikácia Organisms ponúka bohaté možnosti a umožňuje modelovať veľké množstvo situácií a scenárov. V tejto práci sú v rámci experimentov rozobraté 3 reality a je naznačená aj realita popisujúca prostredie s viacerými druhmi organizmov, ktoré sa živia rozličnými druhmi potravy.

Je tu taktiež priestor pre ďalšie rozšírenia aplikácie. Šlo by napríklad vytvoriť ďalšie špecifické uzly kódového stromu, ktorým sa jednotliví agenti riadia. Príkladom môže byť uzol And alebo Or, ktorý by predstavoval spájanie jednoduchších podmienok do komplikovanejších podstromov. Ďalším rozšírením by mohlo byť pridanie kríženia kódov agentov, ktoré bolo v tejto aplikácii vynechané. Je taktiež možné prerobiť samotnú adaptáciu kódu agentov, ktorá je pre potreby tejto práce založená na evolučných algoritmoch, a nahradiť ju lepším algoritmom lokálneho prehľadávania.

6. Prílohy

Prílohou tejto práce je CD, ktoré obsahuje nasledujúce položky:

- Aplikáciu Organisms.exe v priečinku Application/ spolu so zbierkou voľných obrázkov pre ikony blokov a organizmov v priečinku Application/custom/
- Inštalačný archív .NET Framework 4 v priečinku Prerequisites/
- Zdrojový kód aplikácie pre prostredie Microsoft Visual Sudio 2010 v priečinku Sources/
- Dokumentáciu ku zdrojovým kódom vygenerovanú nástrojom Doxygen v priečinku Documentation/
- Priečinok s ukážkovými mapami a mapami použitými v jednotlivých experimentoch v priečinku Maps/
- Text práce vo formáte PDF v priečinku Text/

7. Literatúra

[1] HOLAN, Tomáš. 2008. *Broučci, simulace umělého života* [online]. Dostupné z WWW: <http://ksvi.mff.cuni.cz/~holan/broucci/>

[2] SEKAJ, Ivan. 2005. *Evolučné výpočty a ich využitie v praxi*, Bratislava: Iris, s. 69-71, 11, 44-52, 73-77, ISBN 80-89018-87-4

[3] Michael - AFFENZELLER, Stefan - WAGNER, Stephan - WINKLER, Andreas - BEHAM. 2009. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications,* CRC Press, s. 30-37, ISBN 978-1-4200-1132-6

[4] Erich - GAMMA, Ralph - JOHNSON, Richard - HELM, John - VLISSIDES. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software,* s. 183-195, ISBN 978-032170069-8

[5] LEPŠ, Matěj. 2008. *Genetické programování* [online]. Prezentácia z prednášky na FSV ČVUT, s. 13-21, [cit. 07. 05. 2015] Dostupné z WWW: <http://klobouk.fsv.cvut.cz/~leps/teaching/mmo/prednasky/prednaska12_GP.pdf>

[6] Charles - OFRIA, David - BRYSON, Claus - WILKE. 2014. *Avida: A Software Platform for Research in Computational Evolutionary Biology* [online]. s. 8-9, [cit. 07. 05. 2015] Dostupné z WWW: <http://www.cse.msu.edu/~ofria/pubs/2009AvidaIntro.pdf>

[7] WILENSKY, Uri. 2015. *NetLogo User Manual* [online]. [cit. 07. 05. 2015] Dostupné z WWW: <https://ccl.northwestern.edu/netlogo/docs/>