Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Modulárna implementácia hier v Answer Set Programovaní

(bakalárska práca)

Milutín Krištofič

vedúci:
Mgr.Jozef Šiška

Bratislava, 2007

Čestne vyhlasujem, že predkladanú bakalársku prácu som vypracoval samostatne s využitím uvedenej literatúry.

Milutín Krištofič

Abstrakt

Cieľom bakalárskej práce bolo vyskúšať možnosti modularizácie answer set programov pri implementácií reálnych problémov. V rámci práce sme navrhli predstavu modularizácie logického programu a následne zostrojili v jazyku Python ako rozšírenie pre framework herného enginu od Jozefa Šišku. Tiež sme predstavili samotné logické programy a ich vhodnosť začlenenia do praktických problémov ako je vytváranie počítačových hier.

kľúčové slová: logický program, answer set, modularizácia, počítačová hra, umelá inteligencia

Obsah

Kapitola 1

Úvod

Paradigma answer set programovania je zaujímavou alternatívou k riešeniu programátorských úloh. Našou motiváciou v tejto bakalárskej práci boli priblížiť túto disciplínu z akademickej oblasti bližšie do praxe. Ako vhodná oblasť technického rozvoja na experimentovanie s novátorskými postupmi sa ukázali práve počítačové hry. Ich náročnosť vývoja však požaduje od paradigmy programovania rôzne pomocné nástroje. V rámci tejto práce sme navrhli a implementovali jeden z nich: modularizáciu. Takýto nástroj nielen, že umožňuje rozdeliť projekt na menšie logické časti, na ktorých môžu spolupracovať rôzne programátorské tímy, ale zároveň umožňuje vybrať len istú potrebnú podmnožinu potrebných pravidiel a tak urýchliť výpočet.

Jozef Šiška vytvára framework na tvorbu hier v logických programoch[2], ktorý má poskytnúť rôzne nástroje na vývoj hier. Naša implementácia modularizácie mala za cieľ sa stať súčasťou tohto frameworku, preto sme volili rovnaké nástroje na jej vytvorenie.

Nakoniec sme sa pokúsili na známom príklade o plánovaní z oblasti umelej inteligencie prezentovať nielen možnosti logických programov, ale tiež frameworku na tvorbu hier a hlavne našej implementácie modularizácie.

1.1 Usporiadanie bakalárskej práce

V druhej kapitole sme definovali jazyk logických programov a základné triedy logických programov, aby sme ich priblížili aspoň z formálnej stránky. Následne v tretej kapitole sa snažíme poukázať na vhodnosť využitia logických programov v oblasti počítačových hier a zároveň, aj na potrebu možnosti ich modularizovať. V štvrtej kapitole predstavujeme myšlienku nami preferovaného spôsobu modularizácie, spolu s motiváciou jeho výberu. V piatej kapitole sa venujeme samotnej realizácii modularizácie. Na záver v šiestej kapitole demonštrujeme na príklade výhody modularizácie aj answer set programovania.

Kapitola 2

Answer Set Programovanie

V tejto kapitole predstavíme základné definície logických programov. Najprv zavedieme jazyk logického programu a následne podstatné triedy logických jazykov. Rozhodli sme sa nezaoberať sa definíciami dynamického logického programu, či plánovaniu, keďže modularizáciu robíme len pre answer set programovanie.

2.1 Jazyk logických programov

Pojmy ako term, atóm sú prevzaté z klasickej matematickej logiky.

Definícia 1 (Term) *definujeme rekurzívne:*

1. *premenná alebo konštanta,*
2. *ak $t_1, \ldots t_n$ sú termy a f je funkčný symbol arity n, potom $f(t_1, \ldots t_n)$ je term.*

Základný term neobsahuje žiadne premenné.

Definícia 2 (Atóm) *je výraz tvaru $p(t_1, \ldots, t_n)$, vytvorený z predikátového symbolu p a termov $t_1, \ldots t_n$. Základný atóm neobsahuje žiadnu premennú.*

Definícia 3 (Literál) *je atóm, alebo negácia atómu. Základný literál neobsahuje žiadnu premennú.*

Definícia 4 (Herbrandovské univerzum) *jazyka L je množina všetkých základných termov jazyka. Herbrandovská báza jazyka L je množina všetkých atómov L.*

Definícia 5 (Logický program) *je spočitateľná množina pravidiel formy:*

$$L_0 \leftarrow L_1, \ldots L_n.$$

kde L_i je nejaký literál. Ak $n = 0$, voláme toto pravidlo fakt a zapisujeme ho L_0.

Pravidlá sú jednoduché implikácie, ktorých postačujúce podmienky sú konjukciou literálov a nutná podmienka je len jeden literál. Samozrejme vieme nahradiť všetky klasické logické spojky pomocou týchto pravidiel.

Definícia 6 *Nech r je pravidlo logického programu tvaru $L_0 \leftarrow L_1, \ldots L_n.$, potom L_0 nazývame hlava pravidla r a $L_1, \ldots L_n$ telo pravidla.*

Definícia 7 (Herbrandovská interpretácia) *logického programu P je nejaká podmnožina Herbrandovskej bázy jazyka L programu P.*

Definícia 8 *Uzavretý atóm $p(t_1, \ldots, t_n)$ je pravdivý v interpretácii, ak n-tica objektov priradených príslušným termom patrí do relácie priradenej predikátovému symbolu.*

Definícia 9 (Herbrandovský model) *logického programu P je Herbrandovská interpretácia P, pri ktorej je každá veta z P pravdivá.*

2.2 Triedy logických programov

Logický program, zadefinovaný vyššie, sme popísali ako najvšeobecnejšiu triedu. V nasledujúcich častiach si zadefinujeme striktnejšie triedy, ktoré sú zaujímavé pre túto prácu:

2.2.1 Definitný logický program

Definitné logické programy nepovoľujú negáciu vo svojich pravidlách. Ich sémantikou je jediný najmenší model.

Definícia 10 (Definitný logický program) *je logický program, ktorého pravidlá sú tvaru:*

$$L_0 \leftarrow L_1, \ldots L_n.$$

kde L_i je nejaký atóm.

Definícia 11 (Sémantikou definitného logického programu p) *je najmenší model P. Nech M je množina všetkých modelov P, potom $\bigcap_{m \in M}$ je najmenší model P.*

Kompletná Herbrandovská báza definitného logického programu je jeho interpretáciou. Zjednotenie modelov definitného logického programu je tiež jeho modelom. Z týchto dvoch vlastností, ktoré nie je ťažké dokázať, vyplýva nasledujúca veta:

Veta 1 *Každý definitný logický program má najmenší model.*

2.2.2 Normálny logický program

Normálny logický program povoľuje negáciu v tele pravidla. Bohužiaľ nemôže použiť sémantiku minimálneho modelu, hlavne z dôvodu možnej existencie viacerých minimálnych modelov. Preto si vyberáme model normálneho logického programu iným spôsobom.

Definícia 12 (Normálny logický program) *je logický program, ktorého pravidlá sú tvaru:*

$$L_0 \leftarrow L_1, \ldots L_n.$$

kde $L_i, i \in \{1, \ldots, n\}$ *je nejaký literál a* L_0 *je nejaký atóm.*

Definícia 13 (Gelfond-Lifschitzov operátor) *Nech* P *je normálny logický program a* I *je interpretácia. Gelfond-Lifschitzová transformácia* P *podľa* I *je program* P^I *získaný z* P *pomocou;*

- *odstránením každého pravidla obsahujúceho literál not* A *a* $A \in I$
- *odstranením každého literálu not* A*, kde* $A \notin I$ *z pravidiel programu* P*.*

Program P^I *je definitný logický program, ktorý má jediný najmenší model označovaní* $least(P^I)$*.*

Definícia 14 (Stabilný model normálneho logického programu) *je interpretácia* I *normálneho logického program* P *práve vtedy, keď* $least(P^I) = I$*.*

GL operátor predstavuje intuitívny pohľad: na začiatku zvolíme, ktoré atómy sú pravdivé a ktoré sú nepravdivé. Z tohto predpokladu a pomocou pravidiel vytvoríme definitný logický program. Ak z neho odvodíme všetky predpoklady a nič viac, potom sme získali stabilný model.

2.2.3 Zovšeobecnený logický program

Zovšeobecnený logický program obsahuje negáciu aj v hlave. Pri výpočte stabilného modelu je program P transformovaný na definitný logický program tak, že negácie sa zmenia na pozitívne atómy.

Definícia 15 (Zovšeobecnený logický program) *je definovaný ako logický program.*

Definícia 16 (Stabilný model zovšeobecneného logického programu) *Nech operátor* $'$ *zmení všetky defaultné literály na nové atómy not_A, potom interpretácia* I *zovšeobecneného logického programu* P *je stabilný model práve vtedy, keď*

$$I' = least((P \cup M^-)'^{I'})$$

2.3 Implementácie answer set programovania

Súčasný trend sú implementačné experimenty so stabilnými modelmi. Predovšetkým projekt helsinskej univerzity smodels[5] a viedenskej univerzity dlv[6]. Aj keď sú problémy s výpočtom stabilného modelu, pokroky v tejto oblasti, hlavne optimalizačné, umožnili vypočítať výsledok aj veľkých programov v rozumnom čase. Problémom však asi navždy zostane, že už taká jednoduchá otázka, ako či existuje stabilný model programu P, je NP-úplný problém.

Kapitola 3

Počítačové hry

Počítačové hry sú zaujímavou oblasťou technického rozvoja. Ak neuvažujeme o hrách vytvorených len pre zábavu, zostávajú nám zväčša projekty pokúšajúce sa aproximovať reálny svet pomocou digitálnej techniky. V niektorých oblastiach, hlavne v audiovizuálnej stránke hry dosiahli obrovské pokroky. Hardwarovo je táto oblasť podporovaná špeciálnymi technickými nástrojmi, ako sú zvukové a 2D, 3D grafické karty, softwerovo vznikajú prepracované designy hier, do ktorých sa implementujú stále komplikovanejšie súčasti, ako napríklad simulácie fyzikálnych zákonov. Avšak umelá inteligencia v hrách stagnuje.

Vytvoriť počítačového protivníka je veľmi náročne. Už len keď vidíme aké ťažké bolo zrealizovať ho v tak známych a jednoduchých hrách, ako je Šach, Dáma, či ako stále je to neriešiteľné v jednej z najjednoduchších hier Go. Hráči už v podstate nepotrebujú umelého protivníka, takých si zaobstarajú pomocou pripojenia počítačov a hraní medzi sebou. Ich predstavou je skôr, že svet, v ktorom sa pohybujú bude žiť vlastným životom. V praxi to znamená, že NPC (nehrajúce postavy) budú mať aspoň základné logické uvažovanie, ktoré im pomôže vykonávať rôzne akcie, plánovať a podobne.

3.1 Využitie logických programoch v hrách

Nami popísaná predstava o hernom svete, v ktorom každá NPC má vlastný aparát na tvorbu logických záverov zo stavu, v akom sa nachádza, je logickou a potrebnou aproximáciou sveta. Realizácia za pomoci logických programov môže čiastočne splniť očakávania, veď samou podstatou logických programov je vyplývanie. Ak by sme NPC zadefinovali pravidlá a atómy pre popis stavu sveta, stabilný model by bol ich výsledkom uvažovania. V stabilnom modeli by boli popísané akcie, ktoré má NPC spraviť k naplneniu jej cieľu. Je teoreticky nepodstatné, či si akciu vyberie náhodne, alebo zvolí nejaký spôsob preferencie, zadefinovaný podľa pravidiel.

Pokúsime sa na jednoduchom príklade priblížiť problematiku. NPC postupne, ako ubieha deň dostáva hlad, či už v nejakú určitú dobu, alebo po vykonaní ne-

jakej práce, každopádne to vyplynie z pravidiel. Jej cieľom bude sa najesť a má viacero spôsobov, ako to môže vykonať. Môže nakúpiť jedlo na trhu, ak má dostatok peňazí a na trhu sú predavači s tovarom, alebo si môže ísť uloviť zviera do lesa. Každý stabilný model môže prezentovať jedno riešenie ako napríklad ulovenie zvieraťa.

3.2 Modularizácia programu

Hry boli od počiatkov rozdelené na rôzne levely z viacerých dôvodov. Jeden z hlavných bol, aby sa hra nemusela načítať celá, ale len jedna jej časť - level. Na podobnom princípe funguje modularizácia. Snaha rozdeliť program na logické celky a pokúšať sa aplikovať len tú časť, ktorá je aktuálna. Jedna z možností je rozdeliť hru na oblasti a každú oblasť definovať svojimi pravidlami, ktoré sú pre ňu špecifické. Existuje aj iná možnosť. V hre je veľa objektov, či už ide o NPC, alebo o predmety, s ktorými manipuluje hráč. Pomocou modularizácie môžeme logický program rozdeliť a každý modul bude špecifikovať ovládanie konkrétnej NPC, či manipuláciu s predmetom. Modul sa aktivuje, ak hráč sa bude nachádzať vedľa objektu, ktorý špecifikuje. Použitie jedného zo spôsobov alebo ich kombinácie umožní nielen prehľadnejšie definovať pravidlá hry, ale zároveň urýchli výpočet stabilného modelu.

Kapitola 4

Modularizácia logického programu

V tejto kapitole predstavujeme myšlienku modularizácie logického programu. V prvom rade chceme vysvetliť motiváciu a dôvody, prečo si myslíme, že modularizácia by mala byť ako nástroj v paradigme logického programovania. Ďalej ukazujeme náš výber spôsobu rozdelenia aj spracovania modularizácie, ktorý sme implementovali do frameworku na tvorbu počítačových hier[2]. Nakoniec predstavujeme ešte iný spôsob modularizácie a porovnáme výhody a nevýhody s nami preferovaným spôsobom.

4.1 Motivácia

Dva z hlavných bodov kritiky proti programovaniu v logických programoch (answer set programovaní) sú komplikovanosť písania programov a pomalá rýchlosť ich spracovania. Oba tieto problémy aspoň čiastočne rieši modularizácia.

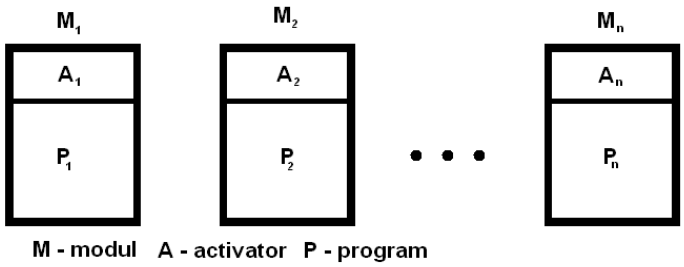
V prvom prípade umožňuje rozdeliť program na podprogramy (moduly), na ktorých môžu pracovať rôzne skupiny ľudí. V imperatívnych jazykoch sú rôzne balíkové, alebo modulové systémy samozrejmosťou. Vďaka týmto nástrojom sa môžu časti kódu znovu použiť v novších projektoch a nemusí sa začínať vždy od začiatku. Ďalšou výhodou je sprehľadnenie programu, vďaka tomu, že môžeme rozdeliť program do logických celkov.

V druhom prípade, problém pomalej rýchlosti spracovania, môžeme riešiť pomocou voľby aktivovaných modulov. Ako sme spomenuli už v predchádzajúcej kapitole s hrami, veľakrát je zbytočné vypočítavať pravidlá pre celý svet, stačí nám iba určitá časť sveta, či iba pravidlá, ktoré sú blízke stavu, v ktorom sa nachádzame. Takto zúžime počet pravidiel, ktoré musí program spracovať a tak zvýšime jeho rýchlosť.

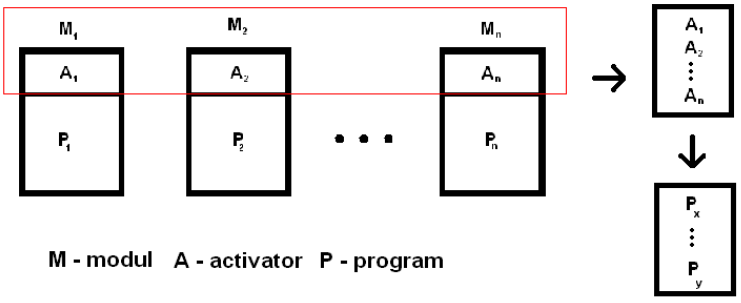
4.2 Zvolený spôsob modularizácie

Existujú rôzne optimalizačné spôsoby ako urýchliť spracovanie programu. V samotnej matematickej logike existujú spôsoby ako zistiť, či nejaké pravidlo nie je zbytočné. Takéto automatické preriedenie je jednou cestou, my sme však zvolili cestu, v ktorej programátor určuje, kedy sú pravidlá zbytočné a kedy sú naopak potrebné.

Programátor si rozvrhne rozdelenie pravidiel do rôznych súborov (modulov), podľa svojho najlepšieho vedomia. Každý modul sa bude skladať z dvoch častí: aktivačnej a programovej. Aktivačná časť bude mať v sebe pravidlá, ktoré budú určovať za akých podmienok, má byť tento modul spustený. Prípadne za akých podmienok je potrebné, aby sa načítali iné moduly. Bude tu rezervovaný špeciálny atóm, ktorý bude ako parameter obsahovať názvy modulov, ktoré treba načítať. Nasledujúci obrázok tieto myšlienky ilustruje.



Spracovanie modularizovaných programov prebehne nasledovným spôsobom: programátor určí, ktoré moduly sa majú spracovať a tie pošle implementácií na spracovanie. Táto implementácia si zoberie z každého programu aktivačnú časť a všetky spojí do jedného logického programu. Tento logický program sa skladá z aktivačných pravidiel a v jeho modeli (náhodne vybranom) budú atómy, ktoré určujú, aké moduly sa majú aktivovať. Potom sa vezmú programové časti z týchto modulov a tie budú tvoriť výsledný logický program, ktorého modely nás zaujímajú. Obrázok ilustruje postup.



Modularizácia nám spôsobuje, že potrebujeme namiesto jedného spracovať dva logické programy. Avšak aktivačné časti sú oproti programovým častiam veľmi malé a preto ich spracovanie je oveľa menej náročné na výpočet, než keby

sme chceli vypočítať model celého logického programu. Uvedený spôsob kladie nároky na programátora, ktorý musí vytvoriť aktivačné programy tak, aby sa mu načítali všetky potrebné programové časti. Je na ňom, či si vytvorí jeden aktivačný program a ostatné nechá prázdne, či v každom module bude mať vlastné pravidlá.

4.3 Alternatívny prístup

Existuje veľa spôsobov ako robiť modularizáciu programu. Jedno zo zaujímavých riešení je takzvaný iteratívny spôsob. Na začiatku by programátor dal logický program modulárnej implementácií, ktorá by ho spracovala. Ak by sa zistilo, že v modeli sa nachádza atóm požadujúci nejaký modul, načítal by sa modul, pridal by sa do programu a takto by sa pokračovalo až by sme nakoniec už nechceli načítať žiadny nový modul.

Nepotrebujeme rozbíjať programy na rôzne časti, stačí nám pridať špeciálny atóm. Na začiatku by sme spustili logický program celého sveta, ktorý by určil pomocou stavu, v ktorej časti sa nachádza náš problém a pridávaním modulov by riešil stále konkrétnejšie body problémov. Programovanie takýmto spôsobom by bolo pravdepodobne komplikované, museli by sme vedieť vyberať rôzne vrstvy pravidiel. Takisto každý logický problém môže mať viacero modelov, preto vývoj spracovania programu by sa skôr podobal na strom, než na spájaný zoznam.

Kapitola 5

Implementácia modularizácie

Táto kapitola predstavuje našu implementáciu modularizácie. Jej myšlienku sme rozobrali v predchádzajúcej kapitole, tu sa len v krátkosti zmienime o spôsobe implementácie. Hneď od začiatku sme počítali s tým, že bude slúžiť ako jeden z nástrojov pre framework na tvorbu počítačových hier[2]. Tento framework je naprogramovaný v programovacom jazyku *python*[8]. Ide o interpretovaný, dynamický objektovo-orientovaný jazyk, vhodný na integráciu s inými jazykmi. Na spracovanie logického programu sme používali *smodels*[5], alebo z viedenskej technickej univerzity program *dlv*[6]. Pomocou nástroja *mkatoms*[7] sme výstupy z týchto dvoch programov normalizovali na jednotný tvar.

5.1 Definovanie modulu

V predchádzajúcej kapitole sme prišli s myšlienkou modulu, ktorý má dve časti: aktivačnú časť a programovú časť. Nastáva otázka, ako v praxi implementovať tieto dve časti. Jedna možnosť je mať obidve časti v jednom súbore s nejakou špeciálnou direktívou, ktorá vie povedať nášmu programu, kde končí aktivačný kód a kde začína programová časť. Iná možnosť je vytvoriť dva súbory, kde v každom bude iná časť. Nakoniec sme zvolili obidva spôsoby.

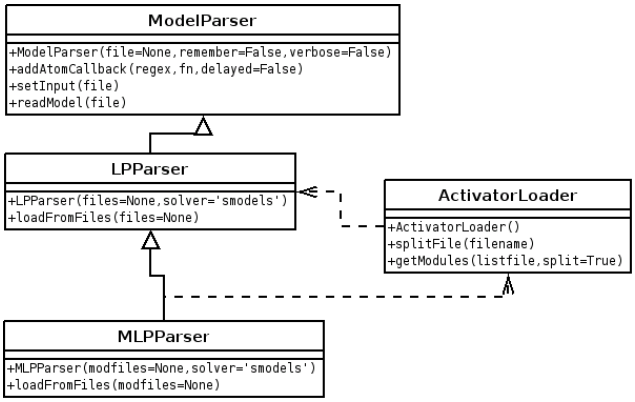
Ak máme len jeden súbor, ten je rozdelený pomocou direktívy "`%::endmodule`". Symbol `%` je v smodels aj dlv symbolom pre začiatok komentáru. Vďaka tomu môžeme takýto súbor používať aj ako normálny logický program. Naša implementácia si potom automaticky rozdelí tento súbor na `názovsúboruspríponou.act`, kde je aktivačná časť a `názovsúboruspríponou.data`, kde je programová časť. Implementácia pripúšťa ako druhý spôsob aj rovno zadať dva súbory .act a .data a v tom prípade ich automaticky použije.

Veľmi dôležité je dohodnúť si aj atóm, ktorý bude informovať vo svojom parametri, ktoré moduly treba načítať. Vybrali sme `load_module()`, kde ako parameter musí byť názov súboru spolu s príponou v úvodzovkách. Napríklad:

```
load_module("ukazka.sm").
```

5.2 Samotný program

Trieda *MLPParser*, parser pre modulárne logické programy, má funkciu *loadFromFiles*, ktorá aktivuje pomocnú triedu *ActivatorLoader*. Táto trieda robí všetko podstatné pre modularizáciu. V prvom rade spracuje každý súbor ako sme vyššie spomínali. Takto získa aktivačné programy, ktoré pošle svojej inštancii triedy *LPParser*, aby z nich spravila jeden logický program a poslala mu atómy z výsledného stabilného modelu, ktoré majú vo svojich parametroch moduly na načítanie. Programové časti týchto modulov pospája a uloží si ich v sebe ako jeden veľký logický program. Tu už môže programátor pracovať s triedou *MLPParser* ako s triedou *ModulParser*, ktorá je základná vo frameworku na tvorbu hier. UML diagram ilustruje základné prepojenia a najdôležitejšie funkcie pre implementáciu modularizácie.



Kapitola 6

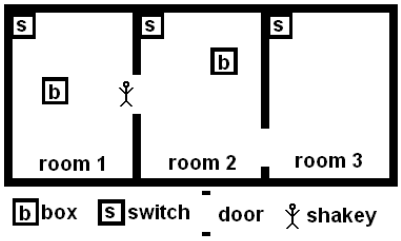
Príklad použitia modularizácie

V tejto kapitole sme sa rozhodli demonštrovať možnosti modularizácie logického programu na jednoduchej hre. Uvedomovali sme si, že skutočnú silu modularizácie využívajú až obrovské projekty a tak v malých hrách môže ísť o zbytočne silný nástroj. To nám však nezabránilo aspoň v ukážke ako môže fungovať samotné rozdelenie logického programu, či ukázať ako jednoducho a intuitívne sa dá hra napísať za pomoci paradigmy logického programu.

6.1 Zadanie

Hra, ktorú sme použili na demonštráciu sa nazýva *Shakey's world*. Je to modifikovaná verzia zo známej knihy o umelej inteligencii Artificial Intelligence: A Modern Approach[3]. Hra bola pôvodne vybraná pre úlohu plánovania.

Shakey je postava, ktorú hrá hráč. Shakeyho svet pozostáva z miestností, vypínačov svetla, dverí a krabíc. Medzi týmito objektami sa Shakey pohybuje a presúva krabice. Na začiatku je zadaný svet, kde sú umiestnené jednotlivé objekty sveta. Na nasledujúcom obrázku môžno vidieť svet, ktorý sme nakoniec použili v samotnej implementácií:



Room je miestnosť, Box krabica, Switch vypínač svetla, Door dvere. V ďal-

šom texte sa budeme odvolávať na anglické výrazy, ak budeme ukazovať časti z implementácie.

Hráč (Shakey) sa môže pohybovať od jedného objektu k druhému. Pomocou vypínačov svetla môže v miestnosti zasvietiť alebo zhasnúť, podľa toho, či je zasvietené, alebo zhasnuté v miestnosti. Pomocou dverí sa môže presúvať z jednej miestnosti do druhej. Krabice si môže zobrať a potom aj kedykoľvek položiť.

V klasickej verzii hry existuje cieľ, ktorý musí agent Shakey splniť, napríklad preniesť všetky krabice do jednej miestnosti. Naša hra nemá cieľ, lebo nejde o úlohu s plánovaním, hráč môže slobodne hrať vo svete a plniť si vlastné ciele, ak má záujem.

6.2 Implementácia

Hru sme naprogramovali v dvoch programovacích jazykoch. Na zápis pravidiel, stavu sveta a vypisovanie akcií, ktoré môže hráč vykonať v danom stave, sme použili answer set programovanie. Konkrétne implementáciu z helsinskej univerzity *smodels*[5]. Na prácu s modularizovanými logickými programami spolu s parsovaním sme použili prácu tejto bakalárskej práce: modulárne rozšírenie frameworku pre tvorbu hier[2]. Toto rozšírenie je, ako sme v predchádzajúcej kapitole vysvetlili, v jazyku *python*[8]. Tiež sme v tomto jazyku vytvorili jednoduchý interface na komunikáciu medzi užívateľom a programom.

6.2.1 Popis implementácie

Logický program je rozložený do viacerých súborov:

init.sm - pravidlá úvodných premenných stavov sveta

state.sm - pravidlá aktuálnych premenných stavov sveta, modul sa aktivuje automaticky.

world.sm - pravidlá nemenných stavov sveta, modul sa aktivuje automaticky.

rules.sm - všeobecné pravidlá, modul sa aktivuje automaticky.

goto.sm - pravidlá pohybu, modul sa aktivuje automaticky.

switchact.sm - pravidlá s manipuláciou vypínačov svetla, modul sa aktivuje, iba ak agent stojí pri vypínači svetla.

boxact.sm - pravidlá s manipuláciou s krabicami, modul sa aktivuje, iba ak agent stojí pri krabici, či drží nejakú krabicu.

Ako môžete vidieť z tabuľky, modularizácia nám umožňuje rozdeliť program na logické časti. Vďaka tomu môžeme jednotlivé pravidlá rozdeliť do súborov podľa toho k čomu slúžia. Ako v príklade vidíme, napríklad každá akcia s nejakým objektom je rozdelená do iného súboru, čím sa môže v programátorskom

tíme rozdeliť práca medzi jednotlivých programátorov. Jeden sa môže venovať pravidlám s krabicami a iný pravidlám s vypínačmi svetla. Navyše modularizácia nám umožňuje optimalizovať výpočet tým, že aktivujeme len tie pravidlá, ktoré je rozumné zapojiť do výpočtu. Ak hráč nie je pri vypínači, je nelogické sa zapodievať tým, akým spôsobom môže zapnúť, či vypnúť svetlo.

Zaujímavá a veľmi pomocná bola možnosť rozdeliť nemenné a premenné stavy sveta. Stav sveta je zadefinovaný pomocou jednoduchých pravidiel logického programu takzvaných faktov, keďže obsahujú len hlavu pravidla. V súbore `world.sm` sú nemenné stavy, napríklad určenie, kde sa nachádzajú dvere, miestnosti a vypínače spolu s celým inventárom vecí, ktoré sa nachádzajú v konkrétnej inštancii sveta. Tieto stavy nie je možné počas hrania meniť.

`state.sm` naopak obsahuje všetky premenné stavy sveta, ktoré sa môžu zmeniť, najčastejšie vplyvom hráča, napríklad umiestnenia krabíc, či informácie o tom, či sa v miestnosti svieti. Tento súbor, ako jediný, behom programu meníme a to podľa toho, ako nejaká akcia zmenila premenné stavy sveta. Na zmenu programu používame naprogramovanú parsovaciu procedúru v pythone.

6.2.2 Algoritmus implementácie

V pythone naprogramovanú časť budeme volať manipulátor. Pri spustení manipulátora sa nastaví akcia, ktorá nič nezmení, v ďalších iterácia už bude nastavovať akciu, ktorú si vybral hráč. Následne sa program pokúsi zmeniť súbor `state.sm`, aby mohol nastavil najnovšie stavy sveta, spôsobené akciou. To vykonáva pomocou pridávania, či odoberania faktov, ktoré mu poradia špeciálne pravidlá z modelu logického programu. Potom spustí modulárne rozšírenie frameworku pre tvorbu hier spolu so všetkými súbormi. Modulárne rozšírenie bude postupovať podľa popisu v predchádzajúcich kapitolách a vyhodí požadované atómy z modelu logického programu. Požadované atómy sú akcie, ktoré môže Shakey vykonať a atómy, ktoré informujú ako sa zmení stav po vykonaní akcie. Nakoniec manipulátor vypíše na konzolu zoznam akcií, ktoré sa môžu vykonať, spolu s možnosťou opustiť hru. Užívateľ si vyberie číslo jeho preferovanej akcie a algoritmus iteruje nový cyklus.

6.2.3 Ukážka pravidiel logického programu

V nasledujúcich ukážkach ukážeme, že pravidlá zapísané v logickom jazyku sú naozaj intuitívne. Napríklad stavy sveta:

» at(door1,room1).

» light(switch2,room2).

» nearby(shakey,door2).

Takýto kód je ľahko pochopiteľný a vďaka tomu sa po dlhom čase nie je problém vrátiť ku kódu a modifikovať ho. Takisto všeobecné pravidlá sú jednoduché:

» thing(X) :- box(X).

» nolight(S,X) :- not light(S,X), room(X), switch(S).

Aktivovanie modulov je veľmi ľahko implementované, vďaka tomu, že sú to tiež pravidlá logického programu:

» load_module("switchact.sm") :- nearby(X,Y), agent(X), switch(Y).

» load_module("boxact.sm") :- agent(A), hold(A,X), box(X).

Nakoniec ešte príklad pravidla pre atóm, ktorý určuje, ktoré akcie môžeme vykonať:

» action(pickup,X) :- nearby(A,X), agent(A), box(X).

» action(goto,X) :- agent(A), box(X), nonearby(A,X), nearby(A,Y), at(Y,R), at(X,R), light(S,R).

6.3 Ukážka z hry

Hra bola pripravená ako ukážka modularizácie logického programu a preto sme sa veľmi nevenovali grafickej stránke hry. Preto sú ukážkami len výstupy z konzoly. Hra na začiatku, keď sme ešte žiadnu akciu nevykonali:

```
============================================================
doing action: do_nothing()
============================================================
at(box1,room1). at(box2,room2). light(switch1,room1). nearby(shakey,door1).

possible actions:
  0: quit()
  1: goto(door2)
  2: goto(switch1)
  3: goto(switch2)
  4: goto(box1)
```

Ak hráč vyberie možnosť 4 ísť ku krabici1, dostaneme takýto výstup:

```
============================================================
doing action: goto(box1)
============================================================
at(box1,room1).at(box2,room2).light(switch1,room1).nearby(shakey,box1).

possible actions:
  0: quit()
  1: goto(door1)
  2: goto(switch1)
  3: pickup(box1)
```

Kapitola 7

Záver

Ako sme v úvode predpokladali paradigma answer set programovania je zaujímavou alternatívou k riešeniu programátorských úloh. Jasne nám to demonštroval aj náš príklad implementácie Shakey World zo šiestej kapitole. Pri spolupráci s answer set programovaním nám dobre poslúžil framework na tvorbu počítačových hier, rozšírený o nami implementavanú modularizáciu, opisanú v piatej kapitole. Jej myšlienku v spôsobe rozdelenia logického programu na rôzne moduly a ich následné spracovanie sme objasňovali v štvrtej kapitole. V tretej kapitole venovanej oblasti počítačových hier sme naznačili využitie, práve nami navrhnutej modularizácie a logických programov. Jazyk a sémantiku logických jazykov, spolu s ich základnými triedami, sme definovali v druhej kapitole. Náš cieľ, priblížiť disciplínu logických programov z akademickej oblasti bližšie do praxe za pomoci rozšírenia nástrojov na vývoj, sme sa pokúsili naplniť pomocou implementácie modularizácie. Pri jej návrhu sme poukázali na výhody, ktoré môže programátorovi poskytnúť. Ako napríklad rozdelenie programu do viacerých logických častí a vďaka tomu väčšia prehľadnosť kódu, či hľadanie stabilného modelu v programe zloženom len z potrebných modulov a tak zrýchlenie výpočtu. Tieto výhody predčili potrebné prevádzkové náklady implementácie modularizácie. Je na programátoroch, akú paradigmu, či nástroj použijú k riešeniu svojich úloh. My veríme, že sme im predložili naozaj zaujímavú ponuku.

7.1 Poďakovanie

Na záver sa chcem poďakovať školiteľovi Mgr.Jozefovi Šiškovi, za možnosť spracovania jeho myšlienky modularizácie logických programov aj za pomoc pri jej implementácií.

Literatúra

[1] Šefránek, J.: Inteligencia ako výpočet

[2] Jozef Šiška. Dynamic Logic Programming and world state evaluation in computer games, In Procs. of WLP06, 2006

[3] Russell, Norvig: Artificial Intelligence, A Modern Approach

[4] http://www.tiobe.com/tpci.htm

[5] http://www.tcs.hut.fi/Software/smodels/

[6] http://www.dbai.tuwien.ac.at/proj/dlv/

[7] http://www.krlab.cs.ttu.edu/mkatoms/

[8] http://www.python.org/