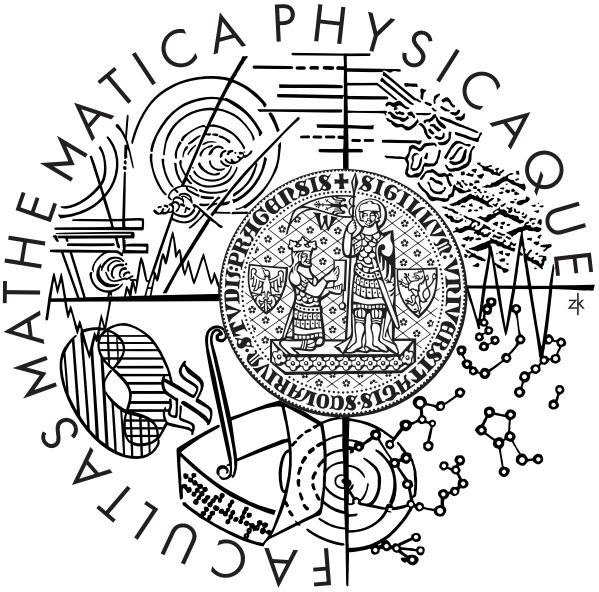
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jan Krajíček

Framework pro implementaci botů pro hru NetHack

Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot
Studijní program: Softwarové inženýrství
Studijní obor: Softwarové systémy

Praha 2015

Rád bych poděkoval svému vedoucímu Jakubovi Gemrotovi a skupině AMIS za podporu a možnost pracovat na zajímavém tématu.

Prohlašuji, že jsem tuto diplomovou 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 Praze dne Jan Krajíček

Název práce: Framework pro implementaci botů pro hru NetHack

Autor: Jan Krajíček

Katedra: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot

Abstrakt: Dosavadní pokusy o implementaci botů pro klasickou roguelike hru NetHack narážely na mnohá úskalí spojená s její rozsáhlostí, obtížností a konzolovým rozhraním. Framework implementovaný v této práci řeší problémy s napojením ke hře a zprostředkovává k ní programátorsky přívětivé rozhraní v jazycích Java a Clojure. Poskytuje možnost programovat pokročilé NetHack boty s využitím komplexního modelu herního světa, knihovny možných akcí a mnoha pomocných funkcí. Framework používá prvků funkcionálního a logického programování a nevyžaduje modifikace vlastního kódu hry. Popsána je také implementace ukázkového bota, který jako první bot vůbec dokáže hru dokončit.

Klíčová slova: agent, framework, NetHack

Title: NetHack Bot Framework

Author: Jan Krajíček

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot

Abstract: Previous attempts at implementing bots for the classic roguelike game NetHack have been hindered by many problems related to its complexity and console-based interface. The framework implemented as part of this work solves the problem of interfacing with the game and provides a programmer-friendly API for the Java and Clojure programming languages. It enables programming sophisticated bots using the provided model of the game world, a library of possible actions and utilities for various aspects of the game. The framework uses elements of functional and logic programming and doesn't require modifications of the game. Also described is an implementation of the first NetHack bot capable of winning the game.

Keywords: agent, framework, NetHack

Obsah

Kapitola 1

Úvod

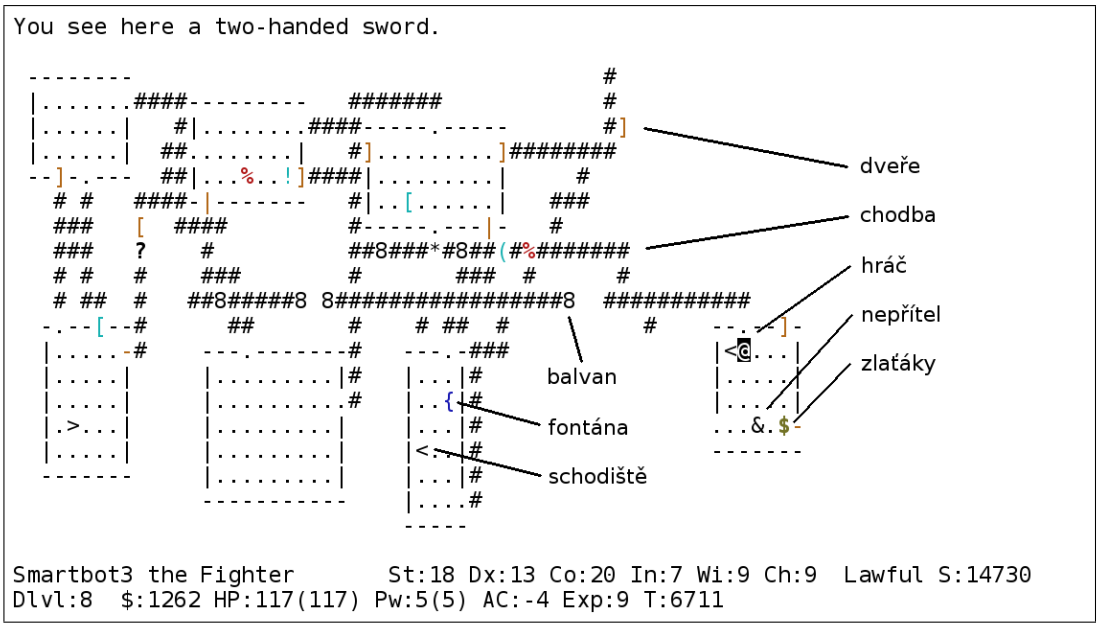
1.1 NetHack a žánr roguelike her

NetHack je klasická počítačová hra pro jednoho hráče, zastupující žánr tzv. roguelike her. Tento žánr je pojmenovaný podle hry *Rogue* (1980), která byla jejím prvním představitelem.

Toto jsou některé typické charakteristiky roguelike her, které se zároveň objevují ve hře NetHack:

- Procedurálně generovaný obsah, díky kterému je podoba každé hry částečně náhodná,
- Absence možnosti hru průběžně ukládat a obnovovat, což znamená pro každou hru pouze jeden pokus,
- Hra na tahy, hráč může o tazích přemýšlet libovolně dlouhou dobu,
- Textové rozhraní s dlaždicovou mapou vykreslenou v terminálu, ve které jsou předměty i postavy reprezentovány ASCII symboly (viz obrázek 1.1),
- Prvky role-playing her ve stylu *Dungeons&Dragons* jako průběžný vývoj vlastností hrdiny, důraz na boj s nepřáteli, správu inventáře a efektivní využívání předmětů, plnění úkolů apod.,
- Mechanika hladu – hráč je zejm. v počátcích hry nucen k neustálému postupu omezeným množstvím jídla, které se ve hře vyskytuje.

Hlavním úkolem hráče NetHacku je projít herní dungeon (mnohopatrovou větvící se jeskyni, ve které se hra odehrává) až na samotné dno, přičemž musí navštívit také 3 vedlejší větve a získat v nich předměty potřebné pro přístup do nejnižší úrovně hlavní větve. V té musí hráč vybojovat cílový předmět *the Amulet of Yendor* a s ním se vrátit zpět na začátek dungeonu. Odsud je hráč nenávratně přenesen do poslední fáze hry



Obrázek 1.1: Snímek herní obrazovky NetHacku s vysvětlivkami

sestávající z pěti speciálních úrovní, kde v poslední z nich musí získaný amulet obětovat svému božstvu.

První verze NetHacku vyšla v roce 1987 a hra je stále ve vývoji. Zdrojové kódy v jazyce C jsou veřejně dostupné a hru lze volně provozovat prakticky na všech moderních platformách. Existuje také řada upravených variant NetHacku, které poskytují grafické nadstavby či hru rozšiřují o další obsah, nebo jen opravují chyby.

1.2 Boti hrající roguelike hry

Roguelike hry od svého počátku inspirují programátory k pokusům o tvorbu programů – botů, kteří by dokázali hrát bez lidského zásahu.

Prvním významným představitelem je Rogomatic [1] (1984), který dokázal dokončit hru Rogue. Dalšími příklady klasických roguelike her, pro které existují úspěšní boti jsou *Angband* (1990) a *Dungeon Crawl Stone Soup* (2006, jde o moderní variantu *Linley's Dungeon Crawl* z roku 1997). Boti pro Angband byli implementováni jako modifikace samotné hry, Dungeon Crawl Stone Soup pak přímo poskytuje skriptovací rozhraní, kterého boti využívají.

V případě NetHacku existuje řada pokusů o implementaci botů a frameworků pro boty, z nichž se nejdále dostali TAEB [2] (implementovaný v jazyce Perl) a Saiph (v jazyce C++) [3]. Dosud se ale přes nemalé úsilí žádnému z nich nepodařilo hru úspěšně dokončit, ba ani překonat její první polovinu.

Z často kladených otázek *Usenet* skupiny rec.games.roguelike.nethack [4] (2008):

```
4.9Q:   Has anyone ever written a program to play NetHack
        automatically (a NetHack "bot")?

4.9A

Not successfully.  People have written "bots", or programs that will
play the game by themselves, for some other roguelikes. These bots
usually play pretty successfully.

However, no one has yet written a well-functioning one for NetHack,
and the commonly held opinion is that NetHack is too complex for a
"bot" that functions well to be currently practicable.
```

1.3 Cíle práce

Cílem práce je návrh a implementace frameworku pro NetHack boty, který programátora odstíní od nutnosti řešit podrobnosti napojení ke hře, interpretace obrazovky terminálu a některých nízkoúrovňových příkazů NetHacku. Framework poskytne programátorsky přívětivé rozhraní ke hře, knihovnu možných akcí k provedení a model herního světa zahrnující reprezentaci map úrovní, předmětů, nepřátel a jejich vlastností.

Framework nesmí využívat vlastních modifikací kódu hry pro usnadnění napojení ke hře nebo jejího ovlivnění, bude využívat stejného rozhraní jako lidský hráč (tj. emulátoru terminálu) a umožňovat i hru po síti na veřejných NetHack serverech s podporou protokolů Telnet nebo SSH.

Dalším cílem je implementace pokročilého ukázkového bota postaveného na tomto frameworku, který by dokázal hru hrát na pokročilé úrovni a s větším úspěchem, než jakého dosáhli předchůdci.

1.4 Struktura práce

Ve druhé kapitole jsou rozebrány hlavní problémy, na které implementace NetHack botů typicky naráží. Třetí kapitola popisuje základní přístupy návrhu implementovaného frameworku, jeho architekturu a klíčová rozhodnutí. Podrobnosti implementace dílčích funkcí frameworku a popis způsobů, jak řešení adresuje problémy z druhé kapitoly jsou v kapitole čtvrté. V páté kapitole je pak popsána implementace ukázkového bota, který frameworku využívá. Vlastnosti výsledného řešení a srovnání s výsledky předchozích prací se nachází v šesté kapitole. V závěru jsou diskutovány dosažené výsledky a navrženy možná budoucí rozšíření. Základní dokumentaci a návody lze nalézt v příloze.

Kapitola 2

Analýza problému

V této kapitole jsou popsány hlavní problémy, se kterými se implementace botů pro NetHack musejí vypořádat a které dále popsaný framework adresuje.

2.1 Problém velikosti herní domény

Z hlediska virtuálního agenta přestavuje svět NetHacku poměrně komplexní prostředí. NetHack se odehrává na až 80 úrovních o rozloze 80×21 dlaždic. Rozložení úrovní dungeonu je stromové – hlavní větev se na několika místech rozvětvuje (samotné podvětve už ale nikoli). Úrovně jsou z velké míry procedurálně generované a obsahují náhodně rozmístěné pasti a jiné prvky jako skryté dveře a chodby, obchody, chrámy a oltáře, vodní a ledové plochy, tekutou lávu apod. Všechny tyto prvky poskytují vlastní specifické možnosti interakce.

Svět NetHacku obývá přibližně 350 druhů převážně nepřátelských stvoření, která se v dungeonu náhodně zjevují, nebo jsou pevně vytvořená ve speciálních úrovních. Každý druh nepřítele má svá specifika a může hráče ohrozit několika způsoby: fyzickým útokem, degradací nebo krádeží výbavy, znemožněním pohybu a kouzly s různými účinky. Hráč je nucen přizpůsobit svou strategii aktuálním hrozbám a dostupné výbavě.

NetHack obsahuje přes 1000 typů předmětů spadajících do třinácti kategorií jako jsou zbraně, zbroj (s podkategoriemi), jídlo, prsteny, svitky, hůlky a další kouzelné předměty. Kapacita inventáře je omezená a při volbě výbavy se nelze vyhnout nepříjemným kompromisům.

Hráč má k dispozici zhruba 50 různých akčních příkazů, které lze v každém tahu použít. Příkazy umožňují tyto hlavní druhy akcí:

- Pohyb po úrovni a mezi úrovněmi, prohledávání okolí, ovládání dveří a dalších prvků úrovní,
- Útoky chladnými nebo vrženými zbraněmi, střelbou nebo kouzly,

- Zobrazení dodatečných informací o předmětech, nepřátelích a prvcích úrovně,
- Správu inventáře – nasaď/sundej zbroj, seber/zahoď předmět, použij předmět (různými způsoby).

Z hlediska umělé inteligence je problematické, že některé příkazy a použití různých předmětů může mít opožděný, případně dočasný efekt a výsledek akcí se může zásadně lišit v závislosti na konkrétní situaci.

2.2 Problémy nejednoznačností rozhraní

Snahy o implementaci botů pro NetHack bez modifikací kódu hry značně komplikuje fakt, že rozhraní NetHacku je přizpůsobeno lidským hráčům a mnoho jeho vlastností je pro strojové zpracování problematické.

Velká část potíží je spojená s nejednoznačnostmi tohoto rozhraní. Pro zobrazení herního světa využívá NetHack pouze barevných ASCII symbolů, které v mnoha případech nestačí k jednoznačné identifikaci zobrazeného prvku. Například znak mezery může na mapě reprezentovat neprůchozí kus skály, ale také průchozí temnou část místnosti, která je mimo dohled, případně i nepřítele – ducha (o průchodnosti nebo neprůchodnosti pole pak nelze usuzovat nic).

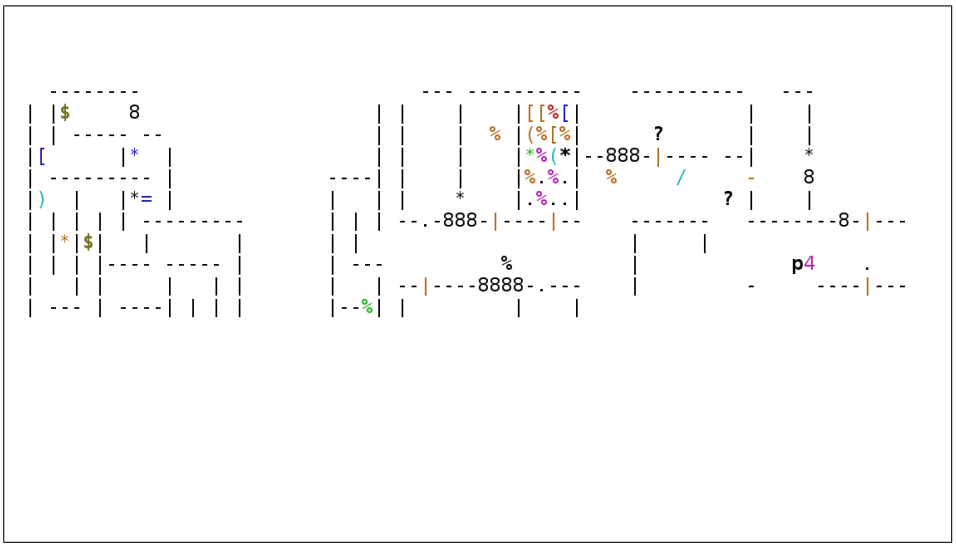
Krom nejednoznačnosti jednotlivých symbolů také nelze vždy určit, v jaké větvi dungeonu se hráč aktuálně nachází. Po sestoupení po schodech nemusí být zřejmé, zda hráč vstoupil do některé vedlejší větve, nebo pokračuje v hlavní větvi, ačkoli jde o podstatnou informaci.

Lidský hráč si dokáže mnoho podobných informací odvodit z kontextu hry, vzhledu okolí apod. a případný dočasný omyl pro něj není problematický. V případě umělé inteligence ale může omyl snadno vést ke špatným rozhodnutím nebo k zacyklení v „neřešitelné“ situaci.

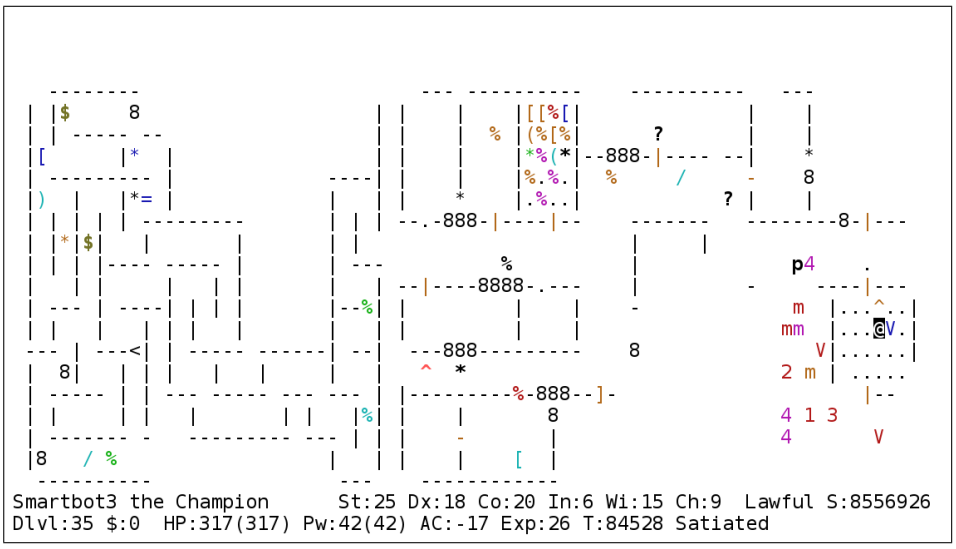
2.3 Problém synchronizace

Problém synchronizace rozhodování a akcí bota s herními tahy NetHacku a překreslováním obrazovky je prvním poměrně obtížně řešitelným problémem, na který programátor bota narazí a jehož řešení má zásadní vliv na funkčnost implementace.

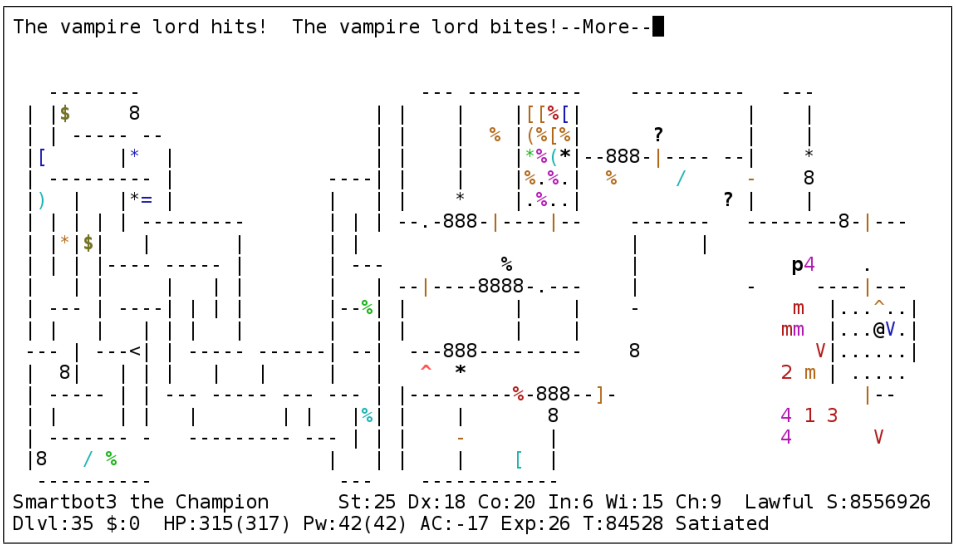
Podstata problému spočívá v tom, že není obecně snadné určit moment, kdy je obrazovka terminálu zobrazující aktuální herní stav plně aktualizovaná a hra očekává vstup od hráče. Komunikace procesu NetHacku s terminálem neprobíhá na aplikační vrstvě výměnou diskrétních paketů, ale jde v principu o proudovou komunikaci bez jasně rozlišitelných hranic mezi jednotlivými tahy. Obzvlášť problematická je situace v případě hry po síti, kde navíc přichází do hry nepředvídatelná latence a fragmentace



Obrázek 2.1: Mezistav herní obrazovky během překreslování po volbě akce



Obrázek 2.2: Další mezistav herní obrazovky, zdánlivě finální



Obrázek 2.3: Finální stav herní obrazovky po dovykreslení událostí

přenášených dat na síťové vrstvě. Každý bot nicméně potřebuje vědět, kdy může zahájit výpočet následující akce s tím, že má k dispozici všechny informace platné ke konci předchozího herního tahu.

V lidském měřítku se obrazovka překresluje poměrně rychle a síťové zpoždění člověk dokáže dobře předvídat, přičemž má také představu o tom, co na obrazovce očekávat. Z pohledu bota je problém složitější a případný omyl jej může dostat do neočekávaných herních kontextů (např. vstoupit do některého herního menu, které bot normálně nevyužívá) nebo mít za důsledek špatnou interpretaci herní situace, typicky bez šance na zotavení.

Obrázky 2.1 až 2.3 demonstrují příklad postupného vykreslování obrazovky NetHacku v rámci jednoho tahu (po akci hráče). Prostřední mezistav na obrázku 2.2 vypadá jako plně vykreslený a mohl by zmást bota k předčasné volbě akce. Jak se ale ukáže na obrázku 2.3, na vykreslení ještě čekaly další herní události.

Toto jsou dříve navržené způsoby řešení, kterých (krom posledního) využívali existující boti a které zároveň nepředpokládají běh hry na některé konkrétní platformě:

- *sleep(1);* neboli čekání konstantní dobu po každém tahu. Tato metoda je poměrně nespolehlivá (zejm. při hře po síti) a vede ke značnému zpomalení běhu bota, které je úměrné délce čekání – „bezpečnější“ hodnoty zpomalují o to více.

- Ping pomocí vedlejšího kanálu (paralelní proces). Takto se dá přesněji odhadnout aktuální síťové zpoždění a doba čekání přizpůsobit. Problémy předchozí metody ale v menší míře stále přetrvávají.

- Ping v rámci Telnet spojení. S touto metodou přišel dříve zmíněný framework TAEB. Ve specifickém případu hry přes protokol Telnet lze „zneužít“ protokolu pro vynucení potvrzení přijetí příkazů bota. Bot po odeslání své akce odešle neplatný příkaz Telnetu a před volbou další akce vyčkává reakce na tento příkaz. Lze očekávat, že nejprve dorazí všechna data o překreslení herní obrazovky v důsledku provedené akce a teprve potom odmítavá reakce na později zaslaný neplatný příkaz. Tato metoda je limitovaná na protokol Telnet (varianta pro SSH by byla implementačně náročnější) a spolehlivost stále není stoprocentní, v okrajových případech může server odeslat odmítnutí zmíněného neplatného Telnet příkazu dříve, než vypočte a odešle výsledky dříve přijaté akce.

- Odhad dle vzhledu obrazovky. Tuto metodu někteří boti kombinovali s předchozími. Bot může usoudit, že obrazovka je vykreslená na základě toho, že jsou plně vykresleny spodní dva stavové řádky, kurzor se nachází na pozici hráče (která ale není vždy jednoznačně určitelná) apod. Jak ilustruje obrázek 2.2, tato metoda trpí možností falešných pozitiv.

- Úprava zdrojových kódů hry. NetHack může pro potřeby botů nebo nadstavbových grafických rozhraní zasílat v rámci datového toku speciální sekvenci znaků značící konec vykreslování každého tahu. Existující patch s názvem *vt_tiledata* nabízí právě tuto funkčnost, přičemž na mnoha veřejných serverech je k dispozici. Spoléhání na úpravy kódu je ale v rozporu s jedním z bodů uvedených cílů této práce.

- Ping v rámci samotného NetHacku. Pokud by v NetHacku existoval příkaz platný v každém herním kontextu, který by měl jednoznačně rozpoznatelný výsledek a neměl přitom vliv na stav hry, dal by se využít pro spolehlivou synchronizaci vykreslování a rozhodování. Takový příkaz v NetHacku bohužel neexistuje. Rozšířenou variantu této myšlenky nicméně využívá zde popsaný framework, což je podrobněji popsáno v části 4.1.

Jak je z popisu patrné, žádná z dosud využitých metod není ideálním řešením synchronizačního problému.

2.4 Problémy spojené s náhodnými jevy

NetHack je hra, ve které zejména ze začátku velmi záleží na náhodě: hráč je vydán na milost a nemilost náhodnému generátoru čísel. Obecně nic nebrání např. situaci, kdy první krok hráče na začátku hry vede do skryté smrtící pasti.

Jak bylo uvedeno, velká většina herních úrovní je generovaná náhodně, hráč tedy nemůže přenášet znalosti o jejich rozložení z jedné hry do druhé.

Protože i předměty jsou až na výjimky generovány náhodně, hráč se nemůže spolehnout, že bude mít k dispozici předměty, které v pozdějších fázích hry potřebuje pro navigaci nebo pro ubránění se silnějším nepřátelům. Mnohdy je třeba neobvyklým způsobem improvizovat s dostupnou výbavou a znát triky, jak obstarat kriticky nezbytné předměty, kterých je v dané hře nedostatek.

Specifikem NetHacku je také náhodné párování mezi vzhledem předmětů a jejich skutečnou identitou. Například lahvička označená jako *„a fizzy potion“* může v jedné hře přestavovat léčivý lektvar a v další způsobovat třeba oslepnutí. Většina předmětů jako kouzelné hůlky, prsteny, amulety, svitky atd. má v každém běhu hry jinou podobu a dokud tato není jednoznačně identifikovaná (herně např. pomocí svitku identifikace nebo mimoherně pomocí triků jako identifikace podle ceny v obchodu), hráč si nemůže být jistý, které předměty jsou užitečné a které škodlivé.

Náhoda má samozřejmě vliv také na výskyt a chování nepřátel, účinky útoků a kouzelných předmětů apod.

2.5 Problémy navigace

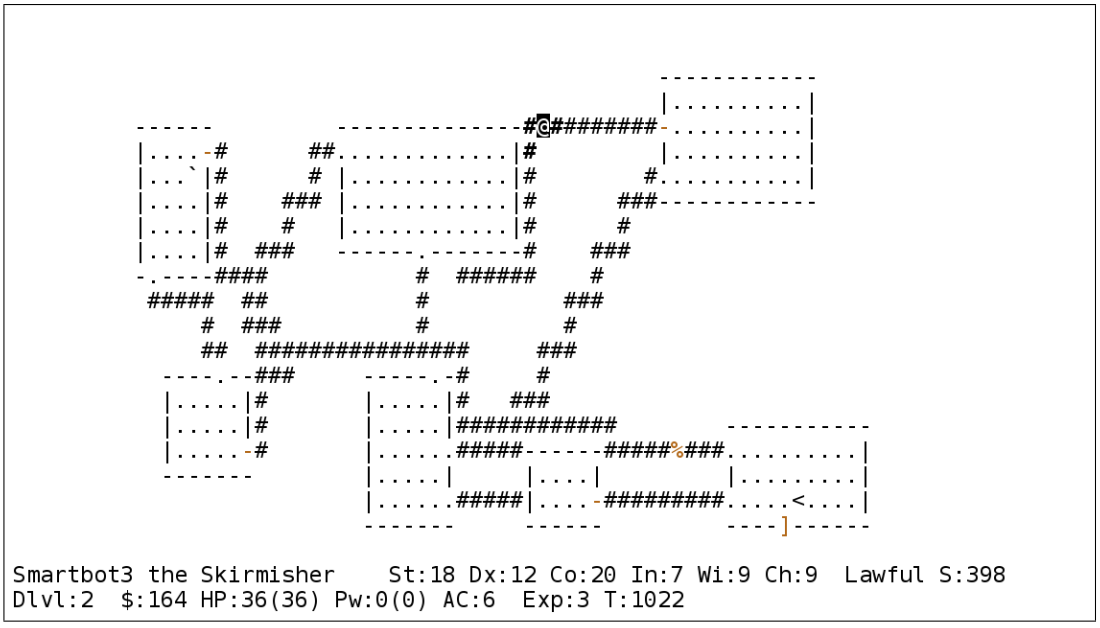
Žádný bot se nedostane velmi daleko bez schopnosti efektivního prozkoumávání a navigace herním prostředím. Špatně implementovaná navigace vede k častým zásekům nebo zacyklení bota ve složitých situacích a zhacení jinak nadějných běhů. Navigace ve světě NetHacku je pro boty obtížná z několika důvodů.

Možnosti navigace jsou výrazně ovlivněné aktuální výbavou hráče. Průchodnost nebo neprůchodnost polí (uzlů i hran navigačního grafu) nelze definovat obecně bez přihlédnutí k předmětům, které hráč může a je ochoten v dané situaci použít.

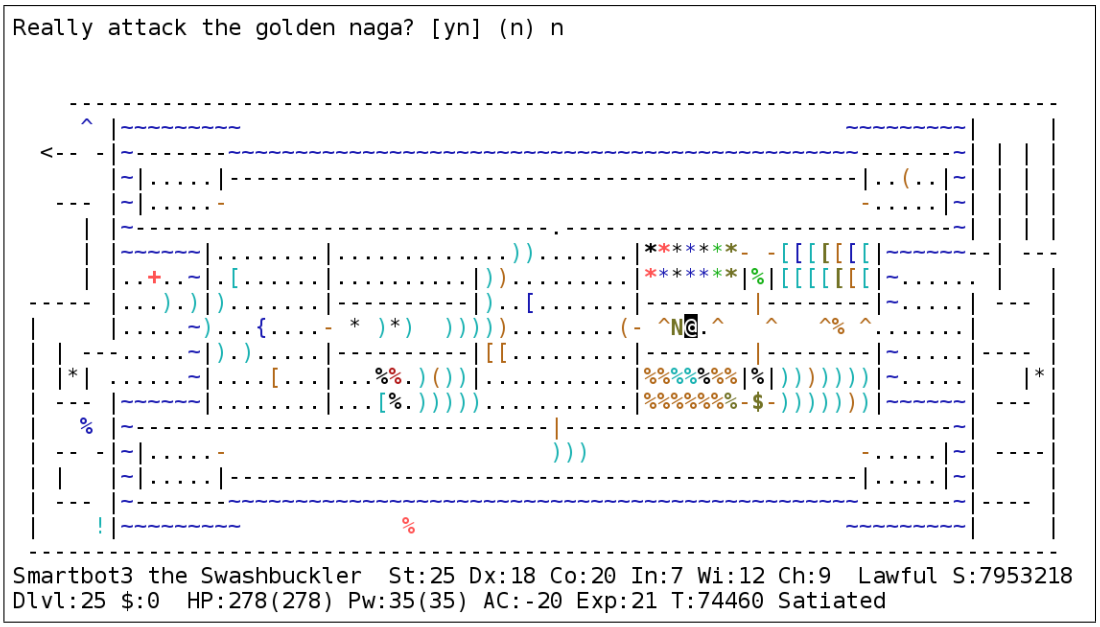
Náhodně vygenerované úrovně NetHacku s jejich osazenstvem jsou náchylné ke vzniku mnoha různých okrajových situací. V některých případech může mít i lidský hráč potíž s hledáním možné cesty. Přizpůsobení chování navigace pro určitou situaci může snadno vést ke zhoršení v jiných okrajových situacích, ačkoli to nemusí být ihned zřejmé.

Například dveře jsou v NetHacku průchozí pouze v přímých směrech a ne diagonálně. Pokud jsou dveře zamčeny nebo v přímém směru zablokovány, je možné je zničit kopnutím a zprůchodnit ve všech směrech, což si ale nelze dovolit v případě obchodů, kde zničení dveří vede k téměř jisté smrti rukou majitele daného obchodu.

V herních úrovních je často nutné aktivně vyhledávat skryté dveře a chodby, bez jejichž nalezení nemusí být úroveň průchozí. Zkušený lidský hráč může být schopen odhadnout, kde se skryté chodby nachází, obecný efektivní algoritmus pro prohledávání ale není přímočaré sestrojit.



Obrázek 2.4: Úroveň se zdá být prozkoumaná, ale chybějící schodiště do následující úrovně naznačuje přítomnost další, dosud neobjevené místnosti



Obrázek 2.5: Přátelské stvoření uvězněné pastí blokuje cestu ke schodišti

V úrovních se také mohou vyskytovat balvany, které za běžných podmínek nejsou průchozí, ale je možné je odvalit z cesty, pokud je v daném směru volná. Stejně tak je ale možné nevhodně odvaleným balvanem cesty zablokovat.

Navigace je komplikována náhodně zjevenými nepřátelskými i přátelskými stvořeními, která mohou zablokovat jinak průchozí cesty (na jednom herním poli může běžně stát jen jedno stvoření, i pokud jde jen o mravence). Přátelská stvoření jsou v tomto ohledu horší než nepřátelská – obecně není žádoucí na ně útočit, protože mnohá z nich jsou až do pozdních fází hry pro hráče neporazitelná a útok na některé klíčové herní postavy může znamenat i nevratné znemožnění úspěšného dokončení hry. Přesto se při hře běžně stává, že se přátelské stvoření (v tomto případě obvykle ne neporazitelné) ocitne na místě, kde blokuje jedinou dostupnou cestu a ze kterého se nikdy dobrovolně nepohne kvůli přítomnosti pastí v okolí, jak ilustruje obrázek 2.5.

Implementace navigace se obecně obtížně ladí a testuje, protože mnoho problematických okrajových situací nemusí v rámci jedné hry nastat. Ověření funkčnosti je možné prakticky pouze dlouhodobým reálným během s průchodem všemi speciálními úrovněmi a s různou dostupnou výbavou.

Kapitola 3

Základní rozhodnutí

V této kapitole jsou popsány prvotní volby provedené před zahájením implementace a obecný přístup navrženého řešení.

3.1 Volba technologií

Implementovaný framework je zacílen na platformu Java, na které dosud žádný framework ani pokročilý ad-hoc bot pro NetHack nebyl vytvořen. Tato platforma poskytuje vyspělé robustní prostředí s adekvátním výkonem a kvalitní volně dostupné nástroje pro vývoj, provoz a monitoring.

Framework nicméně není implementován v jazyce Java, ale v dynamičtějším programovacím jazyce Clojure, který platformy Java využívá. Jde o moderní variantu jazyka Lisp, která se překládá do JVM bajtkódu a poskytuje silné možnosti interoperability s Java kódem.

Použití jazyka Clojure umožňuje přímo aplikovat prvky funkcionálního a logického programování, díky kterým lze minimalizovat dopady rozsáhlosti domény na složitost kódu programu a z velké míry se vyhnout implementační složitosti, která není vlastní řešenému problému ve smyslu např. [5].

V duchu dědictví Lispu jazyk Clojure umožňuje také interaktivní vývoj a ladění programu za běhu s možností načtení a okamžité aplikace nového nebo upraveného kódu a silné možnosti metaprogramování (makra) pro zjednodušení „šablonovitého“ kódu.

Pro emulaci terminálu a komunikaci prostřednictvím protokolů SSH a Telnet je použita Java knihovna *Telnet/SSH/Terminal*.

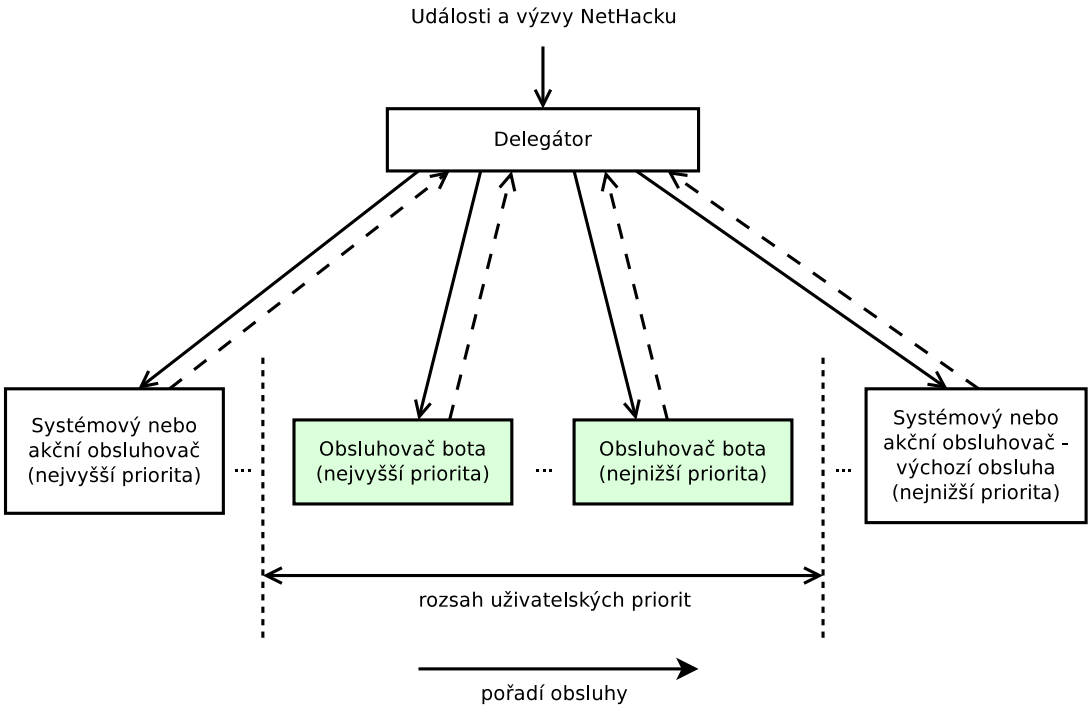
3.2 Architektura frameworku

Framework je implementován jako systém řízený událostmi. Centrální komponentou je „delegátor“ událostí, který registruje obsluhovače (*handlers*) a podle priority zadané při registraci jim události sériově předává. Obsluhovače lze za běhu přidávat nebo vyřazovat, případně měnit jejich priority. Každý typ události je reprezentován Java rozhraním a obsluhovače jsou třídy, které vybraná rozhraní implementují. Uživatelské obsluhovače bota mají oproti systému k dispozici pouze omezený rozsah povolených priorit. Obsluha událostí je zjednodušeně nakreslena na obrázku 3.1. Širší pohled na komunikaci komponent systému je znázorněn na obrázku 3.2.

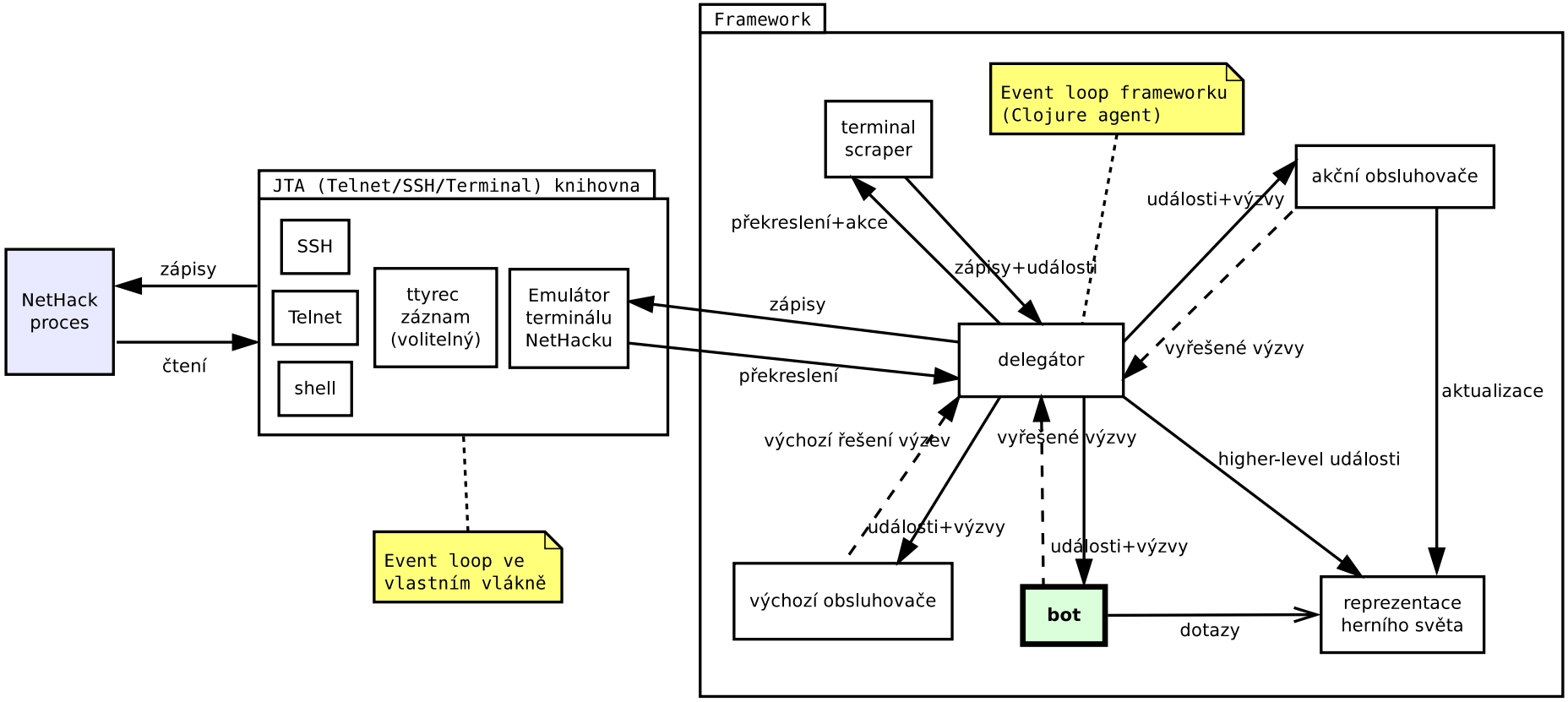
Systém rozlišuje dva druhy událostí: *standardní události (events)* a *výzvy (prompts)*. Delegátor s nimi zachází různým způsobem.

Standardní událost reprezentuje sledovaný jev, který v herním světě nebo v systému nastal. Tyto události jsou předávány všem registrovaným obsluhovačům, které na ni mohou nebo nemusí reagovat. Delegátor v tomto případě neočekává žádnou odpověď. Příkladem je událost překreslení obrazovky virtuálního terminálu, zobrazení informační zprávy NetHacku (*„Zde se nachází předmět XY“*) nebo přechod na jinou úroveň dungeonu.

Výzva reprezentuje jev, který nastává ve chvíli, kdy NetHack očekává reakci hráče. Tyto jsou předávány obsluhovačům pouze než první z nich na výzvu zareaguje (vrátí ne-*null* hodnotu). Zbylé obsluhovače pak volány nejsou. Typ reakce závisí na typu výzvy, může to být např. *string* hodnota, souřadnice na mapě, v případě dotazů typu



Obrázek 3.1: Schéma zacházení s událostmi a výzvami



Obrázek 3.2: Přehledové schéma komunikace komponent frameworku

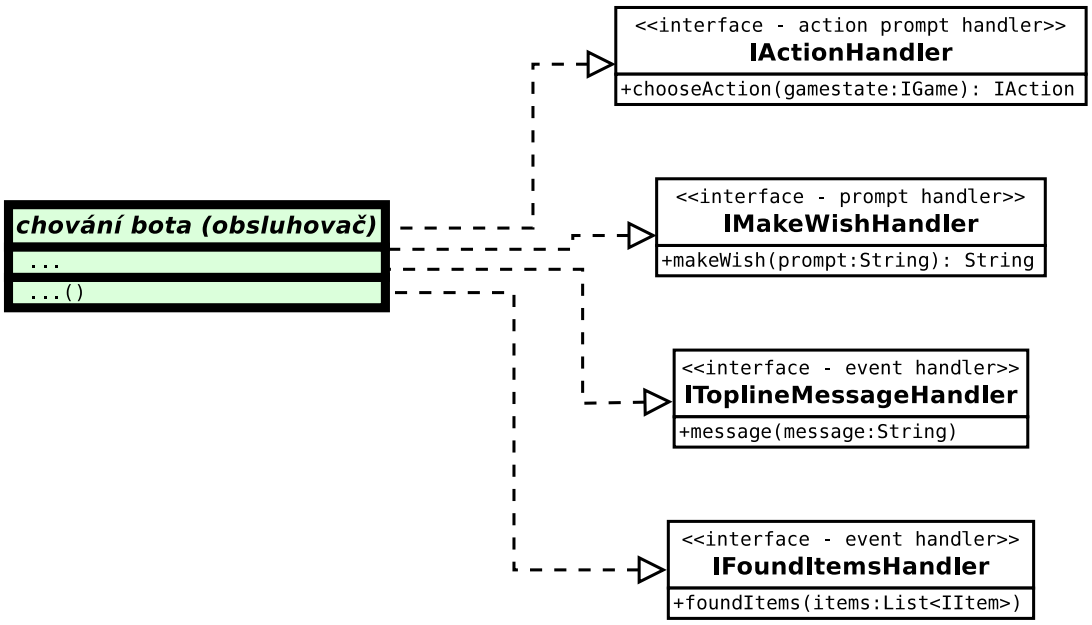
ano/ne pak *boolean*. Základním typem výzvy je výběr akce pro následující tah. Tuto výzvu typicky obsluhuje implementace bota a vrací akci k provedení. Dalšími příklady jsou dotazy NetHacku typu „Kterým směrem chceš vystřelit z luku?“ nebo „Jak chceš pojmenovat tento předmět?“

Pojem *událost* je v dalších kapitolách textu používán pouze ve významu *standardní události* a nezahrnuje události typu *výzva*.

Popsaný model obsluhy událostí umožňuje uživatelským botům i systému dekompozici chování na množinu obsluhovačů s přiřazenými prioritami, které se zabývají pro ně relevantními jevy. Větší rozsah povolených priorit systémových obsluhovačů umožňuje poskytnutí rozumných výchozích reakcí na méně obvyklé nebo méně podstatné výzvy, kterými se pak autor bota nemusí zabývat. Systém má zároveň příležitost přednostně obsloužit události herního světa a aktualizovat jeho model, nebo v akčních obsluhovačích automaticky vyřešit obvyklé herní výzvy spojené s aktuálně prováděnou akcí.

Praktickým důsledkem použití funkcionálního programovacího paradigmatu, ke kterému navádí jazyk Clojure je využívání neměnných (*immutable*) datových struktur a oddělení „čistého“ kódu bez vedlejších efektů, jímž je implementovaná velká část doménové logiky a „nečistého“ kódu, který se objevuje v obsluhovačích, kde realizuje interakce s hrou a spravuje části programu, které se neobejdou bez proměnného stavu. Všechna práce s proměnným vnitřním stavem je realizována explicitními konstrukcemi, které Clojure pro tento účel poskytuje (*atom*, *ref*, *agent*).

API pro boty v jazyce Java je realizováno jako tenká fasáda nad funkcemi implementovanými v Clojure, které jsou obohaceny o typové informace.



Obrázek 3.3: Příklad chování bota, které obsluhuje některé události a výzvy

Kapitola 4

Implementace frameworku

Tato kapitola popisuje významné dílčí prvky implementace frameworku a způsoby řešení problémů uvedených ve druhé kapitole.

4.1 Napojení ke hře

Základním cílem implementovaného frameworku je odstínit programátora od nutnosti zabývat se napojením ke hře. Žádoucí je možnost spuštění hry jako lokálního procesu, ale také hra na vzdáleném serveru pro prověření funkčnosti bota na „neutrální půdě“, kde běží dobře známá verze NetHacku a hru nelze nekalým způsobem ovlivňovat.

Ve frameworku jsou implementovány tři možnosti napojení. NetHack lze buď spustit lokálně, nebo je možno se připojit ke vzdálenému serveru pomocí protokolů Telnet nebo SSH a použít vlastního nebo předpřipraveného „menu bota“ k obsloužení menu serveru, tj. přihlášení uživatele a spuštění hry. Z pohledu programátora bota jde jen o konfiguraci.

V systému je implementován plugin pro knihovnu *Telnet/SSH/Terminal*, který s pomocí tříd této knihovny implementuje emulátor terminálu, tj. virtuální reprezentaci obrazovky NetHacku, které je možno se dotazovat na pozici kurzoru nebo v terminálu vykreslený text a jeho barvu.

Tento plugin generuje událost pro každé překreslení některé části terminálu, včetně změn pozice kurzoru. Tyto události zpracovává komponenta *scraper*, jejíž úlohou je interpretovat základní charakteristiky, které se dají z obrazovky vyčíst (zda je na obrazovce vykresleno menu nebo jiný prvek, který je třeba obsloužit a jejich obsah) a podle toho vyvolávat patřičné odvozené události a výzvy, případně na ně přímo reagovat jediným možným způsobem, např. potvrzením informační zprávy klávesou *Enter*.

Tři klíčové události rozpoznávané z herní obrazovky komponentou scraper jsou:

- Aktualizace stavových řádků (poslední dva řádky terminálu)

- Známá pozice hráče na mapě (na pozici kurzoru)
- Plně vykreslená obrazovka, kdy hra čeká na akci hráče

Klíčovou funkčností komponenty scraper je také řešení problému synchronizace popsaného v části 2.3. Scraper musí ve chvíli přijetí každého dílčího překreslení rozhodnout, zda jde o poslední změnu obrazovky v tomto tahu, nebo zda očekávat další. Je třeba rozlišit a vhodně reagovat na tyto možné výsledky akcí:

1. Jedno nebo více informačních sdělení na prvních řádcích terminálu, které končí (krom posledního) výzvou **`--More--`**
2. Dotaz typu *Ano / Ne*
3. Dotaz vyžadující textovou odpověď
4. Dotaz na volbu pole na mapě (cíle akce)
5. Dotaz na volbu směru akce (z osmi možných směrů)
6. Mnoho různých druhů menu, které lze rozlišit dle hlavičky a prováděné akce
7. Pouze překreslení mapy, hráč může rovnou volit další akci

Mezistavy vykreslení případů 1 až 6 mohou být nerozlišitelné od stavu 7. Po odeslání akce nelze předem určit, který z možných stavů po vykreslení všech výsledků akce nastane. NetHack bohužel nenabízí žádný jeden příkaz, kterým by bylo možné vyvolat jednoznačně rozpoznatelnou nedestruktivní reakci (a tím potvrdit ukončení překreslování) pro každý tento stav. Se znalostí poslední provedené akce ale lze ale sestrojit postup, který bezpečně povede k identifikaci plně vykreslené obrazovky a nezmění přitom stav hry.

Pokud bot provedl akci, která nemůže vést k dotazu na volbu směru (nemůže nastat případ 5), framework ihned po zaslání příkazů k provedení dané akce zašle navíc sekvenci znaků **##'**, která v případě, že bot může volit akci (případ 7) zahájí možnost zadat jeden z rozšířených příkazů NetHacku (*extended command*). Tento výsledek lze z podoby obrazovky jednoznačně rozpoznat, zrušit volbu příkazu a vyslat výzvu k volbě akce bota, přičemž již nehrozí záměna s mezistavem překreslení. Ostatní vyjmenované možnosti buď nejsou touto sekvencí znaků ovlivněny vůbec (případy 2 a 6), nebo jsou ovlivněny jen nedestruktivním způsobem (výzva je zachována – případy 3 a 4) a jejich výskyt je nezaměnitelný s jinými případy. U případu 1 může být zadáním rozšířeného příkazu přepsána poslední informační zpráva, kterou framework dodatečně vyhledá v historii zpráv (příkaz *Ctrl+P*).

Nejproblematičtější je situace, kdy botem provedená akce může vést k nutnosti volit směr (případ 5). Sekvence znaků **##'** má v tomto případě destruktivní efekt – volba

směru je zrušena, hráč může přijít o zdroje (např. jedno nabití kouzelné hůlky, kterou chtěl použít) a herní čas se může posunout. Akce, které mohou vyústit v tento případ jsou proto řešeny zvlášť a pokus o synchronizaci pomocí uvedené sekvence započne pouze ve chvíli, kdy se volba akce na obrazovce projevila a z tohoto projevu je jisté, že volba směru není nutná. Takový projev mají naštěstí všechny akce, které mohou k volbě směru vést.

Implementace tohoto postupu synchronizace je poměrně komplikovaná a je v ní nutno řešit mnoho okrajových případů. Ve výsledku ale nabízí výbornou spolehlivost bez jakýchkoli předpokladů o možném zpoždění sítě a minimálním vlivu na rychlost běhu v případě lokálních her, což jsou hlavní výhody této metody proti dalším možnostem popsaným v části 2.3. Další výhodou je fakt, že nepředpokládá žádné úpravy zdrojového kódu NetHacku, což by byla jediná dosud známá alternativa se srovnatelnou spolehlivostí.

Nevýhodou této metody je násobení případného síťového zpoždění, protože synchronizace v každém tahu vyžaduje několik výměn mezi klientem a serverem.

4.2 Reprezentace herního světa

Každý bot potřebuje znát informace o prostředí ve kterém operuje, na základě kterých se rozhoduje o svých akcích. Přitom je třeba vypořádat se s možnými nejednoznačnostmi rozhraní a pokud možno odstínit nepodstatné detaily.

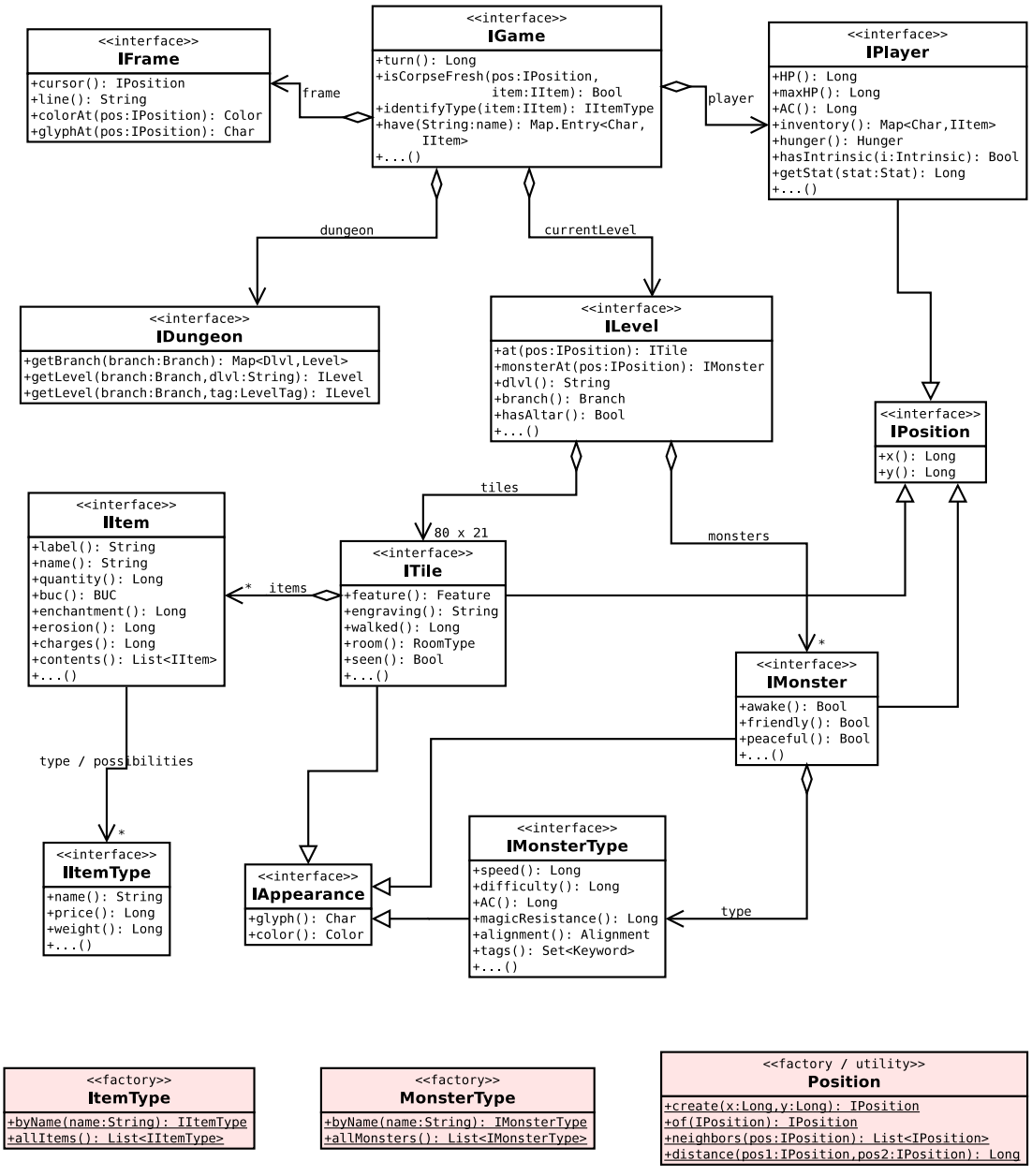
Framework udržuje komplexní model herního světa, který se automaticky aktualizuje na základě vyvolaných událostí. Přehled hlavních tříd a metod, které stav herního světa reprezentují je na obrázku 4.1. Jejich podrobný nízkoúrovňový popis je součástí přiložené JavaDoc dokumentace a tutoriálu (v angličtině).

Ze statických informací o světě NetHacku jsou přímo k dispozici základní údaje o všech typech nepřátel a předmětů, nepřímo vytažené ze zdrojových kódů NetHacku.[1] Na základě těchto údajů může bot usuzovat o nebezpečnosti nepřátel a užitečnosti předmětů, na které narazí.

Významné prvky některých úrovní, které nejsou generované zcela náhodně jsou automaticky zmapovány ve chvíli, kdy je daná úroveň jednoznačně rozpoznaná, tj. po nějaké době prozkoumávání. Podobně jako lidský hráč pak bot může volit optimální cesty známými úrovněmi, aniž by je musel plně prozkoumávat.

Problém nejednoznačností reprezentace nepřátel a prvků úrovní (zejm. typů pastí) lze vyřešit poměrně snadno – NetHack poskytuje herní příkaz, kterým se lze doptat na typ nepřítele, který se skrývá za nejednoznačným symbolem. Tento příkaz nestojí žádný herní čas a nemá vedlejší účinky na stav hry, framework ho tedy automaticky

[1] Základem pro databázi typů předmětů a nepřátel byly datové soubory projektů TAEB [2] a Saiph [3], které vychází ze zdrojového kódu NetHacku. Data byla přepoužita v souladu s licencemi.



Obrázek 4.1: Diagram tříd, které reprezentují stav herního světa

používá kdykoli na takovou nejednoznačnost narazí. Ve chvíli, kdy se bot rozhoduje o následující akci jsou tyto nejednoznačnosti transparentně vyřešeny.

Situace je složitější v případě nejednoznačné aktuální větve dungeonu. Jak bylo popsáno výše, obecně nelze vždy ihned jednoznačně určit, ve které větvi se hráč nachází po přechodu mezi úrovněmi dungeonu. Pro tyto případy jsou pro reprezentaci dané větve použity zástupné hodnoty jako „neznámá větev č. *X*“. Poznatky o úrovních v neznámých větvích jsou po jejím rozpoznání sloučeny.

Pohyb a přítomnost nepřátel na aktuální úrovni je sledován modulem *tracker*, který zaznamenává přítomnost nepřátel, kteří byli botem spatřeni, ale aktuálně nejsou v jeho výhledu. Sleduje také pohyb nepřátel v rámci výhledu a přenáší jejich vlastnosti, což předchází nutnosti v každém tahu opakovaně zjišťovat, o jaký typ protivníka jde, pokud není reprezentován jednoznačným symbolem (framework toto dělá automaticky, ale v úrovních s mnoha nejednoznačnými protivníky by to představovalo znatelné zpomalení).

Tato funkčnost také předchází mnoha druhům oscilací, kterými trpěli předchozí boti v situacích, kdy cesta k zacílenému nepříteli vedla úsekem, kde daný nepřítel není ve výhledu. Pokud bot přítomnost tohoto nepřítele mimo výhled „zapomenul“, ale opět ho spatřil na cestě za jiným cílem, snadno mohl skončit ve smyčce.

Součástí stavu hry jsou také známá fakta o párování mezi vzhledem a identitou předmětů. Toto je podrobněji popsáno v části 4.5.

V duchu funkcionálního přístupu je všechen stav hry zachycen v neměnných (immutable) hodnotách, které využívají perzistentní datové struktury [6] jazyka Clojure. Bot si tak může libovolně ukládat předchozí stavy hry nebo jejich části, volně je sdílet je mezi vlákny, používat jako klíče v hašovacích tabulkách apod.

Výpis 4.1: Příklad reprezentace údajů o identitě předmětu

```
#Armor{
 :name "shield of reflection",
 :glyph [,
 :price 50,
 :weight 50,
 :ac 2,
 :mc 0,
 :material :silver,
 :subtype :shield,
 :appearances ["polished silver shield"
               "smooth shield"],
 :safe true}
```

Výpis 4.2: Příklad reprezentace údajů o typu nepřítele

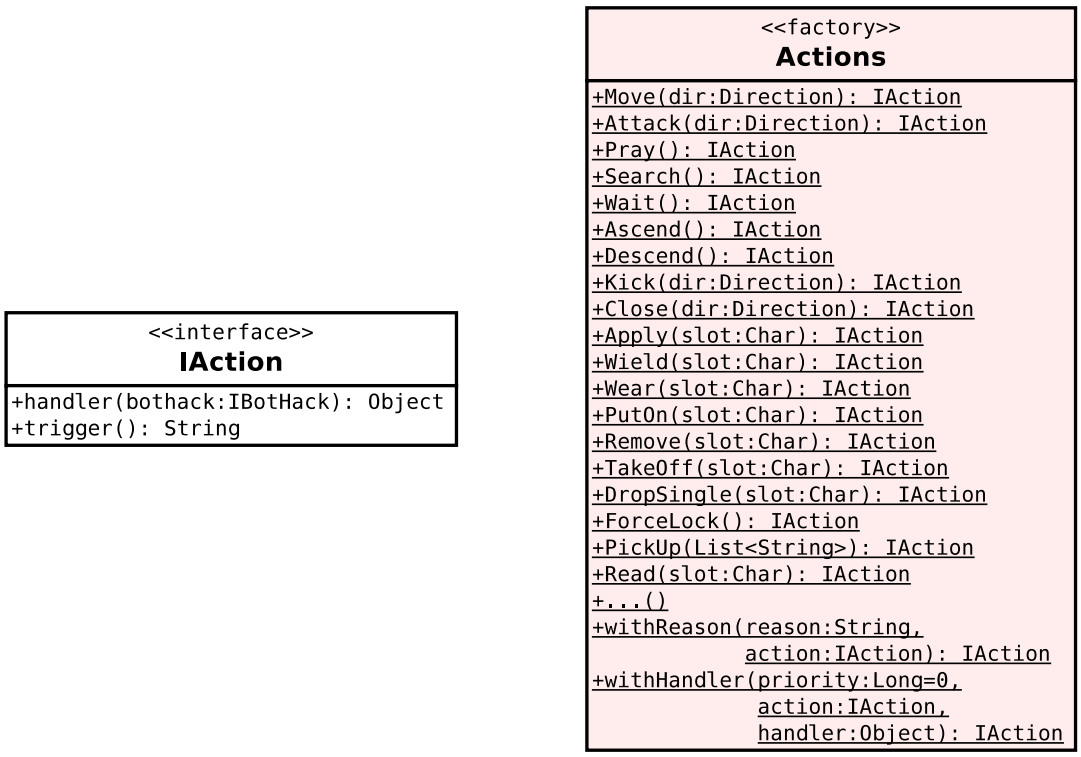
```
#MonsterType{
 :name "Medusa",
 :glyph @,
 :color :bright-green,
 :difficulty 20,
 :speed 12,
 :ac 2,
 :mr 50,
 :alignment -15,
 :gen-flags #{:unique :not-generated},
 :attacks
 [{:type :weapon, :damage-type :physical, :dices 2, :sides 4}
  {:type :claw, :damage-type :physical, :dices 1, :sides 8}
  {:type :gaze, :damage-type :stone, :dices 0, :sides 0}
  {:type :bite, :damage-type :poison, :dices 1, :sides 6}],
 :weight 1450,
 :nutrition 400,
 :sounds :hiss,
 :size :large,
 :resistances #{:stone :elbereth :poison},
 :resistances-conferred #{:poison},
 :tags #{:hostile :nopoly :female :waits :proper-name :strong
         :amphibious :swim :omnivore :fly :poisonous :humanoid
         :infravisible}}
```

4.3 Akce

Framework definuje obecné rozhraní akce, které abstrahuje od nízkoúrovňové interakce s NetHackem nutné pro provádění herních akcí, tj. zadávání herních příkazů a obsloužení výzev spojených s danou akcí.

Rozhraní reprezentující možné akce bota (*IAction*) je znázorněno na obrázku 4.2.

Metoda *trigger* vrací příkaz NetHacku (posloupnost kláves), který způsobí vyvolání této akce. Metoda *handler* pak instanci obsluhovače, který se postará o výzvy, které při vyvolání akce mohou nastat, včetně případných interakcí s herními menu. Při zvolení akce je její obsluhovač automaticky registrován a odregistrován je před výběrem akce následující. Pro příklad: akce *odemkni klíčem truhlu* automaticky obslouží výzvy příkazu použití předmětu (příkaz *'a'pply*) jako „Který předmět chceš použít?“ a „Chceš odemknout tuto truhlu?“

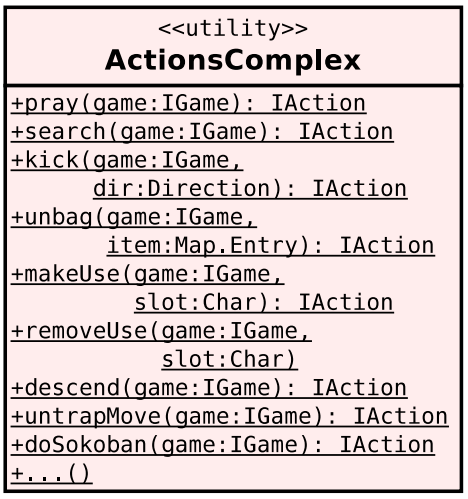


Obrázek 4.2: Rozhraní akce a knihovna atomických akcí

Logika bota musí reagovat na výzvy typu „zvol následující akci“ (rozhraní *IActionHandler*) a odpovědět instancí akčního rozhraní. Ve frameworku je dostupná knihovna akcí (třída *Actions*), které odpovídají hráči dostupným příkazům NetHacku a implementují toto rozhraní.

Mnohé akce NetHacku mají předpoklady, které je třeba před její volbou ověřit. Například nelze vzít do ruky zbraň v situaci, kdy je hráč proměněn v mravence. Pokud bot zvolí neproveditelnou akci, NetHack odpoví oznámením o jejím neúspěchu, které je botovi předáno vyvoláním odpovídající události. Neprovedená akce ale typicky neposune herní čas ani jinak nezmění aktuální situaci, tvrdohlavý bot tedy může snadno skončit v nekonečné smyčce. Pro běžné akce jsou proto k dispozici také funkce představující „chytřejší“ variantu dané akce (rozhraní *ActionsComplex*, obr. 4.3), která předpoklady ověří a v případě nemožnosti jejího provedení vrátí místo samotné akce *null* hodnotu. Akce k provedení vrací také funkce modulu navigace, který je popsán v další kapitole.

NetHack poskytuje velké množství příkazů pro práci s předměty, každý předmět má ale typicky jen jeden užitečný způsob jeho využití. Aby programátor bota nemusel řešit, že například zbroj se nasazuje příkazem *'W'ear* (opak *'T'ake off*), ale doplňky jako prsteny a amulety mají speciální příkaz *'P'ut on* (opak *'R'emove*), jsou součástí rozhraní *ActionsComplex* také metody *makeUse()* a *removeUse()*, které představují jednotný způsob, jak vyvolat použití nebo ustání v používání daného předmětu, přičemž patřičný herní příkaz je zvolen automaticky dle typu předmětu. Tyto dvě metody pokrývají typické případy využití celkem deseti různých atomických akcí NetHacku.



Obrázek 4.3: Třída zpřístupňující komplexní akce

Krom odstínění nízkoúrovňových příkazů se tyto metody v případě použití na výzbroj také automaticky postarají o splnění předpokladů pro nasazení nebo odstranění daného kusu výzbroje, pokud je to možné. Např. pro nasazení trička musí hráč vždy nejprve ručně odstranit další výzbroj, která tričko zakrývá, tj. zbroj a plášť. Ošetřeny jsou i okrajové situace, kdy např. bot nemůže sundat rukavice z důvodu prokletí na zbrani nebo štítu. Framework tak umožňuje výrazné zjednodušení logiky bota, která se výzbrojí zabývá.

Protože uvažování vedoucí ke zvolení určité akce se poměrně obtížně ladí, pokud je známa pouze výsledná akce, k instancím akčního rozhraní je možné pomocí metody *Actions.withReason()* připojovat lidsky čitelný důvod pro její volbu, případně postupný řetěz důvodů. Systém pak v logu vypíše k dané akci i uvedené důvody, což značně usnadňuje lokalizaci chyby při ladění problematického chování bota. V případě zabudovaných komplexních akcí je vždy připojen podrobný výčet důvodů vedoucích k volbě každé dílčí atomické akce.

Výpis 4.3: Ukázka záznamu důvodů k akci v logu

```
Performing action: #bothack.actions.Move{:dir :E}
 reasons:
 ["exploring current level #Tile{:x 65 :y 9 ...}"
  "first exploring main branch up to subbranch entrance"
  "exploring :mines until :minetown"
  "progress (default handler)"]
```

4.4 Navigace

Také hledání cest a průzkum je klíčovou funkčností, kterou nutně potřebují i nejzákladnější boti. Framework by měl poskytovat maximálně flexibilní navigaci, která pokud možno vyřeší obtíže uvedené v kapitole 2.5 a dokáže si poradit i v okrajových podmínkách. Rozhraní navigace musí brát v potaz, že požadavky na hledanou cestu se mohou lišit podle konkrétní situace bota a předmětů, které lze použít pro její zkrácení.

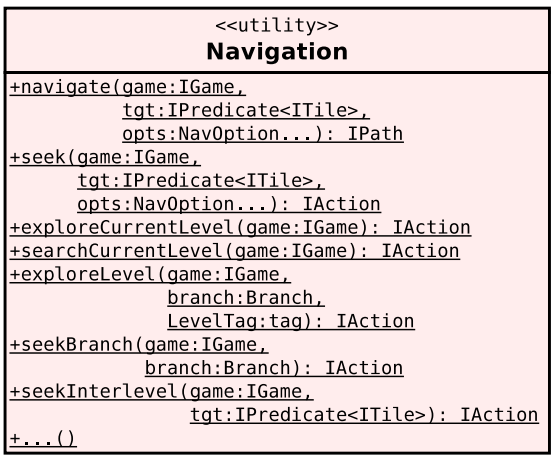
Značné zjednodušení problému navigace v pozdějších úrovních umožňuje náhled, že prakticky všechny problémy navigace řeší tři herní předměty: prsten levitace (*ring of levitation*, umožňuje překonat vodní plochy a lávu), krumpáč (*pick-axe*, umožňuje bourat nové cesty zdmi a skálou) a univerzální klíč (*skeleton key*, pro zamčené dveře). V situaci, kdy hráč má všechny tyto předměty, může se dostat na téměř libovolné místo kterékoli mapy bez spotřeby omezených zdrojů.

Dokud některý z těchto předmětů hráči chybí, situace je mnohem složitější. Může být nutno v úrovni hledat skryté cesty, odvalovat z cesty balvany, nebo používat různé triky pro zprůchodnění herních polí.

Botům je pro průzkum a navigaci poskytnuto několik vysokoúrovňových funkcí, které odstiňují od nutnosti konstruovat primitivní akce pro dílčí kroky a s ohledem na výše uvedené hledají optimální řešení navigačních úloh. Rozhraní navigace je znázorněno na obrázku 4.4.

Metody tohoto rozhraní automaticky využívají uvedené předměty s neomezenou použitelností, pokud jsou dostupné, ale dokáží se bez nich obejít minimálně v první polovině dungeonu, než je možné tyto předměty spolehlivě obstarat. V krajních situacích navigace také útočí na přátelská stvoření, pokud beznadějně blokují cestu a nejde o kritické herní postavy.

Pro navigaci v rámci jedné úrovně používá framework dvě základní metody pro hledání cest: *A** a *Dijkstrův algoritmus*. Algoritmus A* je použitý v případě, kdy bot



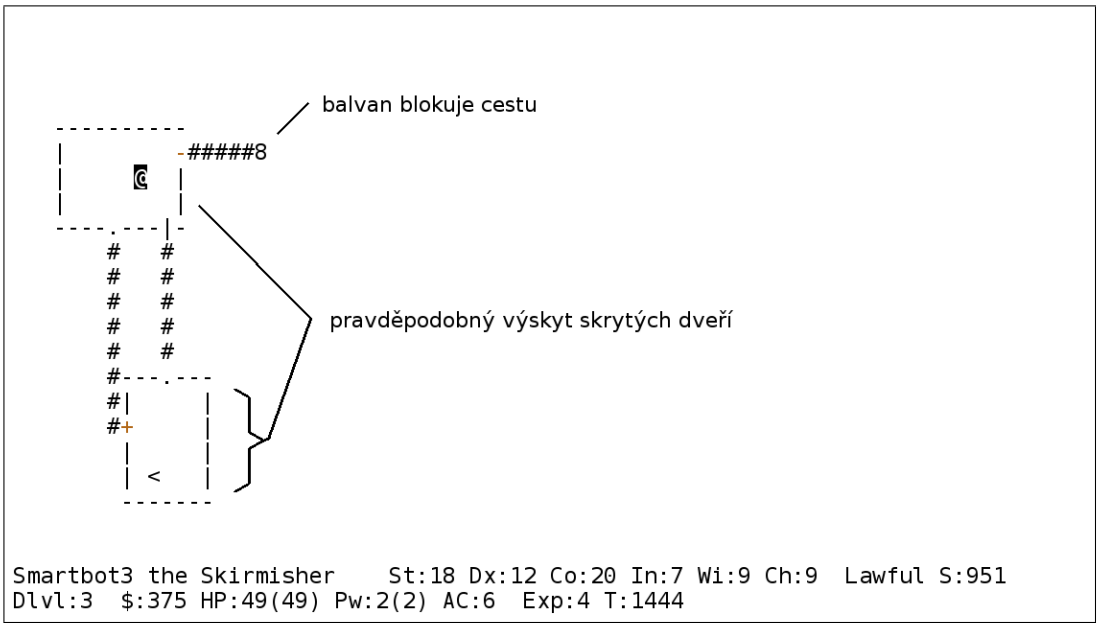
Obrázek 4.4: Rozhraní pro navigaci (hlavní metody)

naviguje k jednomu konkrétnímu cíli, Dijkstrův algoritmus pak v případě, že je třeba nalézt nejkratší cestu k jednomu z více přípustných cílů. Vhodná metoda je zvolena dle konkrétního volání podle počtu odpovídajících cílů v dané úrovni. Váhy hran a uzlů navigačního grafu systém určí automaticky na základě aktuální výbavy bota, přítomnosti přátelských i nepřátelských stvoření na daném poli a jeho domnělé bezpečnosti – např. jsou preferovány pole, kterými hráč dříve bez újmy prošel a nemohou tedy obsahovat pasti.

Funkce průzkumu (*exploreCurrentLevel()*) automaticky provádí hledání pokud narazí na slepé cesty nebo v případě, že už se žádná neprozkoumaná pole v přístupných částí úrovně nenachází, ale v úrovni jsou stále velké neodhalené plochy nebo nebyl nalezen vstup do následující úrovně. Strategie prohledávání preferuje slepé uličky a místa, které jsou blízko rozsáhlejším neodhaleným plochám a s větší pravděpodobností skrývají tajné dveře a chodby (znázorněno na obrázku 4.5). Teprve pokud je vyhledávání v takových místech neúspěšné, je prohledávána celá úroveň.

Navigovat lze také napříč úrovněmi dungeonu do libovolné zadané úrovně včetně dosud nenalezených speciálních úrovní v neprozkoumaných větvích. Framework má zabudované informace o tom, kde a jak kterou speciální úroveň nebo větev hledat a vždy vrací akce, které k jejich nalezení povedou.

Funkce zpřístupňující navigaci v rámci úrovně (*seek()* a *navigate()*) přijímají volitelné parametry představující množinu dodatečných voleb a omezení, pomocí kterých lze např. zakázat použití levitace či krumpáče (což je vhodné během soubojů), povolit pouze cesty, které již bot dříve prozkoumal (pro únik z nebezpečné situace) nebo



Obrázek 4.5: Jak se dostat do zbytku úrovně?

zakázat průchod přes pole, které obsahují pasti. Úplný seznam těchto voleb lze nalézt v JavaDoc dokumentaci výčtového typu *NavOption*.

Díky popsanému rozhraní je problém navigace v prostředí NetHacku z pohledu programátora bota prakticky vyřešen. Autor bota musí pouze zajistit, aby se bot pokusil získat uvedené tři předměty nezbytné pro navigaci v některých pozdějších úrovních. Popsané metody byly odladěny mnoha testovacími běhy ukázkového bota a pouze vzácně jej dostanou do bezvýchodné situace – může se například stát, že se bot pastí propadne do zamčeného obchodu v nižší úrovni, ze kterého bez možnosti odemčení dveří nebo teleportace není úniku. Tyto situace typicky nastávají jen v počátečních fázích hry, kdy botovi chybí výbava, s jejíž pomocí by si navigace jinak dokázala poradit.

4.5 Identifikace předmětů

Úspěšnost hráče NetHacku značně závisí na jeho schopnosti efektivně identifikovat nalezené předměty, tj. zjistit aktuální párování mezi jejich vzhledem a identitou platné pro daný běh hry a vyhnout se používání potenciálně nebezpečných předmětů „na slepo“.

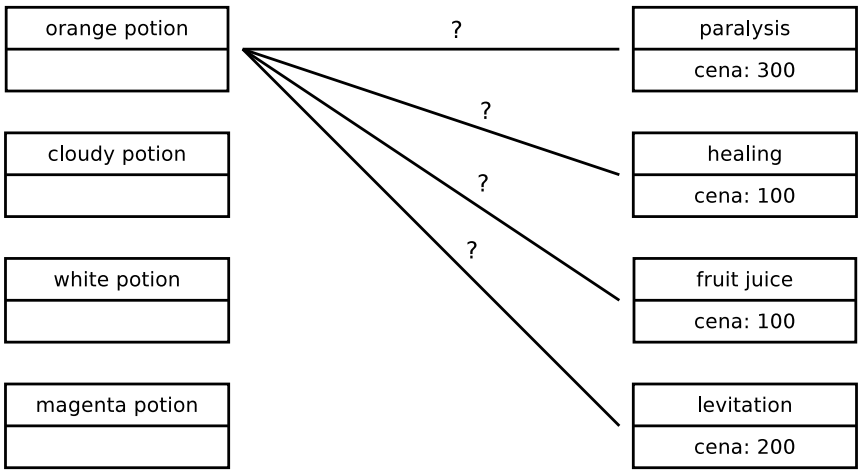
Protože svitek identifikace (*scroll of identify*), který je hlavním prostředkem pro postupné odhalení tohoto párování je omezeným zdrojem a navíc sám o sobě předmětem, který má v každé hře náhodný vzhled, je hráč nucen využívat také jiných prostředků k identifikaci. Toto je jeden z problémů, v jehož řešení mohou být boti efektivnější než lidští hráči a zde implementovaný framework je v tomto výrazně nápomocen.

Pro řešení tohoto problému využívá framework metod logického programování. V průběhu hry automaticky ukládá pozorovaná fakta o vlastnostech neznámých předmětů jako prvky relací v interní databázi. S databází těchto faktů pracuje logický program, který nad ní vyhodnocuje dotazy typu „Co všechno by mohl být tento (neidentifikovaný) předmět?“

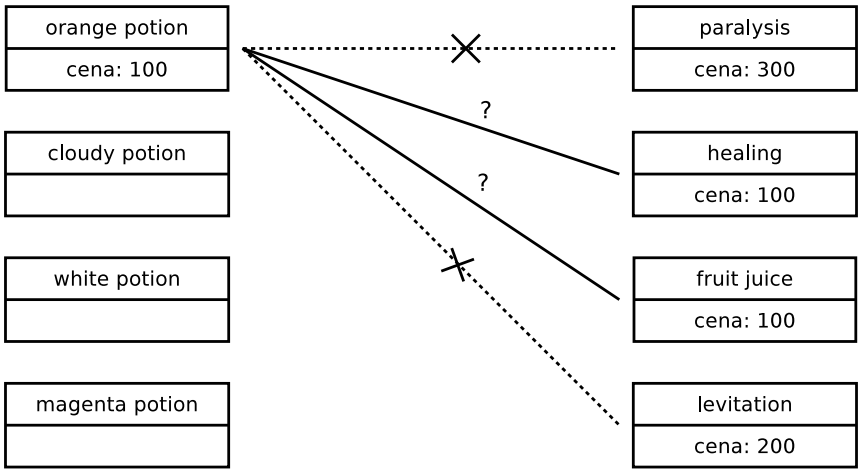
Použitý nástroj pro logické programování je knihovna Clojure *core.logic*, která je portem systému *miniKanren* [8], původně implementovaného v jazyce Scheme.

Framework sleduje několik různých vlastností předmětů, které se nějakým způsobem pozorovatelně projevují, přičemž nejužitečnější vlastností pro identifikaci je cena za předmět nabízená nebo poptávaná v herních obchodech. Ve většině případů nelze určit identitu předmětu pouze z jednoho pozorování ceny, ale lze eliminovat některé možné identity. Je nutné přitom počítat také s vlivem náhody, vlastnostmi hráče (charisma) a s dalšími skrytými vlastnostmi předmětu, které cenu ovlivňují. Přesná pravidla pro tyto možné vlivy lze naštěstí vyčíst ze zdrojových kódů NetHacku.

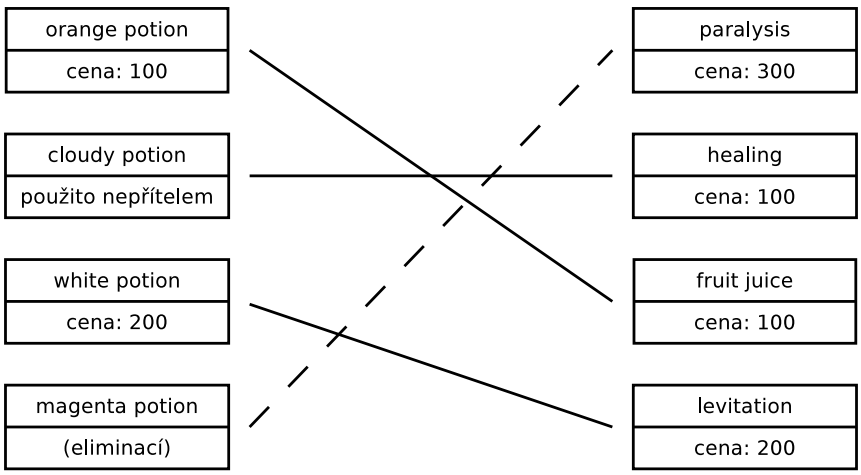
Ve výsledku botům stačí pouze projít předměty v obchodu nabízené (navigace obchody automaticky prozkoumává) a vypozorované ceny jsou ihned použity pro eli-



Obrázek 4.6: Výchozí stav zjednodušeného příkladu, identita všech předmětů na levé straně je neznámá



Obrázek 4.7: Po odhalení ceny oranžového lektvaru lze eliminovat některé možnosti



Obrázek 4.8: Se další znalostí ceny bílého lektvaru a spatření léčivého účinku lze identifikovat vše. V další hře ale může být vše jinak.

minaci párování mezi vzhledem a identitou, které pozorovaným faktům nevyhovují. I v případech, kdy toto nevede k jednoznačné identifikaci předmětu, mohou být odhaleny alespoň dílčí vlastnosti, např. zda jde o jeden z předmětů, které lze bezpečně vyzkoušet a tím identifikovat definitivně.

Framework provádí také identifikaci eliminační metodou – pokud například bylo ve hře jednoznačně identifikováno 8 z 9 možných typů amuletů, poslední typ je automaticky spárován se zbývajícím vzhledem.

Zjednodušený příklad postupné identifikace předmětů typu lektvaru je na obrázcích 4.6 až 4.8. Ve skutečnosti je ve hře 20 druhů lektvarů s náhodným vzhledem.

Pro uspokojivý výkon řešení bylo nutné zavést kešování výsledků opakovaných dotazů se stejnou množinou známých faktů. S touto optimalizací má výsledná funkčnost téměř zanedbatelný dopad na rychlost běhu botů, ačkoli jim poskytuje značnou přidanou hodnotu.

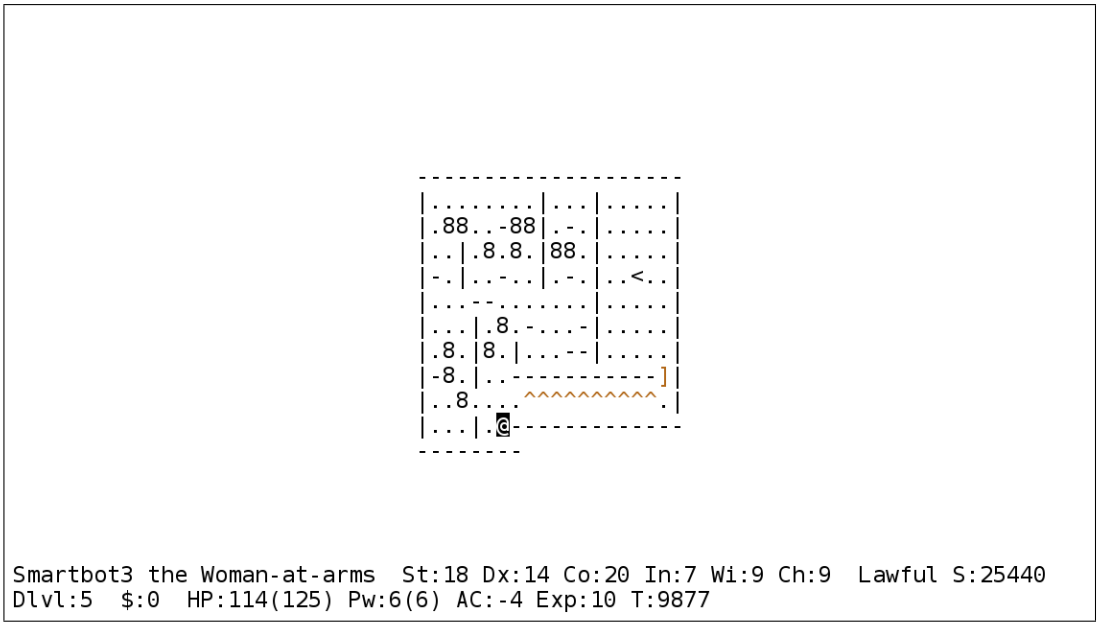
Funkčnost identifikace předmětů popsané v této části by v principu jistě bylo možné implementovat i bez využití logického programování, toto paradigma nicméně dobře pasuje na řešený problém. S využitím knihovny *core.logic* není nutno ručně programovat nízkoúrovňovou reprezentaci znalostí s možností efektivního dotazování a rezoluční metoda umožňuje přímočaré odvozování nových poznatků na základě deklarativního popisu pravidel pro sledované vlastnosti herních předmětů.

Nevýhodou tohoto řešení je poměrně obtížná srozumitelnost výsledného kódu pro programátory bez předchozích zkušeností s logickým programováním.

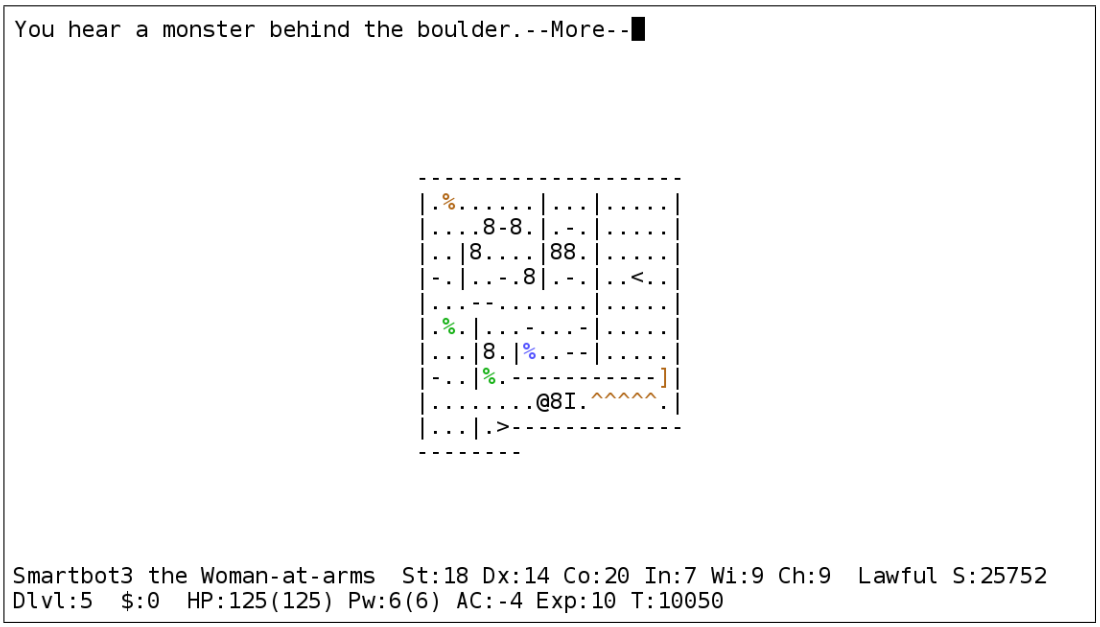
4.6 Sokoban

NetHack obsahuje v jedné z větví hlavního dungeonu minihru v podobě varianty hry *Sokoban*, neboli „Skladník“. Cílem této minihry je vyřešit čtyři úrovně hlavolamu, na jehož konci se nachází odměna v podobě vysoce užitečného předmětu (*amulet of reflection* nebo *bag of holding*). Průchod těmito úrovněmi není nutný pro dokončení hry, ale krom uvedené odměny lze v Sokobanu nalézt také velké množství jídla a dalších náhodně vygenerovaných předmětů, pro které se většinou vyplatí úrovně vyřešit.

Hlavolam spočívá v úloze správným způsobem posouvat balvany (na obrázku 4.9 symbol ‘`8`’) po úrovni a vyplnit jimi neprůchozí díry v zemi (symbol ‘`^`’), díky čemuž se lze dostat do další úrovně nebo k finální odměně. Při řešení je třeba dbát na to, aby nedošlo k zablokování příliš velkého počtu balvanů v rozích nebo jiných místech, ve kterých s nimi nelze dále pohybovat. Pokusy o podvod jako rozbíjení balvanů krumpáčem, tvorba nových balvanů pomocí *scroll of earth* apod. jsou hrou penalizovány a v úrovních Sokobanu platí jiná pravidla pro pohyb než ve zbytku dungeonu, např. balvany nelze posouvat diagonálně a přes díry v zemi není možné projít ani s pomocí levitace.



Obrázek 4.9: Výchozí stav úrovně Sokobanu



Obrázek 4.10: Částečně vyřešená úroveň Sokobanu, kde nepřítel blokuje cestu

O zobecněné variantě hry Sokoban bylo dokázáno, že je PSPACE-úplná [9]. V případě NetHacku ale není třeba problém řešit obecně, protože hra obsahuje celkem pouze osm pevně daných variant úrovní, na které může hráč narazit. Řešení nicméně značně komplikují nepřátelé, kteří se, stejně jako ve zbytku dungeonu, v úrovních Sokobanu náhodně zjevují (obr. 4.10). Tito se mohou vplést do cesty a řešení zkomplikovat, v okrajových případech dokonce posunout, zničit nebo vytvořit nové balvany a znemožnit tak vyřešení hlavolamu obvyklým způsobem.

Framework poskytuje metodu *doSoko()* (součást rozhraní *ActionsComplex*, obr. 4.3), která v úrovních Sokobanu vrací následující akci k provedení vedoucí k postupnému vyřešení úrovně. S využitím této funkce je řešení Sokobanu pro boty velmi jednoduché – implementace bota musí pouze bez posouvání balvanů zajistit likvidaci nepřátel, kteří se v úrovni objeví. Framework navíc automaticky identifikuje podobu odměny v poslední úrovni.

Implementace metody *doSoko()* využívá pevně definované posloupnosti posunů balvanů pro každou z osmi možných úrovní. Z hlediska implementace a odladění této funkčnosti bylo náročné zejména zajistit, aby tato funkce dokázala spolehlivě navázat po přerušení na libovolném místě, jak mohou vynutit nepřátelé. Řešení Sokobanu pomocí této metody nemá stoprocentní spolehlivost a v okrajových případech (když dojde ke zničení balvanu) může vést k zacyklení bota. Velká většina běhů ale na tyto případy nenarazí a vede k úspěšnému dokončení Sokobanu.

Kapitola 5

Implementace ukázkového bota

Tato kapitola se zabývá implementací bota, který využívá funkcí frameworku z předchozí kapitoly a herními technikami, pomocí kterých dokáže hru dokončit a které mohou být obecně vhodné pro využití dalšími boty.

5.1 Model chování

Logika ukázkového bota je implementována jako stochastický rozhodovací strom, jehož uzly v první úrovni mohou uchovávat vlastní stav (dostupný pouze v daném podstromu – sousední uzly se neovlivňují) a případně trvale zanikají po splnění své úlohy. Toto řešení přímočaře zapadá do modelu obsluhovačů popsaného v části 3.2. Má výhodu v relativní jednoduchosti a ukázalo se jako dostatečné pro implementaci veškeré logiky potřebné pro dokončení hry.

Chování bota vznikalo iterativním postupem. První verze bota uměla pouze běhat v kruhu, dokud nezemřela hlady. Postupně byly přidávány funkce, které botovi v danou chvíli pomohly o trochu déle přežít nebo pokročit ve hře. Tento postup poskytnul mnoho prostoru pro odladění základní funkčnosti frameworku jako interpretace obrazovky terminálu a navigace.

Následuje výčet hlavních dílčích chování ukázkového bota, tj. obsluhovačů, které představují první úroveň uzlů rozhodovacího stromu logiky bota v pořadí dle jejich priority. Skutečná implementace jich využívá větší množství, pro jednoduchost popisu byly méně významné obsluhovače vynechány nebo sloučeny.

1. Řešení situací představujících akutní ohrožení na životě. První obsluhovač kontroluje, zda je bot ve stavu, kdy mu hrozí např. smrt zkameněním, utonutím, otravou nebo jiným způsobem, kterému lze předejít okamžitým provedením určité akce.

2. Únik z nebezpečné situace. Pokud má bot velmi málo zdraví, ohrožují ho nepřátelé a nabízí se možná úniková cesta, pak se po ní vydá.

3. Boj s nepřáteli. Volí akci ve chvíli, kdy se v okolí bota vyskytují aktivní hrozby. Zde je implementována strategie popsaná dále v části 5.2.

4. Správa inventáře. Zde bot kontroluje, zda využívá nejlepší dostupnou zbroj a jinou výbavu, případně se zbavuje nepotřebných předmětů.

5. Jídlo. Pokud je bot aktuálně hladový nebo má možnost zkonzumovat rychle se kazící těla nepřátel, které poskytují užitečné vlastnosti, volí akce v tomto směru.

6. Obnova. Když je bot v relativně bezpečném prostředí, ale má málo zdraví nebo dočasné nesmrtící znevýhodnění, pokusí se skrýt na méně exponovaném místě a vyčkává postupné obnovy.

7. Sběr předmětů. Pokud se v okolí nachází dosud neprozkoumané nebo známé užitečné předměty, bot se je vydá sbírat.

8. Používání předmětů. Pokud má bot jednorázové prostředky pro vylepšení své aktuální situace nebo výbavy, případně možnost odstranit kletby pomocí kouzelných svitků, zde je využije.

9. Pokračování v herním plánu. Výchozí obsluhovač bota postupuje dle plánu, který je podrobně popsán v kapitole 5.3.

Ačkoli ukázkový bot nevyužívá dlouhodobějšího plánování ani žádného druhu strojového učení pro adaptaci chování, v principu nic nebrání použití frameworku pro implementaci botů využívajících těchto technik nebo sofistikovanějšího modelu chování.

5.2 Strategie soubojů

Souboje s protivníky v NetHacku představují významnou část hry a jsou zdaleka nejobvyklejší příčinou smrti hráče.

Botům je v přístupu k soubojům poskytnuta plná volnost, framework aktuálně nenabízí specializovanou funkčnost orientovanou na souboje nebo předpřipravené taktiky. V soubojích mohou boti využívat poskytnutou knihovnu akcí a modul navigace.

Ukázkový bot při soubojích hojně využívá faktu, že nápis „Elbereth“ na herním poli způsobuje zastrašení většiny typů nepřátel, což poskytuje výhodu zejm. při soubojích se skupinou nepřátel. Tvorba tohoto nápisu ale stojí herní čas, není zcela spolehlivá a nejsilnější nepřátelé ve hře jej ignorují. Obecně se bot snaží nepřátele nalákat do úzkých chodeb, kde nehrozí obklíčení a ustupuje pokud možno směrem ke schodišti do předchozí úrovně, odkud lze z nebezpečí prchnout a zotavit se. Bot také využívá vržených zbraní proti nepřátelům s pasivními útoky a těm, kteří svými útoky degradují nebo odcizují výbavu hráče.

5.3 Plán průchodu hrou

NetHack nabízí mnoho možností, jak řešit herní problémy za různých podmínek. Spíše než vytvoření co možná nejvíce univerzálního postupu, který by si poradil za jakýchkoli okolností bylo snahou implementovat minimalistickou strategii, která by byla dostatečná pro dokončení hry s alespoň malou pravděpodobností (v řádu jednotek procent, což odpovídá průměrné úspěšnosti lidských hráčů na veřejných serverech) a poskytovala možnost otestovat veškerou funkčnost frameworku. Ukázkový bot je přizpůsoben hře povolání *Valkyrie*, které je pro přímočarý styl hry nejvhodnější.

Počátky hry

V počáteční fázi hry bot vyhledá vedlejší větev dungeonu *the Gnomish Mines*, kde se pokusí získat základní výbavu (trpasličí zbroj, krumpáč) a dosáhnout páté zkušenostní úrovně. V této fázi bot sestoupí maximálně do hloubky speciální úrovně *Minetown*, která poskytuje možnost identifikace předmětů pomocí obchodů a chrámu.

Jakmile má bot dostatek zkušeností a výbavu poskytující základní ochranu (AC hodnoty 3 a lepší), pokusí se vytvořit unikátní zbraň *Excalibur*, se kterou vystačí až do konce hry. Tuto lze získat opakovaným smočením startovního dlouhého meče v náhodně vygenerovaných fontánách.

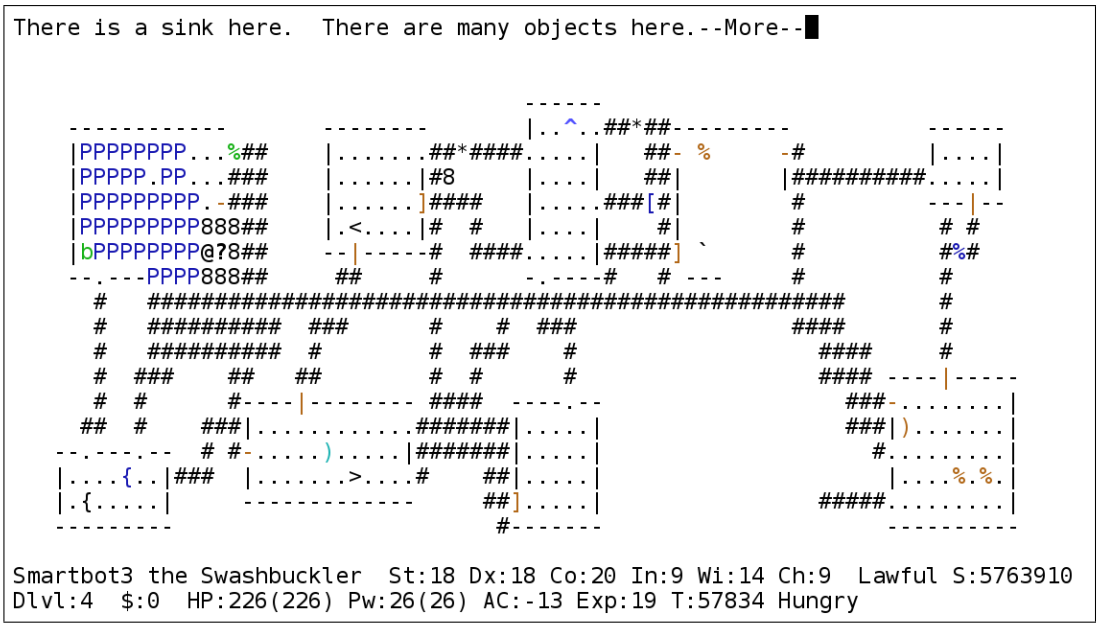
Bot pokračuje průchodem větve *Sokoban* a dokud postrádá základní předměty (klíč nebo šperhák, krumpáč, zbroj), prozkoumává větev *Gnomish Mines* až na její dno. V opačném případě pokračuje prozkoumáváním úrovní hlavní větve a vybavuje se užitečnými předměty, na které náhodně narazí.

V této fázi hry má bot, podobně jako lidští hráči, nejvyšší úmrtnost.

Střední část hry

Od dvacátého podlaží bot přestává plně prozkoumávat úrovně a snaží se co nejrychleji sestoupit do speciální úrovně hradu (*the Castle*). Toto je nejzazší místo hlavní větve dungeonu, kam se dostali nejúspěšnější dosud existující boti. Pro překonání padacího mostu v této úrovni bot předpokládá dřívější nález poměrně běžného předmětu *wand of striking*.

Hrad je obýván zvlášť nebezpečnými nepřáteli, ale nachází se zde jedna garantovaná instance zřejmě nejužitečnějšího předmětu ve hře, totiž hůlky plnící přání (*wand of wishing*), která umožňuje s určitými omezeními získat 4 až 7 hráčem zvolených předmětů. Nalezení této hůlky představuje zlomový moment, kdy pravděpodobnost dokončení dané hry prudce vzroste. Bot ji využije pro doplnění základní chybějící výbavy, zejm. prstenu levitace nutného pro navigaci ve zbytku hry a zdroje ochrany proti kouzlům (*magic resistance*).



Obrázek 5.1: Farming – pro omezený výhled hráče není patrné, že nepřátelé pokrývají celou zobrazenou úroveň

K úspěšnému dokončení hry je nicméně potřeba větší množství předmětů, než lze takto obstarat. Lidští hráči mohou do jisté míry vystačit bez nich nebo využít pokročilejších postupů pro jejich získání. Bot se uchyluje k využití degenerované strategie, tzv. farmaření (*pudding farming*, obr. 5.1):

Ve hře se vyskytují dva typy nepřátel (*black pudding*, *brown pudding*), které se po zasažení kovovým předmětem rozdělí vedví. Záměrným opakovaným dělením těchto nepřátel z bezpečné pozice jimi lze zaplnit celou herní úroveň. Opakovaným pobitím a znovuobnovením jejich populace lze podle množství investovaného času vygenerovat prakticky libovolné množství náhodných předmětů, včetně hůlek plnících přání.

Tato vlastnost hry je dobře známá a není považována za chybu. Lidští hráči tohoto typicky nezneužívají, protože jde o časově náročnou a nesmírně nudnou činnost. V případě bota jde o efektivní způsob, jak si obstarat dostatečné množství zdrojů a zbývající výbavy nutné k dokončení hry. Z hlediska frameworku je to také možnost otestovat některé extrémní situace, jako výskyt tisíců předmětů na jednom poli a chaotického pohybu mnoha nepřítel.

Závěrečná část hry

Po úspěšném zakončení farmaření je bot výborně vybaven a ve zbytku hry je jen několik momentů, kdy mu hrozí reálné nebezpečí. Zde už bot nevyužívá žádných neobvyklých postupů.

Bot pokračuje průchodem větví *Quest* a většinou zbylých úrovní dungeonu, včetně podvětve *Vlad's tower*. Po objevení poslední (cca. padesáté) úrovně dungeonu se bot vydává k úhlavnímu nepříteli *the Wizard of Yendor*. Tento se pak i po své smrti opako-

vaně zjevuje a napadá hráče v průběhu celého zbytku hry. S výbavou, kterou by v této fázi měl bot mít dostupnou ale nepředstavuje zásadní problém.

Bot se bez dalšího prozkoumávání vydá získat cílový předmět *the Amulet of Yendor* a vynese jej zpět do první úrovně, odkud mu zbývá projít posledních 5 speciálních úrovní – *elemental planes* a *Astral plane*. Hlavní výzvou těchto úrovní jsou nejnebezpečnější herní nepřátelé a potřeba vyhledat skrytý, náhodně umístěný portál, který z každé úrovně vede do následující (bez možnosti návratu). V případě poslední úrovně je nutné probojovat se ke správnému oltáři svého božstva, což je jeden ze tří oltářů v úrovni přítomných (v každé hře jiný). Úspěšná hra končí obětováním předmětu *Amulet of Yendor* na tomto oltáři.

Poznámky k plánu

Nabízí se řada možností, jak popsaný plán vylepšit využíváním více druhů předmětů a jiných prvků NetHacku, které bot aktuálně ignoruje nebo využívá jen částečně, a zvýšit tak pravděpodobnost přežití a úspěchu bota za různých okrajových podmínek. Bylo by také možné eliminovat používání degenerovaných strategií a udělat tak herní styl bota podobnější lidským hráčům. Primárním účelem bota nicméně bylo demonstrovat, že framework poskytuje veškerou základní funkčnost nutnou pro průchod celou hrou a poskytnout ukázkové řešení základních herních problémů, což bot úspěšně splňuje a dosahuje přitom výrazně lepších výsledků než jeho předchůdci.

5.4 Ukázka kódu

V této části jsou ukázkové výpisy části kódu jednoduchého bota v jazyce Java, postaveného na popsaném frameworku. Jde o zjednodušenou verzi bota popisovaného v předchozích částech této kapitoly. Kompletní zdroje zjednodušeného i plného ukázkového bota jsou na přiloženém CD.

Na výpisu 5.1 je vidět inicializace a registrace obsluhovačů, které realizují dílčí chování bota a jsou aktivní během celé hry.

Výpis 5.2 je příkladem jednoho z uvedených obsluhovačů. Kód využívá rozhraní pro navigaci, aby vyhledal hromádky zlata vyskytující se na aktuální úrovni a dokud bot nemá 100 a více zlaťáků, všechny nalezené zlaťáky se vydává sbírat.

Výpis 5.1: Hlavní třída zjednodušeného ukázkového bota

```
package bothack.javabots.javabot;

import bothack.bot.IBotHack;

/**
 * This is an example of a bot using the Java API.
 * It is a very basic bot, doing only minimal things to demonstrate
 * and test basic functionality of the API. You can expect it to die
 * fairly quickly.
 * See https://github.com/krajj7/BotHack/blob/master/doc/running.md
 * for how to run it. */
public class JavaBot {
    public JavaBot(IBotHack bothack) {
        bothack.registerHandler(-16, new EnhanceHandler());
        bothack.registerHandler(-11, new MajorTroubleHandler());
        bothack.registerHandler(-6, new FightHandler());
        bothack.registerHandler(-3, new MinorTroubleHandler());
        bothack.registerHandler(1, new FeedHandler());
        bothack.registerHandler(3, new RecoverHandler());
        bothack.registerHandler(6, new GatherItems());
        bothack.registerHandler(19, new ProgressHandler());
        System.out.println("bot initialized");
    }
}
```

Výpis 5.2: Obsluhovač bota – sběr zlata

```
package bothack.javabots.javabot;

import bothack.actions.*;
import bothack.actions.Navigation.IPath;
import bothack.bot.*;
import bothack.bot.dungeon.ITile;
import bothack.bot.items.*;
import bothack.prompts.IActionHandler;

public class GatherItems implements IActionHandler {
    static IItemType GOLD = ItemType.byName("gold piece");

    IPredicate<ITile> hasGold = new IPredicate<ITile>() {
        public boolean apply(ITile tile) {
            for (IItem i : tile.items())
                if (i.type().equals(GOLD))
                    return true;
            return false;
        }
    };

    boolean wantGold(IGame game) {
        return game.gold() < 100;
    }

    public IAction chooseAction(IGame game) {
        if (!wantGold(game))
            return null;
        final IPlayer player = game.player();
        ITile atPlayer = game.currentLevel().at(player);
        IPath res = null;
        res = Navigation.navigate(game,hasGold,NavOption.EXPLORED);
        if (res != null) {
            if (res.step() != null) {
                return res.step();
            } else {
                IItem gold = null;
                for (IItem i : atPlayer.items()) {
                    if (i.type().equals(GOLD)) {
                        gold = i;
                        break;
                    }
                }
                return Actions.PickUp(gold.label());
            }
        }
        return null;
    }
}
```

Kapitola 6

Vlastnosti implementace

V této kapitole jsou popsány provozní charakteristiky řešení a srovnání s výsledky předchozích prací.

6.1 Hardwarové nároky a výkon

Nároky implementovaného frameworku na hardware jsou poměrně malé – ukázkový bot byl úspěšně provozován na stroji s jedním 1.3 GHz CPU a maximální velikostí JVM heapu 122 MB.

V průběhu vývoje byl program profilován volně dostupným nástrojem *VisualVM*. Nejvíce exponovaný kód, zejm. navigace a parsování obrazovky terminálu, byl zoptimalizován použitím *type hints*, které překladači umožňují předejít pomalejším voláním používajícím reflexi a také využitím primitivní aritmetiky bez kontrol přetečení (např. u souřadnic mapy přetečení nehrozí).

Ve výsledku hraje ukázkový bot co do počtu tahů za sekundu u lokální hry výrazně rychleji než člověk i na zastaralém hardwaru. Lidští hráči nicméně dokáží hru vyhrát s výrazně menším počtem odehraných tahů (což je z velké míry důsledkem farmaření bota). Aktuální rekordní čas výhry bota (na serveru s 3.2 GHz quad i7 procesorem a 4 GB heap) je 1 hodina a 17 minut. Rekordní čas lidských hráčů je 1 hodina a 3 minuty.

Rychlost běhu bota je v případě her na vzdáleném serveru nejvíce ovlivněna latencí sítě, jinak je úzkým hrdlem typicky výkon procesoru. V možných optimalizacích jsou stále rezervy, které vzhledem k uspokojivému dosaženému výkonu nebyly plně využity.

6.2 Spolehlivost

Pro dosažení vysoké spolehlivosti frameworku a ukázkového bota bylo provedeno velké množství reálných běhů – bot v průběhu vývoje běžel skoro neustále v několika

paralelních instancích a postupně byly téměř vymýceny situace, které vedly k zásekům a pádům.

Určité procento her ukázkového bota se nicméně stále dostane do bezvýchodné situace, které mohou nastat buď neovlivnitelnou nešťastnou náhodou (např. bot přijde o zdroj levitace a je uvězněn vodou nebo lávou) nebo chybou v kódu bota či vzácněji frameworku.

Framework má zabudováno několik mechanismů, které problémy tohoto typu detekují a dle konfigurace může být daný běh automaticky ukončen. Za nezotavitelný „zásek“ jsou považovány následující situace:

- Žádný z obsluhovačů bota po výzvě nevybral akci k provedení, např. v důsledku chyby v logice nebo neodchycené výjimky, která nastala v uživatelském kódu.
- Bot provedl tisíc akcí po sobě, které nevedly ke zvýšení herního počítadla tahů. Tato situace může nastat, pokud se bot opakovaně pokouší provádět akci, kterou v daném kontextu nelze provést nebo nemá kýžený efekt a zároveň její provedení neposouvá herní čas. Pokud je opakována akce, která herní čas posouvá, tak problém není tolik nutno řešit, protože bot dříve nebo později vyhladoví.
- Bot nezareagoval na výzvu k výběru akce po třech minutách reálného času, např. z důvodu zacyklení.
- Bot se pokouší extrémně dlouho prohledávat jednu a tu samou úroveň. V pokročilé fázi hry může bot s dostatečným množstvím zdrojů přežít velmi dlouhou dobu a bezvýsledně se pokoušet navigovat v úrovni, pro jejíž překonání nemá potřebnou výbavu. Modul navigace v takové situaci provádí prohledávání úrovně, což může často pomoci, pokud ale ani deset iterací prohledávání nemá výsledek, je běh považován za beznadějný.

Díky tomuto mechanismu je možno spouštět opakované běhy bota dlouhodobě bez dozoru i v případě, že bot není zcela spolehlivý, nebo naráží na problematické okrajové situace.

6.3 Srovnání předchozích řešení

Za posledních 10 let vzniklo minimálně 15 různých botů pro NetHack v různých jazycích, jejichž kód je veřejně dostupný. Nejúspěšnější z nich se ale dostali pouze přibližně do poloviny hry a to jen s velmi malou pravděpodobností.

Nevýznamnějšími zástupci jsou framework TAEB [2] (implementovaný v jazyce Perl) a bot Saiph [3] (v jazyce C++).

Framework TAEB poskytoval botům mnoho ze základní funkčnosti popsané v kapitole 4, ale trpěl nedotažeností a velkou chybovostí klíčových částí jako navigace.

Objemem kódu mnohonásobně přesahuje zde popsanou implementaci. Rychlost vývoje frameworku dlouhou dobu trpěla použitou synchronizační metodou, která značně limitovala rychlost běhu botů a tím i možnost důkladného odladění.

Bot Saiph se ve hře se dostal dále než boti využívající TAEB, ale je ad-hoc implementací bez samostatného frameworku, kterého by bylo možno využít v dalších projektech. Podobně jako v případě TAEBu měl problém zejména se spolehlivostí navigace, správou inventáře a využíváním předmětů. I v jeho případě končila velká část her eventuálním pádem nebo nekonečnou smyčkou.

Zde popsané řešení je obecně více dotažené do konce, odladěné a zdokumentované. Funkčnost navigace a identifikace předmětů je výrazně sofistikovanější. Framework si dokáže poradit i s prvky hry, které se vyskytují pouze na jejím konci, kam se žádný z předchozích botů nedostal. Dále je poprvé poskytnuta například funkce sledování a párování pohybujících se nepřátel a podpora obchodů.

V této práci implementovaný ukázkový bot dalece překonává své předchůdce ve směru spolehlivosti i výkonu a opakovaně dosáhl ultimátního cíle dokončení hry na různých veřejných serverech.

Kapitola 7

Závěr

7.1 Výsledky

V rámci práce byl vytvořen framework, který výrazně usnadňuje programování jednoduchých i sofistikovanějších botů pro NetHack a autorům botů umožňuje soustředění na programování umělé inteligence místo nutnosti zabývat se spojením se samotnou hrou a interpretací jejího textového rozhraní.

Framework nevyžaduje vlastní úpravy kódu hry a umožňuje tak provoz botů i na vzdálených serverech provozujících aktuálně nejběžněji používanou verzi hry. V rámci frameworku je poskytnuto mnoho volitelné funkčnosti, která výrazně usnadňuje řešení klíčových aspektů hry. Pro řešení problematické synchronizace s hrou byla navržena a implementována nová metoda s vysokou spolehlivostí a zanedbatelným vlivem na výkon v případě lokálních her.

Jde o první podobné řešení pro platformu JVM a ve srovnání s předchozími pokusy na jiných platformách nejpokročilejší. Projekt demonstruje praktičnost paradigmatů funkcionálního a logického programování v komplexní doméně, přestože nabízí také objektově orientované rozhraní pro jazyk Java. Toto rozhraní je zdokumentováno ve formě JavaDoc a tutoriálů a poskytuje rozšiřitelný základ pro další možné projekty v oblasti umělé inteligence pro hru NetHack.

V práci byl vytvořen také pokročilý ukázkový bot, který jako první NetHack bot vůbec dokázal hru úspěšně dokončit.

7.2 Možná budoucí rozšíření

Pro pohodlnější tvorbu složitějších botů se nabízí možnost implementovat rozšíření frameworku o moduly, které by pomáhaly s dalšími běžnými úkony, které může bot provádět. Největší mezery v tomto ohledu jsou v soubojích a ve správě inventáře.

Framework by mohl nabízet předpřipravené taktiky souboje proti různým typům protivníků, které by vhodně (dle nastavení) využívaly dostupnou výbavu. Podobně jako existující modul navigace pro pohyb a průzkum by toto odstínilo od nutnosti používat nízkoúrovňové příkazy v soubojích.

Velikou pomocí by pro programátory netriviálních botů byl také volitelný modul poskytující funkčnost pro automatický sběr předmětů a volbu výbavy. Tento modul by mohl na základě deklarativního popisu kýžené výzbroje (odpovídající hernímu stylu daného bota), možných alternativ a priorit spravovat výbavu, vždy využívat nejlepší dostupnou zbroj, nahrazovat její opotřebené a degradované části a brát přitom ohled na omezenou kapacitu inventáře. Inspirací pro tuto funkčnost může být kód ukázkového bota, který uvedené úkony řeší vlastním ad-hoc kódem.

Framework je možno rozšířit o podporu většího množství herních rolí NetHacku. Aktuálně jsou podporovány role *Valkyrie* a *Samurai*, které umožňují nejpřímočařejší herní styl. S poměrně malým úsilím by bylo možné přidat podporu dalších ze 13 povolání, které NetHack nabízí. V případě rolí, které mohou využívat kouzla by bylo nutné doimplementovat také související primitivní akce, pro které uvedené dvě role nemají využití.

Jak bylo nastíněno v kapitole 4.1, synchronizaci s NetHackem by bylo možné vylepšit využitím funkčnosti patche *vt_tiledata*, který není dostupný v oficiální verzi hry, ale nejběžněji používaná verze hry jej nabízí. Vylepšení současného řešení synchronizace by přispělo především ke zrychlení běhu botů na vzdálených serverech.

Další projekty v oblasti umělé inteligence a strojového učení mohou framework použít jako svou platformu. Nabízí se například možnost využít metody *reinforcement learning* pro adaptaci strategie soubojů s nepřáteli, čehož využíval v úvodu zmíněný bot Rogomatic [1] pro hru Rogue.

Noví boti mohou experimentovat s pokročilejšími modely chování, než je použit u ukázkového bota. Implementace komplexnějších strategií by byla možná např. s pomocí *behaviour trees* [7].

Další možností je použití metod klasického plánování pro optimalizaci postupu průchodu hrou s ohledem na podmínky konkrétního běhu.

Tyto metody lze implementovat čistě v uživatelském kódu bez zásadních změn frameworku samotného.

Literatura

[1] Mauldin M. L., Jacobson G., Appel A., Hamey L. (1984): *ROG-O-MATIC: A Belligerent Expert System*, Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence, London Ontario
http://www.cs.princeton.edu/~appel/papers/rogomatic.html

[2] Moore S. M., Luehrs J., O'Rear S., et al. (2009-2013): *The Tactical Amulet Extraction Bot* (TAEB)
http://taeb-nethack.blogspot.com

[3] Wahlberg V., O'Rear S., Karl R., Müller P. et al. (2008-2012): *Saiph*
https://github.com/canidae/saiph

[4] Churchill S. W., O'Donnell D. (2000-2008): *RGRN Frequently Asked Questions*, rec.games.nethack.roguelike Usenet newsgroup

[5] Moseley B., Marks P. (2006): *Out of the Tar Pit*, In proceedings of Software Practice Advancement 2006
http://shaffner.us/cs/papers/tarpit.pdf

[6] Kaplan H. (2004): *Persistent data structures*, In Handbook on Data Structures and Applications, Tel Aviv University, CRC Press, ISBN 978-1584884354
http://www.math.tau.ac.il/~haimk/papers/persistent-survey.ps

[7] Colledanchise M., Ogren P. (2014): *How Behavior Trees Modularize Robustness and Safety in Hybrid Systems*, In proceedings of Intelligent Robots and Systems 2014
http://www.cas.kth.se/~petter/Publications/IROS14_co.pdf

[8] Byrd W. E., Friedman D. P. (2009): *Relational programming in miniKanren: techniques, applications, and implementations*, Doctoral dissertation, Indiana University, Indianapolis
https://scholarworks.iu.edu/dspace/bitstream/handle/2022/8777/Byrd_indiana_0093A_10344.pdf

[9] Culberson J. (1997): *Sokoban is PSPACE-complete*, Technical Report TR 97-02, Dept. of Computing Science, University of Alberta
`http://webdocs.cs.ualberta.ca/~joe/Preprints/Sokoban/`

[10] Champandard A. J. (2003): *AI Game Development*, New Riders Games, ISBN 978-1592730049

[11] Russell, S., Norvig, P. (1995), *Artificial Intelligence: A Modern Approach*, Prentice Hall, ISBN 978-0131038059

Příloha A

Obsah CD

Na přiloženém CD se nachází následující adresáře s uvedeným obsahem:

- `bin` – spustitelné JAR soubory
- `BotHack` – obraz repozitáře zdrojových kódů k programu, pomocným skriptům a k dokumentaci
- `cljdoc` – HTML přehled Clojure rozhraní (v angličtině)
- `javadoc` – HTML přehled Java rozhraní (v angličtině)
- `text` – tato práce ve formátu PDF
- `video` – videonahrávka ukázkového běhu bota

Příloha B

Dokumentace

V této příloze se nachází základní dokumentace pro provoz a tvorbu vlastních botů s využitím frameworku popsaného v této práci.

Podrobnější dokumentace v anglickém jazyce s návody pro kompilaci zdrojových kódů frameworku a použití lokální instalace NetHacku se nachází v netištěné dokumentaci (*running.md* a *compiling.md* v adresáři *BotHack/doc*).

Poslední verze kompletní dokumentace je dostupná online v repozitáři na adrese: `https://github.com/krajj7/BotHack`

B.1 Spuštění běhu ukázkového bota

Zde je popsán způsob spuštění ukázkových botů v nejjednodušší konfiguraci, tj. proti veřejnému NetHack serveru (`nethack.xd.cm`). Framework je dostupný jako samostatný JAR soubor v adresáři *bin*. Pro běh je nutná pouze Java verze 7 nebo vyšší a konfigurační soubor, např. *BotHack/config/ssh-config.edn*. Dokumentace konfiguračních voleb je v souboru *BotHack/doc/config.md*.

V případě nefunkčnosti výchozích hodnot přihlašovacích údajů v uvedeném konfiguračním souboru může být pro hru na serveru `nethack.xd.cm` nutná nová anonymní registrace hráče. Tuto lze provést připojením přes SSH na adresu `nethack.xd.cm` s uživatelským jménem *nethack* (bez hesla). Zvolené přihlašovací údaje je třeba zapsat do použitého konfiguračního souboru. Po provedení registrace je také nutné upravit prostřednictvím SSH rozhraní serveru konfigurační soubor *nethackrc* nového účtu a nahradit jeho obsah souborem přiloženým na cestě *BotHack/bothack.nethackrc*.

Bot v jazyce Clojure

Příkaz pro spuštění běhu:

```
java -jar bothack-1.0.0-SNAPSHOT-standalone.jar ssh-config.edn
```

Bot v jazyce Java

Příkaz pro spuštění běhu:

```
java -jar bothack-1.0.0-SNAPSHOT-standalone.jar \
  BotHack/javabots/JavaBot/config/javabot-ssh-config.edn
```

Soubor *JavaBot.jar* z adresáře *bin* přitom musí být obsažen v CLASSPATH.

Alternativně lze bota spustit pomocí nástroje Maven příkazem:
`mvn -Pssh-run test` spuštěným z adresáře *BotHack/javabots/JavaBot*.

B.2 Tvorba vlastního bota

Na přiloženém CD a v repozitáři zdrojových kódů je dostupný tutoriál v angličtině, který podává vysokoúrovňový přehled funkcionality frameworku z pohledu programátora bota (*BotHack/doc/tutorial.md*).

Při tvorbě vlastního bota v jazyce Java lze využít funkční kostry Maven projektu dostupné v adresáři *BotHack/javabots*. Funkční příklady botů v Clojure včetně hlavního ukázkového bota jsou pak v adresáři *BotHack/src/bothack/bots*.

Podrobná dokumentace Java rozhraní v JavaDoc HTML podobě je v adresáři *javadoc* a dostupná online na na adrese `http://krajj7.github.io/BotHack/javadoc/`

Dokumentace Clojure funkcí je online na adrese `http://krajj7.github.io/BotHack/cljdoc/`