Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Využitie všeobecného počítačového hráča v spoločenských hrách

Bakalárska práca

2021
Vladimír Bačinský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Využitie všeobecného počítačového hráča v spoločenských hrách

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	RNDr. Jozef Šiška, PhD.

Bratislava, 2021
Vladimír Bačinský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Vladimír Bačinský
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Využitie všeobecného počítačového hráča v spoločenských hrách
General game playing and desktop games

Anotácia: General game playing sa venuje tvorbe algoritmov schopných hrať akúkoľvek hru podľa všeobecného popisu. Najrozšírenejší je jazyk GDL (game description language), ktorý umožňuje popisovať konečné hry s úplnou informáciou, bez náhody. Jeho rozšírenie, GDL-II, umožnuje popisovať hry s neúplnou informáciou a prvkami náhody.
Existuje veľa implementácií GGP algoritmov nad GDL a taktiež veľa súťaží, kde sa tieto algoritmy porovnávajú, avšsak už menej algoritmov pre GDL-II. Pri ich evaluácii sa používajú skôr hry so striktnejšími pravidlami a jasnými cieľmi, ale bolo by zaujímavé vyskúšať ich aj na komplexnejšej hre dizajnovanej pre ľudí.

Cieľ: Popísať / implementovať vybranú spoločenskú hru v jazyku GDL / GDL-II a vyskúšať na nej rôzne implementácie GGP hráčov, ideálne vo verzii s neúplnou informáciou a náhodou.

Literatúra: Love, N; Genesereth, M; Hinrichs, T. General game playing: game description language specification. Technical Report, Rep. LG-2006-01", Stanford University. 2006

Thielscher, M. A general game description language for incomplete information games. AAAI, AAAI Press. 2010

Kľúčové slová: General Game Playing, Desktop Games, Game design

Vedúci: RNDr. Jozef Šiška, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 16.11.2020

Dátum schválenia: 04.01.2021 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Chcem sa poďakovať svojmu školiteľovi RNDr. Jozefovi Šiškovi, PhD. za cenné rady, odbornú pomoc a usmernenie pri písaní bakalárskej práce.

Abstrakt

Všeobecné hranie hier je systém schopný hrať hocijakú hru popísanú v nejakom jazyku na popis hier. Medzi takýto najznámejší jazyk patrí Game Desription Language (GDL), ktorý sa syntaxou podobá prológu. Táto práca sa venuje prevodu pravidiel hry Slovania obchodníkmi do pravidiel logického programovacieho jazyka GDL. Slovania obchodníkmi je stolová strategická hra určená pre 2 hráčov, pričom hráči v nej pohybom svojich figúrok po hracej ploche plnia misie. Následne implementáciu pravidiel spomínanej hry v GDL otestujeme na všeobecných počítačových hráčoch a v krátkosti rozanalyzujeme výsledky.

Kľúčové slová: všeobecný počítačový hráč, stolová hra, logické programovanie, GDL

Abstract

General Game Playing (GGP) is a system able to play any game written in some language that describes games. Game Description Language (GDL), whose syntax is similar to Prolog, is one of the most known languages of that type. This bachelor thesis deals with translation of the rules from the game Slovania obchodnikmi to logic programming language of GDL. Slovania obchodnikmi is strategic board game for 2 players where they move their pawn on the board. We test our implementation of GDL rules for the aforementioned game on GGP and we will shortly analyze the results.

Keywords: general game playing, desktop game, logic programming, GDL

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

História spoločenských hier siaha do dávnej ľudskej minulosti. Prvé hry sú dátované do roku 4000 pred naším letopočtom. Od tejto doby sa postupne vyvíjali a boli neoddeliteľnou súčasťou všetkých kultúr. Hráči sa počas ich hrania primárne zabávajú, ale môžu zlepšovať aj napríklad strategické myslenie pri strategických hrách alebo sa naučiť nové informácie pri vzdelávacích hrách.

Postupným rýchlym vývojom počítačov sa aj hry stali ich súčasťou. V dnešnej dobe existuje mnoho počítačových hráčov, avšak takmer všetci boli vytvorení na základe pravidiel konkrétnej hry. Všeobecní počítačoví hráči (označujeme GGP z anglického General Game Playing) vznikli, aby dokázali hrať efektívne viacero hier, avšak hry musia byt popísané v nejakom jazyku na popis hier, napriklad GDL (celým názvom Game Description Language). GDL je logický programovací jazyk, ktorý popisuje stavy hry ako množinu faktov a herné mechanizmy popisuje pomocou logických pravidiel.

Cieľom tejto bakalárskej práce je implementovať spoločenskú hru Slovania obchodníkmi v jazyku GDL a následne na nej vyskúšať GGP hráčov. Slovania obchodníkmi je strategická stolová hra vytvorená autorom. Hra je určená pre dvoch hráčov a obsahuje hraciu plochu, ktorá je podobná hre Človeče, nehnevaj sa. Cieľom spoločenskej hry je získať 20 bodov za dokončenie misií, ktoré hráči môžu plniť pohybom svojej figúrky po hracej ploche.

V prvej kapitole sa oboznámime, čo to vlastne všeobecný počítačový hráč je a vysvetlíme si základne algoritmy, ktoré používa a zároveň si zadefinujeme jazyk GDL, ktorý všeobecní počítačoví hráči využívajú. V druhej kapitole si podrobne vysvetlíme hru Slovania obchodníkmi a jej pravidlá. V tretej kapitole popíšeme implementáciu spomínanej hry v jazyku GDL. V poslednej kapitole sa zameráme na testovanie hry Slovania obchodníkmi a následným skúšaním všeobecných počítačových hráčov na nej.

Kapitola 1

Úvod do problematiky

V tejto kapitole si zadefinujeme základné pojmy, ktoré budeme používať. Predstavíme si, čo je to všeobecný počítačový hráč a oboznámime sa s najdôležitejším nástrojom práce a to jazykom na popis hier Game Description Language (GDL).
V nasledujúcom texte budeme pod *neohodnoteným grafom* uvažovať dvojicu $G = (V, E)$, pričom V je množina elementov, ktoré nazývame vrcholy a E je množina dvojíc vrcholov a pod pojmom *strom* budeme rozumieť súvislý acyklický graf, teda graf neobsahujúci kružnicu a medzi každou dvojicou vrcholov existuje cesta.

1.1 Všeobecný počítačový hráč

Pre mnoho spoločenských hier existujú programy a algoritmy, ktoré nám umožňujú ich hranie, napríklad program hrajúci buď šach alebo dámu. Ak by sme chceli program, ktorý umožňuje hranie aj dámy aj šachu, tak tento program by potreboval vedieť pravidlá oboch hier dopredu. Práve najväčšou výhodou všeobecného počítačového hráča je, že dokáže hrať hry napísané v nejakom jazyku na popis hry bez toho, aby poznal ich pravidlá vopred. Medzi jeden z najznámejších jazykov popisu hry patrí jazyk GDL. GGP je teda schopný hrať viacero druhov hier, vrátane dvoch vyššie spomenutých. Musia však mať spoločnú štruktúru a to konkrétne, že hra musí mať konečný počet hráčov, konečný počet stavov a konečný počet akcií v ktoromkoľvek stave. Navyše jeden zo stavov musí byť počiatočný [7]. Aby hra bola hrateľná, tak musí existovať aspoň jeden taký stav, ktorý je koncový a zároveň existuje taká postupnosť legálnych ťahov, že sa pomocou nich vieme dostať z počiatočného stavu do koncového. Formálne môžeme reprezentovať model hry nasledovne:
Nech n je počet hráčov a S je množina stavov, pričom

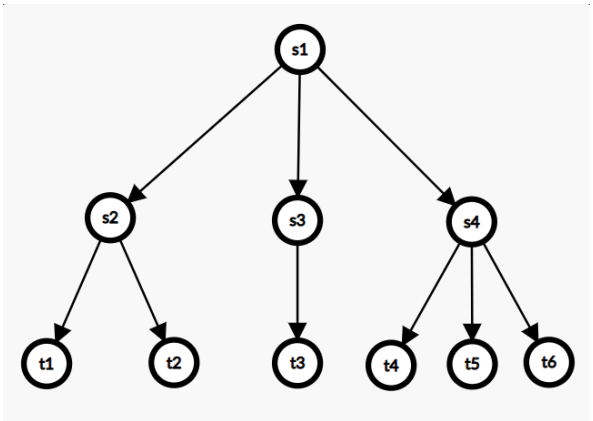
- $A_1, \ldots, A_n$ - A_i je konečná množina akcií (legálnych ťahov) i-teho hráča, $i \in \{1, \ldots, n\}$

- $f : S \times A_1 \times \ldots \times A_n \longrightarrow S$ - funkcia na aktualizovanie, teda slúži na prechod z jedného stavu do druhého, pomocou legálnych ťahov hráčov
- $s_1 \in S$ - s_1 je počiatočný stav
- $T \subseteq S; T \neq \emptyset$ - T je množina terminálnych, teda koncových stavov

Takýto formálnejší model hry využijeme neskôr pri demonštrácií konkrétných algoritmov, ktoré GGP hráči využívajú. Všeobecní počítačoví hráči sa líšia stratégiou, pomocou ktorej vyberajú akcie, teda legálne ťahy v aktuálnom stave. K najtriviálnejším patria Legálny Hráč (značíme LegalPlayer) a Náhodný Hráč (značíme RandomPlayer). LegalPlayer funguje jednoducho a to tak, že z pomedzi zoznamu legálnych ťahov vyberie prvý v poradí. RandomPlayer hrá taktiež elementárne a zo zoznamu akcií vyberá náhodne. Existujú však aj omnoho komplikovanejší hráči, ktorí pri vyberaní ťahu využívajú efektívne algoritmy, ktoré im pomáhajú vybrať najoptimálnejšiu akciu a práve s niektorými z nich sa oboznámime. V našej práci sa budeme zaoberať hrou, ktorá je určená pre dvoch hráčov a je konkurenčná, teda víťazom sa stáva len jeden hráč. Narozdiel od hier určených pre jedného hráča, v tých viacnásobných je výber najlepšieho ťahu závislý od potenciálneho budúceho ťahu protihráča. Navyše žiaden hráč nevie priamo odhadnúť ťah, ktorý súper zahrá, preto musíme počítať so všetkými možnými akciami protihráča. Celú hru potom môžeme reprezentovať ako strom, pričom koreňom je počiatočný stav, vnútorné vrcholy sú stavy, ku ktorým sa vieme pomocou funkcie f dostať z počiatočného stavu, listy sú terminálne stavy a orientované hrany reprezentujú prechod medzi stavmi pomocou funkcie f. Jednoduchá reprezentácia by mohla vyzerať ako na obrázku 1.1. Počet koncových stavov sa rovná počtu všetkých možných priebehov danej hry a ich hodnota sa rovná počtu dosiahnutých bodov v tomto stave. Na výber optimálneho ťahu v aktuálnom stave existuje napríklad algoritmus nazývaný Minimax.

Minimax

Jeho funkcionalitu budeme demonštrovať na príklade, kde hráč zahrá jeden ťah a po ňom nasleduje hneď súper, teda ak je hráč na ťahu v hĺbke h, tak v hĺbke $h+1$ bude na ťahu súper. Na pochopenie jeho funkčnosti nám pomôže obrázok 1.2. Aby Minimax fungoval, potrebuje mať ohodnotené listy v strome, teda v našom prípade to sú ohodnotené terminálne stavy. Následne sa hráč na ťahu v aktuálnom stave snaží pomocou Minimaxu maximalizovať svoj optimálny legálny ťah, za predpokladu, že súper si vždy vyberie pre seba taktiež najlepší ťah. Algoritmus Minimax začína počítať v o jedna menšej hĺbke celého stromu. Všetky stavy v tejto hĺbke reprezentujú ťah súpera. Pre protihráča je teda najefektívnejšie vybrať minimum z hodnôt svojich synov (vrcholy, do ktorých vedie hrana z daného vrcholu) v o jedna väčšej hĺbke a naopak pre nás je najlepšie vybrať maximum zo



Obr. 1.1: Reprezentácia hry stromom, pričom vrchol v hĺbke 0 je počiatočný stav, vrcholy v hĺbke 1 sú stavy, ktoré vznikli prechodom z počiatočného stavu pomocou funkcie f a vrcholy v hĺbke 2 sú koncové stavy. Šípky v tomto strome znázorňujú jednotlivé prechody medzi stavmi a to na základe funkcie f, teda legálnych ťahov každého hráča v danom stave.

synov. Analogicky postupným znižovaním hĺbky a striedaním maxima a minima v každej hĺbke sa dostaneme až k aktuálnemu stavu, kde sme dostali najväčšie číslo z vrcholov v o jedna väčšej hĺbke, teda najoptimálnejší ťah je práve do vrchola s najväčšou hodnotou, ktorú minimax vypočítal.

Nevýhodou tohto algoritmu je, že zakaždým musí prehľadať celý strom, ktorý má vo všeobecnosti veľkú hĺbku, lebo ťahov, ktoré vedú od počiatočného do koncového stavu može byť veľa a preto existuje vylepšený Minimax.

Optimalizovaný Minimax

V prípadoch takých stromov, kde poznáme absolútnu maximálnu respektíve minimálnu možnú hodnotu v terminálnych stavoch, tak nebudeme musieť vždy prehľadávať celý strom. Napríklad ak vieme, že absolútna minimálna hodnota je 0 a zároveň algoritmus Minimax v stave, kedy počíta minimum svojich synov (takýto stav označujeme minnode) má v jednom z nich 0, tak logicky už nemusí hľadať v ďalších synoch, pretože menšiu hodnotu už nenájde, teda vráti 0. Analogicky ak vieme, že napríklad absolútna maximálna hodnota v koncových stavoch je 100, tak ak Minimax v stave, kde počíta maximum (označujeme maxnode) nájde v jednom zo svojich synov hodnotu 100, rovnako už nemusí hľadať ďalej, pretože väčšiu hodnotu už nenájde a preto vráti 100. Tento optimalizovaný Minimax nám vie pomôcť v niektorých prípadoch, avšak ukážeme si ešte silnejšiu verziu tohto optimalizovaného minimaxu a to Alfa-Beta vyhľadávanie.

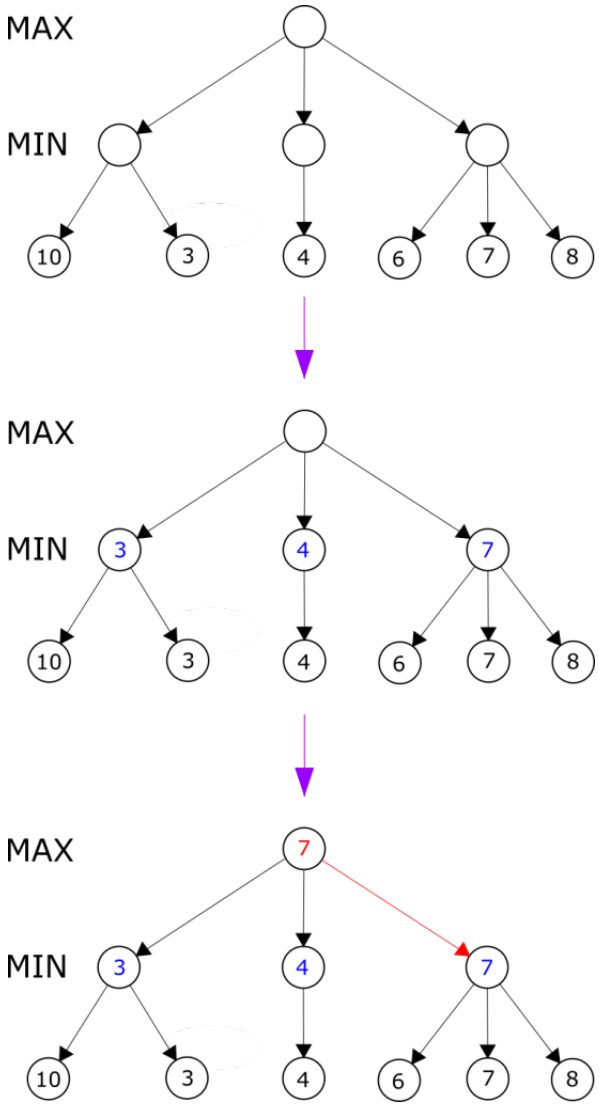
Alfa-Beta vyhľadávanie

Tento algoritmus si počas behu udržiava doposiaľ najmenšiu nájdenú hodnotu

nazývanú alfa (označujeme α) a najväčšiu nájdenú hodnotu nazývanú beta (označujeme β). Algoritmus môžeme rozdeliť na dva prípady, jeden ak vylúčime vetvu stromu v minnode a druhý v maxnode. Rozoberme si teda prvú možnosť a to ak v minnode nájde Minimax v synovi väčšiu hodnotu ako α (označme túto hodnotu x), tak potom už nebude musieť prehľadávať ďalších synov a jednoducho vráti hodnotu α. Ak by aj v ďalších synoch bola väčšia hodnota ako x, tak tá je pre nás irelevantná, pretože minnode aj tak vyberie minimum, teda α. Čo v prípade, ak v ďalších synoch je menšia hodnota ako x? Tá sa taktiež javí ako zbytočná, pretože vieme, že neskôr v o jedna menšej hĺbke bude následne maxnode vyberať maximálnu hodnotu zo synov a tam vždy vyberie α pred tou našou menšou hodnotou v synovi. Tým sme ukázali, že obidva prípady synov, už nebudeme potrebovať a preto túto vetvu nebudeme musieť ďalej prehľadávať.
Podobne budeme postupovať v druhej možnosti, ak v maxnode nájde Minimax v synovi väčšiu hodnotu ako beta (označme túto hodnotu y). V prípade, že v ďalších synoch je menšie číslo ako y, tak je táto vetva zbytočná, pretože maxnode vyberie maximum. Ak je v neprehľadaných synoch väčšia hodnota ako y, tak tá je taktiež nepodstatná, pretože v ďalšom kroku bude Minimax v o jedna menšej hĺbke, teda bude v minode vyberať minimum a tam opäť zvíťazí stav s hodnotou y.
Neskôr v tejto bakalárskej práci budeme testovať spoločenskú hru práve na GGP hráčoch a jedným z nich je hráč, ktorý využíva Monte Carlo Vyhľadávanie.

Monte Carlo Vyhľadávanie

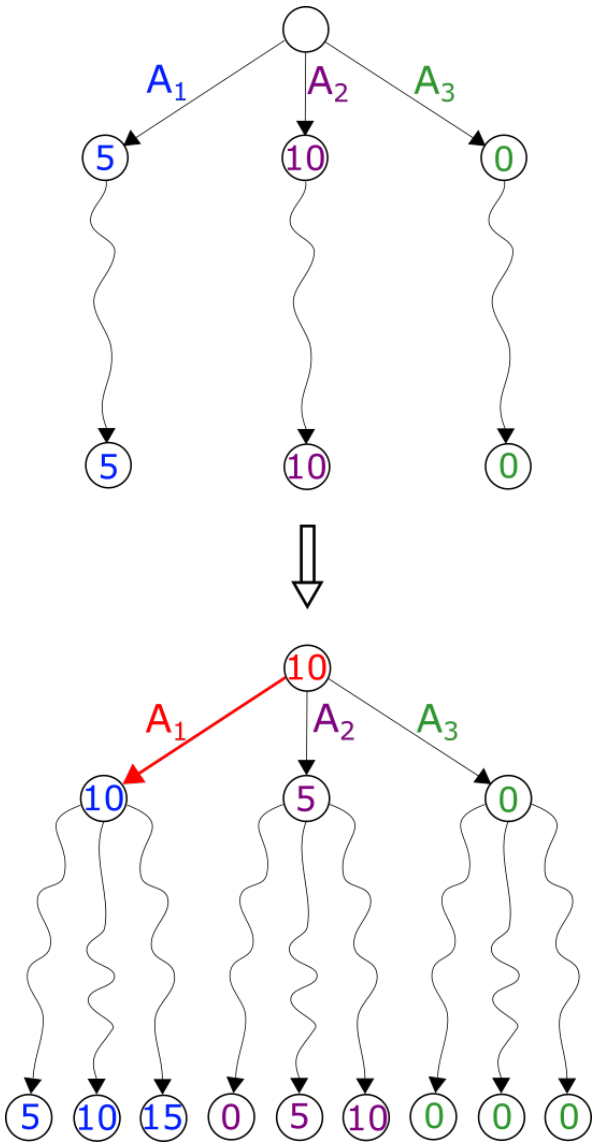
Tento algoritmus označovaný tiež ako MCS (z anglického Monte Carlo Search) sa snaží nájsť, rovnako ako vyššie spomínané algoritmy, optimálny legálny ťah pre hráča na ťahu v aktuálnom stave. Pre lepšie pochopenie nám pomôže obrázok 1.3. Ak všeobecný počítačový hráč hrá hru napísanú v GDL, tak má vždy stanovený nejaký časový limit, kým sa musí rozhodnúť pre výber svojho ťahu. MCS spustí simuláciu postupne pre každú možnú akciu v danom stave, teda v každom synovi. V simulácii algoritmus vyberá ťahy náhodne až kým sa nedostane k terminálnemu stavu s hodnotou totožnou s počtom dosiahnutých bodov na konci hry. Túto hodnotu si uloží ako ohodnotenie daného syna a pokračuje z aktuálneho stavu analogicky ďalšími synmi. Ak prejde všetkých synov, tak simuláciu začne znovu chronologicky od najľavejšieho syna a postup opakuje s takým rozšírením, že hodnotu ktorú našiel v koncovom stave pripočíta k danej hodnote syna a zároveň si zapamätá počet simulácii z daného stavu. Ak sme dosiahli časový limit, tak postupne MCS v každom synovi predelí jeho hodnotu počtom opakovaní a následne vyberie z týchto hodnôt maximum. Hrana, ktorá smeruje



Obr. 1.2: Ukážka simulácie minimaxu na jednoduchom strome. Minimax vypočítal legálny ťah reprezentovaný červenou (doprava) šípkou.

k tomuto maximu je náš optimálny ťah, ktorý MCS vyberie.

Ukázali sme si základné algoritmy, ktoré všeobecní počítačoví hráči môžu využívať pri vyberaní najoptimálnejšieho ťahu v aktualnom stave.
Zaujímavosťou v oblasti GGP je, že už od roku 2005 sa na konferencii AAAI (z anglického Association for the Advancement of Artificial Intelligence) konajú každoročné GGP súťaže. V prvej časti súťaže sa zúčastnení hodnotia na základe ich schopnosti vykonávať legálne ťahy, získavať prevahu a čo najefektívnejšie dokončiť hry. V druhom kole sa účastníci stretávajú proti sebe v čoraz zložitejších hrách. Program, ktorý v tejto fáze vyhráva najviac hier, tak víťazí v súťaži a do roku 2013 jej tvorca získaval dokonca cenu 10 000 dolárov [1].



Obr. 1.3: Ukážka simulácie Monte Carlo, pričom A_1, A_2 a A_3 sú legálne akcie hráča na ťahu. MCS v tomto prípade simuluje náhodne ťahy cez všetkých troch synov spolu trikrát. Postupne v každom synovi počiatočného stavu následne sčíta hodnoty vo „svojich“ listoch a predelí súčet počtom simulácií a hodnotu pridá danému synovi. MCS potom už len vyberie maximum z týchto synov a práve hrana, ktorá smeruje do maxima je náš optimálny ťah.

1.2 Jazyk General Game Description

GDL bol vyvinutý, aby vykonávanie logických úloh pre všeobecné hranie hier bolo efektívne. GDL popisuje stavy hry ako množinu faktov a množinu logických pravidiel [6]. Napríklad, ak máme fakty

$$rodic(jozef, fero)$$
$$rodic(fero, julka)$$

tak je faktom, že Jozef je rodičom Fera a Fero je rodičom Julky. Ak zároveň máme pravidlo

$$staryrodic(X, Z) :- \quad rodic(X, Y), rodic(Y, Z)$$

ktoré hovorí, že ak X je rodičom Y a zároveň Y je rodičom Z, tak potom X je starým rodičom Z, tak z našich faktov a pravidla môžeme odvodiť

$$staryrodic(jozef, julka)$$

teda Jozef je starý rodič Julky.

Zadefinujme si teraz formálnejšie základné pojmy používané v GDL, term a pravidlo podľa [6]. Medzi slovnú zásobu v GDL patria objektové konštanty (označujeme písmenami zo začiatku abecedy: a, b, c), funkčné konštanty (označujeme malými písmenkami abecedy: f, g, h), relačné konštanty (označujeme malými písmenkami abecedy: p, q, r) a premenné (označujeme veľkými písmenami, prevažne z konca abecedy: X, Y, Z).

Termom v GDL sú objektové konštanty, premenné alebo funkčné koštanty arity n aplikované na n termov (označujeme $f(a, X, b)$). *Atóm* je relačná konštanta aplikovaná na n termov - označujeme $p(a, f(a, b))$. Negácia je výraz, ktorý sa skladá zo znakov negácie a atómu - označujeme $\neg p(a, f(a, b))$. Literál je atóm alebo negácia atómu - označujeme $p(a, f(a, b))$ alebo $\neg p(a, f(a, b))$.

Pravidlo v GDL sa skladá z hlavy a zoznamu literálov, ktoré sú oddelené čiarkou, označujeme

$$h :- \quad l_1, ..., l_n$$

Pravidlo vyhodnocujeme ako konjunkciu [1] literálov. Zoznam literálov nazývame telo. Inštancia v hlave je pravdivá vtedy, ak všetky pozitívne literály sú pravdivé (true) a všetky negatívne literály sú nepravdivé (false). Ešte pred tým ako si prejdeme postupne všetky základné relácie Jazyka Popisu Hry, tak si ukážeme konkrétnu syntax v GDL nazývanú formát výmeny vedomostí (označujeme KIF z anglického Knowledge Interchange Format). V KIF pred premenné píšeme otáznik, teda premmenna X je písaná ako `?x`. Funkčnú konštantu $f(a, X)$ píšeme ako `(f a ?x)` alebo $true(control(X))$ píšeme ako `(true (control x))`. Negáciu $\neg(true(control(X)))$ píšeme ako `(not (true (control x)))` a pravidlo $p(X) \Leftarrow q(X) \wedge \neg r(X)$ píšeme ako `(<= (p ?x) (q ?x) (not (r ?x)))` [6]. Začiatok komentárov značíme bodkočiarkou.

[1] konjunkcia (označujeme $\wedge$) je operátor logickej spojky. Množina operandov *and* je pravdivá, ak všetky operandy sú pravdivé

Medzi základné relácie v GDL patria: *role, true, init, next, legal, does, goal a terminal.* Poďme si ich postupne všetky vysvetliť. Príklady budeme demonštrovať na jednoduchej hre piškvorky 3x3 (Tic-Tac-Toe). Hru hrajú dvaja hráči (označme ich hráč x a hráč o) na hracej ploche štvorca, v ktorom je 9 menších štvorcov, teda 3x3. Ťahy hráčov spočívajú v položení, respektíve nakreslení svojho znaku na konkrétny štvorec, pričom po nakreslení nasleduje ťah druhého hráča. Na každom políčku hracej plochy môže byť maximálne jeden znak. Cieľom hry je získať trojicu svojich znakov, buď v niektorom z riadkov, stĺpcov alebo na diagonálach. V prípade, že niektorý z hráčov získa spomínanú trojicu, tak vyhráva a hra končí.

Predikát *role* definuje hráčov hry, teda `(role x)` znamená, že jedným z hráčov tejto hry je x.

Predikát *true* (*true(fact)*) popisuje skutočnosti, ktoré sú pravdivé v aktuálnom stave. Potom `(true(cell 1 1 b))` znamená, že na hracej ploche na pozíciii 1 1 v bunke je blank [2].

Predikát *init* je analogický s *true*, ale je určený len pre začiatok hry, teda pre počiatočný stav. Príkladom v piškvorkách tohto predikátu je `(init(cell x y b))` $\forall x, y \in \{1, 2, 3\}$, pretože na začiatku Tic-Tac-Toe sú všetky bunky hracej plochy prázdne.

Relácia *next* je analogická s *true*, ale predstavuje fakty, ktoré budú platiť v nasledujúcom stave. Nech `(control x)` znamená, že na ťahu je hráč x, potom `(<= (next (control o)) (true (control x)))` znamená, že v nasledujúcom stave bude platiť, že hráč o bude na ťahu, ak je pravda, že hráč x je na ťahu v aktuálnom stave.

Existuje špeciálna akcia *noop*, ktorá indikuje, že hráč neurobil žiadnu akciu.

Relácia *legal*: (*legal*(*hrac*, *krok*)) predstavuje legálne kroky hráčov v aktuálnom stave. Napríklad `(<= (legal x noop) (true (control o)))`, čiže legálnym krokom pre hráča x je akcia *noop*, ak je práve na ťahu druhý hráč o.

Relácia *does* indikuje kroky vykonané hráčmi na ťahu. Nech `(mark x y)` je faktom, ktorý znamená krok zaznačenia (v reálnej papierovej verzii nakreslil svoj znak) na pozícii x y, potom pravidlo `(<= (next (cell 1 1 x)) (does x (mark 1 1)))` hovorí o tom, že v ďalšom stave hry bude na pozícii 1 1 v bunke symbol `x`, ak platí mark, teda že hráč x zaznačil na pozícii 1 1.

Predikát *goal* určuje odmenu pre hráčov, napríklad za dokončenie úlohy alebo ako v Tic-Tac-Toe za výhru. Nech `(line ?player)` je true práve vtedy ak je nejaký riadok, stĺpec alebo diagonály zaplnený rovnakými znakmi alebo po diagonálach, potom `(<= (goal ?player 100) (line ?player))` znamená, že hráč dostáva 100 bodov, ak platí *line*.

Relácia *terminal* definuje koniec hry. Nech *open* hovorí o tom či je ešte nejaký blank na hracej ploche, potom `(<= terminal (not open))` hovorí, že koniec hry nastane

[2] prázdnota, prázdne miesto

napríkald vtedy, ak už na hracej ploche nie je bunka s blankom.

Kapitola 2

Spoločenská hra "Slovania obchodníkmi"

V tejto kapitole podrobne rozoberieme spoločenskú hru s názvom Slovania obchodníkmi, ktorú sme vytvorili v rámci ročníkového projektu [2]. Popíšeme akého typu táto hra je, z akých pravidiel sa skladá, čo je jej cieľom, aké má špecifické prvky a podobne.

2.1 Úvod k hre

Slovania obchodníkmi je typická strategická stolová hra. Hrá sa na konkrétnom hracom pláne, po ktorom sa hýbu figúrky hráčov podľa nejakých pravidiel. Hrací plán je špecifický pre tento konkrétny druh hry a vždy je rovnaký. Je podobný hre Človeče, nehnevaj sa, ale s tým rozdielom, že figúrky hráčov sa môžu hýbať oboma smermi. V princípe hracím plánom je jednoduchý neohodnotený graf. Ak hráč hodil na kocke číslo 6, tak si následne môže vybrať každý vrchol vo vzdialenosti práve 6, alebo hrad/mesto vo vzdialenosti menej ako 6. Hráč nemá úplnú kontrolu nad pohybom svojej figúrky, pretože hádže kockou a tým tu vstupuje činiteľ náhody, teda táto hra patrí k nedeterministickým. Úlohou každého hráča je plniť misie. Misia sa skladá zo začiatočného mesta alebo hradu, kde konkrétny tovar musí hráč vyzdvihnúť a končiaceho mesta alebo hradu, kde zásielku doručí. Každá misia je obodovaná podľa dĺžky najkratšej cesty (teda počtu políčok najkratšej cesty). Napríklad môžeme mať misiu, ktorej úlohou je doručiť kožu z Šarišského hradu do hradu Šašov, pričom počet políčok najkratšej cesty je 20 a odmenou je 5 korún. Plnenie misií však v tejto hre nie je také jednoduché, pretože hráči si môžu kúpiť negatívne(škodiace) žetóny, ktoré neskôr môžu vkladať na políčka ciest.
Cieľom každého hráča je získať 20 bodov pomocou plnenia misií.

2.2 Pravidlá

Hra je určená pre 2 hráčov. Na ťahu je vždy práve jeden hráč. Jeho ťah sa začína hodením kocky (označme hodnotu čísla hodeného na kocke x), následne si hráč musí vybrať políčko kam presunie svoju figúrku, avšak môže ju posunúť len na políčka cesty, ktoré sú vo vzdialenosti presne x od aktualnej pozície figúrky a políčka hradov, ktoré sú vo vzdialenosti maximálne x. Potom hráč môže nakupovať negatívne žetóny, ktoré nemusí nutne v tom istom ťahu klásť na hraciu plochu, alebo pozitívne žetóny. Slovania obchodníkmi sa skladajú z týchto prvkov:

Hracia plocha

Je neohodnotený graf, v ktorom vrcholy sú všetky hrady/mestá a cesty a hrany sú medzi tými vrcholmi, ktoré sú susedné. Ak napríklad hrad A má vstupnú aj výstupnú jedinú cestu, tak hrana je medzi vrcholom(hradom) A a susedným políčkom cesty.

Misie

Každá misia sa skladá zo začiatočného(A) a konečného(B) hradu/mesta, počtu políčok najkratšej cesty z A do B, odmeny a názvu predmetu, konktrény príklad nájdeme v tabuľke 2.1.

Materiál	Z hradu/mesta	Do hradu/mesta	Počet políčok	Odmena
Železo	Šarišský hrad	Fiľakovo	28	5
Koža	Spišský hrad	Bratislava	24	4

Tabuľka 2.1: Ukážka dvoch misií.

Negatívne žetóny

Negatívne žetóny sa nemôžu vkladať na políčka hradov alebo miest. Sú určené len na políčka ciest. Na každom políčku cesty môže byť maximálne jeden negatívny žetón. Žetóny sú 4 farieb, pretože chceme aby boli medzi hráčmi odlíšiteľné. Ak hráč stúpi na políčko, na ktorom je cudzí negatívny žetón, tak sa negatívny žetón zoberie z tohto políčka preč. Negatívne žetóny sú 3 typov:

- Z–Zbojníci: Ak hráč vstúpi na políčko, na ktorom je cudzí žetón Z, tak ho zbojníci olúpia o náklad. To znamená, že ak hráč už vyzdvihol svoj materiál, tak na dokončenie misie sa poň bude musieť vrátiť.
- RC–Rozbitá cesta: Ak hráč stúpi na políčko s týmto negatívnym žetónom a taktiež je iného hráča, tak putuje do väzenia.
- B–Bažina: Ak hráč vstúpil na políčko, na ktorom je cudzí žetón B, tak sa vracia na políčko odkiaľ prišiel.

Pozitívne žetóny

Hráč si pozitívne žetóny môže kúpiť na svojom ťahu. Ich funkcionalitou je brániť sa voči negatívnym žetónom. Ku každému negatívnemu existuje pozitívny, ktorý slúži na ubránenie voči negatívnemu. Konkrétne:

- Ochranka: ochrana pred zbojníkmi
- Nové koleso: imunita voči rozbitej ceste
- Sprievodca: imunita voči bažine

V prípade, že hráč má nakúpený správny (napríklad stúpil na políčko s bažinou a má sprievodcu) pozitívny žetón a zároveň stúpi na políčko, na ktorom je cudzí negatívny, tak sa automaticky vyhne efektu negatívneho žetónu a svoj pozitívny žetón stráca. Následne sa negatívny žetón odstráni z políčka.

Peniaze

Platidlom v tejto hre je mena koruna. Každý negatívny žetón stojí 1 korunu a každý pozitívny žetón stojí 2 koruny. Hráč začína s desiatimi korunami a za každú ďalšiu splnenú misiu dostane toľko korún, koľko je odmena danej misie.

Väzenie

Ak hráč stúpil na poličko, na ktorom je bažina cudzieho hráča a zároveň nemá pozitívny žetón sprievodcu, tak putuje do väzenia. Automaticky končí svoj ťah, teda nemôže nakupovať žetóny ani vkladať negatívne na hraciu plochu a svoj ťah začína protihráč, ktorý po skončení ťahu nasleduje ešte raz. Následne ak ukončí ťah druhýkrát po sebe tak nasleduje hráč, ktorý bol vo väzení. V prípade, že hráč A na svojom ťaju stúpil na bažinu a nemá sprievodcu a následne hráč B na svojom ťahu putuje do väzenia, tak zjavne nepôjde dvakrát po sebe, ale hneď ukončí svoj ťah a nasleduje normálne hráč A a obaja sú z väzenia von.

Nakoľko pozícia figúrky sa rovná začiatočnému hradu misie, tak na splnenie prvej misie stačí ak hráč donesie materiál len do cieľového hradu/mesta. Ak hráč vyzdvihol tovar a jeho figúrka sa nachádza v cieľovej destinácii misie, tak automaticky získava toľko bodov a toľko peňazí aká je odmena misie a daný hráč dostane náhodne ďalšiu misiu na splnenie. Hráč môže použiť svoj kúpený negatívny žetón v ktoromkoľvek svojom ťahu, teda nie je povinný ho použiť vtedy, kedy ho aj kúpil. Na konkrétnom políčku hracej plochy môžu byť v jednom momente figúrky všetkých hráčov.

Príprava hry: Každému hráčovi je na začiatku hry náhodne pridaná misia a figúrka sa postaví na hrad/mesto, v ktorom začína hráčova misia. Následne obaja hráč dostaú po 10 korún. Potom hráči hádžu kockou a svoj ťah začína hráč, ktorý hodil väčšie číslo.

Koniec hry: Ak niektorý z hráčov získa 20 a viac bodov, tak hra sa automaticky ukončí a víťazom sa stáva hráč, ktorý tento limit dosiahol.

Kapitola 3

Implementácia hry v GDL

V tejto kapitole popíšeme implementáciu hry Slovania obchodníkmi v jazyku GDL. V prvej časti si zadefinujeme potrebné aritmetické operácie a potom samotnú hru.

3.1 Aritmetické operácie

Narozdiel od iných programovacích jazykov, kde čísla a základné aritmetické operácie sú samozrejmosťou, GDL je jedinečný v tom, že ich nepozná a je nutné si ich dodefinovať. Práve preto, ešte pred písaním konkrétnych pravidiel týkajúcich sa hry, musíme zaviesť tieto operácie.

3.1.1 Číslo

V našej hre budeme pracovať len s prirodzenými číslami vrátane 0. Ich základnou vlastnosťou je následnosť. Definujme teda predikát nasledovník(succ z anglického succeed) nasledovne:

```
(succ 0 1)
(succ 1 2)
```

Čiže 1 je nasledovníkom 0, 2 je nasledovníkom 1 a tak ďalej. Predikát *succ* budeme v našej práci veľmi často využívať. Je dobrým nástrojom na rýchle zmenšenie alebo zväčšenie čísla o 1. Taktiež tento predikát bude výborným nástrojom pri definovaní ďalších operácií ako sčítavanie alebo väčší než. Definícia čísla x potom vyzerá takto:

```
(<= (number ?x) (succ ?x ?y))
(<= (number ?x) (succ ?y ?x))
```

Alebo je y nasledovníkom x, alebo x je nasledovníkom y.

3.1.2 Sčítavanie

Sčítavanie patrí k najzákladnejším operáciam v programovacích jazykoch a nevyhneme sa mu aj v našej práci. Pri definovaní predikátu *pluss* si vystačíme s operáciou nasledovníka a jednoduchou rekurziou. Prvá časť pravidla definuje triviálny prípad $x+0 = x$. Druhá časť reprezentuje $x + y = result$. Nakoľko vieme, že operácia sčítania je komutatívna, tak môžeme napríklad x zväčšiť o 1 a y zmenšiť o 1 a následne sa rekurzívne zavolať. Tým dosiahneme, že sa y postupne rekurziou dostane až na 0 a následne postupným vynorením z rekurzie dostaneme správny výsledok.

```
(<= (pluss ?res 0 ?res)
    (succ ?res ?x))
(<= (pluss ?x ?y ?res)
    (succ ?x ?xP1)
    (succ ?yM1 ?y)
    (pluss ?xP1 ?yM1 ?res))
```

3.1.3 Odčítavanie

Opäť si pravidlo rozdelíme na dva prípady a to elementárny príklad, kde pravidlo vypočíta $x - 0 = x$ a druhý prípad $x - y = result$, pričom v našej hre si vystačíme s podmienkou $x \geq y$. Potom definícia je takmer identická s operáciou sčítavania:

```
(<= (minuss ?res 0 ?res)
    (succ ?res ?x))
(<= (minuss ?x ?y ?res)
    (succ ?xM1 ?x)
    (succ ?yM1 ?y)
    (minuss ?xM1 ?yM1 ?res))
```

3.1.4 Väčší než

Ak už máme funkčný predikát *pluss*, tak následne s jeho použitím môžeme jednoducho zapísať operáciu väčší než, teda x je väčšie ako y, ak existuje z rôzne od nuly a zároveň platí $y + z = x$:

```
(<= (greaterThen ?x ?y)
    (pluss ?y ?z ?x)
    (not (succ ?z 1)))
```

3.2 Implementácia hry

Ak už máme všetky potrebné operácie pripravené, môžeme začať s písaním pravidiel samotnej hry. Hra je určená pre dvoch hráčov. Prvého hráča budeme názývať *blue* a druhého *red*. Nakoľko GDL nepodporuje náhodu, tak vytvoríme špecialneho hráča *radnom*, ktorý v rámci svojich legálnych akcií pridá do hry prvky náhody. Následne pri simulácii hry necháme za *random* hrať všeobecného počitačového hráča *RandomPlayer*. Konkrétna implementácia rolí vyzerá triviálne takto:

```
(role red)
(role blue)
(role random)
```

3.2.1 Fakty

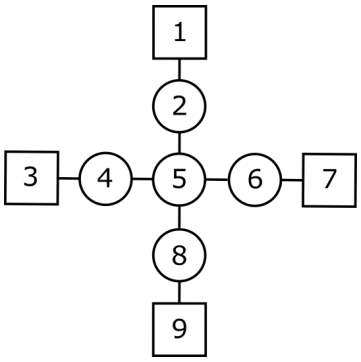
Zadefinujme si teraz základné fakty, ktoré budeme v implementácii mnohokrát využívať. Tieto fakty sú počas priebehu hry nemeniace a vždy pravdivé. Hracia plocha sa skladá z políčok ciest a hradov, ktoré pre jednoduchosť budeme identifikovať číslom, teda každé políčko bude mať svoje špecifické identifikačné číslo (označujeme ID). Už samotná hra je oproti ostatným hrám napísaným v GDL obšírna, preto som sa rozhodol hraciu plochu od tej skutočnej zjednodušiť a zmenšiť, aby sme skrátili dĺžku trvania hry, teda počtu všetkých stavov daného priebehu hry z počiatočného do koncového stavu. Hracia plocha bude neohodnotený graf, pričom vrcholy budú políčka hradov a ciest, ktoré potrebujeme od seba odlíšiť, teda konkrétne

```
(castle 1) (castle 3) (castle 7) (castle 9)
(path 2) (path 4) (path 5) (path 6) (path 8).
```

Ak vedie hrana z políčka A do políčka B, tak automaticky vedie hrana z B do A, preto by sme mohli hraciu plochu reprezentovať neorientovaným grafom, no napriek tomu pre zjednodušenie písania pravidiel som sa rozhodol faktom $edge(idFrom, idTo)$ reprezentovať len hranu z $idFrom$ do $idTo$. Pomocou tohto faktu následne budeme reprezentovať hrany v orientovanom grafe.

```
(edge 1 2) (edge 2 1) (edge 2 5) (edge 5 2)
(edge 5 6) (edge 6 5) (edge 6 7) (edge 7 6)
(edge 5 8) (edge 8 5) (edge 8 9) (edge 9 8)
(edge 4 5) (edge 5 4) (edge 3 4) (edge 4 3)
```

Pre lepšiu predstavu nám pomôže jednoduchá vizualizácia hracieho plánu na obrázku 3.1. Cieľom oboch hráčov je získať 10 a viac bodov tým, že plnia misie, ktoré sú im



Obr. 3.1: Príklad hracej plochy, na ktorej sú políčka ciest(kruhy), políčka hradov(štvorce), ktoré sú vyznačené svojim unikátnym identifikačným číslom a hrany, teda čiary spájajúce políčka.

pridané náhodne. Misia sa skladá z identifikačného čísla ID, z hradu/mesta kde ma hráč tovar vyzdvihnúť a z hradu/mesta kam tento tovar má doniesť. V našom prípade množina misií sa skladá zo všetkých kombinácií hradov a miest, avšak začiatočný hrad sa musí líšiť od toho cieľového:

```
(mission 1 1 3) (mission 2 3 7) (mission 3 1 9)
(mission 4 7 1) (mission 5 7 3) (mission 6 7 9)
(mission 7 3 1) (mission 8 3 7) (mission 9 3 9)
(mission 10 9 7) (mission 11 9 1) (mission 12 9 3)
```

3.2.2 Počiatočný stav a generovanie hry

Aby všeobecný počítačový hráč bol schopný hrať hocijakú hru, tak je nutnou podmienkou, aby aspoň jeden zo stavov hry bol počiatočný. Explicitne inicializujeme všetky dôležité hodnoty hráčom:

```
(init (generateGame true))
(init (pozitiveChips blue 0 0 0))
(init (pozitiveChips red 0 0 0))
```

```
(init (negativeChips blue 0 0 0))
(init (negativeChips red 0 0 0))
```

Pri pozitívnych žetónoch sú atribútmi postupne meno hráča, počet sprievodcov, ochraniek a nových kolies a u negatívnych žetónoch v slede meno hráča, počet bažín, zbojníkov a rozbitých ciest. Spoločenská hra Slovania obchodníkmi umožňuje zahrávať akcie len hráčovi, ktorý je na ťahu. Teda ak je na ťahu napríklad hráč *red*, tak potom *blue* a *random* nerobia nič(označujeme noop z anglického no operiaton). Na vygenerovanie hry budeme potrebovať pridať misie hráčom náhodne a to zabezpečíme pomocou legálneho ťahu hráča *random*. Legálne ťahy pri generovaní vyzerajú potom nasledovne:

```
(<= (legal ?anyPlayer noop)
    (true (generateGame true))
    (role ?anyPlayer)
    (distinct ?anyPlayer random))

(<= (legal random (generateMissions ?idBlueM ?idRedM))
    (true (generateGame true))
    (mission ?idBlueM ?from ?to)
    (mission ?idRedM ?from2 ?to2)
    (distinct ?idBlueM ?idRedM))
```

Prvé pravidlo nám hovorí, že ak sme v stave, kedy generujeme hru, tak hráči *blue* a *red* nerobia nič a druhé pravidlo slúži práve na vygenerovanie misií pre hráčov, teda *random* bude môcť vybrať zo všetkých kombinácií faktov misií, avšak nemôžu mať obaja hráči rovnaké ID misie. Atribúty v *generateMissions* sú v poradí prvý pre *blue* a druhý pre *red* hráča. Na reprezentovanie všetkých dôležitých atribútov hráča budeme používať predikát *playerStat*, ktorého atribúty sú v tomto poradí, hráč, pozícia figúrky, peniaze, ID misie, dosiahnuté body a či už vyzdvihol tovar. Napríklad konkrétny predikát môže vyzerať takto: $playerStat(blue, 1, 10, 5, 6, false)$. Aby sme generovanie úspešne dokončili, potrebujeme ešte definovať pomocou relácie *next*, čo bude platiť v nasledujúcom stave. Inicializovanie pre hráča *blue* vyzerá takto:

```
(<= (next (playerStat blue ?from 5 ?idBlueM 1 true))
    (does random (generateMissions ?idBlueM ?idRedM))
    (mission ?idBlueM ?from ?to))
```

Hráč *blue* dostane id misie podľa toho, čo vygeneroval hráč *random* a podľa faktu misie pridáme aj aktuálnu pozíciu figúrky, ktorá je totožná so začiatkom misie a zároveň môžeme pridať *true* pre atribút *vyzdviholTovar* a práve preto inicializujeme aktuálny

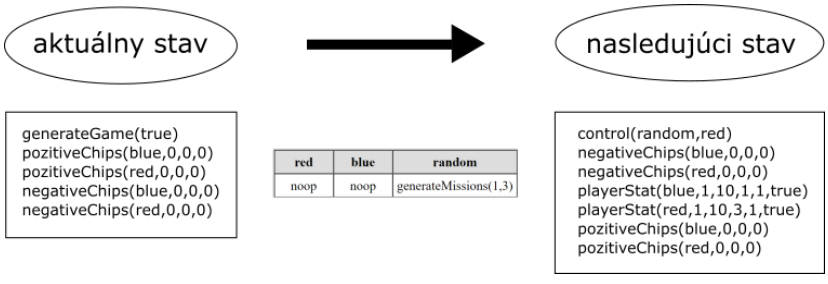
počet bodov na 1 (z pravidiel vieme, že ak hráč vyzdvihne tovar tak jeho aktuálny počet bodov sa zvíši o 1). Na dokončenie všetkých atribútov predikátu *playerStat* pridáme ešte 5 korún. Počet sme znížili kvôli menšej mape. Inicializovanie pre hráča *red* je analogické, len ID misie je v poradí druhý atribút v *generateMissions*, čiže:

```
(<= (next (playerStat red ?from 5 ?idRedM 1 true))
    (does random (generateMissions ?idBlueM ?idRedM))
    (mission ?idRedM ?from ?to))
```

Na reprezentovanie informácie, ktorý hráč je práve na ťahu a kto bude po ňom nasledovať bude slúžiť predikát $control(aktualnyHrac, nasledujuciHrac)$. Každému stavu, kedy bude môcť *blue* alebo *red* hrať svoje akcie predchádza ťah hráča *random*, ktorý vygeneruje číslo na kocke, o ktoré sa následne hráč bude môcť posunúť. Preto potrebujeme zadefinovať, ktorý hráč bude na ťahu v nasledujúcom stave od počiatočného a to zabezpečíme týmito pravidlami:

```
(<= (next (control random blue))
    (does random (generateMissions ?idBlueM ?idRedM))
    (greaterThen ?idBlueM ?idRedM))
(<= (next (control random red))
    (does random (generateMissions ?idBlueM ?idRedM))
    (greaterThen ?idRedM ?idBlueM))
```

Teda v budúcom stave bude na ťahu hráč *random*, aby vygeneroval číslo hodené na kocke a po ňom bude nasledovať ten hráč, ktorý dostal vyššie id misie, čo je v podstate analógia s inými spoločenskými hrami, kde začína hráč, ktorý hodil vyššie číslo na kocke. Na lepšiu predstavu simulácie generovania hry nám pomôže obrázok 3.2.



Obr. 3.2: Ukážka generovania hry, teda prechodu z aktuálneho(počiatočného) stavu do nasledujúceho pomocou legálnych akcií hráčov. Keďže generujeme hru, tak *red* aj *blue* nerobia nič a *random* zahral svoju akciu vygenerovania misií s id 1 pre *blue* a 3 pre *red*. V nasledujúcom stave bude na ťahu *random*, aby sme napodobnili hod kocky a po ňom pôjde hráč *red*, pretože má vyššie ID misie.

3.2.3 Legálne akcie hráčov

Ak sme už novú hru úspešne vygenerovali, tak môžeme postupne vysvetliť všetky legálne akcie hráčov. Ukážme najprv najzákladnejšiu akciu, ak hráč nie je na ťahu, tak nerobí nič:

```
(<= (legal ?anyPlayer noop)
    (true (control ?player ?other))
    (role ?anyPlayer)
    (distinct ?player ?anyPlayer))
```

Základný prvok náhody, ktorý využívame v hre pred začiatkom ťahu hráčov je hod kockou a zabezpečíme ho pomocou hráča *random*. Pravidlo pre hod čísla 1 vyzerá potom takto:

```
(<= (legal random (roll 1))
    (true (control random ?next))
    (not (true (moved ?next)))
    (not (true (generateGame true))))
```

Čiže ak platí, že hráč *random* je na ťahu, nasledujúci hráč sa ešte nepohol (táto podmienka je nutná, pretože ak hráč dokončí misiu a *random* mu následne pridá novú, tak nechceme aby mal na výber hodiť číslo) a zároveň hra už je vygenerovaná. Kocku sme si v našej hre prispôsobili a to tak, že má len hodnoty 1 a 2 nakoľko sme si hraciu plochu zmenšili. Implementácia hodenia kocky 2 je analogická ako vyššie s tým rozdielom že v *roll* namiesto 1 je číslo 2. Legálne akcie mimo ťahu sme už popísali a vysvetlili, teraz sa zamerajme na legálne akcie hráčov na ťahu a rozdeľme ich na 3 časti:

1. posun figúrkou
2. nákup negatívnych a pozitívnych žetónov, vkladanie negatívnych žetónov na políčka ciest
3. ukončenie ťahu

Začnime prvým bodom, ktorý je nevyhnutný, čiže každý hráč začína svoj ťah posunom figúrky o číslo, ktoré vygeneroval *random*. Ako zistíme aké číslo hodil *random* na kocke? Pomocou predikátu *rollRess*, ktorý získame pomocou relácie *next*:

```
(<= (next (rollRess ?num))
    (does random (roll ?num)))
```

Ak *random* hodil číslo, tak v nasledujúcom stave bude platiť predikát *rollRess* s rovnakým číslom. Je nevyhnutné aby to bolo takto definované cez nový predikát a reláciu *next*, pretože GDL nepodporuje závislosť relácie *legal* od relácie *does*. Akcia zahrania posunu figúrky vyzerá nasledovne:

```
(<= (legal ?player (move ?act ?next))
    (true (rollRess ?num))
    (or (vrcholy_vo_vzdialenosti ?act ?next ?num)
    (hrady_vo_vzdialenosti ?act ?next ?num))
    (true (playerStat ?player ?act ?m ?idM ?p ?collected))
    (true (control ?player random)))
```

Pravidlo nám hovorí, že hráč, ktorý je na ťahu môže pohnúť figúrku z aktuálnej pozície na nové políčko, o ktorom musí platiť aspoň jedna z týchto podmienok:

- Ak random hodil číslo x, tak nové políčko(cesta) musí byť vo vzdialenosti presne x od aktuálnej pozícii figúrky.
- Ak random hodil číslo x, tak nové políčko(hrad) musí byť vo vdzialenosti maximálne x.

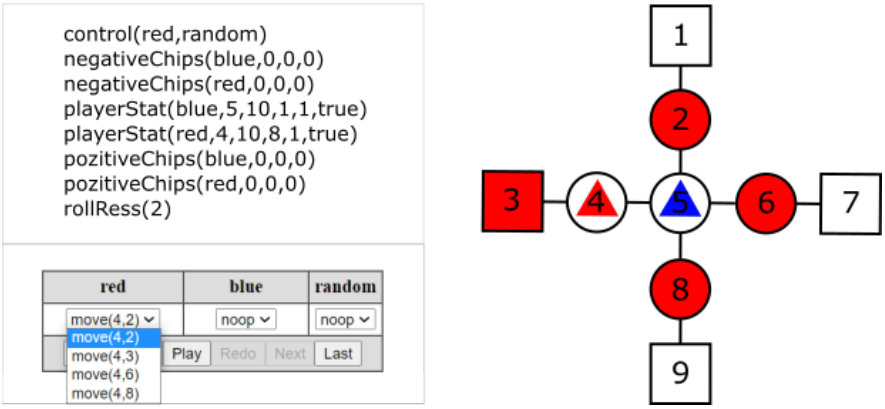
Prvá podmienka závisi od pomocného predikátu, ktorý vyzerá takto:

```
(<= (vertexInDistance ?a ?b 1)
    (edge ?a ?b))
(<= (vertexInDistance ?a ?b ?n)
    (greaterThen ?n 1)
    (edge ?a ?c)
    (succ ?x ?b)
    (succ ?nM1 ?n)
    (not (equals ?a ?b))
    (vertexInDistance ?c ?b ?nM1))
```

Prvá časť definuje triviálny prípad, kedy vrchol A je vo vzdialenosti presne 1 od vrcholu B, ak existuje hrana z A do B. Druhá časť nám hovorí o tom, že vzdialenosť vrcholu A od vrcholu B sa rovná n ak platí, že vrcholy A a B sú odlišné, vzdialenosť n je väčsia ako 1 a zároveň existuje vrchol C, ktorý je vo vzdialenosti od B presne $n-1$ a to vypočítame pomocou jednoduchej rekurzie.
Druhá podmienka závisí od následujúceho doplnkového pravidla:

```
(<= (castleInDistance ?a ?cas ?n)
    (castle ?cas)
    (greaterThen ?x 0)
    (greaterThen ?n ?x)
    (vertexInDistance ?a ?cas ?x))
```

Hrad je vo vzdialenosti n od vrcholu A ak platí, že hrad je skutočne hradom a zároveň je vo vzdialenosti 1 až $n-1$ od vrcholu A. Táto vlastnosť pôsobí mätúco, ale má svoj význam. Ak sa napríklad nachádzame na políčku, ktoré je vo vzdialenosti od hradu kam máme doniesť tovar misie presne 1 a na kocke hodíme číslo 2, tak práve toto pravidlo nám slúži na to, aby sme mali možnosť pohnúť sa s figúrkou aj na tento hrad. Na lepšie pochopenie demonštrujeme tento legálny ťah na obrázku 3.3.



Obr. 3.3: V ľavej časti môžeme pozorovať aktuálny stav a taktiež zoznam legálnych akcií, ktoré môže hráč *red* zahrať. V pravej časti sa nachádza jednoduchá vizualizácia tohto stavu, pričom trojuholník reprezentuje figúrku a políčka vyfarbené červenou farbou implikujú legálnu akciu pohybu figúrky na ne.

Prejdime teraz do druhej časti legálnych akcií hráča na ťahu. Hráč môže nakúpiť negatívne žetóny a to bažinu, zbojníkov a rozbitú cestu. Na všetky tri platia rovnaké podmienky, preto si ukážeme len nákup bažiny a analogicky je potom definovaný aj nákup zbojníkov či rozbitej cesty, len s inými názvami, konkrétne *buyN_Bandits* a *buyN_CrushedRoad*.

```
(<= (legal ?player buyN_Swamp)
    (true (playerStat ?player ?act ?m ?idM ?p ?collected))
    (true (moved ?player))
    (true (control ?player random))
    (not (true (inJail ?player)))
    (greaterThen ?m 0))
```

Hráč môže nakúpiť negatívny žetón s názvom bažina ak je na ťahu, už sa pohol figúrkou,

nie je vo väzení a má dostatok peňazí, konkrétne negatívne žetóny stoja jednu korunu za jeden kus, preto stačí ak má počet peňazí vyšší ako 0. Aby sme pochopili do detailov toto pravidlo, tak si musíme ešte ukázať, kedy je pravdivý predikát *moved*(*player*):

```
(<= (next (moved ?player))
    (does ?player (move ?from ?to)))
(<= (next (moved ?player))
    (true (control ?player random))
    (not (does ?player endTurn)))
(<= (next (moved ?player))
    (does random (generateNewMission ?idM))
    (true (control random ?player)))
```

Toto pravidlo nám slúži na to aby sme vedeli kedy hráč môže zahrať aj iné ťahy ako len *move*. Prvá časť slúži na vytorenie predikátu, teda zjavne v nasledujúcom stave bude platiť moved, ak hráč práve zahral akciu *move*. Druhá časť nám slúži na zotrvanie predikátu, teda ak hráč je na ťahu a ešte neukončil ťah, tak predikát sa nemení. Tretia časť je pridaná pre situáciu, ak hráč dokončil misiu a následne budeme potrebovať pomocou hráča *random* vygenerovať novú misiu, tak aby hráč, ktorý dokončil misiu mohol po vygenerovaní pokračovať v hraní legálnych akcií. Pre úplne pochopenie ešte ukážeme, kedy platí predikát *inJail*. Pravidlo sa skladá z dvoch častí, tá prvá slúži na vytvorenie predikátu a vyzerá nasledovne:

```
(<= (next (inJail ?player))
    (true (field ?newP ?opponent crushedRoad))
    (does ?player (move ?oldP ?newP))
    (true (pozitiveChips ?player ?guide ?guard 0))
    (distinct ?player ?opponent))
```

Hráč bude v nasledujúcom stave vo väzení ak vstúpil na políčko, na ktorom je súperov negatívny žetón *crushedRoad* a zároveň nemá pozitívny žetón *newWheel*(ochranu voči rozbitej ceste). Druhá časť slúži na kopírovanie predikátu v nezmenenej podobe:

```
(<= (next (inJail ?player))
    (true (inJail ?player))
    (not (removeJail ?player)))
```

Teda jednoducho hráč ostáva vo väzení ak v ňom je a zároveň nenastane možnosť, aby sa z neho dostal preč. Prípad kedy sa hráč má dostať von z väzenia je definovaný týmto spôsobom:

```
(<= (removeJail ?player)
    (does ?opponent endTurn)
    (role ?player)
    (distinct ?player ?opponent))
(<= (removeJail ?player)
    (true (inJail ?player))
    (true (inJail ?opponent))
    (distinct ?player ?opponent))
```

Hráč sa dostáva z väzenia ak ukončil ťah súper alebo ak je vo väzení aj súper.
Hráč môže taktiež nakúpiť pozitívne žetóny aby sa dokázal brániť voči negatívnym. Môže si vybrať z týchto žetónov a to sprievodcu, ochranku a nové koleso. Už z názvov je zrejmé, že sprievodca slúži na obranu voči bažine, ochranka na zbojníkov a nové koleso na rozbitú cestu. Nákup ochranky je podobný nákupu bažiny a to:

```
(<= (legal ?player buyP_Guard)
    (true (control ?player random))
    (not (true (inJail ?player)))
    (true (playerStat ?player ?act ?m ?idM ?p ?collected))
    (true (moved ?player))
    (greaterThen ?m 1))
```

Hráč môže zahrať legálnu akciu nákupu ochranky ak je na ťahu, pohol sa, nie je vo väzení a zároveň jeho počet peňazí je väčší ako 1, nakoľko pozitívne žetóny stoja 2 koruny za jeden kus. Analogicky sú zadefinované nákupy zbojníkov a nových kolies len s iným názvom. Negatívne žetóny môžu hráči na svojom ťahu vkladať na hraciu plochu, preto ukážeme pravidlo, ktoré reprezentuje legálnu akciu vloženia bažiny na políčko:

```
(<= (legal ?player (dropSwamp ?f))
    (path ?f)
    (role ?anyPlayer)
    (true (negativeChips ?player ?s ?b ?cr))
    (true (moved ?player))
    (true (control ?player random))
    (not (isField ?f))
    (not (true (inJail ?player)))
    (greaterThen ?s 0))
```

Hráč môže vložiť bažinu na políčko, ak políčko je voľná cesta, teda neobsahuje iný

negatívny žetón, a zároveň platí, že hráč je na ťahu, už pohol figúrkou, nie je vo väzení a má práve nejakú bažinu kúpenú, teda počet bažín je väčší ako 0. Analogicky sú definované legálne akcie vloženia zbojníkov, avšak s tým rozdielom, že počet práve nakúpených zbojníkov je väčší ako 0 a podobne u legálnej akcie vloženia rozbitej cesty počet nakúpených rozbitých ciest musí byť väčší ako 0. Pre úplne pochopenie si špecifikujme predikát *isField*.

```
(<= (isField ?f)
    (true (field ?f ?player ?chip)))
```

Teda políčko je obsadené ak platí že na tom políčku už je nejaký negatívny žetón nejakého hráča. Kedy však platí predikát *field*? Máme tri pravidlá, ktoré zabezpečujú vytvorenie predikátu *field* v nasledujúcom stave. Ukážeme si jedno z nich:

```
(<= (next (field ?f ?player swamp))
    (does ?player (dropSwamp ?f)))
```

Na políčku bude v nasledujúcom stave bažina ak hráč zahral legálnu akciu vloženia bažiny na políčko. Analogicky na políčku v budúcom stave budú zbojníci, respektíve rozbitá cesta, ak hráč zahral *dropBandits*, respektíve *drobCrushedRoad*. Nasledujúce pravidlo slúži na zachovanie predikátu v identickej podobe.

```
(<= (next (field ?f ?player ?chip))
    (true (field ?f ?player ?chip))
    (not (removeField ?f ?player ?chip)))
```

Žetón ostane na políčku v nastavajúcom stave, ak tam už je a neplatí, že má zmiznúť. Ukážeme si najprv kedy bažina a rozbitá cesta sa má odstrániť z políčka:

```
(<= (removeField ?newP ?playerPut ?chip)
    (true (field ?newP ?playerPut ?chip))
    (distinct ?chip bandits)
    (does ?otherPlayer (move ?old ?newP))
    (distinct ?otherPlayer ?playerPut))
```

Ak negatívny žetón je odlišný od zbojníkov a súčasne je položený na políčku a zároveň iný hráč stúpil na dané políčko, tak negatívny žetón sa odoberie. Odstránenie zbojníkov je analogické avšak s pridaním jednej podmienky a tou je, že zbojníci okrádajú len už vyzdvihnutý tovar, teda ak chceme aby zmizli, tak cudzí hráč, ktorý stúpi na políčko musí už mať tovar vyzdvihnutý. Konkrétna implementácia vyzerá nasledovne:

```
(<= (removeField ?newP ?playerPut bandits)
    (true (field ?newP ?playerPut bandits))
    (does ?otherPlayer (move ?old ?newP))
    (true (playerStat ?otherplayer ?old ?m ?idM ?p true))
    (distinct ?otherPlayer ?playerPut))

      (<= (legal random (generateNewMission ?idM))
          (true (finished_missionn ?player))
          (mission ?idM ?from ?to))
```

Predposledným legálnym ťahom v hre Slovania obchodníkmi je generovanie novej misie:
Hráč *random* môže zahrať akciu generovania novej misie ak platí, že nejaký hráč dokončil misiu a zároveň existuje misia s daným ID. Pomocný predikát *finished_missionn* vyzerá takto:

```
(<= (next (finished_missionn ?player))
    (does ?player (move ?act ?newP))
    (true (playerStat ?player ?act ?m
          ?actMission ?actPoints true))
    (mission ?actMission ?from ?newP))
```

Hráč dokončil misiu ak sa pohol na políčko, ktoré sa rovná políčku cieľového hradu aktuálnej misie. Poslednou legálnou akciou je ukončenie ťahu. Hráč môže ukončiť svoj ťah ak sa pohol a zároveň je na ťahu.

```
(<= (legal ?player endTurn)
    (true (moved ?player))
    (true (control ?player random)))
```

3.2.4 Následujúci stav

Každá hra napísaná v GDL využíva reláciu *next*, ktorá nám hovorí, čo bude platiť v nasledujúcom stave. Nejaké pomocné pravidlá využívajúce *next* sme si už ukázali v predošlých podkapitolách, avšak nedefinovali sme ešte základné predikáty, ktoré musia platiť pre oboch hráčoch v každom stave a to *negativeChips*, *pozitiveChips* a *playerStat*. Prvý z nich slúži na reprezentáciu počtu nakúpených určitých negatívnych žetónov konkrétnym hráčom. V počiatočnom stave sme si inicializovali počty bažín, zbojníkov a rozbitých ciest na 0. Ukážeme si najprv pravidlá, ktoré hovoria o zmene

predikátu z aktuálneho do nasledujúceho stavu. Príklad budeme demonštrovať na žetóne bažina:

```
(<= (next (negativeChips ?player ?newS ?b ?cr))
    (true (negativeChips ?player ?oldS ?b ?cr))
    (does ?player buyN_Swamp)
    (succ ?oldS ?newS))
(<= (next (negativeChips ?player ?newS ?b ?cr))
    (true (negativeChips ?player ?oldS ?b ?cr))
    (does ?player (dropSwamp ?f))
    (succ ?newS ?oldS))
```

To jest počet bažín, ktoré hráč vlastní sa zvýši o jedna, ak žetón kúpil a zníži, ak ho položil na políčko hracej plochy. Analogicky u zvyšných dvoch žetónoch sa podobne počet zvýši o jedna, ak hráč kúpil daný žetón a zníži, ak položil daný žetón. na hraciu plochu
Potrebujeme ešte zadefinovať pravidlo, ktoré hovorí o tom, že predikát ostane v nezmenenom stave:

```
(<= (next (negativeChips ?player ?s ?b ?cr))
    (true (negativeChips ?player ?s ?b ?cr))
    (not (buyNegativeChip ?player))
    (not (droppedNegativeChip ?player)))
```

Teda počet negatívnych žetónov sa nezmení ak hráč nekúpil ani jeden z negatívnych žetónov a zároveň ani jeden z nich nepoložil na hraciu plochu.
Pomocou predikátu *pozitiveChips* reprezentujeme jednotlivo aktuálny počet všetkých troch pozitívnych žetónov daného hráča v poradí sprievodca, ochranka a nové koleso. Tie slúžia hráčom na obranu voči negatívnym, konkrétne v takýchto dvojiciach, sprievodca na bažinu, ochranka na zbojníkov a nové koleso na rozbitú cestu. Analogicky ako u negatívnych sa počet daného pozitívneho žetóna zvýši o jedna ak hráč zahral legálnu akciu nákupu daného žetóna. Na druhej strane zníženie počtu o jedna je odlišné a nastane vtedy, ak pozitívny žetón splní svoju funkciu. Pravidlo ukážeme na pozitívnom žetóne sprievodca.

```
(<= (next (pozitiveChips ?player ?guideM1 ?guard ?nw))
    (stepOnSwampAndHaveGuide ?player)
    (true (pozitiveChips ?player ?actGuide ?guard ?nw))
    (greaterThen ?actGuide 0)
    (succ ?guideM1 ?actGuide))
```

Čiže počet sprievodcov sa zníži o jeden ak hráč má aspoň jedného sprievodcu a platí predikát *stepOnSwampAndHaveGuide*, ktorý vyzerá nasledovne:

```
(<= (stepOnSwampAndHaveGuide ?player)
    (does ?player (move ?oldP ?newP))
    (true (field ?newP ?opponent swamp))
    (distinct ?player ?opponent)
    (true (pozitiveChips ?player ?gi ?ga ?nw))
    (greaterThen ?gi 0))
```

Teda predikát je pravdivý, ak hráč vstúpil figúrkou na políčko, na ktorom je cudzia bažina. Analogicky sa počet nových kolies, respektíve ochraniek zníži o jedna ak hráč vstúpil na cudziu rozbitú cestu. U zbojníkov musí zároveň hráč už mať vyzdvihnutý tovar, pretože len vtedy môžu zbojníci olúpiť hráča o náklad. Počet pozitívnych žetónov hráča ostane v budúcom stave rovnaký ak nekúpil ani jeden z pozitívnych žetónov a zároveň ani jeden z nich nesplnil svoju funkciu. Pri generovaní hry sme inicializovali predikát *playerStat*, teraz ukážeme pár pravidiel s reláciou next, ktoré nám hovoria buď o zmene tohto predikátu v budúcom stave alebo o jeho zachovaní v nezmenenom tvare. Napríklad pravidlo:

```
(<= (next (playerStat ?player ?actP ?m ?idM ?p ?collected))
    (does ?player noop)
    (not (true (finished_missionn ?player)))
    (true (playerStat ?player ?actP ?m ?idM ?p ?collected)))
```

Ak hráč nie je na ťahu, tak stav hráča ostáva nezmenený, čiže všetky jeho dôležité atribúty ostávajú rovnaké. Ďalším pravidlom demonštrujúcim zhodný predikát v nasledujúcom stave je:

```
(<= (next (playerStat ?player ?act ?m ?idM ?p ?collected))
    (true (playerStat ?player ?act ?m ?idM ?p ?collected))
    (or (does ?player endTurn)
    (droppedNegativeChip ?player)))
```

Ak hráč ukončil svoj ťah alebo položil na stôl negatívny žetón, tak sa nemení žiaden atribút z nasledovných: aktuálna pozícia, počet peňazí, id misie, dosiahnuté body, či atribút reprezentujúci vyzdvihnutie tovaru.
Znázornime teraz nejaké prípady zmeny predikátu *playerStat*, napríklad ak sa hráč pohol figúrkou na políčko odkiaľ začína jeho aktuálna misia, tak vyzdvihne tovar, konkrétne pravidlo vyzerá takto:

```
(<= (next (playerStat ?player ?newPos ?m ?actMis ?newP true))
    (true (playerStat ?player ?actPos ?m ?actMis ?p false))
    (succ ?p ?newP)
    (does ?player (move ?actPos ?newPos))
    (mission ?actMis ?newPos ?to))
```

Môžeme si všimnúť, že sa zmení atribút jednak pozície figúrky kam sa hráč pohol, ale taktiež aj počet dosiahnutých bodov sa zvýši o 1, pretože hráč už vyzdvihol tovar a zároveň atribút indikujúci vyzdvihnutie tovaru sa zmení na *true*. Kvázi jednoduché zmeny nastanú ak hráč nakúpi nejaký negatívny žetón, tak zmenší sa o jedna len atribút počtu peňazí, podobne ak nakúpi hráč pozitívny žetón tak sa zmenší počet peňazí o 2, pretože pozitívne žetóny stoja 2 koruny za jeden kus. Ďalší prípad je, ak hráč vstúpi na políčko kde začína misia, tak sa zmení atribút vyzdvihnutia tovaru na *true*, alebo ak hráč vstúpi na políčko na ktorom je cudzí negatívny žetón bažina, tak sa nepohne ale ostáva na pôvodnom políčku a podobne.
Na záver tejto podkapitoly si ešte ukážeme niekoľko pravidiel obsahujúcich predikát control, ktorý nám slúži na zastupovanie hráča na ťahu a taktiež hráča, ktorý bude nasledovať po ňom. Nasledujúce pravidlo nám hovorí prípad, kedy predikát ostane v budúcom stave v nezmenenom stave:

```
(<= (next (control ?player random))
    (true (control ?player random))
    (not (finished_mission ?player))
    (not (does ?player endTurn)))
```

To nastane vtedy, ak hráč neukončil svoj ťah a zároveň nedokončil misiu. Síce je pravda, že ak hráč dokončí misiu tak teoreticky nekončí svoj ťah, ale prakticky potrebujeme aby nasledoval *random* a vygeneroval novú misiu:

```
(<= (next (control random ?player))
    (finished_mission ?player))

(<= (next (control ?player random))
    (true (control random ?player))
    (does random (generateNewMission ?idM)))
```

Teda v nasledujúcom stave bude môcť vygenerovať misiu *random*, ak hráč ukončil misiu a následne po vygenerovaní bude na ťahu opäť rovnaký hráč.

3.2.5 Terminálny stav a odmeny

Hra sa dostane do koncového stavu ak niektorý z hráčov má aktuálny počet bodov väčší ako 9:

```
(<= terminal
    (true (playerStat ?player ?actP ?m ?actMiss ?actPoints ?collected))
    (greaterThen ?actPoints 9))
```

Pridanie bodov hráčom je triviálne:

```
(<= (goal ?player ?actPoints)
    (true (playerStat ?player ?actP ?m ?actMiss ?actPoints ?collected)))
    (goal random 0)
```

Hráč dosiahne na konci hry počet bodov rovný aktuálnemu počtu získaných bodov. A *random* samozrejme dostane 0 bodov.

3.3 Varianty hry

Zamerajme sa teraz na stav hry, v ktorom sa hráč rozhoduje kam presunie svoju figúrku z aktuálnej pozície na pozíciu vo vzdialenosti presne hodeného čísla. Stavový automat vypočíta legálne akcie pohybu figúrky podľa toho ako vyzerá v GDL práve pravidlo $legal(player, move)$. Dostávame sa k hlavnému rozdielu, kde buď môžeme v každom stave počítať políčka vo vzdialenosti hodeného čísla alebo môžeme pridať do pravidiel hry predvypočítané fakty, ktorými budeme reprezentovať vzdialenosti medzi vrcholmi, pričom maximálna vzdialenosť sa bude rovnať maximálnej hodnote hodenia kocky. Tento fakt bude vyzerať nasledovne: $fromToDistance(from, to, distance)$. Prvý spôsob, kde vzdialenosti počítame v každom stave sme demonštrovali v kapitole implementácia hry. V druhej metóde musíme spomínané fakty pridať do pravidiel. Ako si ich môžeme jednoducho vygenerovať? Práve pomocou predikátov *vertexInDistance*, *castleInDistance* a relácie *next*. Konkétne pre prípad maximálnej hodnoty hodenia kocky 3 vyzerá pravidlo nasledovne:

```
(<= (next (fromToDistance ?act ?next ?dis))
    (greaterThen ?dis 0)
    (greaterThen 4 ?dis)
    (or (vertexInDistance ?act ?next ?dis)
    (castleInDistance ?act ?next ?dis)))
```

V nasledujúcom stave potom dostaneme zoznam týchto faktov vo formáte HIF(z anglického Human Readable Format), ktorý potom už len jednoducho konvertujeme do KIF, pomocou napríklad prekladača [3]. Tento zoznam faktov už len potom pridáme do pravidiel. Pre zaujímavosť pseudokód na vytvorenie takéhoto predikátu pre nie logický jazyk nájdeme v algortime 1. Ďalší atribút hry, ktorý môžeme zmeniť je hracia plocha. Vytvorili sme mapu s viacerymi vrcholmi aj hranami a tým pádom teda dostávame spolu 4 hry:

- classicMap a biggerMap - základná a väčšia mapa, pričom vzdialenosti vrcholov bude musieť stavový automat počítať v každom stave
- classicMapFTD a biggerMapFTD - základná a väčšia mapa, pričom vzdialenosti vrcholov sú predpočítané, teda pridané k pravidlám hry ako fakty

input : graph and maxDistance
output: listOfPredicates

```
1  listOfPredicates ← ∅;
2  foreach vertex v such that v ∈ graph.vertices() do
3      for i ← 1 to maxDis do
4          listOfTo ← FindVerticesInDistance(graph v i);
5          foreach to such that to ∈ listOfTo do
6              predicate ← (FromToDistance v to i);
7              listOfPredicates ← listOfPredicates ∪ predicate;
8          end
9      end
10 end
11 return listOfPredicates
```

Algorithm 1: generátor zoznamu predikátov fromToDistance

Kapitola 4

Testovanie počítačových hráčov

Táto kapitola sa zaoberá testovaním už napísanej hry v GDL jazyku a následným skúšaním všeobecných počítačových hráčov na nej.

4.1 Validácia

Aby sme hru mohli rozumne testovať, tak je dôležité aby sme overili správnosť implementácie. Na validáciu hry sme použili verejne dostupný balíček [4], ktorý je napísaný v jazyku Java. V spomínanom kóde sme využili dva simulátori na kontrolu hier napísaných v GDL a to konkrétne *StaticValidator* a *SimulationValidator*. Prvý overuje napríklad odlišnosť názvov funkcií a premenných, aritu funkcií, taktiež kontroluje bezpečnosť pravidiel, teda či všetky premenné sú v pozitívnom podcieli a zároveň preveruje vymedzenie rekurzívnych pravidiel. Druhý validátor slúži na jednoduchú simuláciu danej hry, pričom pred samotnou simuláciou ešte musíme zadať počet simulácií a zároveň maximálne trvanie času hry, teda dĺžku postupnosti stavov hry z počiatočného do koncového stavu. Následne tento validátor napríklad kontroluje, či v každom stave môže hocijaký hráč zahrať aspoň jednu legálnu akciu, a zároveň overí, či hra skončí skôr ako je maximálna vstupná dĺžka hry a nakoniec preverí, či každý hráč v koncovom stave dostane ohodnotenie. Všetky varianty hry sme úspešne pomocou týchto dvoch validátorov skontrolovali.

4.2 Predpočítanie pohybu

V tejto časti využijeme vyššie zmienený *SimulationValidator*, avšak s menšou úpravou. V prvom rade si však v krátkosti vysvetlime ako konkrétna simulácia prebieha. V každej simulácii inicializujeme stavový automat, pomocou ktorého reprezentujeme stavy hry. Najprv inicializujeme jeho počiatočný stav podľa pravidiel hry a následne iterujeme kým sa nedostaneme do koncového stavu alebo kým nepresiahneme maxi-

málnu dĺžku hry, pričom v každom stave stavový automat vyberá legálne akcie pre všetkých hráčov náhodne, konkrétne pomocou objektu *Random*. My však budeme chcieť testovať rozličné implementácie hry na rovnakej postupnosti stavov z počiatočného do koncového stavu a to zabezpečíme využitím funkcie *Random.seed*(). Počas simulácie budeme sledovať čas trvania behu. Nakoľko na všetkých variantoch hry budeme kopírovať rovnakú postupnosť stavov, tak tým zabezpečíme identickosť priebehu hry. Simulácie sa budú líšiť len spôsobom počítania legálnych akcií *move* pomocou stavového automatu a veľkosťou hracej plochy. Výsledky testovania sú v tabuľke 4.1. Môžeme si všimnúť, že v základnej verzii mapy trvali simulácie takmer rovnaký čas.

mapa	**počet simulácií**	**ms základná verzie**	**ms ftd verzia**
classic	100	93195	93252
bigger	100	345482	299891

Tabuľka 4.1: Testovanie predikátu fromToDistance

Pri väčšej mape však simulácie s predpočítaným predikátom $fromToDistance$ skončili skôr.

4.3 Monte Carlo proti Random

Správnosť hry si ukážeme na konkrétnej ukážke zápasu medzi všeobecnými počitačovými hráčmi, pričom prvým hráčom bude *MonteCarlo*, s ktorým sme sa už oboznámili v tejto práci a druhým triviálny *Random*. Hod kockou a generovanie misií budeme reprezentovať taktiež hráčom *Random*. Na demonštrovanie zápasov týchto hráčov využijeme balíček [5], ktorý napríklad ponúka pripájanie hráčov a následný výber konkrétnej hry napísanej v jazyku GDL. Medzi vstupné údaje k spusteniu zápasu patrí počet simulácií a taktiež maximálny čas, ktorý hráči budú mať na výber svojej legálnej akcie. Práve pri testovaní vyskúšame rôzne hodnoty tejto dĺžky, pretože ako sme si už spomínali v tejto bakalárksej práci *MonteCarlo* počíta výber svojej akcie až dokonca. V tabuľke 4.2 sú výsledky zápasov hráčov *MonteCarlo* a *Random*. Môžeme si všimnúť že na základnej mape v oboch parametroch maximalného času na ťah vyhral vo všetkých prípadoch *MonteCarlo*. Pri zväčšení mapy, teda aj väčšej dĺžky postupnosti stavov z počiatočného do koncového, bol *MonteCarlo* pri 5 sekundách vyrovnaným súperom hráča *Random*, avšak keď sme zvýšili parameter času na ťah, tak *MonteCarlo* rapídne zvýšil svoju efektivitu.

názov hry	**počet zápasov**	**čas na ťah v s**	**výhry Monte Carlo v %**	**výhry Random v %**
classicMap	12	5	100	0
classicMap	6	10	100	0
biggerMap	10	5	60	40
biggerMap	6	10	83.33	16.66

Tabuľka 4.2: Testovanie Monte Carlo proti Random

4.4 Monte Carlo proti Monte Carlo

Podobne ako v predošlej kapitole využijeme balíček [5] na ukážku zápasu, kde proti sebe postavíme rovnakých hráčov. Pri testovaní sa upriamime na počet vyhratých zápasov hráča *blue* a hráča *red* a následne overíme či obaja hráči sú vyvážení a že pravidlá hry nie sú viac naklonené pre jedného z hráčov. Výsledky testovania nájdeme v tabuľke 4.3. Pre obmedzenie času sme nedokázali simulovať veľké množstvo zápasov, avšak z dát môžeme vidieť, že hra je vyvážená. Pravidlá pre oboch hráčov boli tvorené identicky.

názov hry	**počet zápasov**	**výhry blue v %**	**výhry red v %**
classicMap	10	60	40

Tabuľka 4.3: Testovanie Monte Carlo proti Monte Carlo

Záver

V rámci tejto bakalárskej práce sme sa oboznámili so všeobecnými počítačovými hráčmi a v krátkosti sme demonštrovali aké algoritmy používajú. GGP hráči sú schopní hrať hry, ktoré su popísané v nejakom jazyku na popis hier. Medzi najznámejší jazyk popisu hry patrí GDL, preto sme sa následne v práci zaoberali definovaniu pojmov v tomto jazyku a neskôr teóriu demonštrovali na príklade hry Tic-Tac-Toe.
Ďalej sme sa oboznámili podrobne s pravidlami hry Slovania obchodníkmi, teda z čoho sa hra skladá, aký je jej cieľ a taktiež jej špecifické prvky v podobe žetónov. Neskôr sa nám podarilo dosiahnuť hlavný cieľ a to výsledná implementácia spomínanej hry v jazyku GDL, ktorá je validná. Následne sme boli úspešní vo vytvorení rôznych variantov hry, ktoré sme taktiež úspešne zvalidovali a následne testovali ich rozdiel. Potom sme hru otestovali aj na reálnych všeobecných počítačových hráčoch, pričom sme sledovali, ktorý hráč je silnejší a ako napríklad rozhoduje parameter maximálneho času výberu legálnej akcie pre hráča. Testovaním sme ukázali, že MonteCarlo je očividne lepším hráčom ako Random a zároveň, že parameter dĺžky výberu ťahu je pre MonteCarlo kľúčový.
Poslednou, nanešťastie, nedokončenou snahou, o ktorú sme sa snažili bolo odskúšanie hry na lepšom hráčovi, konkrétne viacnásobnom víťazovi súťaže v hraní hier na konferencii AAAI , Cadiaplayer. Ďalším krokom by bolo vhodné teda oskúšať hru na Cadiaplayer a zároveň simulovať zápas Cadiaplayer proti MonteCarlo a zanalyzovať výsledky.
Aj táto práca, respektíve implementácia hry v tejto práci nie je dokonalá a dá sa zlepšovať, napríklad jednoduchou vizualizáciou, kde by sme v každom stave reprezentovali hraciu plochu. Ak by hra obsahovala vizualizáciu, tak tým pádom sa nám otvárajú nové možnosti v podobe simulovania zápasu človek proti všeobecnému počítačovému hráčovi. Taktiež dobrým námetom do budúcna by mohla byť implementácia hry v GDL-II, teda s novým parametrom neúplnej informácie. Hráči by nemali prístup ku všetkým informáciam hry, napríklad v našej hre Slovania obchodníkmi by sme mohli skryť akutálne negatívne a pozitívne žetóny súperov. Táto bakalárska práca môže byť tatiež dobrým základom pre diplomovú prácu, kde by sme sa mohli zaoberať už konkrétnym vytvorením všeobecného počítačového hráča.

Literatúra

[1] https://en.wikipedia.org/wiki/General_game_playing. [Citované 2021-02-10].

[2] http://davinci.fmph.uniba.sk/~bacinsky6/. [Citované 2021-05-12].

[3] http://ggp.stanford.edu/public/gameconverter.php. [Citované 2021-05-12].

[4] https://github.com/hardiecate/ggp-base. [Citované 2021-05-12].

[5] https://github.com/ggp-org/ggp-base. [Citované 2021-05-12].

[6] Michael Genesereth Nathaniel Love, Timothy Hinrichs. General Game Playing: Game description language specification. Technical report, Stanford University, 2006.

[7] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107, 2011.

Príloha A: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádza zdrojový kód programu. Zdrojový kód je zverejnený aj na stránke `https://github.com/vbacinsky/bachelor_thesis`.

Príloha B: Používateľská príručka

Systemové požiadavky: Java 8 (Update 251, 64-bit), Java SE Development Kit 8 (Update 111, 64-bit)
Najprv odzipujte súbor v prílohe a otvorte priečinok *ggp-base-master*.
Príklady príkazov (v systéme Windows použite gradlew.bat):
Potvrdenie verzie a kompilácia:

```
./gradlew -v
./gradlew assemble
```

V jednom termináli spustíme aplikáciu hráčov, kde intuitívne môžeme pridať konkrétnemu hráčovi určitý port:

```
./gradlew player
```

V druhom termináli spustíme aplikáciu servera:

```
./gradlew server
```

Následne vyberieme lokálny repozitár a vyberieme hru, ktorú chceme testovať a pridáme hráčov. Kliknutím na tlačidlo *startnewmatch* začne simulácia zápasu.
Pre jednoduchú simuláciu hry len s ľudskou interakciou môžete jednoducho skopírovať pravidlá hry a pridať ich do sekcii *rulesheet* na stránke:
http://ggp.stanford.edu/public/gametester.php
Po kliknutí na tlačidlo *initialize* sa spustí hra.