Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Automatický systém na hranie stolových hier

Bakalárska práca

2017
Ján Paholík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Automatický systém na hranie stolových hier

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Pavel Petrovič, PhD.

Bratislava, 2017
Ján Paholík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Ján Paholík
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: Automatický systém na hranie stolových hier
Automatic System for Playing Table Games

Cieľ: Pre účely letnej školy S3 bol v roku 2013 zostrojený robotický systém, ktorý dokáže hrať rozličné stolné hry pre jedného alebo dvoch hráčov. Systém pozostáva z robotického ramena a kamery a pripojeného stolného počítača, ktorý prostredníctvom kamery sleduje ťahy hráča a vymýšľa a vykonáva svoje ťahy pomocou ramena. Systém je navrhnutý všeobecne tak, aby sa dokázal naučiť hrať nové hry - stačí vyplniť špecifikáciu pravidiel v samostatne navrhnutom špecifikačnom jazyku. V doterajších prácach boli takto zadefinované rôzne hry, napríklad padacie piškvorky (connect 4). Tento systém však má niekoľko nedostatkov, kvôli ktorým nie je dobre prezentovateľný na verejnosti: jeho súcasťou nie je systém inverznej kinematiky, takže pozície polí na hracom pláne treba nastavovať ručne, pravidlá sa špecifikujú vo formáte "čo sa dá presunúť odkiaľ - kam", pričom tento formát síce vyhovuje kontrole správnosti ťahu, ale už menej vyhovuje generovaniu ťahov nových. Úlohou študenta je naučiť systém hrať novú hru, avšak s tým, že navrhne a implementuje nový spôsob špecifikácie pravidiel a doplní systém o modul inverznej kinematiky. Výsledný systém overí a nájde vhodný všeobecný algoritmus umelej inteligencie, ktorý bude hru hrať.

Literatúra: Petrovic P. (2015) Jazyk pre specifikáciu pravidiel stolných hier ako základ automatického inteligentného hracieho robotického systému, Kognícia a umelý zivot, p. 141-150, Trencianske teplice, Máj 2015. Repozitár projektu, dostupné online: github.com/Robotics-DAI-FMFI-UK/s3games
Marko Cvijović, Mislav Unger, Jan Corazza: The Art of Playing Games, 55-th International Symposium ELMAR-2013.

Vedúci: Mgr. Pavel Petrovič, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 03.10.2016

Dátum schválenia: 24.10.2016

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

....

študent vedúci práce

Poďakovanie:

Chcel by som poďakovať môjmu školiteľovi Mgr. Pavlovi Petrovičovi, PhD. za odbornú a ochotnú pomoc pri písaní tejto bakalárskej práce.

Abstrakt

Aplikácie umelej inteligencie naberajú v posledných rokoch na význame. Preto je štúdium praktických aj teoretických aspektov inteligentných systémov veľmi aktuálne. Názornou a zrozumiteľnou ukážkou pre širokú verejnosť je využitie umelej inteligencie a robotického systému na hranie stolových hier proti ľudskému hráčovi. Naša práca nadväzuje na predchádzajúci vývoj systému, ktorý automaticky hrá stolové hry. Pozostáva z robotického ramena, kamery, riadiaceho počítača a herného plánu. Obsahuje rôzne prehľadávacie algoritmy a heuristiky, ktoré je možné aplikovať na ľubovoľné stolové hry. Výnimočnou vlastnosťou systému je jeho univerzálnosť - definuje vlastný špecifikačný jazyk, v ktorom je možné zadefinovať pravidlá ľubovoľnej stolovej hry. Systém potom dokáže využiť algoritmy umelej inteligencie na hranie takto zadefinovanej hry proti ľudskému hráčovi. V našej práci analyzujeme nedostatky tohto systému - ktoré sú predovšetkým chýbajúca inverzná kinematika pre robotické rameno a nepraktický spôsob generovania susedných stavov v stavovom prehľadávacom priestore. Ponúkame riešenie oboch problémov - definujeme novú, efektívnejšiu verziu jazyka, čo demonštrujeme v uskutočnenom experimente, upravený špecifikačný jazyk výrazne zrýchlil hranie hier. Naše testy inverznej kinematiky úspešne overili podstatné zjednodušenie interakcie robota s hracím plánom a pripravili systém na prezentáciu robotickej skupiny na verejných podujatiach, čo bolo jedným z cieľov práce.

Kľúčové slová: inverzná kinematika, systém na hranie hier, umelá inteligencia, deklaratívny jazyk, robotika

Abstract

Aplications of artificial intelligence are getting more and more important in the last years. That is the reason, why is studying of practical and also theoretical aspects of intelligent systems so actual. Using artificial intelligence and robotic system for playing table games against human player is schematic and clear illustration for general public. Our thesis continues in previous development of system, which automatically plays table games. It consists of robotic arm, camera, control computer and game plan. It contains various search algorithms and heuristics, which can be applied for arbitrary table games. Special attribute of this system is its versatility - it defines own specification language which can be used to define rules of arbitrary table game. Then the system can use algorithms of artificial intelligence for playing a game, which was defined with the specification language, against human player. In this thesis we analyze disadvantages of this system, which is mainly missing inverse kinematics for robotic arm and impractical way of generating adjacent states in state search space. We offer solution for both of the problems - we define new, more effective version of the language, which is demonstrated in performed experiment and modified specification language accelerated playing game significantly. Our inverse kinematics tests successfully confirmed simplification of robot's interaction with game plan and prepared system for presentation of robotic group on public events, which was one of the main goals of this thesis.

Keywords: inverse kinematics, system for playing games, artificial intelligence, declarative language, robotics

Obsah

Zoznam obrázkov

Zoznam ukážok kódu

Úvod

Hranie hier proti počítačom má už niekoľko rokov svoju popularitu. Či už išlo o jednoduché alebo zložitejšie hry, ako napríklad šach, či hranie starej čínskej hry Go. Už pred asi 15 rokmi sa počítaču podarilo poraziť ľudského hráča v šachu, no hra Go predstavovala oveľa väčšiu výzvu, vzhľadom na počet možností jednotlivých ťahov. Hoci by to pred 10 rokmi málokto čakal, minulý rok sa podarilo počítaču poraziť jedného z najlepších ľudských hráčov práve v hre Go. Išlo o veľmi veľký úspech v oblasti umelej inteligencie.

Vytvoriť takýto automatický systém na hranie stolových hier, v našom prípade obohatenom o robotické rameno, je komplexná záležitosť, ktorá vyžaduje zručnosti v mnohých oblastiach. Ide predovšetkým o zručnosti elektrotechnické, programátorské, súvisiace prirodzene s logickým a matematickým uvažovaním, ale taktiež si vyžadujú kreatívne myslenie. Potrebné je avšak okrem všetkých týchto zručností aj množstvo času, prípadne väčší tím zaangažovaných ľudí. Keďže je táto práca bakalárska, čas aj tím sú záležitosti obmedzené a preto sa zameriavame iba na niektoré časti nášho projektu.

Pre túto bakalársku prácu som sa rozhodol kvôli zameraniu na robotiku, ktorá spája viaceré technické odvetvia. Okrem toho hranie hier v umelej inteligencii je oblasť, kde sa dá mnoho naučiť a vyskúšať - nové algoritmy, práca so štatistikami a konkrétne pri tejto téme si budem môcť vyskúšať prácu so špecifikačným jazykom na hranie hier, jeho úpravu a spracovanie.

Hry vo všeobecnosti majú veľký význam. Nejde len o príjemne strávený čas, ale aj o rozvíjanie logického myslenia. Hry pre jedného hráča formujú uvažovanie jednotlivca a hry pre dvoch hráčov zase vyžadujú aj dávku intuície, či uvažovania za protihráča.

Je zaujímavé hrať hru proti počítaču a ešte zaujímavejšie je hrať proti robotickému systému v reálnom svete. Často môže aj takáto hra spôsobiť motiváciu začať sa niečomu venovať. Boli by sme veľmi radi, ak sa systém s ktorým v tejto práci pracujeme, stane takýmto motivátorom pre deti a mládež na rôznych podujatiach, aby sa začali venovať informatike. Veríme, že bude dobre prezentovať aj našu fakultu a pritiahne tak nových šikovných študentov.

Vo všeobecnosti systémy hrajú hry pomocou rôznych algoritmov. Môže ísť o prehľadávania stavov hry a hľadanie cesty k víťaznej konfigurácii, či odhadnutie stavu a

následného ťahu pomocou heuristických metód. Oba prístupy majú svoje výhody aj nevýhody.

Práve spomenuté metódy sú využiteľné pri vytváraní všeobecných systémov, napríklad na hranie hier, kedy hru definujeme pomocou faktov o nej. Zapíšeme ju určitým deklaratívnym jazykom a všeobecné algoritmy už vedia pomocou takto zadefinovaných pravidiel hru odohrať.

Táto práca sa dotkne viacerých tém, ktoré sme tu načrtli a ukáže konkrétny pohľad do niektorých z nich.

Kapitola 1

Východiská

Táto práca stavia na už existujúcom projekte nazvanom S3Games [4]. Projekt bol vyvinutý pre účely Letnej školy vedy v Požege ešte v roku 2013. Na jeho vývoji sa podieľali viacerí ľudia - spomeňme za všetkých aspoň školiteľa mojej bakalárskej práce Mgr. Pavla Petroviča, PhD., ktorý bol jedným z hlavných členov. Ako východiskový bod sme tak mali už veľa vecí spravených. V tejto kapitole si ukážeme, ktoré to konkrétne boli a taktiež popíšeme základnú špecifikáciu celého systému.

1.1 Všeobecná špecifikácia systému

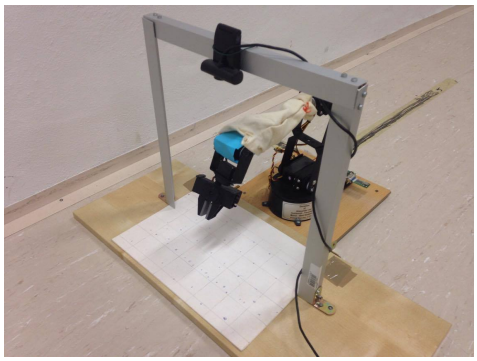
Ako už názov práce napovedá, našim cieľom je vytvoriť systém, ktorý by vedel hrať stolové hry. Vo všeobecnosti ide o hry pre jedného, alebo dvoch hráčov, ktoré spĺňajú nasledovné pravidlá [7]:

- systém bude hrať hry pre jedného alebo dvoch hráčov, ktorí sa v ťahoch striedajú
- hra pozostáva z postupnosti ťahov
- hra má cieľ, ktorému spravidla zodpovedá nejaká konfigurácia hry alebo jej parametre
- jeden ťah pozostáva z jedného alebo viacerých preložení figúrok, ktoré sa môžu klásť len na definované miesta
- ťahy by mali byť realizovateľné robotickým ramenom v štýle akcií "chyť a umiestni"
- situáciu hry a najmä jednotlivé kroky ťahov dokáže rozpoznať kamera
- hru je možné hrať aj virtuálne – na simulovanej hracej ploche
- na každom mieste môže byť naraz iba jedna figúrka – ostatné prípady sa riešia zmenou stavu figúrky, prípadne duplicitnými miestami

- každá figúrka aj miesto, kam sa kamene ukladajú, majú svoj jednoznačný identifikátor a typ

1.2 Hardvérová časť

Systém na hranie stolových hier, ktorý máme aktuálne k dispozícií sa skladá z robotického ramena Lynxmotion AL5D[1], webkamery a hracej plochy. Môžeme ho vidieť na obrázku 1.1.



Obr. 1.1: Fotografia celého systému

Rameno, ktoré môžeme vidieť na obrázku 1.2, má šesť stupňov voľnosti a je zakončené chápadlom, ktoré má maximálne rozpätie 3,175 cm. Servomotory ramena sú riadené rozhraním SSC-32U, ktoré prijíma a interpretuje príkazy na sériovej linke. Táto linka je pripojiteľná cez adaptér do bežného portu USB. Pristupovať k nej môžeme napríklad pomocou knižnice RXTX[3] z programovacieho jazyka Java.

Obr. 1.2: Robotické rameno Lynxmotion AL5D

Webkamera je taktiež pripojiteľná cez klasický USB port. Je umiestnená nad hracou plochou na kovovej konštrukcii a umožňuje tak detekovať aktuálny stav hry. Za týmto účelom je využívaná samostatná aplikácia napísaná v C++, ktorá na získavanie obrazu používa knižnicu OpenCV[2].

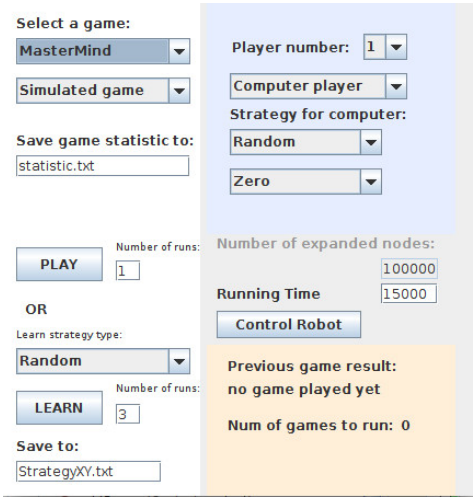
Hracia plocha je veľká približne ako papier A4. Figúrky využívame priamo z hraných hier. V prípade, že nevyhovujú robotickému ramenu, nahradíme ich alternatívnymi figúrkami, vyrobenými z dreva a farebne odlíšenými.

1.3 Softvérová časť

Softvérová časť tejto bakalárskej práce sa skladá z dvoch hlavných častí - frameworku a jazyka na špecifikáciu pravidiel stolových hier. V nasledujúcom texte si ich viac priblížime.

1.3.1 Framework

Aplikácia napísaná v jazyku Java tvorí softvérovú časť. Z pohľadu používateľa ide o grafickú aplikáciu, ktorá umožňuje hrať s naším systémom niektorú z aktuálne dostupných hier, ktoré systém vie hrať.



Obr. 1.3: Úvodné okno aplikácie s nastaveniami

Pozostáva z hlavného okna, kde je možné vybrať, ktorú hru chceme hrať, či pôjde o simulovanú, alebo hru v reálnom svete a taktiež je možné vybrať, ktorou stratégiou umelej inteligencie bude systém hrať. Môžeme ho vidieť na obrázku 1.3. K dispozícii je niekoľko známych prehľadávacích algoritmov a heuristických metód, vrátane prehľadávania do hĺbky (Depth First Search), prehľadávania do šírky (Breadth First Search), MiniMax, MonteCarlo, ale taktiež náhodný výber. Ďalej je možné nastaviť kam sa majú ukladať štatistické údaje, prípadne môžeme robota učiť hrať novú hru pomocou niekoľkých metód. K dispozícii je aj okno pre manuálnu kontrolu pohybov robota.

Užitočnou súčasťou je taktiež možnosť nechať hrať počítač hru v simulácii automaticky niekoľko krát za sebou. Výsledky z týchto hier si môžeme nechať uložiť do súboru a ďalej s nimi pracovať.

Po vybratí a spustení konkrétnej hry sa nám zobrazí okno s jej vizualizáciou. Teda pri simulovanej verzii hry v tejto vizualizácii aj posúvame figúrky, pomocou počítačovej myši, za ľudských hráčov. V systéme je ale možné nastaviť, že počítač bude hrať za oboch hráčov sám, v tomto prípade vo vizualizácii len sledujeme priebeh hry. Pri reálnej verzii hry v nej vidíme reprezentáciu toho, čo je na hracej ploche.

Do aplikácie je možné pridávať nové algoritmy, pridaním nových potomkov tried *Player* a *Strategy*. Pridanie novej heuristiky je tiež možné vytvorením nového typu potomka triedy *Heuristic*.

Framework zodpovedá aj za prečítanie pravidiel zo špecifikačného súboru, jeho spracovanie a uloženie do internej reprezentácie. Je v ňom zakomponovaná celá logika hrania hier, ovládanie robotického ramena, či hranie v simulácii.

1.3.2 Jazyk na špecifikáciu pravidiel

Typov programovacích jazykov je viacero - imperatívne, deklaratívne, procedurálne, funkcionálne, logické a mohli by sme ešte pokračovať mnohými ďalšími. Funkcionálne, logické a deklaratívne jazyky majú mnoho spoločného s procedurálnymi. Pomenovania, určenie rozsahu (scoping), taktiež typy, výrazy a koncepty riadenia priebehu výberu a rekurzie sa objavujú v každej z týchto skupín. Všetky jazyky musia byť postupne prehliadnuté a analyzované. [10]

Imperatívne jazyky by sme mohli popísať tým, že predstavujú riešenie, či už konkrétneho alebo všeobecného problému. Hovoria teda, ako sa ide niečo robiť.

Deklaratívne jazyky sa vyznačujú tým, že popisujú problém namiesto toho, aby určovali to, ako sa má riešiť. Náš špecifikačný jazyk sa radí práve medzi deklaratívne jazyky. Ide nám totiž o to, aby sme popísali pravidlá a nie o to, aby sme v špecifikačnom súbore riešili stratégiu hrania, respektíve algoritmus, ktorý sa použije.

Ak chceme zadefinovať novú hru, môžeme tak urobiť vytvorením a vyplnením nového textového súboru - špecifikácie hry. Pozrime sa, čo by mal, vo všeobecnosti, obsahovať.

Keďže systém má vedieť hrať ľubovoľnú hru, v rámci kritérií stanovených v podkapitole 1.1, bola potreba nájsť spôsob, ako zadávať pravidlá konkrétnych hier. Za týmto účelom vznikol nový špecifikačný jazyk.

Každá hra je definovaná n-ticou (ET, LT, L, E, EX, SC, EG, GR), ktorej význam je nasledovný [7]:

- ET (element types) – množina typov figúrok
- LT (location types) – množina typov miest, kam môžu byť figúrky umiestnené
- L (locations) – množina miest, kde môžu byť figúrky umiestnené

- E (elements) – množina figúrok a ich počiatočné umiestnenie
- EX (expressions) – pomocné funkcie, ktoré sa môžu využívať v pravidlách
- SC (scoring) – pravidlá o získavaní bodov v rozličných situáciách
- EG (end of game) – pravidlá o ukončení hry v rozličných situáciách
- GR (game rules) – pravidlá hry, podľa ktorých je možné ťahať.

Avšak k definícii hry spomenutá n-tica nestačí. Je potrebné zadefinovať jednotlivé farby figúrok, aby boli rozpoznateľné cez kameru, ďalej súradnice servomotorov na robotickom ramene pre jednotlivé políčka. Treba taktiež zvoliť obrázky pre jednotlivé figúrky a obrázok pozadia hracej plochy pre vizualizáciu hry, ktorá sa vytvára či už pri hre reálnej, alebo virtuálnej. Definovať je potrebné aj klikateľné oblasti elementov vo vizualizácii hry.

Keď zhrnieme potrebné informácie na špecifikáciu hry, dostaneme podobný výsledok ako je n-tica spomenutá vyššie. Avšak rozdiely, ktoré sú na prvý pohľad malé, doplnia podstatné informácie, bez ktorých by hra nefungovala.

Špecifikačný súbor sa skladá zo sekcií. Každá sekcia začína jej názvom uvedenom v hranatých zátvorkách. Uvádzame zoznam všetkých potrebných sekcií s krátkym popisom a príkladmi:

- BOARD (hracia plocha) – zadefinovanie obrázku ako pozadia hracej plochy vo virtuálnej verzii hry

```
1 [BOARD]
2 image=pozadiezabky.png // obrázok na pozadí
```

Ukážka kódu 1.1: Definícia hracej plochy

- REALBOARD ELEMENT TYPES (typy figúrok v reálnej hre) – zadefinovanie názvov, farieb (formát HSV), veľkosti figúrok

```
1 [REALBOARD ELEMENT TYPES]
2 name=zelenaZabka // názov
3 state=1 // stav
4 hueMin=71 // odtieň (minimum)
5 hueMax=192 // odtieň (maximum)
6 saturationMin=0.27 // sýtosť (minimum)
7 saturationMax=1.0 // sýtosť (maximum)
8 valueMin=0.09 // jas (minimum)
```

```
valueMax=0.9 // jas (maximum)
sizeMin=200 // minimálna veľkosť
sizeMax=5000 // maximálna veľkosť
```

Ukážka kódu 1.2: Definícia reálnej figúrky

- ELEMENT TYPES (typy figúrok) – názvy, stavy, obrázky a umiestnenie figúrok

```
[ELEMENT TYPES]
name=zelenaZabka // názov
states=1 // stav
image=zelenazabka.png // obrázok
point=1,1 // pozícia
```

Ukážka kódu 1.3: Definícia virtuálnej figúrky

- LOCATION TYPES (typy miest) – názvy, obrázky a tvary typov pozícií

```
[LOCATION TYPES]
name=kamen // názov
image=kamen.png // obrázok
shape=circle(30) // tvar
point=1,1 // pozícia
```

Ukážka kódu 1.4: Definícia typu pozície

- LOCATIONS (pozície) – názvy, typy, umiestnenie pozícií v hre, kde môžu byť umiestnené figúrky, nastavenie uhlov kĺbov pre robotické rameno

```
[LOCATIONS]
name=kamen(1) // názov
type=kamen // typ
point=40,250 // umiestnenie
camera=20,135
robot=-32.0,-10.0,15.0,-66.0,77.0,-32.0,-37.0,15.0,-66.0,77.0
    // nastavenie uhlov kĺbov robotického
    ramena
```

Ukážka kódu 1.5: Definícia pozície

- PLAYER NAMES (mená hráčov)

```
[PLAYER NAMES]
```

```
2   1=hrac1 // meno hráča
```

Ukážka kódu 1.6: Definícia mena hráča

- MOVABLE ELEMENTS (pohyblivé figúrky) – názvy, umiestnenie a vlastníci všetkých figúrok v hre

```
1   [MOVABLE ELEMENTS]
2   name=zelenazabka(1) // názov
3   type=zelenazabka // typ
4   player=1 // hráč
5   location=kamen(1) // pozícia
```

Ukážka kódu 1.7: Definícia figúrky

- EXPRESSIONS (výrazy) – vlastné definované výrazy, ktoré pomáhajú pri definícii pravidiel

```
1   KamenObsadeny($POS) // názov výrazu
2   $POS > 0
3   $POS <= 7
4   NOT EMPTY("s($POS)")
5   END // koniec výrazu
```

Ukážka kódu 1.8: Definícia výrazu overenia, či je pozícia voľná

- END OF GAME (koniec hry) – stavy, kedy sa hra končí a kto ju za daných okolností vyhrá

```
1   [END OF GAME]
2   situation=ZabkyDoma // názov výrazu, ktorý ak je
        splnený, nastane koniec hry
3   winner=1 // víťaz
```

Ukážka kódu 1.9: Definícia situácie, kedy hra končí

- GAME RULES (pravidlá hry) – názov, figúrka, pozície odkiaľ a kam, podmienka pravidla

```
1   [GAME RULES]
2   name=zelenaSkok // názov
3   element=zelenazabka($J) // figúrka
```

```
4    from=kamen($K) // odkiaľ
5    to=kamen($L) // kam
6    condition=KamenObsadeny($K+1) AND ($L==$K+2) //
         podmienka
```

Ukážka kódu 1.10: Definícia jedného pravidla hry

V rámci týchto sekcií (okrem sekcie EXPRESSIONS) sú dvojice *atribút=hodnota*, ktoré je potrebné nastaviť. V sekcii EXPRESSIONS sú definované vlastné, pomenované funkcie. Obsahujú výrazy priradenia, logického vetvenia, cykly, či niektoré z operátorov jazyka alebo zabudovaných primitívnych funkcií. Za spomenutie stojí, že v našom špecifikačnom jazyku vieme pracovať s množinami.

Medzi operátormi jazyka môžeme nájsť výrazy pre zistenie stavu figúrky, jej aktuálne umiestnenie, aktuálneho vlastníka alebo jej typ. Podobne vieme zisťovať informácie o mieste - obsah (aká figúrka sa na ňom aktuálne nachádza), typ, alebo, či miesto nie je prázdne. Pozrieť sa vieme aj na aktuálneho hráča a jeho nahrané body.

Primitívne funkcie slúžia na priamu manipuláciu hry, keď nastanú určité okolnosti. Vieme takto presunúť figúrku z miesta na miesto a zmeniť jej stav, či vlastníka. Po úspešnom vykonaní ťahu vieme tiež zmeniť hráča, ktorý je aktuálne na ťahu.

Pri hľadaní ďalšieho ťahu a teda pri prehľadávaní stavového priestoru môže nastať takzvaná explózia. Jedná sa o mnoho identických stavov, ktoré sú považované za rozličné. Táto situácia môže nastať napríklad vtedy, keď sú niektoré figúrky ešte mimo hracej plochy akoby v zásobníku. Keď nastane určitá udalosť a chceme z tohto zásobníka pridať figúrku do hry, je nám jedno, ktorá z figúrok to bude a teda nepotrebujeme mať pre každú z nich vygenerovaný samostatný ťah. Tento problém rieši náš špecifikačný jazyk označením políčka ako irelevantného. Figúrky, ktoré sú na ňom poukladané, sú považované akoby za jednu, teda aj vygenerovaný ťah je len jeden.

Kapitola 2

Podobné systémy

Existuje viacero systémov, ktoré sa zaoberajú podobnou problematikou akou sa zaoberáme v našej práci. Niektoré sa venujú inverznej kinematike, iné zase hraniu hier a algoritmom. V tejto kapitole stručne analyzujeme niekoľko takýchto prác.

Na našej fakulte bola vypracovaná bakalárska a diplomová práca na podobné témy. Ide o práce, ktoré sa dotýkali inverznej kinematiky, ale aj systému, s ktorým pracujeme.

Bakalárska práca Petra Pukančíka [8] riešila inverznú kinematiku pre podobné robotické rameno. V práci sa všeobecne uvádzajú možné prístupy riešenia tejto problematiky. Celkovo práca rieši prevažne hardvér, komunikačné protokoly a elektroniku. Inverznú kinematiku popisuje teoreticky, no praktickú ukážku výpočtu pri danom robotickom ramene nepopisuje.

Druhou prácou, ktorá sa dotýkala podobnej problematiky je diplomová práca, taktiež od Petra Pukančíka. [9] Pracuje s rovnakým systémom, aký je použitý v našej práci. Zaoberá sa kalibráciou kamery, doprednou a inverznou kinematikou. Taktiež v nej spracovali simuláciu herného procesu a implementovali rôzne algoritmy, ktoré je možné použiť pri hraní hier – náhodný výber ťahov, ale aj algoritmy umelej inteligencie Minimax a Minimax s alfa-beta orezávaním. Inverzná kinematika avšak nebola do systému implementovaná.

Príkladom systému, ktorý hrá jednu konkrétnu hru – Tic Tac Toe – s ľudským hráčom, je projekt [5], ktorý vytvoril Dr. Rainer Hessmer. Súčasťou systému je robotické rameno, typu AL5A, taktiež od firmy Lynxmotion, kamera a aplikácia, ktorá spracúva obraz z kamery, spraví rozhodnutia na základe aktuálnej situácie a presunie figúrku na určité miesto. Jedná sa tu o systém, ktorý vie hrať len jednu hru, teda má implementovaný algoritmus na riešenie len jedného konkrétneho problému.

Kapitola 3

Ciele práce

V tejto kapitole uvedieme, kam má naša práca smerovať a čo chceme dosiahnuť. Ako sme už spomenuli, k dispozícii máme relatívne komplexný systém na hranie stolových hier. Avšak, vždy sa nájde niečo, čo by sa dalo vylepšiť. V našom prípade teda budeme robiť zmeny na existujúcom systéme.

Prvou z nich je inverzná kinematika pre robotické rameno. Môže ísť o pomerne zložitý problém, ktorý vyžaduje náročné výpočty. Pozrieme sa, ako sa to bude dať riešiť v našom prípade, načrtneme si robotické rameno a uvážime, aký prístup zvoliť. Následne vytvoríme postup výpočtu inverznej kinematiky, ktorý aj implementujeme do existujúceho frameworku. Týmto novým spôsobom by sa nám malo zjednodušiť definovanie nových herných pozícií v špecifikačných súboroch hier. Chceme, avšak, ponechať systém kompatibilný aj s predošlou verziou definovania herných pozícií, aby bolo možné hrať už existujúce hry.

Druhou oblasťou, ktorej sa budeme v našej práci venovať, je úprava špecifikačného jazyka hier. Pôjde o úpravu, ktorá má zabezpečiť efektívnejšie prehľadávanie herného priestoru a generovať tak ťahy rýchlejšie.

Následne upravíme dve hry, jednu pre jedného hráča, druhú pre dvoch hráčov, do novej verzie špecifikačného jazyka a všetky úpravy tak otestujeme. Pozrieme sa aj na to, či sa naše zmeny prejavia na rýchlosti hry systému.

Spomínané ciele podrobne popíšeme v príslušných kapitolách. Zahrnieme do nich všetky návrhy, úpravy, testovania a komplikácie, s ktorými sme sa počas práce stretli. V závere zhrnieme, ktoré z cieľov sa nám podarilo alebo nepodarilo naplniť.

Kapitola 4

Inverzná kinematika

Robotické ramená sa skladajú, ako tie ľudské, z niekoľkých kĺbov. Každý jeden z týchto kĺbov je zvlášť ovládateľný a dá sa nastaviť do určitého uhla. Takéto ovládanie je veľmi prirodzené pri ručnej manipulácii s ramenom, kedy manuálne nastavíme každú časť ramena presne do takej polohy, ako potrebujeme.

Avšak, keď ku robotickému ramenu priradíme ľubovoľnú súradnicovú sústavu, situácia sa zmení. Ak totiž chceme, aby sa chápadlo ramena nachádzalo v určitom konkrétnom bode tejto sústavy, nastavenie uhlov už nie je priamočiarym riešením. Tu vstupuje na scénu inverzná kinematika, ktorá vie byť veľmi nápomocná.

Inverzná kinematika sa zaoberá hľadaním vhodnej konfigurácie jednotlivých kĺbov ramena, aby sa koncový bod ramena nachádzal v požadovanom bode priradenej súradnicovej sústavy a v konkrétnom uhle. [8]

V našej práci sa manipulovalo robotickým ramenom pomocou nastavenia uhlov jednotlivých kĺbov. Tento prístup ale nebol výhodný pre hranie hier, kedy musíme chápadlom ramena prísť do konkrétneho bodu, uchopiť figúrku a premiestniť ju do iného konkrétneho bodu.

Dôvod implementácie inverznej kinematiky bol teda prostý - zjednodušiť definovanie nových hier a zjednodušiť tiež prístup k jednotlivým políčkam daných hier pri programovaní. Ak sme totiž chceli v pôvodnom systéme zadefinovať novú hru, museli sme robotické rameno manuálne nastaviť na každé políčko hracej plochy. Keď sme sa dostali na určité políčko, uložili sme konfiguráciu uhlov ramena do špecifikačného súboru a pokračovali sme na ďalšie políčko. Je zrejmé, že takéto definovanie pozícií pre novú hru bolo veľmi zdĺhavé.

Chceli sme teda dosiahnuť, aby sme programu mohli povedať súradnice na hracej ploche a on sám dopočíta, do akých uhlov treba nastaviť jednotlivé kĺby, aby sa koncový bod ramena nachádzal na daných súradniciach.

Takýmto spôsobom je možné jednoduchšie definovať aj všetky pozície na hracej ploche tak, že zadáme súradnice napríklad prvého hracieho políčka a vzdialenosť, ktorá

medzi jednotlivými políčkami je. Ušetríme si tak mnoho práce, ktorá dlho trvá a dá sa pri nej ľahko pomýliť.

Riešení pre inverznú kinematiku existuje viacero. Spomeňme niektoré z nich:

- geometrický prístup
- algebraický prístup
- iteratívny prístup

S narastajúcim počtom stupňov voľnosti narastá aj zložitosť výpočtov. Dokonca už pre rameno s dvoma stupňami voľnosti máme dve riešenia. Ak sú ramená viac komplexné môže to viesť k potenciálne nekonečne veľa možným riešeniam. [6]

4.1 Počiatočné problémy

Ak sme chceli začať navrhovať a implementovať novú funkcionalitu, bolo potrebné sfunkčniť už vytvorenú aplikáciu, ktorá bola súčasťou toho, z čoho sme už vychádzali. Narazili sme na niekoľko komplikácií počas tohto procesu.

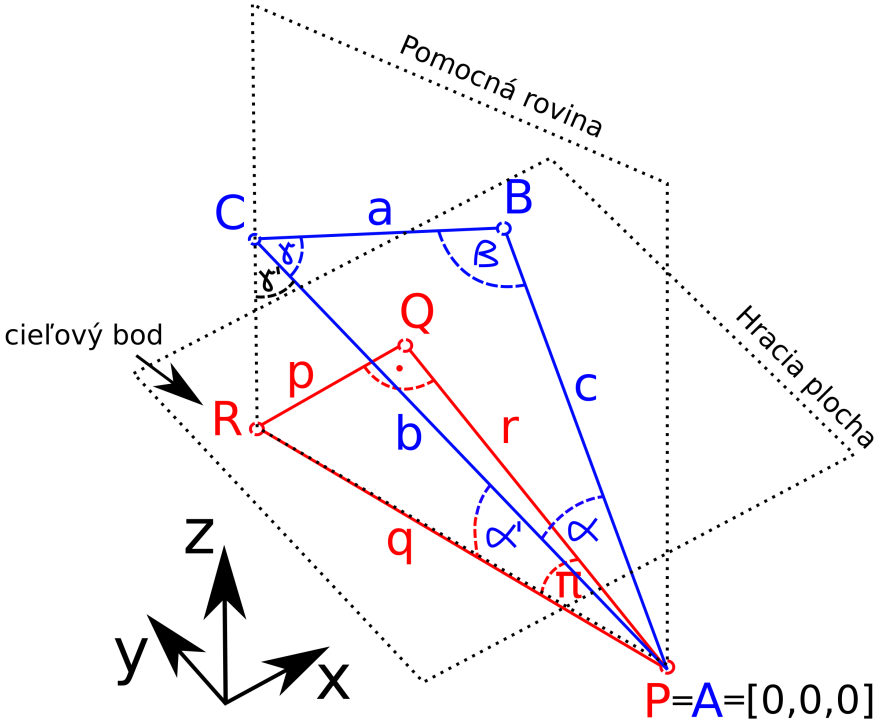
Aplikácia bola napísaná pre 32-bitovú verziu operačných systémov, avšak počítač, ktorý sme mali k dispozícií pre túto prácu mal 64-bitový operačný systém. Druhý problém bol v ovládači pre robotické rameno, ktorý bol tiež vydaný len pre 32-bitové operačné systémy. Posledným problémom bolo, že aplikácia bola vyvíjaná na systéme Windows a nebola kompatibilná so systémom Linux, ktorý bol nainštalovaný na počítači, ktorý sme mali k dispozícii.

Prvé dva problémy boli pomerne ľahko vyriešiteľné - stačilo použiť 32-bitové verzie vývojového prostredia a vývojárskej sady jazyka Java. Tretia komplikácia si vyžadovala zásah do kódu aplikácie, pretože niektoré z použitých funkcií boli špecifické pre systém Windows. Nešlo avšak o funkcie, ktoré by neboli ľahko nahraditeľné inými, prístupnými na systéme Linux. Podarilo sa tak úspešne spustiť aplikáciu a aj komunikáciu s robotickým ramenom a kamerou.

4.2 Návrh výpočtu

Keď už pôvodná aplikácia fungovala korektne, mohli sme začať s návrhom výpočtu inverznej kinematiky. V odbornej literatúre sa hovorí o viacerých možných prístupoch.

Začali sme nákresom nášho robotického ramena, ku ktorému sme priradili súradnicovú sústavu s osami x, y a z. Môžeme ho vidieť na obrázku 4.1. Os x je v smere naľavo a napravo od ramena a má kladné hodnoty napravo od ramena. Os y je v smere dopredu a vzad od ramena a naberá kladné hodnoty smerom vpred a os z je v smere



Obr. 4.1: Nákres ramena v trojuholníkoch

hore a dole, pričom kladné hodnoty má smerom hore. Základňa robotického ramena predstavuje bod $[0, 0, 0]$. Za jednotku súradnicovej sústavy sme zvolili centimeter, kvôli jednoduchému meraniu vzdialeností políčok na hracích plánoch hier.

V nákrese sme vyznačili kĺby ramena ako body a zakreslili modelovú situáciu, kedy sa koncový bod ramena nachádza v konkrétnej pozícii. Vyznačili sme uhly kĺbov a doplnili do trojuholníkov.

Pre ilustratívne účely, predstavme si robotické rameno ako ľudskú ruku, ktorá sa skladá z ramena, predlaktia a zápästia. Potom úsečka AB predstavuje rameno, úsečka BC predlaktie a úsečka CR zápästie.

Ukázalo sa, že na výpočet inverznej kinematiky bude v našom prípade stačiť spočítanie niekoľkých uhlov v dvoch pravouhlých trojuholníkoch a jednom rôznoramennom trojuholníku.

Najskôr ukážeme čiastkové výpočty uhlov. Poznáme súradnice bodu R (označme ich x, y, z) kam sa má dostať koncový bod robotického ramena a všetky dĺžky jednotlivých častí ramien, teda úsečky AB, BC a CR. Samotné pomocné výpočty teda vyzerajú

nasledovne:

$$\begin{aligned} \alpha &= \cos^{-1}\left(\frac{b^2+c^2-a^2}{2bc}\right) \\ \beta &= \cos^{-1}\left(\frac{c^2+a^2-b^2}{2ac}\right) \\ \gamma &= 180-\alpha-\beta \\ \pi &= \sin^{-1}\left(\frac{p}{q}\right) \\ \alpha' &= \sin^{-1}\left(\frac{|CR|}{b}\right) \\ \gamma' &= \sin^{-1}\left(\frac{q}{b}\right) \end{aligned} \tag{4.1}$$

Keď máme vypočítané čiastkové uhly, použijeme ich na výpočet uhlov, do ktorých nastavíme jednotlivé kĺby ramena. Musíme zohľadniť aj to, akú má každý z kĺbov počiatočnú polohu. V konečnej verzii to vyzerá nasledovne (uo – uhol otočenia, ur – uhol ramena, up – uhol predlaktia, uz – uhol zápästia):

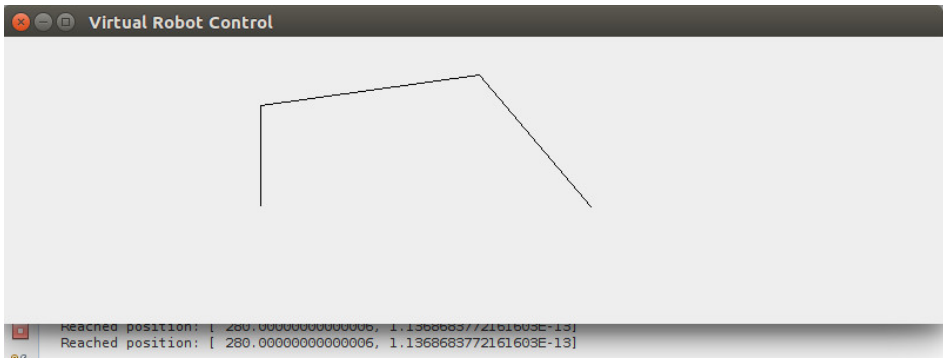
$$\begin{aligned} uo &= \begin{cases} \pi & | \quad x >= 0 \\ -\pi & | \quad inak \end{cases} \\ ur &= -(90-(\alpha+\alpha')) \\ up &= -(90-\beta) \\ uz &= -(180-(\gamma+\gamma')) \end{aligned} \tag{4.2}$$

4.3 Implementácia

Po ukončení týchto teoretických výpočtov sme prešli k implementácii. Chceli sme dosiahnuť, aby sme mohli definovať to, kde sa nachádzajú jednotlivé miesta hracích plôch dvomi spôsobmi. Prvý spôsob bol pôvodný, kedy sme zadali konkrétne uhly pre každý z kĺbov, pre každú pozíciu na hracej ploche, viď ukážka kódu 1.5. Druhý spôsob bol náš nový – definovanie pomocou súradníc v centimetroch, viď ukážka kódu 4.1.

```
1 [LOCATIONS]
2 name=stone(1) // pozícia kameň č. 1
3 // ostatné údaje
4 robot=-9,20,0 // súradnice pozície v centimetroch
5
6 ... // ostatné pozície
```

Ukážka kódu 4.1: Definovanie pozícií v hre Žabky – nová verzia



Obr. 4.2: Vizualizácia robotického ramena, virtuálny test inverznej kinematiky

V našom frameworku drží informácie o pozíciach trieda *RobotLocation*, ktorá vyžaduje pri vytváraní novej inštancie textový reťazec. Do tohto reťazca sú vždy dosadené hodnoty atribútov *robot* jednotlivých lokácií. Aby naše dve verzie zadávania miest mohli fungovať spolu, doplnili sme do nášho frameworku jednoduché rozhodovanie podľa počtu zadaných čísel v textovom reťazci. Ak je čísel päť, bez ďalšieho spracovania sa uložia do poľa *angles* (uhly). Ak sú tri, prebehne výpočet uhlov zo súradníc a následne sa tiež uložia do poľa *angles*. Ak sa stane, že dostaneme súradnice, ktoré sú pre naše rameno nedosiahnuteľné, program vyhodí výnimku a skončí.

4.4 Testovanie

Prvotné testovanie sme uskutočnili virtuálne. Vytvorili sme jednoduchú aplikáciu, ktorá simuluje naše robotické rameno. Môžeme ju vidieť na obrázku 4.2. Do konštruktora objektu, ktorý vykresľuje rameno, zadáme súradnice bodu, v ktorom má byť koncový bod ramena. Následne prebehne výpočet vzájomných uhlov jednotlivých úsečiek, ktoré predstavujú časti ramena a otvorí okno so zobrazeným ramenom. V konzole môžeme vidieť, aké sú súradnice koncového bodu "zápästia" robotickej ruky. Toto testovanie ukázalo správnosť implementácie inverznej kinematiky.

Testovali sme aj priamo s robotickým ramenom. Vyskúšali sme hru Žabky (popísaná v časti 5.1) a robotické rameno s malými odchýlkami dosahovalo chcené pozície.

Definovanie miest je teda omnoho jednoduchšie, no neprišli sme ani o možnosť hrať hry, ktoré boli definované starým spôsobom.

Kapitola 5

Úprava špecifikačného jazyka pre stolové hry

Jedným z cieľov tejto bakalárskej práce bolo upraviť špecifikačný jazyk, v ktorom sa v našom systéme špecifikujú stolové hry. V tejto kapitole popíšeme navrhované úpravy, ich implementácie a uvedieme jednoduché príklady ako upravený jazyk používať. Spomenieme taktiež, prečo boli zmeny v jazyku potrebné.

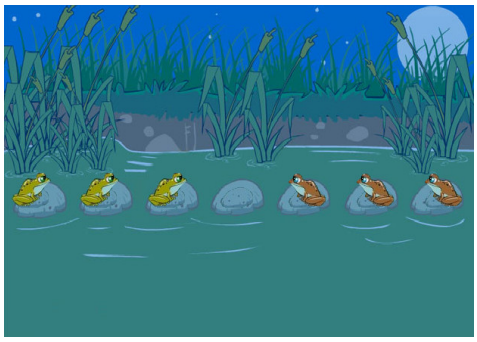
5.1 Pôvodný špecifikačný jazyk

Na úvod vysvetlíme ešte niektoré podrobnosti, ako pôvodný špecifikačný jazyk fungoval a ako sa v ňom dali zadefinovať hry. Ako sme už v predošlom texte spomenuli, definícia hry predstavuje samostatný textový súbor.

V zadefinovanom špecifikačnom súbore sa nachádzajú všetky potrebné informácie ku hraniu danej stolovej hry, či už vo virtuálnej alebo reálnej verzii. Celý súbor sa pri spúšťaní konkrétnej hry rozanalyzuje. Následne sa vytvorí jeho interpretácia vo vnútri spusteného frameworku a inicializuje sa počiatočný stav hry. S týmto vnútorným stavom sa dá počas hry manipulovať. Po začatí hry už nie je nutné zo špecifikačného súboru nič čítať, ale všetky potrebné informácie sú už načítané v spustenej aplikácii.

Ako príklad si zoberme jednoduchú logickú hru pre jedného hráča s názvom Žabky, viď obrázok 5.1. Cieľom je presunúť zelené žabky na miesta hnedých a naopak hnedé na miesta zelených. Pohybovať sa môžu o krok vpred, ak je pred nimi voľný kameň, alebo môžu preskočiť jednu ľubovoľnú žabku, pokiaľ sa za ňou nachádza voľný kameň. Cúvať alebo skákať späť žabky nemôžu. Spôsob, akým by sme tieto jednoduché pravidlá zapísali do špecifikačného jazyka, je na ukážke kódu 5.1.

```
1 name=zelenaKrok // pravidlo pre posun o pozíciu vpred
2 element=zelenazabka($J) // posúvame zelenú žabku s indexom J
```



Obr. 5.1: Hra Žabky

```
3  from=kamen($K) // z kameňa s indexom K
4  to=kamen($L) // na kameň s indexom L
5  condition=$L==$K+1 // pričom musí platiť podmienka, že index K
       je o 1 väčší ako index L
6
7  name=zelenaSkok // pravidlo o skákaní
8  element=zelenazabka($J)
9  from=kamen($K)
10 to=kamen($L)
11 // výraz (funkcia) KamenObsadeny je definovaná v časti
       EXPRESSIONS (výrazy)
12 condition=KamenObsadeny($K+1) AND ($L==$K+2) // kameň pred
       žabkou musí byť obsadený a musí skákať na kameň vzdialený
       o dva
```

Ukážka kódu 5.1: Časť špecifikácie hry Žabky

Všimnime si, že v definícii pravidla máme kľúčové slovo *condition* (podmienka), v ktorom je vložený výraz. Ak je splnený, pravidlo je možné uplatniť.

Dôležitou súčasťou definície pravidiel je ešte jeden atribút s názvom *followup* (nasledujúce, nasleduje). Ide o zadanie akcie, ktorá sa má udiať hneď potom, ako sa vykoná ťah podľa konkrétneho pravidla. Napríklad v hre Alquerque, ktorú bližšie popisujeme v sekcii 5.4, keď figúrka hráča č. 1 preskočí figúrku hráča č. 2, musí byť figúrka hráča č. 2 odstránená. Do atribútu *followup* by sme teda vložili funkciu, ktorá odstráni preskočenú figúrku z hracej plochy na definované irelevantné miesto.

Či už chceme definovať akciu, ktorá sa má stať po vykonaní ťahu, alebo overiť podmienku pri pravidlách, alebo vyhodnotiť, či už nastal koniec hry a kto vyhral, potrebujeme definovať vlastné funkcie a použiť v nich funkcie vstavané a operátory jazyka.

Vlastné funkcie si môžeme definovať v časti EXPRESSIONS (výrazy) špecifikačného súboru. Ide v podstate o logické formuly, pričom jednotlivé riadky sú prepojené operátorom konjunkcie. Všeobecný a existenčný kvantifikátor predstavujú výrazy s názvom *FORALL* (pre všetky) a *FORSOME* (pre niektoré, pre aspoň jedno).

Funkcie sa začínajú názvom, nasleduje žiadny alebo niekoľko argumentov. V ďalších riadkoch je, klasicky, telo funkcie a kľúčové slovo *END* (koniec) funkciu ukončuje.

V tele môžeme vytvárať premenné, ktoré musia začínať znakom $ a priraďovať im hodnoty. Tieto môžu byť logické, číselné, čisté textové reťazce, textové reťazce s premennými, alebo množiny.

Funkcia sa vyhodnocuje riadok po riadku, až kým sa niektorý z riadkov nevyhodnotí na *false* (nepravda). Všetko ostatné, či už čísla, reťazce, množiny a, samozrejme, hodnota *true* (pravda) je považované za pravdivé a teda neukončuje funkciu. Návratová hodnota je posledný výraz vo funkcii, alebo *false* (nepravda), ak sa funkcia skončila pred vyhodnotením posledného riadku.

Treba si dávať pozor na používanie názvov premenných, nakoľko všetky premenné sú globálne a môže sa ľahko stať, že si prepíšeme hodnotu v určitej funkcii a následne očakávame, že bude nezmenená.

Teraz uvedieme základné operátory jazyka výrazov. Ide o porovnávania rovnosti, väčší, menší, väčší alebo rovný, menší alebo rovný, ďalej sčitovanie, odčitovanie, násobenie, delenie, zvyšok po delení a absolútna hodnota. Pracovať môžeme aj s množinami a zisťovať, či je určitá množina podmnožinou inej, či sa prvok nachádza v množine. Samozrejme môžeme robiť operácie ako zjednotenie, prienik a rozdiel množín.

Nezaobišli by sme sa ani bez možnosti stanovenia podmienok pomocou *IF* (ak). Ich syntax v našom špecifikačnom jazyku je na ukážke kódu 5.2

```
1   IF (podmienka, vyraz c. 1, vyraz c. 2).
```

Ukážka kódu 5.2: Ukážka syntaxe podmienky IF

Podmienka sa vyhodnotí a ak bola splnená, vykoná sa výraz č. 1, ak splnená nebola, vykoná sa výraz č. 2. Ako výrazy môžeme napísať aj názvy funkcií.

Keďže sa v jazyku nachádza podmienka, prirodzene by sme mohli očakávať aj cyklus *WHILE* (kým). Tento sa, avšak, v našom jazyku nenachádza. Nahrádzajú ich výrazy *FORALL* (pre všetky) a *FORSOME* (pre niektoré), ktoré majú syntax ako je na ukážke kódu 5.3.

```
1   FORALL (index, od, do, vyraz)
2   FORSOME (index, od, do, vyraz)
```

Ukážka kódu 5.3: Ukážka syntaxe FORALL a FORSOME

Cyklus *FORALL* na začiatku priradí do premennej *index* hodnotu *od*, vyhodnotí výraz a postupne pokračuje priraďovaním hodnôt o jeden väčších do indexu, až kým nedosiahne hodnotu *do*. Celý cyklus sa vyhodnotí na *true* (pravda), ak sa každý z vyhodnotených výrazov počas jeho behu vyhodnotil na pravdivú hodnotu, pričom máme

na mysli akúkoľvek hodnotu okrem *false* (nepravda).

Cyklus *FORSOME* funguje podobne, iba s tým rozdielom, že sa vyhodnotí na *true* (pravda), ak sa aspoň jeden z výrazov počas jeho behu vyhodnotil na pravdivú hodnotu.

5.2 Pôvodné generovanie nových ťahov

Pozrime sa teraz, ako prebiehalo generovanie prípustných ťahov vo frameworku z pôvodnej verzie špecifikačného jazyka. Pri ich hľadaní sa do každého pravidla dosadili postupne všetky figúrky. Následne sa prehľadávalo cez všetky pozície na hracej ploche. Zisťovalo sa, či môžeme figúrku presunúť v súlade s pravidlom na novú testovanú pozíciu. Ukázanie princípu v pseudokóde je na ukážke 5.4.

```
1 for (Pravidlo p: pravidla) {
2    for (Figurka f: figurka) {
3       for (Pozicia poz: pozicie) {
4          if ( ... ) { // ak sa dá presunúť kameň k na pozíciu
               poz podľa pravidla p
5             pridajDoMoznychTahov()
6          }
7       }
8    }
9 }
```

Ukážka kódu 5.4: Princíp generovania nových ťahov

5.3 Návrh a implementácia úprav

V tejto časti si priblížime, aké zmeny sme sa rozhodli spraviť v špecifikačnom jazyku a samotnom frameworku. Priblížime taktiež dôvody, ktoré nás k týmto rozhodnutiam viedli.

Z ukážky kódu 5.4 je na prvý pohľad vidieť, že nejde o veľmi efektívne riešenie generovania nových možných ťahov a robíme veľa zbytočnej práce navyše. Formálne by sme zložitosť mohli vyjadriť tak, ako je uvedené v zápise 5.1 (p – počet pravidiel, f – počet figúrok, poz – počet pozícií).

$$T(p, f, poz) \in O(p \cdot f \cdot poz) \tag{5.1}$$

Jedna z variant, ako by sme to mohli spraviť rýchlejšie, je pozmeniť generovanie tak, aby nám každá figúrka v hre "povedala", kde všade môže ísť. To znamená, že by nám stačilo "opýtať sa" všetkých figúrok hráča, ktorý je na ťahu, kam sa môžu pohnúť

a zoznam všetkých možných ťahov by sme mali k dispozícii. Tým by sme dosiahli zložitosť uvedenú v rovnosti 5.2 (p – počet pravidiel, f – počet figúrok, poz – počet pozícií).

$$T(p, f, poz) \in O(p \cdot f) \tag{5.2}$$

Práve takéto zefektívnenie sme sa rozhodli implementovať. Môžeme si všimnúť, že sme sa úplne nezbavili závislosti od počtu pravidiel, nakoľko pre konkrétnu figúrku musíme vždy zistiť, ktoré pravidlo jej prislúcha. Na druhej strane sme sa zbavili závislosti od celkového počtu políčok na hracej ploche.

5.3.1 Špecifikačný jazyk

Najprv sme sa museli rozhodnúť, akým spôsobom budeme zapisovať pravidlá hry v upravenom špecifikačnom jazyku. Keď sa pozrieme napríklad na ukážku kódu 5.1, vidíme, že v pravidlách môžeme zadávať pozície, z ktorej kam chceme ísť, pomocou indexov. Indexy sú označované so znakom $ na začiatku ich názvu. Keď sa overuje, či na tieto pozície môžeme použiť aktuálne testované pravidlo, doplnia sa do týchto indexov čísla pozícií automaticky.

Na tej istej ukážke si môžeme všimnúť aj to, že pozícia z ktorej ideme je uložená v atribúte *from* (z). Hodnota tohto atribútu sa nám zíde aj v našej novej verzii špecifikačného jazyka, pretože práve podľa nej budeme určovať, kam sa môže figúrka ďalej hýbať. Avšak, atribút *to* (do, na) budeme chcieť zmeniť na nový, nami zadefinovaný atribút *toLocations* (na pozície). Do tohto nového atribútu vložíme všetky možné ťahy, ktoré môžeme spraviť z pozície uloženej v atribúte *from*.

Princíp úpravy je teda jednoduchý: nahradíme kľúčové slovo *to* kľúčovým slovom *toLocations*. Priradíme do neho množinu všetkých možných ťahov, ktorú vygenerujeme v časti EXPRESSIONS (výrazy) nášho špecifikačného súboru.

Pozrime sa na samotnú úpravu špecifikačného súboru hry Žabky. V ukážke kódu 5.5 vidíme, že sme pridali náš nový atribút *toLocations* a vložili sme do neho výsledok funkcie *GetPossibleMovesGreenFrog* (získaj možné ťahy pre zelené žabky). Podmienka je nastavená konštantne na pravdivú, lebo si vždy môžeme vypýtať zoznam možných ťahov, či už nejaké ťahy bude obsahovať, alebo nie.

Funkciu na získanie možných ťahov sme si definovali vopred a môžeme ju vidieť v ukážke 5.6. Ako argument berie pozíciu a vracia všetky ťahy, ktoré sú uskutočniteľné z tejto pozície. To znamená, že všetky pravidlá, pre tento typ figúrok musia byť zahrnuté v tejto funkcii.

V prvom riadku sme premennej s názvom *POSSIBLEMOVES* (možné ťahy) priradili prázdnu množinu. Následne sme v ďalších dvoch riadkoch do premenných *STEP-*

POS (pozícia krok) a *JUMPPOS* (pozícia skok) uložili indexy miest, ktoré by zodpovedali kroku, resp. skoku z pôvodnej pozície.

V piatom až siedmom riadku overujeme, či môžeme spraviť skok – tzn. či je obsadené miesto s indexom *POS + 1* a či je voľná pozícia s indexom *JUMPPOS*. Ak sú podmienky splnené, pridáme do množiny možných ťahov miesto zodpovedajúce skoku.

V ôsmom až desiatom riadku analogicky overujeme, či môžeme spraviť krok z pôvodnej pozície a ak áno, pridáme ho do množiny všetkých možných ťahov.

V jedenástom riadku vrátime získanú množinu ťahov a funkcia sa končí.

```
1  name=greenFrogMove
2  element=greenfrog($J)
3  from=stone($K)
4  toLocations=GetPossibleMovesGreenFrog($K)
5  condition=true
```

Ukážka kódu 5.5: Nové definovanie pravidiel hry Žabky

```
1  GetPossibleMovesGreenFrog($POS)
2  $POSSIBLEMOVES = {}
3  $STEPPOS = ($POS + 1)
4  $JUMPPOS = ($POS + 2)
5  IF(StoneOccupied($POS+1) AND MoveAvailable($JUMPPOS),
6     $POSSIBLEMOVES = $POSSIBLEMOVES UNION {"stone($JUMPPOS)"},
7     true)
8  IF(MoveAvailable($STEPPOS),
9     $POSSIBLEMOVES = $POSSIBLEMOVES UNION {"stone($STEPPOS)"},
10    true)
11 $POSSIBLEMOVES
12 END
```

Ukážka kódu 5.6: Funkcia, ktorá vracia všetky možné ťahy z pozície $POS

5.3.2 Framework

Keď sme mali takto pripravený atribút *toLocations* (na pozície) v rámci každého pravidla pre daný typ figúrok, ktorý nám vráti všetky možné ťahy z konkrétnej pozície, mohli sme začať s úpravou frameworku.

Ako prvé sme museli preštudovať, ako prebieha načítanie a spracovanie špecifikačného súboru. Tento proces môžeme rozdeliť do dvoch krokov:

1. čítanie súboru po riadkoch a detekovanie jednotlivých sekcií,

2. analýza atribútov sekcií a transformácia ich hodnôt do objektov príslušných typov.

Samotná transformácia textových reťazcov do vnútornej interpretácie frameworku je jednou z najnáročnejších častí tohto procesu.

Nás bližšie zatiaľ zaujímalo spracovanie časti GAME RULES (pravidlá hry), keďže sme do nej pridali nové kľúčové slovo. Potrebovali sme ho identifikovať, uložiť jeho hodnotu do príslušného objektu a pri generovaní nových ťahov zistiť, či sú pravidlá zadané starým alebo novým spôsobom.

Po analýze frameworku sme zistili, že trieda *GameRule* (pravidlo hry) obsahuje všetky informácie každého z pravidiel. Doplnili sme do nej teda vlastnosť *toLocations* (na pozície). Taktiež sme pridali príkaz na spracovanie tejto vlastnosti do triedy *GameSpecificationParser* (syntaktický analyzátor špecifikácie hry), ktorá zabezpečuje spracovanie textu súboru do internej reprezentácie.

Následne sme potrebovali zmeniť samotný spôsob generovania nových ťahov. V triede *GameRule* (pravidlo hry) sa nachádza metóda *GetMatchingMoves* (získaj prípustné ťahy), ktorá zabezpečovala zhromaždenie ťahov, ktoré boli uskutočniteľné podľa daného pravidlo v určitom stave hry. Ako argument berie, okrem iných, objekt figúrky. Podľa tohto objektu zistí, na akej pozícii sa figúrka nachádza a potom prechádza cez všetky možné pozície na hracej ploche a zisťuje, či je posun z pôvodnej na novú pozíciu platný podľa daného pravidla. Okrem toho metóda ešte kontroluje niekoľko, pre nás irelevantných, vecí.

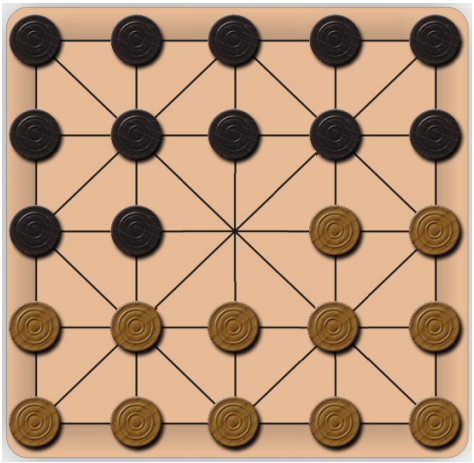
Do tejto metódy sme teda doplnili rozdelenie behu programu na základe toho, či máme pre dané pravidlo definovanú vlastnosť *toLocations*, alebo nie. Vnútri nami pridaného bloku sme už len nechali vyhodnotiť vlastnosť, aby sme získali vygenerovanú množinu.

5.3.3 Testovanie

Keď sme mali hotovú implementáciu, mohli sme prejsť k testovaniu. Ako to často pri programovaní býva, nič nevyjde na prvýkrát. Zistili sme, že pôvodná verzia frameworku mala nesprávne naprogramovanú prácu s množinami, ktorá bola kľúčová pre zmeny, ktoré sme implementovali. Problémom bolo, že nebolo možné vytvoriť prázdnu množinu ani priradiť do nej niektoré prvky.

Prešli sme teda do druhej fázy štúdia frameworku, zvlášť do časti, kde sa spracováva a analyzuje špecifikačný súbor, sekcia EXPRESSIONS (výrazy). Zistili sme, že výrazy sú rozdelené na lexémy a tie sú zvlášť analyzované.

Po podrobnom preskúmaní, skúšaní a krokovaní (debugovaní) programu sme prišli na to, že chyba bola v nesprávnej analýze lexém a chybnom vytvorení objektu prislúchajúceho množine. Medzi identifikátormi lexém chýbali práve znaky, ktoré mali



Obr. 5.2: Hra Alquerque

definovať množiny ("{}"). Následne sme ešte upravili triedu reprezentujúcu množiny a testovanie bolo poprvýkrát úspešné a náš systém si konečne zahral zefektívnenú verziu hry Žabky.

5.4 Úprava hry Alquerque

Úspešne sa nám už podarilo zefektívniť hru pre jedného hráča, no chceli sme ešte upraviť aj jednu hru pre dvoch hráčov. Vybrali sme si hru Alquerque.

Jedná sa o hru, ktorú tvorí 12 bielych a 12 čiernych figúrok rozostavených na hracej ploche 5x5 bodov. Body sú pospájané úsečkami horizontálne, vertikálne a po niektorých uhlopriečkach. Všetky spojenia bodov a začiatočnú konfiguráciu hry je možné vidieť na obrázku 5.2. Bod v strede je voľný. Cieľom je odstrániť všetky súperove figúrky. Pravidlá hry sú nasledovné:

- hru začína hráč s bielymi figúrkami
- hráči sa striedajú po jednom ťahu
- figúrka sa môže posunúť v ľubovoľnom smere na susedný bod, pokiaľ je voľný
- figúrka musí skákať, pokiaľ je na jednom z jej susedných bodov súperova figúrka a bod za ňou je voľný
- figúrka musí spraviť aj viac skokov za sebou, ak to situácia umožňuje, pričom po každom skoku môže meniť smer
- ak figúrka mala skákať a neskákala, je odstránená z hry

- hra končí ak boli odstránené všetky figúrky jedného z hráčov, alebo ak sa jeden z hráčov dostal do obkľúčenia a nemôže uskutočniť ťah, aj keď je na rade; v oboch prípadoch vyhráva jeho súper

5.4.1 Problémy v upravenom jazyku

Ako sme už spomenuli v podkapitole 5.1, pri definovaní pravidiel máme možnosť určiť akciu, ktorá sa má vykonať potom, ako sa pravidlo uplatní. Toto riešenie bolo v pôvodnom jazyku intuitívne a priamočiare, pretože každé pravidlo určovalo presun figúrky z jedného miesta na iné jedno miesto. To znamená, že sme jednoducho vedeli zistiť napríklad to, ktorá figúrka bola preskočená a teda sme ju aj vedeli odstrániť.

Problém nastal vtedy, keď sme chceli skombinovať naše nové definovanie pravidiel s akciami po uplatnení pravidiel. Z pravidla, ktoré definovalo jedno konkrétne miesto sme totiž spravili také, podľa ktorého sa dalo dostať na všetky miesta, kam konkrétna figúrka mohla ísť. To znamená, že ak by sme napríklad mohli v hre Alquerque s jednou figúrkou preskočiť rôzne súperove figúrky do viacerých strán, nevedeli by sme následne určiť, ktorý z týchto možných ťahov sa vykonal. Tým pádom by bolo nemožné vyhodiť figúrku z hry a tá by nenapredovala.

Tento problém sa ukázal, všeobecne pre hry, ako principiálne veľmi zásadný a bolo potrebné ho vyriešiť.

5.4.2 Návrh úpravy a implementácia

Už spočiatku, po objavení problému, bolo jasné, že bude potrebné prerobiť definíciu množiny možných ťahov na množinu možných ťahov s udalosťami, ktoré majú nastať po nich tak, aby boli určitým spôsobom spojené. Takto by sme vedeli presne určiť, ktorý pohyb figúrky sa vykonal a tak aj vykonať príslušnú následnú akciu.

Keď sme prvýkrát upravili pravidlá hry Alquerque do novej verzie špecifikačného jazyka, vyzeralo to, principiálne, ako na ukážke kódu 5.7. Jednoducho sme priradili overenú pozíciu do množiny možných ťahov.

```
1 $TAHY = $TAHY UNION {"pozicia($X,$Y)"}
```

Ukážka kódu 5.7: Priradenie pozície do množiny možných ťahov

Teraz sme chceli dosiahnuť, aby zápis množiny vyzeral ako na ukážke kódu 5.8. Potrebovali sme zabezpečiť, aby *AkciaPoTahu* dostala všetky potrebné informácie o vykonanom ťahu v argumentoch. Jednalo sa o pôvodnú a koncovú pozíciu a voliteľne ešte typ figúrky.

```
$TAHY = $TAHY UNION {{"pozicia($X,$Y)", AkciaPoTahu(...)}}
```

Ukážka kódu 5.8: Priradenie pozície a následnej akcie do množiny možných ťahov

V špecifikačnom súbore sme teda upravili zápisy množiny, ktoré vracia atribút *toLocations* (na pozície) a funkciu, ktorá sa mala o prípadné vyhodenie figúrok postarať.

Po prvých krátkych testoch sme si ale uvedomili ešte niekoľko podstatných faktov. Prvým bolo, že premenné sú v celom špecifikačnom súbore a počas celého behu hry globálne. Druhým faktom bolo to, že akcia, ktorá sa má zavolať a vykonať až po ťahu, prebehne už pri hľadaní všetkých možných ťahov. Keď, totiž, framework vyhodnocuje čiastkové množiny s jednotlivými pozíciami, vyhodnocujú sa aj akcie, ktoré k nim prináležia. Problémom, avšak, bolo, že ak by sa aj tieto akcie nevyhodnocovali hneď s danými množinami, ale až po ťahu, boli by v nich argumenty, ktoré by už nemuseli byť aktuálne. Do premenných, ktoré sa používali ako argumenty do jednotlivých funkcii, sme totiž postupne vkladali súradnice všetkých ostatných možných koncových pozícií. To znamená, že všetky akcie po ťahoch by sa správali, akoby bola figúrka presunutá na miesto, ktoré bolo testované ako posledné.

Po uvažovaní sme dospeli k záveru, že ideálne by bolo mať v špecifikačnom jazyku možnosť vynútiť vyhodnocovanie výrazov, v určitých prípadoch len do určitej hĺbky. Takýmto spôsobom by sme vedeli zabezpečiť napríklad to, že pri vyhodnocovaní a zjednocovaní čiastkových množín sa vyhodnotí textový reťazec s premennými pozície a vyhodnotia sa argumenty akcie, ktorá sa má zavolať po ťahu. Tieto by sa do nej uložili už ako konštanty.

Rozhodli sme sa zaviesť do jazyka nový typ množín s označením "[ ]" (hranaté zátvorky). Pomenovali sme ho *LazySet* (lenivá množina), nakoľko sa vyhodnotí, len ak ju k tomu skutočne prinútime. Upravili sme zápis množín v špecifikačnom súbore, tak ako to môžeme vidieť na ukážke kódu 5.9

```
$TAHY = $TAHY UNION {["pozicia($X,$Y)", AkciaPoTahu(...)]}
```

Ukážka kódu 5.9: Zápis možného ťahu a následnej akcie v novom type množiny

Následne sme prešli k úprave frameworku. Vytvorili sme triedu *LazySet* (lenivá množina), ktorá je potomkom triedy *Expr* (výraz) a implementovali sme potrebné metódy. Kľúčová pre nás bola metóda *eval* (vyhodnoť), ktorá zabezpečuje rekurzívne vyhodnocovanie výrazov. Prechádzame v nej cez všetky prvky, ktoré množina obsahuje a vyhodnocujeme každý prvok, ktorý nie je volaním funkcie. Ak narazíme na funkciu, vyhodnotíme jej argumenty a uložíme ich do nej už ako konštanty.

Zavolať samotnú funkciu môžeme potom tak, že si ju vyberieme z množiny do samostatného výrazu a ten vyhodnotíme. Druhý spôsob je použitie našej metódy *forceEval* (vynúť vyhodnotenie), ktorú môžeme zavolať na objekte triedy *LazySet* (lenivá

množina). Táto metóda vyhodnotí všetky výrazy v množine, vrátane funkcií.

5.4.3 Testovanie

Po implementácii predošlých úprav sme prešli k testovaniu. Za oboch hráčov sme zvolili počítač s algoritmom náhodného vyberania ťahov z množiny možných ťahov. Pozorovania vyzerali sľubne, no po niekoľkých ťahoch prvého aj druhého hráča sme narazili na prvé komplikácie. Chyba bola zvláštna v tom, že systém nedovoľoval spraviť ťah, ktorý bol, z ľudského pohľadu, legitímny. Druhou zvláštnosťou bolo, že chyba sa pri viacerých testovaniach opakovala v úplne rozličných časoch.

Vyzeralo to, že pôjde o nejaký náhodný jav, ktorý sa niekedy objaví a niekedy nie. Po dlhom krokovaní programu a mnohých testovaniach sme prišli na to, že vyhodnocovanie akcií po ťahoch sme síce zastavili v metóde *eval* (vyhodnoť) našej triedy *LazySet* (lenivá množina), no rovnaké vyhodnocovanie prebieha aj v metóde *equals* (rovná sa), ktorú náš systém niekedy zavolá. To znamená, že v rôznych chvíľach behu programu sa zavolali funkcie, ktoré sa mali zavolať až po konkrétnych ťahoch. V týchto funkciách sa okrem iného diala aj zmena hráča, ktorý je na ťahu. Preto sa niekedy stalo, že ťah, ktorý chcel systém vykonať bol síce možný, no na ťahu bol iný hráč.

Okrem testovania pomocou algoritmu na náhodný výber z možných ťahov sme testovali aj niektoré zložitejšie algoritmy, ktoré prehľadávajú stavový priestor a snažia sa vybrať najlepší ťah. Išlo o algoritmy MiniMax a Monte Carlo. Systém dokázal s týmito algoritmami odohrať hru korektne podľa pravidiel.

Záver

V tejto práci sme popísali automatický systém na hranie stolových hier. Analyzovali sme jeho časti - robotické rameno, framework a špecifikačný jazyk na definovanie hier.

Zamerali sme sa na konkrétne problémy a ich riešenia. Začali sme inverznou kinematikou - návrhom jej výpočtu a následne sa nám podarilo ju úspešne implementovať. Teraz je možné zadať do špecifikačného súboru miesto hernej pozície jednoduchšie. Môžeme ho zapísať v súradniciach v centimetroch, pričom počiatočný bod predstavuje základňa robotického ramena. Predošlý spôsob definície herných pozícii podľa uhlov robotického ramena sme ponechali k dispozícii.

Testy inverznej kinematiky sme uskutočnili vo zvlášť vytvorenej aplikácii, ktorá simuluje robotické rameno. Uhly medzi ramenami sme získali z našich výpočtov inverznej kinematiky a doprednou kinematikou sme overili správnosť dosiahnutého koncového bodu. S reálnym ramenom sme na hre Žabky overili funkčnosť celej implementácie. Všetky testy boli úspešné.

Ďalej sme popísali ako framework pri hraní hier generoval nové možné ťahy. Kvôli tomu, že pôvodné generovanie vykonávalo niektoré kroky zbytočne, navrhli a upravili sme algoritmus na efektívnejší. Dosiahli sme, že nám každá figúrka vie "povedať", na ktoré miesta sa môže v konkrétnom stave hry pohnúť. Zjednodušili sme tak generovanie všetkých ďalších prípustných ťahov v každom stave hry.

Táto zmena si vyžadovala zásah aj do štruktúry špecifikačného jazyka a jeho spracovania vo frameworku. Do definícií pravidiel v špecifikačnom jazyku sme pridali atribút *toLocations* (na pozície). Do neho je možné uložiť množinu prípustných nasledujúcich ťahov pre konkrétny typ figúrky pomocou užívateľom definovaných výrazov. Množiny, do ktorých sa ukladajú možné ťahy a akcie, ktoré majú nastať po nich, sme vytvorili a prispôsobili my. Vďaka nim je zabezpečené to, že akcia, ako napríklad vyhodenie figúrky po preskočení súperom, nastane naozaj vo chvíli keď bol takýto ťah vykonaný. S množinami, ktoré boli v pôvodnom špecifikačnom jazyku toto nebolo možné.

Do upravenej verzie špecifikačného jazyka sme prerobili pravidlá dvoch hier - Žabky a Alquerque, na ktorých sme ukázali, že naša úprava je funkčná a použiteľná. Hru Žabky sme testovali s algoritmom prehľadávania do hĺbky a hru Alquerque s algoritmami MiniMax a Monte Carlo. Systém odohral obe hry korektne podľa pravidiel.

Využili sme funkciu frameworku, ktorá umožňuje beh rovnakej hry niekoľko krát

za sebou. Nechali sme systém zahrať hru Žabky 10000 krát za sebou podľa pôvodne špecifikovaných pravidiel a potom podľa nami upravených pravidiel. Do štatistického súboru sme zaznamenali celkový čas trvania odohrania hier podľa prvej aj druhej verzie pravidiel. Celkový čas, ktorý ubehol, kým systém odohral hry s pôvodnou špecifikáciou bol *1064744 milisekúnd*, čo je *17,7 minút*. Čas hrania hier s našou verziou pravidiel bol *899010 milisekúnd*, čo je *15 minút*. Ako sme predpokladali v teoretickom porovnaní zložitostí pôvodného a nového algoritmu, naše zmeny zrýchlili hranie systému a to dokonca o *15,25%*.

Ciele práce sa nám prevažne podarilo naplniť. Ďalšie pokračovanie v tejto práci by mohlo priniesť nové zabudované *followup* (nasledujúce) akcie po ťahoch, do ktorých by systém automaticky dodal informácie o vykonanom ťahu a pozícii, kam sa figúrka posunula. Spôsob, ktorý sme implementovali je síce všeobecnejší a viac prispôsobiteľný, no možno pre jednoduché použitie a pohodlnosť písania jednoduchých hier by to mohli niektorí ľudia oceniť.

Literatúra

[1] Lynxmotion Robotic Arm AL5D. http://www.lynxmotion.com/c-130-al5d.aspx.

[2] OpenCV - Open Source Computer Vision. http://opencv.org/.

[3] RXTX Wiki. http://rxtx.qbang.org/wiki/index.php/Main_Page.

[4] S3Games zdrojový kód. https://github.com/Robotics-DAI-FMFI-UK/s3games.

[5] Tic Tac Toe Playing Robotic Arm. http://www.hessmer.org/robotics/lynxmotion-robotic-arm.html.

[6] T.D. Viappiani P. Jaeger, M. Nielsen. *Twelfth Scandinavian Conference on Artificial Intelligence*. IOS Press, 2013.

[7] Pavel Petrovič. Jazyk pre špecifikáciu pravidiel stolných hier ako základ automatického inteligentného hracieho robotického systému. Kognícia a umelý život, p. 141-150, 2015.

[8] Peter Pukančík. Riadiaci systém s inverznou kinematikou pre mobilné robotické rameno. Bakalárska práca, 2012.

[9] Peter Pukančík. Robotický systém pre hranie stolovej hry. Diplomová práca, 2015.

[10] Michael L. Scott. *Programming Language Pragmatics*. Elsevier Science, 2005.