Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Herní strategie robotického fotbalu

Plzeň 2014 Jaromír Lepič

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 13. května 2014

Jaromír Lepič

Abstract

The goal of this thesis is to design, implement and test algorithms that can be later used in the game strategy module of robotic soccer control software. A testing application written in C# is also provided in order to visualize the output of implemented algorithms. Used algorithms should be stable and fast. First part deals with discretization of the game field in order to limit the state space of further algorithms. In second part various statistics are created and then used to select roles and actions for robotic players. Third part concentrates on computing weights of discretized fields to create a weighted graph. Finally the fourth part deals with graph-search algorithms in order to select strategically correct path to player's destination. Additionally, this thesis contains user's manual describing how to control the testing application. Remaining appendices describe configurations of testing machines and show some examples of output from the testing application.

Cílem této práce je navrhnout, implementovat a otestovat algoritmy, které mohou být později použity v modulu herní strategie řídícího softwaru robotického fotbalu. Testovací aplikace napsaná v C# je navíc vytvořena za účelem vizualizace výstupu implementovaných algoritmů. Použité algoritmy by měly být stabilní a rychlé. První část se zaobírá diskretizací herního pole, aby se omezil stavový prostor následujících algoritmů. V druhé části jsou vytvořena určitá hodnocení a použita k výběru rolí a akcí robotických hráčů. Třetí část se soustředí na výpočet vah diskretizovaných polí za účelem vytvoření váženého grafu. Nakonec ve čtvrté části práce zkoumá algoritmy prohledávání grafu, které vybírá strategicky správnou cestu k hráčově cíli. Tato práce dále obsahuje uživatelský manuál s popisem ovlávání testovací aplikace. Zbylé přílohy popisují konfiguraci testovacích strojů a ukazují příklady výstupu z testovací aplikace.

Poděkování

Děkuji panu Ing. Kamilu Ekšteinovi, Ph.D. za vedení a rady během vypracovávání této práce. Zároveň děkuji Ing. Petru Altmanovi a ostatním členům týmu robotického fotbalu, bez nichž by tento zajímavý projekt nemohl vzniknout.

Obsah

1 Úvod **1**

2 Projekt Robotického fotbalu **2**

2.1 Co je robotický fotbal 2
2.2 Pravidla ligy MiroSot Middle League 2
2.3 Tým robotického fotbalu 3
2.4 Řídící software 4
2.5 Požadavky 7
2.5.1 Požadavky na strategický modul 7
2.5.2 Požadavky na tuto práci 8
2.5.3 Omezení cíle této práce 8

3 Teoretický rozbor **10**

3.1 Motivace 10
3.2 Diskretizace 11
3.2.1 Řídká a hustá síť 11
3.2.2 Čtvercová síť a ostatní varianty 15
3.3 Hodnocení hráčů 17
3.4 Výběr rolí a akcí 19
3.4.1 Rozhodovací stromy 20
3.4.2 Strojové učení 21
3.5 Vážení grafu 26
3.6 Prohledávání grafu 28
3.6.1 Prohledávání do šířky 30
3.6.2 Prohledávání do hloubky 31
3.6.3 Dijkstrův algoritmus 32
3.6.4 Greedy best-first search 33
3.6.5 Algoritmus A* 35

4 Realizace **37**
4.1 Diskretizace 37
4.2 Hodnocení hráčů 40
4.2.1 Vzdálenost k míči 40
4.2.2 Natočení k míči 41
4.2.3 Vlastnictví míče 41
4.2.4 Viditelnost míče 42
4.2.5 Vzdálenosti k brankám 43
4.2.6 Natočení k soupeřově bráně 44
4.2.7 Viditelnost spoluhráčů 44
4.2.8 Inverzní útočné hodnocení 45
4.2.9 Časová náročnost 46
4.3 Výběr rolí a akcí 46
4.3.1 Útočník 48
4.3.2 Záložník 50
4.3.3 Brankář 50
4.3.4 Obránci 51
4.3.5 Časová náročnost 51
4.4 Vážení grafu 51
4.4.1 Pozice hráčů 51
4.4.2 Pohybové vektory soupeře 54
4.4.3 Časová náročnost 59
4.5 Prohledávání grafu 60
4.5.1 Optimalizace kódu 61
4.5.2 Optimalizace heuristiky 64
4.5.3 Vzdálenost sousedních bodů 68
4.5.4 Zhodnocení 69
4.6 Budoucí rozšíření 70

5 Implementace **72**
5.1 GUI.cs 73
5.2 Processor.cs 74
5.3 Skripty prostředí Octave 76

6 Závěr **77**

A Uživatelská dokumentace **81**
A.1 Soubory a překlad 81
A.2 Testovací aplikace 81
A.3 Skripty jazyka *Octave* 86

B Konfigurace testovacích strojů **88**
B.1 Stroj 1 88
B.2 Stroj 2 88

C Obrázkové přílohy **89**

1 Úvod

V roce 2010 vnikl z iniciativy studentů Fakulty aplikovaných věd a Fakulty elektrotechnické projekt robotického fotbalu Západočeské univerzity. Projekt začal být studenty vypracováván v následujím roce v rámci bakalářských prací. Práce byla logicky rozdělena na řídící software, jehož vývojem byl pověřen tým Katedry informatiky a výpočetní techniky Fakulty Aplikovaných věd, a na vlastní fyzické roboty, které měla vyvíjet skupina studentů Fakulty elektrotechnické ve spolupráci s Katedrou mechaniky Fakulty aplikovaných věd.

Jako člen týmu Katedry informatiky a výpočetní techniky jsem se podílel na vývoji první fáze řídícího softwaru. Výsledkem byla modulární platforma schopná zobrazovat herní stav pomocí 3D vizualizace. Protože v té době existovali roboti jen jako koncept, řídící software pracoval pouze se simulací herního pole, kde vlastní simulací byla zmíněná vizualizace vyvinutá Ing. Robertem Ecksteinem. Jádro modulárního softwaru vytvořil Ing. Petr Altman. Dále platforma obsahovala modul elementární inteligence vyvinutý Ing. Jakubem Vališem a modul herní strategie, který jsem v rámci bakalářské práce vyvinul já.

Na modulech řídícího softwaru se dále pracovalo v rámci semestrálních a diplomových prací. Ing. Jakub Vališ vylepšil v loňském roce svůj modul elementární inteligence a práce Bc. Vojtěcha Friče a Ing. Roberta Ecksteina vedly k vytvoření modulu počítačového vidění. Protože i v době psaní této práce nejsou fyzické stroje robotických hráčů hotové, pracuje modul počítačového vidění pouze s daty generovanými počítačovou grafikou, případně s maketami hráčů položenými na skutečném hřišti.

Cílem mé práce je vyvinout spolehlivý modul herní strategie. Tato práce nenavazuje přímo na mou předchozí práci na modulu herní strategie [1]. Namísto stavových automatů přistupuje k problému z jiného úhlu. Jako cíl práce jsem si stanovil vytvořit modul schopný podle situace na herním poli rozdělit role robotů a na základě pozic a pohybu soupeře vybrat vhodnou trasu robota-útočníka. Tento modul je tudíž vyvíjen s přihlédnutím na možnost jeho rozšíření v budoucnosti.

2 Projekt Robotického fotbalu

2.1 Co je robotický fotbal

Robotický fotbal je obdobou fotbalu tradičního, ale místo živých hráčů se ho účastní hráči robotičtí. Vzhled robotů i pravidla zápasů se liší dle kategorií, stanovených organizací FIRA (Federation of International Robot-soccer Association) [2]. Naše kategorie je označována jako *MiroSot* (Micro-Robot World Soccer Tournament) a je jednou z mnoha. V ligách *HuroCup* či *AndroSot* hrají autonomní humanoidní roboti, kteří se pohybují na dvou nohách. Ligy *AmireSot* a *RoboSot* hrají autonomní či semi-autonomní roboti nehumanoidních tvarů. *NaroSot* je liga velmi podobná lize *MiroSot*, jen roboti a hřiště jsou menší. Nakonec kategorie *SimuroSot* nemá fyzické roboty a zápasy se odehrávají v simulovaném počítačovém prostředí [3].

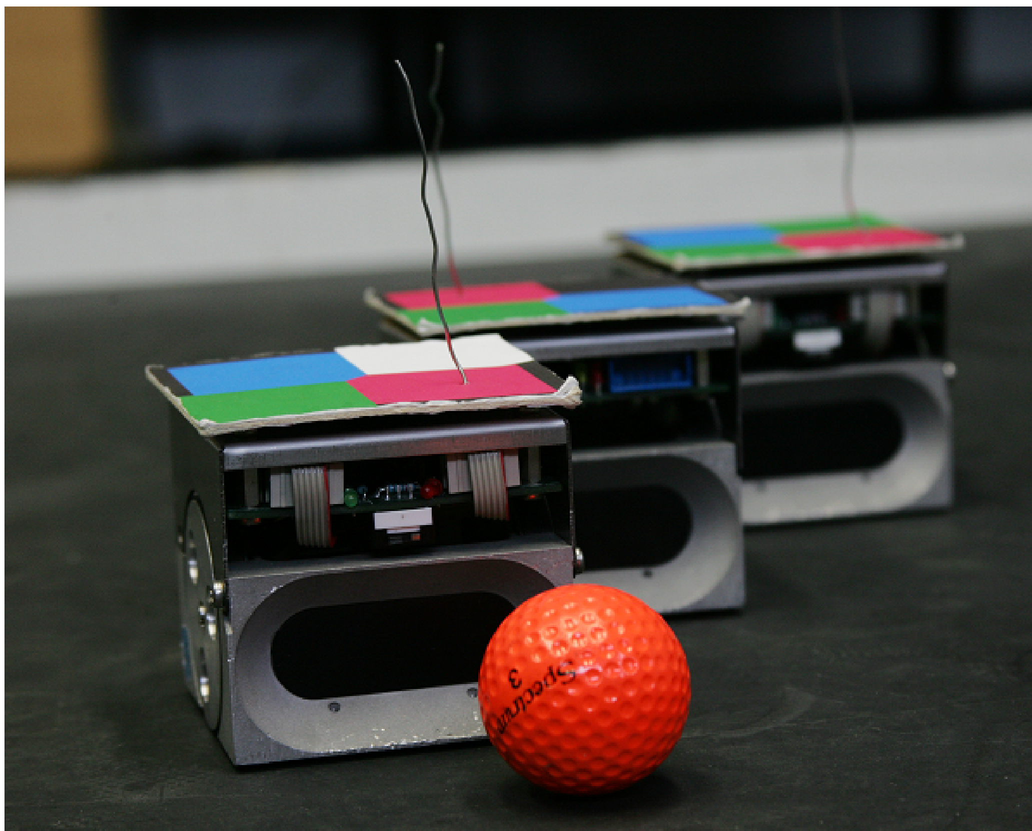
2.2 Pravidla ligy MiroSot Middle League

Liga *MiroSot* se dále ještě dělí na trojici kategorií podle počtu hráčů a velikosti herního pole. Projekt Západočeské univerzity cílí na kategorii *Middle League*, tedy střední ligu.

Ve střední lize hrají proti sobě týmy o pěti hráčích. Každý hráč je omezen maximálními rozměry 7,5 $cm \times$ 7,5 $cm \times$ 7,5 cm. Na svrchní straně robota se nachází barevný štítek sloužící k identifikaci modulem rozpoznání obrazu. Barvy žlutá či modrá jsou přiděleny organizátory a barevné kombinace na štítcích nesmí obsahovat přidělenou barvu soupeře.

Herní pole tvoří dřevěná deska o rozměrech 220 $cm \times$ 180 cm s 5 cm vysokými mantinely. Povrch desky je černý, nereflexní a strukturou je stejný jako stůl na stolní tenis. Důležité herní pozice a oddělovací čáry jsou na hřišti přítomny, branka je široká 40 cm. Míček je oranžový o průměru 42,7 mm.

Typický vzhled robotů této kategorie demonstruje obrázek 2.1. Další podrobnosti, omezení a informace o průběhu hry lze najít v oficiálním dokumentu FIRA [4].



Obrázek 2.1: Ukázka hráčů ligy MiroSot s míčem

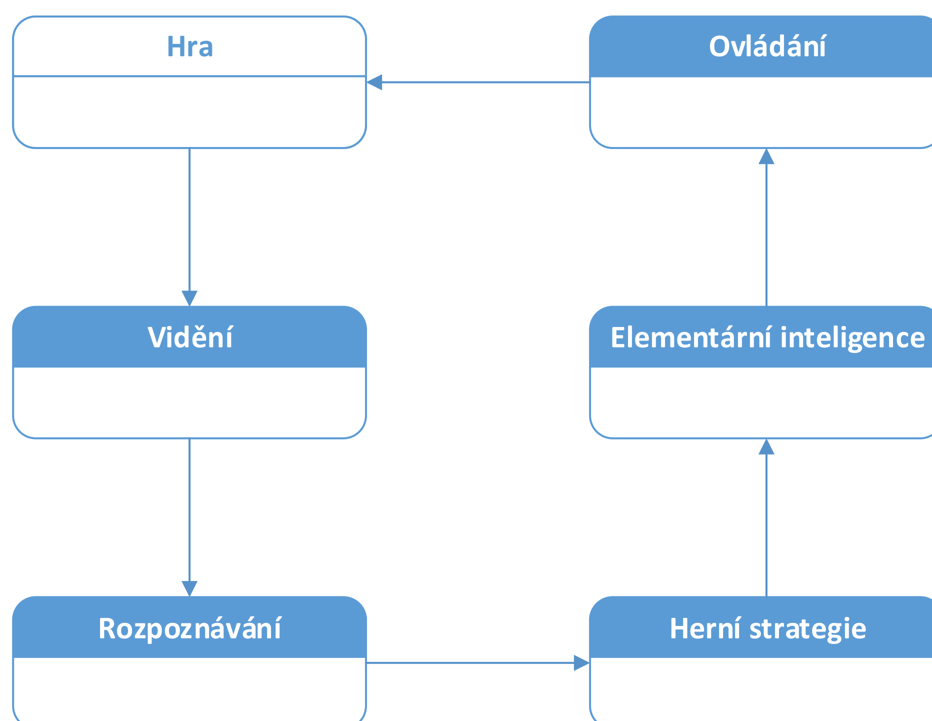
2.3 Tým robotického fotbalu

Projekt robotického fotbalu se v současné době dá rozdělit na dvě skupiny. Tým z Fakulty elektrotechnické spolupracuje s Fakultou strojní a Katedrou mechaniky Fakulty aplikovaných věd a vyvíjí vlastní stroje-hráče s cílem postavit silné a rychlé roboty schopné obstát v zápase. Druhý tým, jehož jsem součástí, je z katedry informatiky fakulty aplikovaných věd.

Náš tým má za cíl vytvořit řídící software projektu, který koordinuje roboty na hřišti. Tento software zahrnuje mimo jiné moduly rozpoznání obrazu, vizualizace, simulace, herní strategie a elementární inteligence.

2.4 Řídící software

Řídící software je program, který je spuštěn na počítači mimo hřiště a představuje jedinou rozhodovací jednotku celého týmu, protože samotní roboti žádné samostatné rozhodování neprovádějí. Z kamery zavěšené nad hřištěm přicházejí data do programu. Ten z nich produkuje hráčům příkazy, které jsou posílány bezdrátově.



Obrázek 2.2: Abstraktní model softwaru

Řídící software byl implementován poprvé v rámci bakalářských prací v roce 2011 v programovacím jazyce C#. Software má modulární podobu, kde nejdůležitější je jeho jádro. Ostatní moduly jsou načítány dynamicky v podobě knihoven DLL a komunikují spolu pomocí připraveného systému zasílání zpráv. Tento systém umožňuje snadné nahrazení jedné verze modulu za jinou a byl implementován Ing. Petrem Altmanem v rámci jeho bakalářské práce [5]. Abstraktní model modulů softwaru zobrazuje obrázek 2.2.

Hra

Hra není modul řídícího softwaru, ale v uvedeném schématu představuje část smyčky mimo program, která reprezentuje skutečnou hru na hřišti. Na jedné straně přijímá příkazy k pohybu robotů a na straně druhé je pomocí kamery stav hry snímán pro další iteraci zpracování.

Vidění a rozpoznávání

V původním návrhu struktury řídícího softwaru byly moduly vidění a rozpoznávání odděleny. Modul vidění měl za úkol získávat snímky herního pole a posílat je modulu rozpoznávání. V praxi je v současné době funkcionalita modulu vidění součastí modulu rozpoznánání. Modul rozpoznávání má pak na starosti rozpoznání obrazu z kamery. Je třeba z obrazu rychle a spolehlivě získat pozice všech robotů a míče. Pozice se určují vzhledem k hernímu poli. Takto získaná data se posílají herní strategii. Tento modul byl implementován Ing. Robertem Ecksteinem v rámci jeho diplomové práce, kde vylepšil řešení vytvořené v bakalářské práci Bc. Vojtěcha Friče [6; 7].

Herní strategie

Modul herní strategie představuje mozek celého týmu. V závislosti na získaných datech z rozpoznání obrazu vytvoří akce, které směřují ke strategické hře a konečnému cíli - vítězství. Pro každého robota je vygenerován příkaz a parametry, pokud je onen příkaz vyžaduje. Příkazy jsou následně předány k provedení pohybovému modulu. Implementací herní strategie za pomoci multiagentního systému se zabývala má bakalářská práce [1]. Tato práce zvolila jiný přístup a ačkoliv není kompletně funkční jako modul, představuje značné vylepšení inteligence hráčů oproti předchozí implementaci.

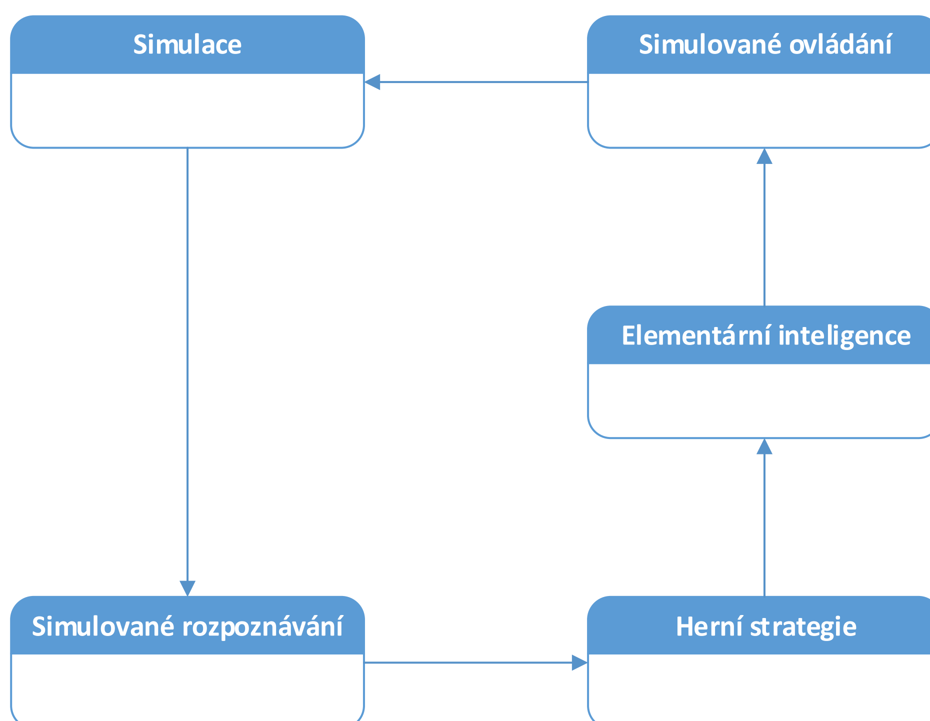
Elementární inteligence

Pohybový modul bývá typicky označen jako modul elementární inteligence, protože využívá metody umělé inteligence k optimalizaci pohybu po hřišti. Modul přijímá od modulu herní strategie cílový bod pohybu, případně celou sadu bodů, které by měl robot projet. Výsledkem by měla být definitivní trasa robota po hřišti. Tento modul byl implementován Ing. Jakubem Vališem v rámci jeho diplomové práce [8].

Ovládání

Modul ovládání zodpovídá za komunikaci s roboty na hřišti a předává jim základní příkazy pro pohyb po hřišti.

Obrázek 2.2 zobrazuje požadovanou podobu softwaru pro skutečnou hru, tedy pro hru s fyzickými roboty na skutečném hřišti. V době psaní diplomové práce je k dispozici herní stůl i kamera, ale robotičtí hráči ještě ne. To umožňuje odladit algoritmy počítačového vidění na maketách, ale testování modulů herní strategie a elementární inteligence potřebuje plynulou hru s měnícím se stavem na hřišti. Proto je vše nutné zatím simulovat na počítači. Simulace má i tu výhodu, že poslouží k testování nových či vylepšených algoritmů bez potřeby přístupu ke stolu a hráčům.



Obrázek 2.3: Simulační model softwaru

Zatímco moduly herní strategie a elementární inteligence získávají a vysílají data nezávislá na způsobu běhu programu, ostatní moduly jsou již navázány na reálný svět, a proto je třeba je nahradit jejich simulovanými protějšky, jak je ukázáno na obrázku 2.3. I po získání skutečných robotů bude simulace užitečná pro testování nových přístupů v obou zmíněných modulech.

Simulace
Modul simulace nahrazuje skutečné hřiště simulovaným pouze v počítači. Modul vidění zde odpadá, protože zde neexistuje žádná kamera, ze které by se získávaly snímky hřiště.

Simulované rozpoznávání
Modul simulovaného rozpoznávání je simulační obdobou modulu rozpoznávání. Tento modul získává pozice robotů a míče z vlastní simulace v počítači a posílá je herní strategii.

Herní strategie
Modul herní strategie se nemění.

Elementární inteligence
Modul elementární inteligence se nemění.

Simulované ovládání
Modul simulovaného ovládání nahrazuje modul ovládání a místo posílání příkazů robotům posílá data pouze simulaci, která je zpracuje a vygeneruje další iteraci herní smyčky.

2.5 Požadavky

2.5.1 Požadavky na strategický modul

Strategický modul musí splňovat podmínky spolehlivosti, inteligence a rychlosti.

- Spolehlivost znamená, že herní strategie musí reagovat na každou situaci na herním poli nějakou akcí. Neexistuje možnost nečinnosti celého modulu. Akce nečinnosti určitého robota jako vygenerovaný příkaz přípustná je.
- Inteligence znamená, že modul musí reagovat na situaci na hřišti způsobem, který připomíná inteligentní strategické uvažování. V potaz by měl modul brát především pozice a pohyb spoluhráčů, soupeřů a míče.
- Rychlost je faktor určený vybavením. V současné době používaná kamera generuje 50 snímků za sekundu, tedy čas na zpracování jedné iterace v řídícím systému je 20 ms. Tento čas se dále musí rozdělit mezi

tři výpočetně náročné moduly – vidění, herní strategii a elementární inteligenci. Výsledný modul herní strategie by tedy měl být schopen dokoknčit včas všechny výpočty, aby poskytl zbylým modulům čas na provedení jejich výpočtů.

2.5.2 Požadavky na tuto práci

- Prozkoumat principy a algoritmy používané pro generování herních strategií v elektronicky řízených hrách, především pak v robotickém fotbalu.
- Tyto principy a algoritmy upravit pro potřeby týmu robotického fotbalu Západočeské univerzity.
- Navrhnout strukturu modulu herní strategie pro využití těchto algoritmů.
- Otestovat různé varianty těchto algoritmů a porovnat generované výsledky.
- Implementovat herní strategii v jazyce C# pro zachování kompatibility a čitelnosti ostatními členy týmu.

2.5.3 Omezení cíle této práce

- Cílem není vytvořit plnohodnotný modul herní strategie připravený k použití v řídícím systému, ale samostatná aplikace, jejíž algoritmy (ideálně jedna třída) se snadno použijí v budoucím modulu herní strategie.
- Aplikace bude v závislosti na herním stavu generovat role hráčů a zvolenému útočníkovi přiřadí určitý příkaz. Tento příkaz bude rozpracován v inteligentně nalezenou diskretizovanou trasu pohybu útočníka.
- Generování příkazů ostatním rolím nebude součástí této práce. Jejich generování bude předmětem případných pokračování této práce. Předpokládá se použití stejného algoritmu hledání diskretizované trasy po hřišti.

- Předzpracování vstupů z modulu rozpoznání obrazu ani následné zpracování výstupů do modulu elementární inteligence nebude implementováno. Aplikace bude používat pozice zadané uživatelem, případně náhodně vygenerované, a výsledná trasa pohybu bude zobrazena uživateli. Přesto je vhodné, aby vstupní a výstupní formát dat byl snadno čitelný a transformovatelný.

3 Teoretický rozbor

3.1 Motivace

Jako první hledisko je třeba vzít data, která poskytuje modul rozpoznávání. Tento modul ze snímků kamery získává pozice všech robotů a jejich natočení, nikoliv však pohybové vektory. Pro určení vhodné strategie je nicméně dobré znát i pohybové vektory robotů, hlavně soupeřů, aby bylo možno předvídat jejich pozice v krátkém časovém úseku. Také následný modul elementární inteligence požaduje pro určité funkce znalost těchto pohybových vektorů robotů. Problematika získávání vektorů pohybu z pozic robotů v po sobě následujích iteracích nenáleží jednoznačně ani do funkcionality rozpoznání obrazu ani do funkcionality herní strategie. V rámci týmu nebylo dohodnuto, komu bude tento úkol přidělen, a tato práce se jím nezabývá. Předpokládá ale, že v době, kdy budou zde zkoumané algoritmy herní strategie implementovány do modulu herní strategie, bude naprogramován i modul předzpracování (vektorizace).

Kromě již zmíněných vektorů pohybu (a samozřejmě pozic objektů na hřišti), vyžaduje modul elementární inteligence vykonávanou akci určitého hráče. Modul pracuje především s akcí pohybu po hřišti. Pokročilé akce *střela* a *přihrávka* byly zpracovány pouze teoreticky, a proto se i tato práce bude pokoušet vše řešit pouze pomocí příkazů pohybu. Vlastní příkaz pohybu se dá předat jako pohyb do cílového bodu nebo jako pohyb do cílového bodu přes několik dalších bodů. V obou případech bude docházet k vyhýbání překážkám (pokud jsou) a nalezení optimální křivky pro trasu robota (například vyhlazení ostrých hran v plánované trase robota). Druhý zmíněný fakt, že se modul elementární inteligence hledáním hladkých pohybových křivek stará, aby zadaná trasa byla transformována na optimální z hlediska pohybových schopností robota, znamená, že pro herní stragii stačí generovat trasu, která je optimální z hlediska strategie. Tedy, že je možné použít diskretizaci stavového prostoru hry.

Funkcionalita modulu elementární inteligence ale není jedinou ani hlavní motivací pro použití diskretizace. Znamená pouze, že nebude třeba transformovat trasu nalezenou v diskretizovaném prostoru hry na pohybově optimální trasu. Hlavní motivací diskretizace je omezení možného stavového prostoru hry. V okamžiku, kdy je stavový prostor (určený především pozicemi hráčů)

spojitý, je nekonečný. A jeho prohledávání je pak daleko složitější.

Podoba této diskretizace by měla odpovídat typu úlohy a druhu proměnných, které stavový prostor definují. V případě robotického fotbalu je stav hry definovaný pozicí hráčů a míče. Jako intuitivní a srozumintelná varianta diskretizace se zde proto nabízí rozdělit samotné hřiště na menší pole a veškeré pozice se poté diskretizují do těchto polí. Takto rozdělené hřiště může představovat graf, který je možno prohledat grafovými algoritmy hledání cesty. Před vlastním prohledáváním je ale nutno do grafu zanést dodatečné informace reflektující stav hry (pozice a pohyb robotů). Zároveň je třeba na základě diskretizovaného či nediskretizovaného stavu hry vybrat útočníka a zvolit jeho akci.

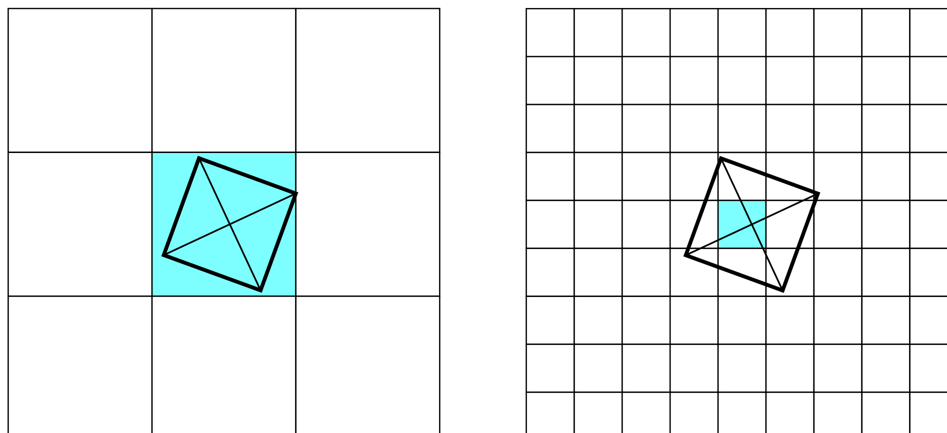
3.2 Diskretizace

Diskretizované hřiště musí umět nést dostatečnou informaci o stavu hry. Ideálně v té míře, že ztráta informace způsobená diskretizací je zanedbatelná. To vylučuje rozdělení hřiště jen na několik málo sekcí, kdy informace o přesné pozici by se prakticky ztratila. Zůstala by jen informace o obsazenosti částí hřiště, která by ale mohla sloužit jen jako pomocná informace, nikoliv jako hlavní stavový prostor pro použití grafových algoritmů. Možnosti diskretizace se proto nabízí v podobě husté či relativně řídké sítě. Druhé hledisko podoby diskretizovaného hřiště je tvar polí. Nabízí se čtvercová síť nebo nějaká jiná nepravoúhelná varianta.

3.2.1 Řídká a hustá síť

Obrázek 3.1 zobrazuje rozdíl mezi hustou a řídkou sítí. Řídká síť je složena z polí větších než jsou rozměry robota, zatímco hustá síť je složena z malých polí, podstatně menších než je robot. Barevně je vyznačeno pole, na které je diskretizována pozice hráče, určená jeho středem.

Výhodou řídké sítě je větší omezení stavového prostoru, které musí později algoritmy hledání cesty prohledávat. To znamená větší rychlost jejich provádění. Daní za tuto rychlost je ale přesnost funkcí následujících po diskretizaci, včetně nalené trasy, která bude zákonitě méně odpovídat ideální trase. Když je herní pole takto větší než robot, mělo by ve většině případů



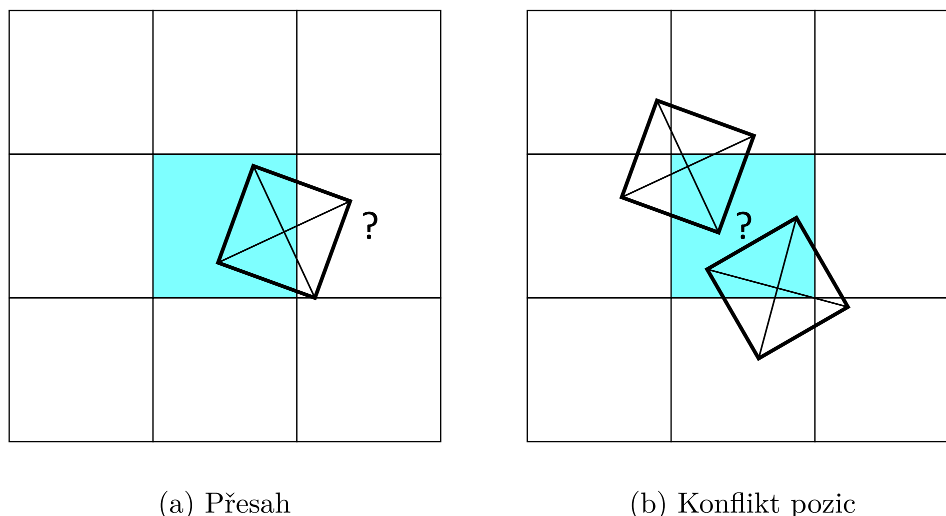
(a) Řídká síť (b) Hustá síť

Obrázek 3.1: Ukázka varitant hustoty sítě

pojmout skoro celého robota, včetně jeho natočení, a představovat tak jeho manévrovací prostor. Nicméně ne vždy se tak stane. Drobné přesahy částí robota mimo jeho pole nepředstavují problém, ale situace zobrazená na obrázku 3.2a už problémem být může. Robot zde skoro polovinou svého těla zasahuje do sousedícího pole. Nastává otázka, jak označit ono sousedící pole. Označit, že robotu přísluší, je stejně nepřesné, jako označit opak. Pokud by robotu náleželo, pak jeden robot najednou zabírá velký prostor, ačkoliv většina z těchto dvou polí je stále volná. Naopak označení, že robot na poli není, je logicky nepřesné, protože robot využívá pole téměř stejně, jako své původní pole.

Variantou předchozího problému je situace, kdy dva roboti mají své středy na stejném poli, ačkoliv velkou část svého těla mají na polích okolních (viz obrázek 3.2b). Podle souřadnic středu je jim přiřazeno, že se vyskytují na stejném poli. Tato situace může nastat vždy, když je pole větší než robot. I u pole stejně velkého jako robot může nastat situace, kdy v rozích pole stojí dva roboti natočení o 45°. Čím jsou rozměry pole blíže velikosti robota, tím je tato situace méně častá, ale vyloučit nelze. Jelikož musí herní strategie reagovat na všechny možné (ač nepravděpodobné) situace, nelze tyto okrajové případy zanedbat. Pokud na herním poli stojí spoluhráč i protihráč, nelze například říci, který tým ovládá konkrétní pole.

Možným řešením by mohla být reprezentace obsazení pole pomocí fuzzy logiky a jejího principu stupně příslušnosti. Stupeň příslušnosti určuje míru,



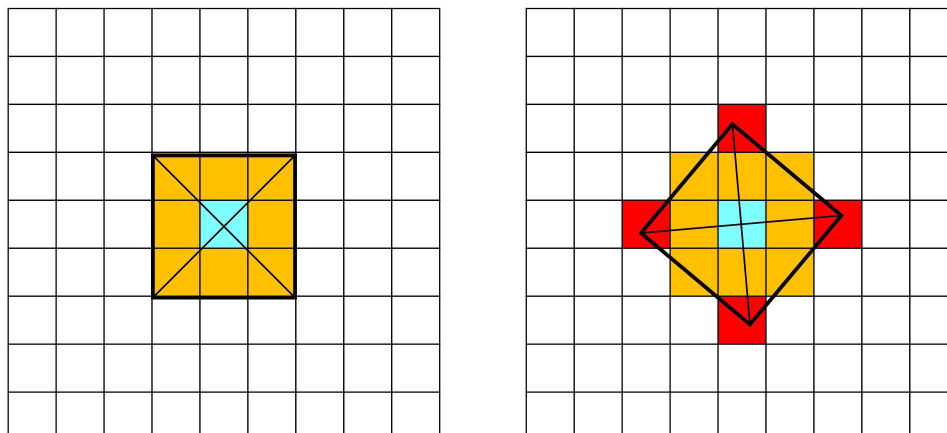
(a) Přesah (b) Konflikt pozic

Obrázek 3.2: Problémy v řídké síti

s jakou nějaká proměnná náleží do určité množiny z mnoha. Jeho velikosti se pohybují v rozmezí 0 až 1 a umožňují tak vyjádřit slova jako „hodně“, „málo“ nebo „možná“. Fuzzy logika označuje tato slova jako *lingvistické proměnné* [9]. To by umožnilo v tomto případě vyjádřit pozici robota jako „spíše“ na původním poli, ale „částečně“ už na poli sousedícím. Pokud by mělo každé pole stupně příslušnosti pro všechny roboty, vyřešila by se situace náležitosti dvou robotů na stejné pole.

Použití fuzzy logiky by znamenalo nutnost používat pro pole v síti struktury či třídy, které jsou schopny informaci o stupních příslušnosti uchovat. Dále vytvořit vlastní funkci pro určení stupňů příslušnosti a především pak veškeré následující algoritmy navrhnout se schopností tuto informaci zpracovat. Nicméně ani stupeň příslušnosti není dostatečné řešení. Vzájemná poloha robotů na sdíleném poli a směr, z něhož robot zasahuje na pole vedlejší, je pro správné strategické uvažování prakticky důležitější, než správně určené stupně příslušnosti. Tyto informace není složité zpracovat ani uložit, ale dále to komplikuje činnost dalších algoritmů.

Druhou variantou je hustá síť, kde rozměry robota pojmou jedno až několik menších polí (viz obrázek 3.1b). Z hustoty sítě plynou opačné výhody a nevýhody než u sítě řídké. Vyšší počet polí znamená větší prostor pro následné zpracování a prohledání grafovými algoritmy. To znamená větší složitost a časovou náročnost těchto algoritmů, nicméně na druhou stranu budou podstatně přesnější.



(a) Obsazení polí (b) Obsazení polí při natočení

Obrázek 3.3: Problémy v husté síti

Také u husté sítě vznikají další problémy, na které je třeba brát ohled. Obrázek 3.3a ukazuje síť o rozměrech pole několikrát menších než je velikost robota, zde třikrát. Pozice robota se zarovná na určité pole (tyrkysová barva), nicméně robot přitom obsazuje další pole, znázorněná oranžovou barvou. V případě husté sítě tedy nestačí určit pro obsazenost polí pouze diskrétní pozice robotů, ale je třeba takto označit i určitý počet sousedních polí.

V husté síti jsou pole tak malá, že i když robot zakrývá pole jen částečně a ono je mu přiřazeno jako obsazené, jde pouze o malou a zanedbatelnou chybu vzhledem k velikosti celého hřiště. Není proto třeba nutné uvažovat o částečném obsazení pole jako u řídké sítě. Ovšem pro uvedený příklad hustého pole nelze jen jednoduše označit všechna sousedící pole jako obsazená, protože při určitých natočeních robota může hráč výrazně pokrývat i další pole, jak ukazuje obrázek 3.3b. Pole vyznačená červeně jsou z nemalé části pokryta robotem. Toto by bylo ještě výraznější na některých z červených polí, pokud by byl střed robota posazen spíše u kraje tyrkysového pole. Je nutno proto nějakým způsobem vzít v potaz natočení robota pro určování obsazených polí.

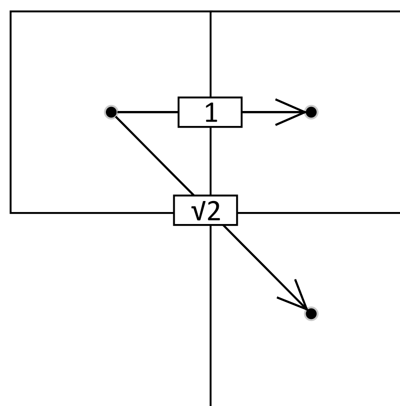
3.2.2 Čtvercová síť a ostatní varianty

Kromě velikosti polí v diskretizovaném hřišti je třeba také zvolit správnou podobu tohoto pole. V předchozích odstavcích jsem pracoval se čtvercovou sítí. Zároveň ale existují jiné možnosti, které mají své výhody i nevýhody v porovnání se standardní čtvercovou sítí.

Obyčejná síť čtverců v pravoúhlé mřížce má podstatnou výhodu snadné reprezentace v datových strukturách programu. Takováto mřížka naprosto přesně odpovídá dvourozměrnému poli, kde index do pole značí index do mřížky. Samozřejmě za předpokladu, že pole v mřížce je určeno svým indexem, ale bylo by neefektivní používat jiný přístup. Tohle vše umožňuje velice snadné zpracování polí hřiště v dalších algoritmech. Indexace herních polí je ukázána na obrázku 3.4a.

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]
[3,0]	[3,1]	[3,2]	[3,3]

(a) Indexace polí

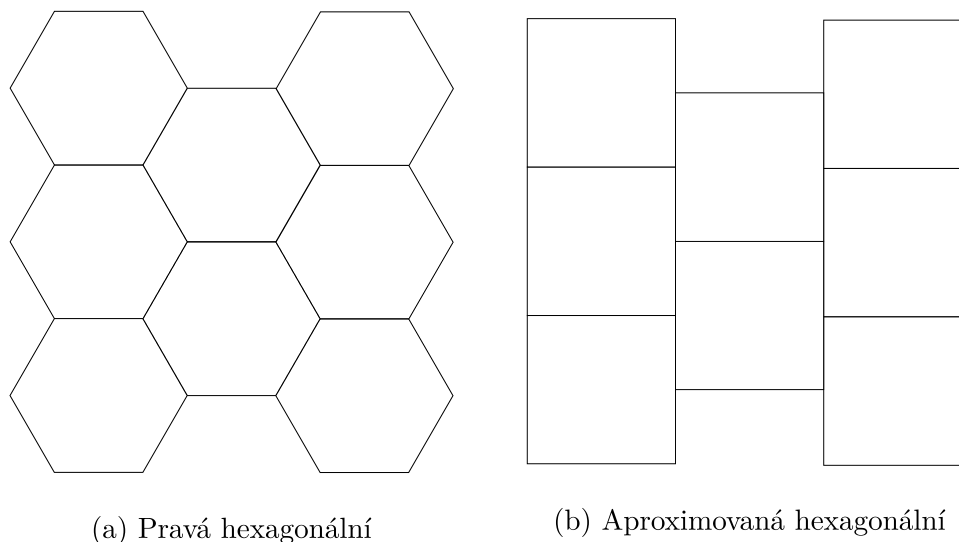


(b) Vzdálenost polí

Obrázek 3.4: Vlastnosti čtvercové sítě

Jedinou malou nevýhodou čtvercové sítě je, že vzdálenost mezi sousedními poli není vždy stejná (viz obrázek 3.4b). Jde vždy o vzdálenost 1 nebo $\sqrt{2}$. Tyto vzdálenosti jsou relevantní pro později uvedené grafové algoritmy. Vlastnosti čtvercové sítě nutí mezi těmito vzdálenostmi rozlišovat. Alternativou, která by eliminovala dvě velikosti vzdáleností, je hexagonální síť či její aproximace, jak ukazuje obrázek 3.5.

Výhoda jednotné vzdálenosti je prakticky jediná výhoda hexagonální sítě. První nevýhoda hexagonální sítě je v její reprezentaci v datových struktu-



(a) Pravá hexagonální (b) Aproximovaná hexagonální

Obrázek 3.5: Varianty hexagonální sítě

rách. Indexy do dvourozměrného pole zde nesouhlasí s pozicemi herních polí. Bylo by nutno zvolit jinou indexaci hexagonálních polí a od ní odvodit nové řešení sousednosti polí na základě indexů. Ve čtvercové síti jsou sousední pole všechna ta, jejichž jeden či oba indexy se liší o jedna. V hexagonální síti to tak jednoduché není a pravděpodobně by bylo třeba zavést složité přepočty souřadnic, případně zvolit úplně jiný souřadný systém. Některé tyto systémy existují a jsou úspěšně použity ve strategických počítačových hrách (například Civilization V). Například systém, kde každý hexagon odpovídá krychli v trojrozměrném prostoru [10].

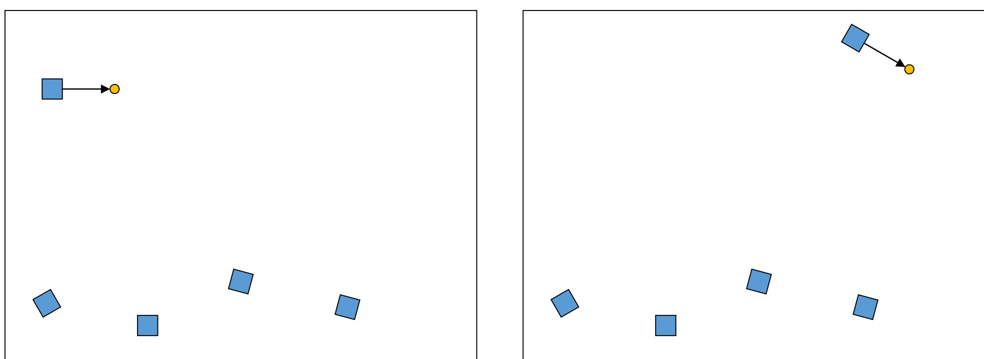
Druhým problémem je určení vlastní diskrétní pozice. Ve čtvercové síti jde o pouhé zaokrouhlení dvou souřadnic na určitou nejbližší hodnotu, ale v hexagonální by bylo třeba převést reálné souřadnice pozice do jiného systému a zde zaokrouhlit. Částečným řešením by bylo pouze aproximovat hexagonální pole čtvercovými poli (viz obrázek 3.5b). Ačkoliv by to jistě zjednodušilo výpočet diskrétní pozice, neodstraňuje to předchozí problém (nutnost nového systému souřadnic), ani neřeší problém následující.

Výrazným problémem hexagonální sítě oproti čtvercové je omezení počtu sousedních polí z osmi na šest. To limituje možnosti při hledání cesty. Místo otočení o násobky 45° se lze otáčet jen o 60°, což snižuje přesnost nalezené cesty.

Výše uvedené problémy a nutnost přepočtu mezi souřadnými systémy znamenají, že použití hexagonální sítě jsem v této fázi teoretického výzkumu zavrhl. Použití hexagonální sítě by mohlo být námětem pro alternativní verzi této práce, kdy veškeré algoritmy budou upraveny pro hexagonální souřadný systém.

3.3 Hodnocení hráčů

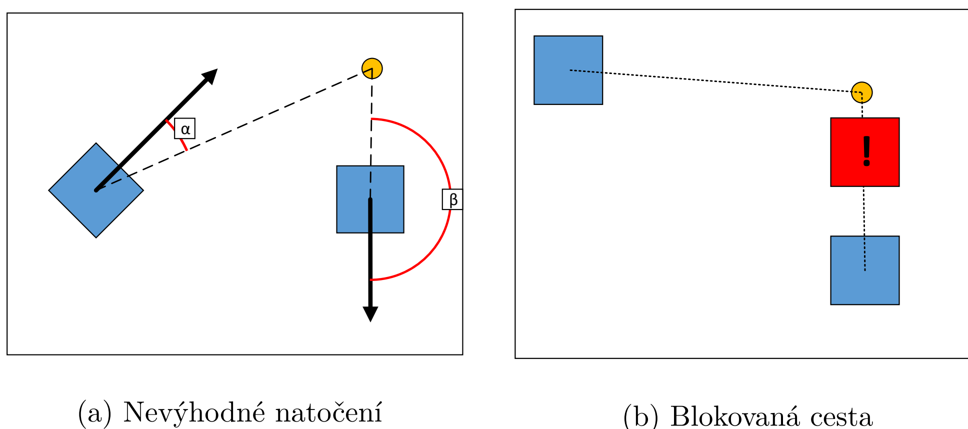
Na základě diskretizovaných či nediskretizovaných pozic hráčů je třeba určit ve vlastním týmu role jednotlivých hráčů a tyto dále určí jejich akce. Nicméně samotné pozice hráčů nestačí. Výběr musí fungovat na základě obecnějších vstupních hodnot. Ideální hodnoty jsou takové, které popisují vztah mezi objekty na hřišti, aniž by byly nějak závislé na skutečných pozicích. Například hráč, který je 10 kroků daleko (jeden krok je vzdálenost mezi dvěmi sousedními poli na hřišti) od míče, je vhodným kandidátem na roli útočníka, pokud všichni ostatní z týmu jsou nejméně 20 kroků daleko. Toto platí bez ohledu na to, kde jsou onen hráč a míč umístěni. Tuto skutečnost demonstruje obrázek 3.6. Nákres stavu hry je zjednodušený o pozice soupeře, protože tento příklad pracuje jen se vzdáleností od míče. Na obrázku není proto ani znázorněno, kde je branka soupeře. Bez ohledu na pozici na hřišti je jako útočník (tedy hráč, který se zaměřuje na míč) zvolen nejbližší hráč.



Obrázek 3.6: Zjednodušený příklad určení útočníka podle jeho vzdálenosti od míče

Samozřejmě pouhá vzdálenost od míče je pro výběr rolí a akcí příliš jednoduchá. Nezohledňuje soupeře ani směr pohybu hráčů. Jeden hráč může

být sice míči nejblíže, ale nemusí už být v pozici nejvýhodnější pro útok. Proto může dalším užitečným parametrem být relativní natočení hráče vůči míči a viditelnost míče hráčem. Tyto parametry demonstruje obrázek 3.7. Obrázek 3.7a ukazuje, že hráč, který se pohybuje směrem od míče, je horší útočník, protože by nejprve musel zcela obrátit směr svého pohybu. Oproti tomu jen o něco více vzdálený spoluhráč musí jen mírně upravit svůj směr jízdy. Stejně tak je důležitá viditelnost míče, jak dokazuje část 3.7b. Jeden hráč je sice míči blíže, ale míč nevidí, překáží mu soupeř. Druhý, vzdálenější spoluhráč má výhled čistý a může bez problémů zaútočit na míč, aniž by v tomto okamžiku musel řešit objíždění soupeře.



(a) Nevýhodné natočení

(b) Blokovaná cesta

Obrázek 3.7: Příklady, kdy nejbližší hráč není nejvýhodnější

Na hřišti může dojít k situaci, kdy je určení útočícího hráče jednoznačné. To je v případě, kdy hráč vlastní míč. Určení vlastnictví míče hráčem ale už jednoznačné není. Pouhá vzdálenost k míči, tedy skutečnost, že míč je hned vedle robota, nestačí. Je třeba také vzít opět v potaz relativní úhel směru pohybu vůči míči. Pokud by se robot totiž pohyboval od míče, jakkoliv blízkého, pak míč ve skutečnosti nevlastní, protože v dalším okamžiku se od míče vzdálí, bez jakékoliv akce s míčem.

Další užitečnou proměnnou je vzdálenost od obou branek. Zatímco pro útočníka může být nedůležitá, defenzivní role (brankář, obránce) jistě využijí parametr vzdálenosti od domácí branky. Podobným způsobem je vzdálenost od soupeřovy branky vhodná pro určení role podpory útočníka. Zbylými paramatery, pro které jsem v této fázi přípravy nevymyslel účel, ale mohou být časem prospěšné, jsou úhel natočení vůči soupeřově brance a viditelnost spoluhráčů.

3.4 Výběr rolí a akcí

Role hráčů v této práci jsou přiřazovány dynamicky, což umožňuje větší flexibilitu herní strategie. Také to dokonce dovoluje měnit složení rolí v týmu. To by do budoucna mohlo znamenat implementaci algoritmů měnící počet určitých rolí v závislosti na stavu hry. Například v situaci, kdy tým prohrává o pouhý gól by se jeden z obránců nahradil druhým hráčem podporujícím útok. Implementaci výběru rolí a akcí definují stanovená omezení této práce. Soustředění se na vývoj algoritmů pro roli útočníka, především prohledávání grafu, a absence akce přihrávky (která nebyla v modulu elementární inteligence implementována) znamenají, že hráčem operujícím s míčem bude vždy útočník. Díky dynamickým rolím se tak každý hráč může stát útočníkem a jiný hráč zastoupí jeho původní roli. Navržená sestava rolí a jejich povinnosti vypadá takto:

Útočník: Útočníkem se stává hráč, který je v nejvýhodnější pozici pro práci s míčem. Jako takový má za cíl míč získat a zavést jej před soupeřovu branku a tam vystřelit. Střelbu může v případě nedostupnosti akce střely nahradit zavezení míče až do branky. V případě, že bude dostupná akce přihrávky, tak může hráč přihrát na záložníka, pokud ten je ve výhodnější pozici.

Záložník: Záložník představuje hráče, který podporuje útočníka ve snaze vstřelit gól. Jeho cílem je vyčkávat v útočném postavení na odražený míč, případně přihrávku, a stát se tak novým útočníkem a pokusit se skórovat.

2 Obránci: Obránci jsou první obrannou linií, která má zabránit soupeři ve vstřelení gólu. Jejich cílem je vyčkávat v obranných pozicích, které budou v cestě soupeře s míčem. Když bude soupeř dostatečně blízko, logika výběru útočníka by měla určit obránce jako útočníka, který se pak pokusí získat míč.

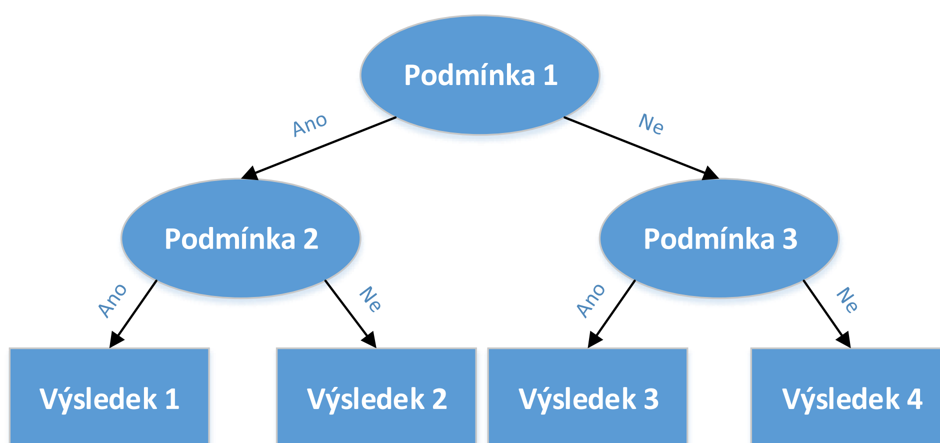
Brankář: Brankář je druhou a poslední obrannou linií. Jeho cílem je střežit bránu proti střelám a hráčům, kteří projdou přes obránce. Po získání míče se jako útočník pokusí míč vyvézt, zatímco nejbližší hráč se stává novým brankářem.

Z výše uvedeného výčtu je vidět, že určení role podstatně omezuje možné akce hráče. Výběr akcí je tudíž rozdělen do dvou kroků – výběr role a výběr

akce. Zároveň je zřejmé, že dokud nebudou implementovány pokročilé akce v modulu elementární inteligence, budou akce omezeny na pohyb, ať už s míčem nebo ne. Z možností, jak implementovat určování rolí a akcí jsem uvažoval o dvou: rozhodovacích stromech a metodách strojového učení.

3.4.1 Rozhodovací stromy

Rozhodovací strom je struktura, kde uzly stromu představují jednotlivé testované skutečnosti a listy pak představují určitá rozhodnutí. Přechody mezi uzly stromu se aktivují v závislosti na výsledku testovaných skutečností. Obecný příklad takového stromu ukazuje obrázek 3.8. Strom samozřejmě nemusí mít vždy jen dva potomky v každém uzlu, může jich mít více, neboli být N-ární. Přechodů z uzlu může být více podle velikosti testované proměnné.



Obrázek 3.8: Obecný příklad rozhodovacího stromu

Výhodou rozhodovacích stromů je především jejich snadná čitelnost. Díky tomu se snadno převedou na sadu pravidel algoritmu a ten lze naopak snadno převést na stromové schéma. Rozhodovací stromy lze snadno upravovat, a pokud se tak učiní, není nutné přetrénovat klasifikátor jako v případě strojového učení. Nicméně v případě komplexních stromů může znamenat změna některé proměnné nutnost přepracovat celý strom nebo jeho velkou část. Další nevýhodou rozhodovacích stromů je fakt, že jsou založeny na očekávaných datech. Data, která byť jen těsně spadají mimo tato očekávaná data, budou klasifikována velmi nepřesně. Často nemusí být ani možné stanovit celou množinu

očekávaných dat, a tak může strom vést ke špatným rozhodnutím. Ačkoliv lze tedy rozhodovací strom sestavit na základě malého množství trénovacích dat, může být zároveň velmi nepřesný [11].

Poslední zmíněnou nevýhodu ohledně očekávaných dat lze eliminovat, pokud autor předem zná celkovou podstatu dat a ví tedy přesně, jaká data lze očekávat. Toho v případě této práce dosáhnu, protože podobu trénovacích dat (hodnocení robotů) sestavuji sám a znám závislost těchto dat. Pomáhá také, že tato data budou pravděpodobně jednoduchá, získatelná několika snadnými výpočty. Hlavní nevýhodou tedy zůstává velká složitost stromu, protože je třeba řešit mnoho situací, včetně těch málo nastávajících – například co dělat, když dva roboti mají určeno, že vlastní míč.

3.4.2 Strojové učení

Strojové učení jsou algoritmy umělé inteligence, které jsou schopné se naučit řešit daný problém na základě zadaných (takzvaných *trénovacích*) dat, aniž by byly k tomu explicitně naprogramovány. Programy se učí na základě své *zkušenosti* s daty a na základě dosažených výsledků upravují své zpracování dat. Algoritmy strojového učení jsou tedy schopny pracovat s dosud neviděnými informacemi, protichůdnými informacemi a chybami v datech.

Existují dva přístupy k technikám strojového učení – učení s učitelem a bez učitele. První přístup využívá algoritmy, kterým je v rámci trénovacích dat předán i změřený výsledek těchto dat. Algoritmy se pak snaží tomuto výsledku přiblížit s minimální chybou. Algoritmům učení bez učitele je předána pouze množina vstupních dat a program sám už musí v datech najít nějakou strukturu. Z popisu obou přístupů je zřejmé, že na použití v rozhodování o rolích a akcích hráčů je použitelné učení s učitelem. V tom případě by byla algoritmu předána sada spočtených hodnocení robota a výsledná role či akce, kterou by učitel robotu na základě těchto dat přiřadil. Protože výsledkem je zařazení do jedné z několika tříd, jde o problematiku *klasifikace*. Použitelné techniky jsou zde *logistická regrese* a *neuronové sítě* [12].

Logistická regrese

Logistická regrese je metoda, která na základě vstupních hodnot odhaduje pravděpodobnost zařazení do jedné konkrétní třídy. Pro případ klasifikace do několika tříd, jako v případě robofotbalu, kde třída odpovídá roli či akci, je třeba vytvořit klasifikátor pro každou třídu a výslednou třídou je ta, jejíž klasifikátor dosáhl nejvyššího výsledku. Klasifikátor se během trénování pokouší přiřadit vstupním hodnotách x_i váhu θ_i a použije následující vzorec:

$$h(X) = \frac{1}{1 + e^{-\Theta^T X}} \tag{3.1}$$

X je zde vektor vstupních hodnot, Θ je vektor vah vstupních hodnot a $h(X)$ je *hypotéza*, neboli výsledná pravděpodobnost zařazení do určité třídy. Tato hypotéza je poté použita ve funkci ceny, v angličtině *cost function*, která vyjadřuje chybu oproti očekávanému výsledku zadanému učitelem. Funkce vypadá následovně:

$$Cost(h_\Theta(X),y) = -y \log(h_\Theta(X)) - (1 - y) \log(1 - h_\Theta(X)) \tag{3.2}$$

V této rovnici y představuje výsledek vstupních hodnot zadaný v datech. Funkce $Cost(h_\Theta(X),y)$ se pak sečte přes všech M trénovacích příkladů:

$$J(\Theta) = \frac{1}{M} \sum_{i=1}^{M} Cost(h_\Theta(X^{(i)}),y^{(i)}) \tag{3.3}$$

$J(\Theta)$ je tedy celková cena klasifikace pro konkrétní vektor Θ. Algoritmus se nyní snaží vektor pozměnit tak, aby cena $J(\Theta)$ klesa a nakonec byla minimální:

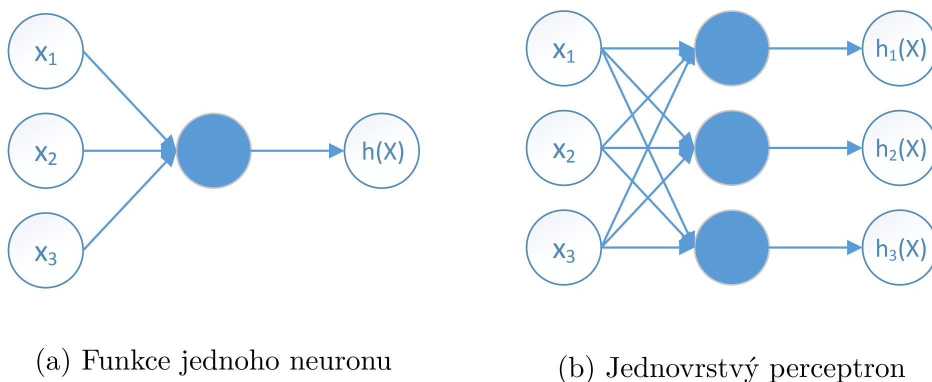
$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\Theta) \tag{3.4}$$

$$\theta_j = \theta_j - \alpha \sum_{i=1}^{M} (h_\Theta(X^{(i)}) - y^{(i)}) x_j^{(i)} \tag{3.5}$$

Tato úprava θ_j (složky vektoru Θ) proběhne souběžně pro všechny hodnoty $j = 1..N$, kde N je počet vstupních parametrů v jednom trénovacím příkladě. Výše uvedené vzorce se v cyklu aplikují, dokud není změna vektoru Θ nulová nebo dostatečně malá. Proměnná α ve vzorci určuje rychlost konvergence k řešení.

Neuronové sítě

Neuronové sítě jsou složitější struktury, kde jeden uzel (*neuron*) vykonává stejnou funkci jako celý klasifikační algoritmus logistické regrese. Tyto neurony jsou vedle sebe umístěny v jednotlivých vrstvách, kterých může být jedna i více. Vazby mezi vrstvami mohou být různé podle typu sítě, ale já zde budu uvažovat pouze o sítích s dopřednou propagací, takzvaných *perceptronech*. Jednovrstvý perceptron o k neuronech pak požadovanými výsledky odpovídá logistické regresi do k tříd. Příklad neuronu a jednovrstevného perceptronu je na obrázku 3.9, kde modré uzly jsou jednotlivé neurony. Nákres jednoho neuronu může zároveň sloužit také jako schéma funkce logistické regrese [13].



(a) Funkce jednoho neuronu

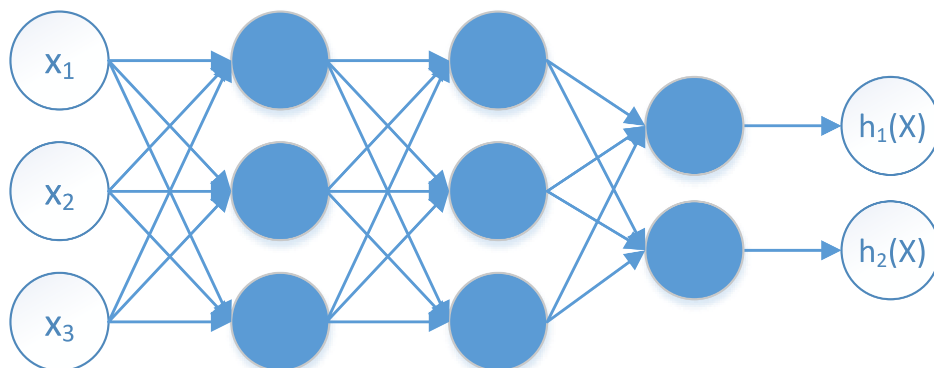
(b) Jednovrstvý perceptron

Obrázek 3.9: Neuron a perceptron

Vícevrstvé perceptrony fungují na pricipu, kdy kromě vstupní vrstvy (proměnných) a výstupní vrstvy existují ještě skryté vrstvy. Pak se výstupy jedné vrstvy stanou vstupními parametry další vrstvy. Těmto výstupům z jednotlivých vrstev se říká *aktivace* a spočítají se obdobně jako výstupy u linerání regrese:

$$a^{(l)} = \frac{1}{1 + e^{-\Theta^{(l-1)} a^{(l-1)}}} \tag{3.6}$$

Zde $a^{(l-1)}$ je aktivace z předchozí vrstvy. Perceptrony mají pouze dopřednou propagaci, proto pro první aktivaci platí $a^{(1)} = X$, neboli jde o vektor vstupů. Na druhé straně pak při L vrstvách platí, že $a^{(L)} = h_\Theta(X)$, tedy jde o výsledné hypotézy. Počet neuronů v jednotlivých vrstvách se může lišit, pouze počet v poslední vrstvě musí odpovídat počtu klasifikovatelných tříd, protože poslední vrstva je ta, která poskytuje výsledné hypotézy. Schéma takového vícevrstvého perceptronu ukazuje obrázek 3.10.



Obrázek 3.10: Příklad vícevrstvého perceptronu

Trénování probíhá podle podobných principů jako u logistické regrese, jen je ve vzorcích třeba vzít v úvahu možnost několika hypotéz naráz. Stejně tak váhy v jednotlivých vrstvách perceptronu nebudou vektory, ale matice, a pro každou vrstvu bude existovat samostatná taková matice. Oprava vah podle chyby se řeší pomocí zpětné propagace chyby od výstupní vrstvy k vstupní, kde chyby se postupně přenásobují vahami jednotlivých přechodů. Protože je matematická podstata stejná jako u logistické regrese, ale zato vzorce jsou podstatně složitější, nebudu zde tyto vzorce zbytečně rozepisovat.

Výhody a nevýhody

Obecnou výhodou metod strojového učení je, že po důkladném natrénování je algoritmus schopný bez problémů zpracovat i data, která dosud

neviděl, pokud logicky vychází z trénovacích dat. Dalším plusem je absence nutnosti předělávat algoritmus, pokud se nějak změní druh vstupních dat (například přibyde nebo ubyde proměnná), na druhou stranu je nutno klasifikátor znovu natrénovat. Samotné trénování zabírá určité množství času, příliš velké na to, aby se daly parametry klasifikátoru měnit průběžně během hry, zatímco nahrazení rozhodovacího stromu jiným je záležitostí okamžiku. Především pak u trénování vícevrstvých neuronových sítí jde o časově náročnou záležitost.

Co se týče trénování klasifikátoru, je potřeba velké množství trénovacích dat, ve kterých učitel označí požadované výsledky. Při poskytnutí dostatečného množství trénovacích dat vznikne spolehlivý klasifikační algoritmus, ale vytvoření takovýchto trénovacích dat je náročná úloha. Při pěti sledovaných robotech a necelých deseti parametrech (hodnocení hráčů) by mohlo jít až o stovky ručně označených testovacích vstupů.

Další podstatnou nevýhodou je implementační složitost. Algoritmy strojového učení stojí na složitém matematickém modelu a je tak snadné při implementaci udělat chybu. Takováto chyba je těžko odhalitelná, protože ze samoučící podstaty algoritmů často tato chyba nezpůsobí selhání funkcí a mnohdy ani na první pohled špatné výsledky. Pro první iteraci vývoje algoritmů herní strategie by bylo tedy nejspíše vhodnější použít rozhodovací strom, jako spolehlivý odrazový můstek.

Co se týče rozhodnutí zda použít logistickou regresi nebo perceptron, není zde jednoznačná odpověď. Princip jednovrstevného perceptronu je velmi podobný principu klasifikace logistickou regresí do více tříd, tudíž v případě použití strojového učení jde především o rozhodnutí, zda použít logistickou regresi či vícevrstevný perceptron. Volba, zda použít skyté vrstvy a tím pádem skončit u vícevrstevného perceptronu, není jednoznačná. Různé typy úloh pracují nejlépe s jiným počtem skytých vrstev a jiným počtem neuronů v těchto vrstvách, a proto nelze určit, jaká možnost by byla vhodná pro potřeby robotického fotbalu. Bylo by nutné otestovat mnoho možností. Kvůli tomuto a s ohledem na nutnost velkého množství trénovacích dat je studie o použitelnosti metod strojového učení vhodná pro práci cílící přímo na tento problém.

3.5 Vážení grafu

Než se přikročí k vlastnímu hledání cesty v grafu, je třeba diskrétním polím na hřišti přiřadit určité váhy. Ty by měly určovat nevhodnost pole pro pohyb robota. Váhy tedy značí negativní hodnocení, které by později grafový algoritmus měl vzít v potaz. Identifikoval jsem 3 základní faktory určující nevhodnost pole:

- Pozice soupeře
- Pozice spoluhráče
- Pohybový vektor soupeře

Pozice všech hráčů v poli jsou samozřejmě nežádoucí pole, protože trasa naplánovaná přes tato pole obsahuje překážku. To znamená, že modul elementární inteligence bude nucen provést akci vyhnutí se překážce, která silně pozmění plánovanou trasu robota. Hráč by měl tedy v rámci hledání trasy najít takovou trasu, která robota v poli objede za podmínek určených herní strategií a minimalizuje tak nutnost použít vlastní akci vyhnutí překážce.

Pohybový vektor soupeře je důležitý pro hledání trasy, protože určuje sadu polí, na kterých se v dohledné době tento soupeř může přesunout. Při plánování trasy je tedy vhodné vyhnout se polím, kde se bude soupeř vyskytovat s velkou pravděpodobností (pole blízko soupeřova robota). Naopak v určitých případech se může vyplatit riskovat a zkrátit trasu přes málo ohrožená pole, kam se soupeř může přesunout jen možná.

Pohybový vektor spoluhráče není tolik podstatný jako protihráčův a to ze dvou důvodů: zaprvé je vyhýbání dvou spřátelených robotů známé a modifikovatelné. Algoritmus se může upravit tak, aby jeden z robotů mohl mít označenu prioritní trasu a ten druhý by mu z ní uhnul. Druhým důvodem je, že herní strategie by měla ve výsledku zkoordinovat plány robotů tak, aby ke kolizím nedocházelo. Zatímco u soupeřova robota je budoucí pozice neznámá, u spřáteleného je známa a měla by být mimo trasu robota s důležitějším úkolem (typicky útočník).

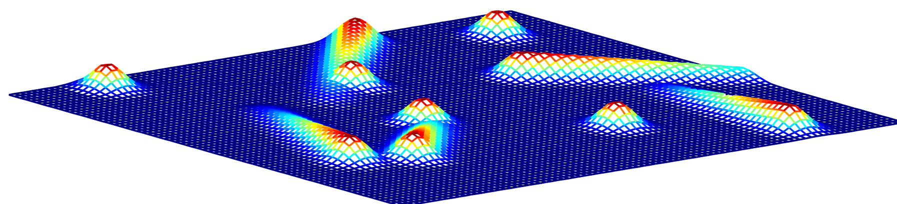
Z výše uvedených specifikací nevhodnosti pole vyplývá, že míra této nevhodnosti by neměla být skoková, ale růst postupně, jak se robot pohybuje směrem k více nežádoucím polím. Toto by mělo hráči poskytnout na trase

důležitý manévrovací prostor pro řešení nečekaných situací. Jako ideální vzorec pro rozprostření vah na pole v okolí pozic robotů se nabízí Gaussovská funkce s obecným vzorcem pro dvourozměrný prostor:

$$f(x,y) = Ae^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}}, \tag{3.7}$$

kde $[x_0,y_0]$ jsou souřadnice středu, vůči kterému se funkce počítá, a σ je parametr, určující tvar funkce. Čím je vyšší, tím je charakteristický tvar zvonu Gaussovské funkce širší. Nakonec A je parametr určující výšku, kterou bude mít zvonovitý tvar funkce. Při $A = 1$ je výška funkce ve středu 1. Pro účely váhování diskrétních polí bude A představovat maximální přidělenou váhu, umístěnou na pozici hráče.

Pro váhování polí podle vektoru pohybu soupeře bude třeba aplikovat obdobný vzorec jednorozměrné Gaussovské funkce v kolmých směrech na vektor pohybu. Zároveň ale musí tyto váhy ve směru vektoru postupně klesat. Zde lze použít lineární pokles, případně jinou, ať už rychleji či pomaleji klesající funkci. Nakonec je nutné otestovat druh pásma vah ve směru vektoru. Šířka může zůstávat konstatní nebo se může s rostoucí vzdáleností od robota rozšiřovat. Toto rozšiřování by odráželo skutečnost, že robot může (a pravděpodobně bude) měnit směr svého pohybu. Rozšíření pásma umožní hráči naplánovat trasu s větší opatrností, které ovšem při příliš vysokých vahách může trasu zbytečně prodloužit.



Obrázek 3.11: Příklad zobrazení vážené sítě jako 3D mapy

Výše uvedenými principy váhovaná síť polí může představovat trojrozměrnou mapu hřiště. Pole bez přiřazené váhy jsou základní rovinou, z níž vystupují kopce přiřazených vah. Při této interpretaci přestává být vzdálenost dvou sousedních polí počítána ve dvou rozměrech, přibývá třetí rozměr odpovídající váze pole. Tímto již není vzdálenost dvou sousedních polí kon-

stantní (1 či $\sqrt{2}$ podle vzájemné pozice). V případě použítí trojrozměrné interpretace mapy vah v prohledávacím grafovém algoritmu už není relevantní hlavní výhoda užití hexagonální diskretizační sítě. Příklad zobrazení vah jako výškové mapy ukazuje obrázek 3.11

3.6 Prohledávání grafu

Jak jsem zmínil dříve, diskretizovaná pole hřiště budou představovat graf. V tomto grafu sousedí každý uzel až s osmi dalšími uzly. V případě interpretace grafu jako trojrozměrné mapy je vzdálenost dvou uzlů rovna jejich eukleidovské vzdálenosti v prostoru. Ovšem existují i jiné možnosti interpretace vzdálenosti, používané především pro optimalizaci algoritmu. Nejprve je třeba rozebrat standardní prohledávací algoritmy a jejich vhodnost pro problematiku herní strategie.

```
1  prohledávání(počátek, cíl){
2    seznam uzavřeno = prázdný seznam uzlů;
3    seznam otevřeno = počátek;
4
5    while(otevřeno není prázdné) {
6      u = vyber uzel z otevřeno;
7      test nalezení cíle;
8      odstraň u z otevřeno;
9      přidej u do uzavřeno;
10     for (uzel soused in sousedi(u)){
11       if (uzavřeno obsahuje soused) continue;
12       cena = spočítej_cenu(u, soused);
13       if ((otevřeno neobsahuje soused)
14         or (cena < soused.cena) {
15         if (otevřeno neobsahuje soused)
16           přidej soused do otevřeno;
17         soused.cena = cena;
18         přiřaď sousedu další statistiky;
19         soused.předchozí = u;
20       }
21     }
22   }
23   sestav_cestu(počátek, cíl);
24 }
```

Kód 3.1: Pseudokód prohledávání grafu

Kód 3.1 ukazuje obecný pseudokód prohledávání grafu. Tučně zvýrazněné části jsou ty, které se liší podle použitého algoritmu. Použity jsou dva seznamy uzlů. Seznam *otevřeno*, který obsahuje uzly čekající na otestování, a seznam *uzavřeno*, který se skládá z bodů již kompletně otestovaných. Podle typu prohledávání běží algoritmus v cyklu, dokud není seznam *otevřeno* prázdný, tedy dokud nejsou prohledány všechny uzly grafu. Druhou možností ukončení je dosažení cílového uzlu. Pokud je algoritmus nastaven tak, aby prohledával přednostně optimální cesty, nebo pokud postačuje nalezení jakékoliv cesty do cíle, končí prohledávání prvním nalezením cílového uzlu. Pokud prohledává algoritmus graf slepě, je pro nejlepší řešení třeba prohledat celý graf. Toto ukončení při nalezení cíle značí v pseudokódu ***test nalezení cíle***.

Dalším obecně uvedeným příkazem je ***u = vyber uzel z otevřeno***. Značí, že každý algoritmus přednostně vybírá ze seznamu jiný prvek, nikdy ne na slepo. Podle typu výběru zde prováděného je pak v konkrétní implementaci prohledávání často datový typ seznamu *otevřeno* nahrazen jiným typem, například frontou, zásobníkem či prioritní frontou. Uzel vybraný v tomto kroku se stává výchozím bodem pro prohledávání cesty do svých sousedů. Pokud je nějaký soused v seznamu *uzavřeno*, tak se neprohledává, protože do onoho uzlu již nejlze najít lepší cestu.

Řádka ***cena = spočítej_cenu(u, soused)*** značí spočtení celkové ceny pohybu z počátku do uzlu *soused* přes uzel *u*. Typicky k již známé ceně uzlu *u* přičte nově spočítanou cenu přesunu z *u* do *souseda*. To, v čem se zde algoritmy liší, je způsob jakým se počítá vzdálenost mezi uzly. Mnohdy se počítání vzdáleností liší i v rámci stejného typu grafového algoritmu, většinou v závislosti na daném problému, potřebách přesnosti a rychlosti.

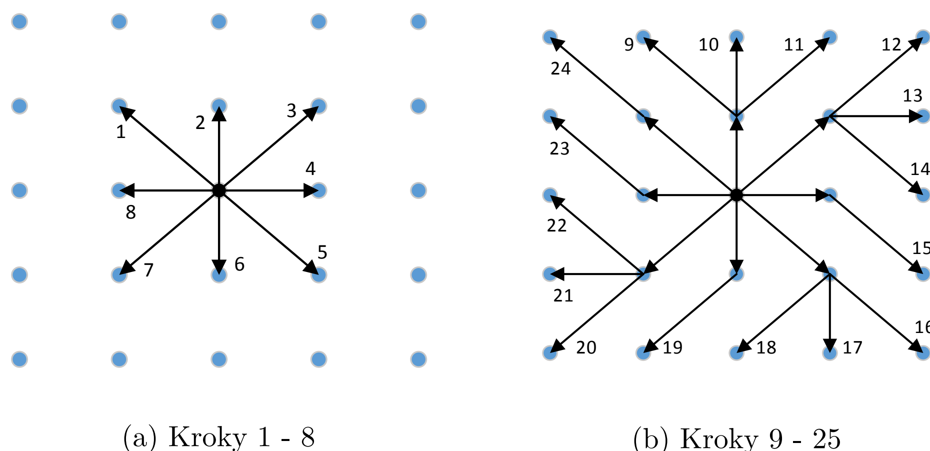
Pokud zkoumaný sousední uzel není v seznamu *otevřeno*, tak se do něj přidá, protože při známé ceně pohybu do něj (tedy známé částečné cestě) je vhodný jako potenciální výchozí bod dalšího prohledávání. Pokud už uzel v seznamu je, upraví se mu cena, pokud ta je nižší, neboli byla nalezena výhodnější cesta. V obou případech se uzlu nastaví odkaz na předchozí uzel, z něhož byl objeven. Nakonec se uzlu mohou přiřadit ještě dodatečné statistiky, které často ovlivňují způsob výběru uzlů ze seznamu *otevřeno*. Takovéto statistiky zahrnují například heuristické funkce.

Po skončení prohledávání se na základě odkazů na předchozí uzly sestaví celá cesta. Začne se u cílového uzlu a postupně se získávají jeho předchůdci na objevené cestě. Takto získaný seznam uzlů se nakonec obrátí. Pokud je předchůdce cílového uzlu v závěru nenastaven, znamená to, že buď došlo

k chybě v algoritmu (výjimka či chyba implementace), nebo je cílový uzel nedosažitelný, ať už vinou nespojitého grafu nebo neprostupných uzlů.

3.6.1 Prohledávání do šířky

Algoritmus prohledávání do šířky (anglicky **Breadth-first search**) je spolu s prohledáváním do hloubky jeden z nejzákladnějších a nejjednodušších grafových algoritmů. Při prohledávání do šířky se nejprve prohledávají uzly nejdříve objevené. Algoritmus tedy nejprve zkoumá ty uzly, do kterých se dostane nejméně kroky, bez ohledu na velikost těchto kroků (vzdálenost uzlů). Prohledávání na jednoduchém grafu ukazuje obrázek 3.12.



(a) Kroky 1 - 8

(b) Kroky 9 - 25

Obrázek 3.12: Příklad prohledávání grafu do šířky

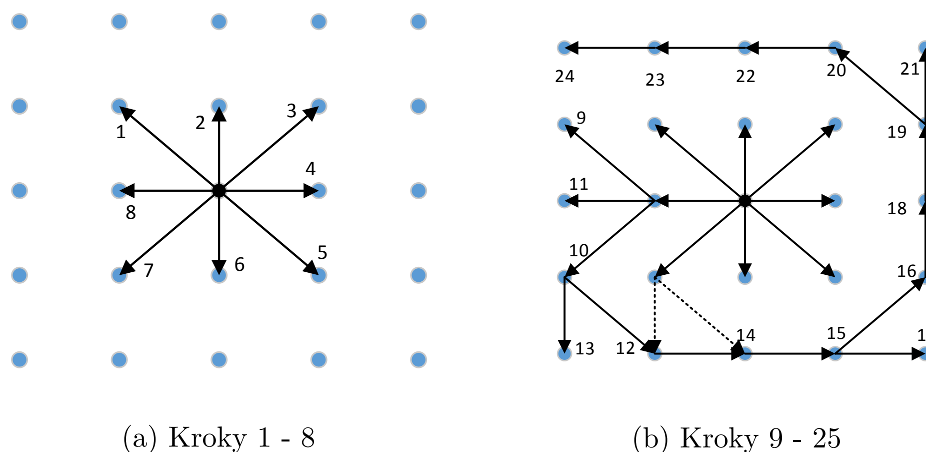
Na obrázku je šipkami znázorněno postupné objevování uzlů. Nejprve jsou objeveny sousední uzly počátku a následně pak sousedi těchto sousedů. V podstatě algoritmus objevuje nové uzly ve vrstvách nabalených na předchozí objevené uzly. Z obrázku také lze vyčíst skutečnost, že na grafu odpovídajícímu mřížce se dvěma konstatními kroky mezi uzly (rovný a šikmý), objeví graf hned všechny nejkratší cesty do všech bodů bez nutnosti cestu přepočítávat při nalezení „zkratky“ mezi dvěma body. Velkou nevýhodou je, že v případě váženého grafu algoritmus nerespektuje váhy uzlů při vybírání dalšího prohledávaného. Proto je nutno tento algoritmus nechat prohledat všechny uzly, pokud je třeba najít nejkratší cestu do cíle.

Implementačně prohledávání do šířky používá pro seznam *otevřeno* datovou strukturu fronty (struktura typu **FIFO - First-In-First-Out**) a žádné

dodatečné statistiky se mimo ceny vzdálenosti nepočítají.

3.6.2 Prohledávání do hloubky

Prohledávání do hloubky (anglicky **Depth-first search**) je algoritmus podobný prohledávání do šířky s jediným rozdílem. Při výběru nového zkoumaného uzlu se bere poslední nalezený uzel místo toho nejdříve nalezeného. Algoritmus vždy jako další krok prohledávání vybere poslední nalezený uzel. V okamžiku, kdy tento uzel již další nezpracované sousedy nemá (nebo nemá již vůbec žádné), vrací se prohledávání po objevené cestě zpět, dokud nenarazí na uzel, který nezpracované (či levněji dosažitelné) sousedy ještě má. Zde pokračuje v hledání a vytváří tak v tomto místě další prohledávanou „větev“. Takové prohledávání do hloubky na jednoduchém grafu ukazuje obrázek 3.13.



(a) Kroky 1 - 8 (b) Kroky 9 - 25

Obrázek 3.13: Příklad prohledávání grafu do hloubky

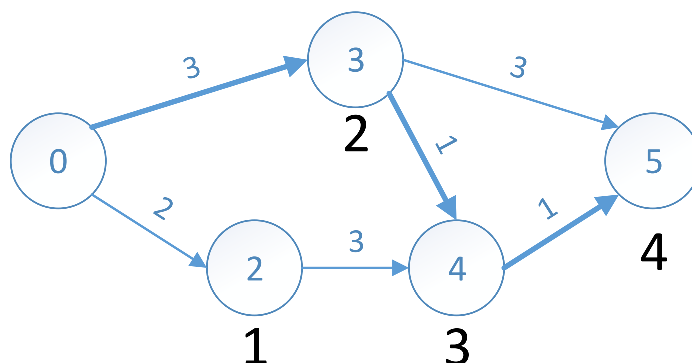
Obrázek ukazuje pouze objevování nových uzlů. Situací na druhé části obrázku algoritmus nekončí, ale je třeba objevit ještě výhodnější cesty do známých uzlů, jak naznačují tečkované šipky z uzlu 8. To demonstruje, že na grafu odpovídajícímu mřížce je algoritmus náročnější, protože dochází k přepočtům nalezených cest. Nevýhodu prohledávat celý graf pro nalezení nejlepšího řešení má algoritmus společnou s prohledáváním do šířky.

Implementačně je prohledávání do hloubky shodné s prohledáváním do šířky, až na způsob implementace, kde místo fronty je použit zásobník (datová struktura typu **LIFO - Last-In-First-Out**).

3.6.3 Dijkstrův algoritmus

Dijsktrův algoritmus patří k asymptoticky nejrychlejším grafovým algoritmům pro grafy s nezápornými vzdálenostmi mezi uzly (hranami). Algoritmus nezkoumá uzly slepě, ale vybírá si je podle jejich výhodnosti. Vychází z logiky, že uzel s nejmenší cenou je v nejlepší pozici najít cílový uzel při zachování minimální ceny. Algoritmus tedy jako zkoumaný uzel, z něhož se prozkoumávají další uzly, vybírá ten, který má v současné době cenu nejnižší. Algorimus skončí v okamžiku, kdy je jako uzel s nejnižší cenou vybrán cílový uzel. Není třeba již prohledávat zbytek grafu, protože ostatní uzly mají cenu vyšší, přestože ještě nejsou v cíli. Protože v grafu nejsou záporné hrany, nelze již najít kratší cestu.

Pro případ grafu se zápornými hranami není Dijsktrův algoritmus vhodný. Pro tyto grafy se používá *Bellman-Fordův algoritmus*, který má ale podstatnou nevýhodu větší výpočetní složitosti [14]. Protože prohledávaný graf v této práci neobsahuje záporné hrany, a tedy Dijkstrův algoritmus je lepší varianta, nebude tato práce Bellman-Fordův algoritmus rozebírat.



Obrázek 3.14: Ukázka Dijkstrova algoritmu

Obrázek 3.14 ukazuje příklad funkce Dijkstrova algoritmu na velmi jednoduchém grafu. Čísla nad hranami jsou jejich váhy, čísla v uzlech jsou jejich ceny v době dokončení algoritmu a černá čísla u uzlů jsou jejich pořádí výběru. Uzel s černým číslem 1 byl objeven jako první a z něho byl vedlejší uzel nalezen s cenou 5. Jenže přes vrchní uzel byl hned tento vedlejší uzel objeven s nižší cenou 4. Stejný případ nastal i pro cílový uzel. Ten byl nejprve objeven s cenou 6, ta byla později upravena na 5. S touto cenou byl také uzel vybrán

a algoritmus skončil.

Logika je oproti prohledávání do šířky a do hloubky změněná pouze o výběr uzlu ze seznamu *otevřeno* a je zde navíc ukončení v případě výběru cílového uzlu. Proto se i implementace Dijkstrova algoritmu liší v těchto dvou bodech. Přibylo testování cílového uzlu a ze seznamu *otevřeno* je vybírán prvek s nejnižší cenou. Pro optimalizaci rychlosti algoritmu je vhodné tento seznam implementovat jako prioritní frontu. Ta funguje jako obyčejná fronta, ale při vložení prvku se prvek automaticky zařadí na místo odpovídající jeho ceně. Složitost výběru je pak pouze O(1).

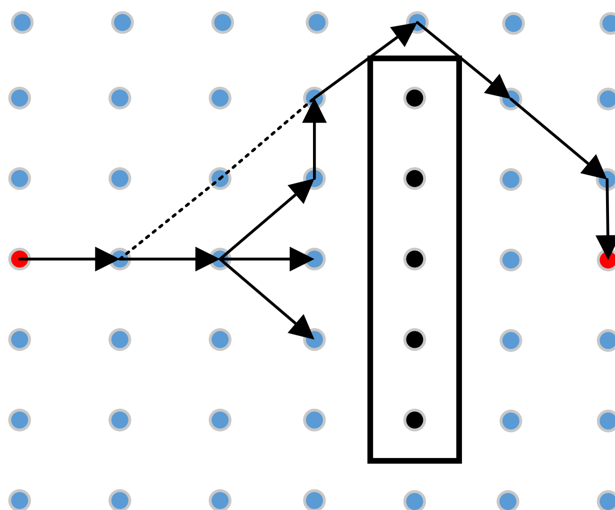
Dijkstrův algoritmus je výhodnější než předchozí zmíněné algoritmy, nicméně v hustém grafu odpovídajícímu mřížce stále prohledává velké množství uzlů. Například vybírá uzly sice s nejmenší cenou, ty ale mohou ležet úplně na druhou stranu od cílového uzlu. Je proto potřeba algoritmus, který započítává do výběru i postavení uzlů k cílovému bodu. Těmto algoritmům se říká **best-first search** algoritmy, česky překládané jako **uspořádané prohledávání**.

3.6.4 Greedy best-first search

Jak již anglický název napovídá, algoritmy uspořádaného vyhledávání vybírají ze seznamu *otevřeno* takový uzel, o kterém předpokládají, že je vzhledem ke zbývající cestě do cíle nejvýhodnější. Tento předpoklad se nazývá *heuristika*, značí se typicky $h(x)$ a znamená odhad ceny zbývající vzdálenosti do cíle. Narozdíl od ceny u uražené vzdálenosti není heuristika přesná, protože algoritmus nemá přesnou představu, jak vypadá zbývající cesta do cíle. Heuristika může například odhadnout vzdálenosti na základě eukleidovské vzdálenosti, jenže ve zbytku cesty se mohou nacházet neprostupná pole, takže nakonec bude výsledná cena podstatně vyšší.

Pod označení best-first algoritmů spadá algoritmus, který vybírá uzly čistě na základě heuristiky a je nazýván **greedy best-first search**. Podle prvního slova názvu patří do skupiny *greedy* (česky *hladových*) algoritmů. Tyto algoritmy vybírají jako další krok lokálně optimální řešení a snaží se tedy k cíli dorazit s řešením, které je typicky dobré, ale ne nejlepší. Greedy algoritmy tedy dokáží rychle najít nějakou cestu k cíli, často ale ne tu optimální. Greedy best-first algoritmus jako kritérium vybírá sousední uzel, pokud jeho heuristika je menší než heuristika předchůdce. Neboli vybírá uzel, o němž předpokládá, že ho nejvíce přiblíží k cíli ze současného uzlu. Pokud

žádný ze sousedů nemá lepší heuristiku, je soused vložen do prioritní fronty a vybrán je prvek s nejnižší heuristikou ze všech [15].



Obrázek 3.15: Ukázka greedy best-first prohledávání

Obrázek 3.15 je příkladem prohledávání pomocí greedy best-first algoritmu. Heuristika prohledávní určuje, že pohyb směrem k cíli snižuje hodnotu této heuristky, neboli je vhodným krokem vpřed. V okamžiku setkání s neprůchodnou překážkou algoritmus sice nenajde souseda s heuristikou lepší než současný bod, nicméně sousední body (původně objeveny z bodu předcházejího) mají nejnižší globální heuristiku. Z jednoho z těchto bodů je opět třeba přistoupit na krok, který zvýší heuristiku, ale poté už se heuristika opět snižuje, dokud se nedorazí do cíle. Část skutečně optimální trasy, kterou prohledávání nenajde, je vyznačena tečkovanou čarou.

Ačkoliv je výhodou rychlost a prohledávání obecně správným směrem, chybí v algoritmu nalezení optimální cesty. Aplikace heuristiky do Dijkstrova algoritmu způsobí oříznutí zbytečných částí prohledávání a podstatně tak urychlí vlastní algoritmus. Taková kombinace se nazývá **A***.

3.6.5 Algoritmus A*

A* (někdy také zapisován jako *A-star*) je best-first algoritmus, který hodnocení uzlu zakládá jak na ceně již uražené vzdálenosti, tak na heuristice zbývající vzdálenosti. Kromě obvyklé ceny pohybu do bodu x se počítá také celkové skóre (označované jako *f-skóre*) tohoto bodu vzorcem $f(x) = g(x)+h(x)$, kde $g(x)$ je právě cena pohybu do bodu (*g-skóre*) a $h(x)$ je heuristika. G-skóre je definováno jako $g(x) = g(x') + d(x',x)$, kde $g(x')$ je g-skóre předchozího uzlu a $d(x',x)$ je cena přechodu mezi těmito dvěma uzly. F-skóre je nejdůležitějším hodnocením uzlu v tomto algoritmu a podle něj se vybírají nové uzly pro další prohledávání.

A* je kvůli použití heuristiky sice také greedy algoritmus, ale při správně zvolené heuristice může spolehlivě vracet optimální řešení. Takto správně zvolená heuristika se nazvývá *přípustná*, anglicky *admissible*. To znamená, že heuristika jakéhokoliv bodu je vždy stejná jako nejnižší možná cena cesty $L(x)$ z tohoto bodu do cíle, nebo je podhodnocením této cesty. Matematicky $\forall x : h(x) < L(x)$. Pokud heuristika někdy cenu nadhodnocuje, dojde k oříznutí některých řešení. A* pak najde řešení rychleji, ale méně optimálně. Čím více je heuristika nadhodnocena, tím více se A* podobá výše uvedenému greedy best-first prohledávání. A naopak, čím více je heuristika podhodnocena, tím více vedlejších řešení se prohledává. Pokud je hodnota heuristiky vždy 0, stává se A* Dijkstrovým algoritmem [16].

Další vlastností důležitou pro nalezení optimálního řešení je *monotónnost*, někdy nazývaná *konzistence*. Monotónnost je definována podmínkou $h(x) - h(y) \leq d(x,y)$, pro všechny sousedící páry bodů x,y. To znamená, že jakýkoliv krok v grafu, který zmenší heuristiku, musí mít cenu tohoto kroku minimálně stejně velkou. Jinými slovy žádný krok nesmí vést k menšímu f-skóre bodu, než má jeho předchůdce. Díky tomuto omezení nelze znovuobjevit lepší cestu do bodů v seznamu *uzavřeno*, protože nově objevené trasy do známých bodů budou mít zákonitě větší f-skóre. Nedodržení monotónnosti by znamenalo, že seznam *uzavřeno* je zbytečný a výpočetní složitost by se zhoršila.

V ohledem na výše uvedené je důležité při výběru heuristiky určit správné měření vzdálenosti. Ačkoliv je eukleidovská vzdálenost nejintuitivnější, kvůli operaci odmocniny je její počítání pomalé. Podle výsledného naměřeného výkonu hledání trasy bude možná nutné zvážit použití jiné, rychleji počítané heuristiky.

Implementačně využívá A* pro seznam *otevřeno* prioritní frontu, kde pri-

oritou řazení je f-skóre bodu. Stejně jako v Dijkstrově algoritmu skončí nalezením první cesty do cíle. První nalezená cesta bude i nejoptimálnější, pokud heuristika dodrží pravidla přípustnosti a monotónnosti. Výpočet ceny sousedních uzlů probíhá standardně přiřazením g-skóre, nicméně při zařazení či znovuzařazení bodu do seznamu *otevřeno* je třeba bodu přiřadit celkové f-skóre na základě ceny a heuristiky.

4 Realizace

4.1 Diskretizace

Jak jsem již psal v teoretickém rozboru, rozhodl jsem se použít čtvercovou mřížku pro diskretizaci hřiště. Dále se nabízí volba mezi hustou a řídkou sítí. Jako nejmenší možná velikost řídké sítě se nabízela délka pole o hraně 10 centimentrů. Jakákoliv menší velikost se nedala do hřiště o rozměrech 220 $cm \times 180$ cm umístit. Pole husté sítě jsem pak zvolil o délce hrany 2,5 cm, kdy nenatočený hráč obsadí takřka přesně devět těchto polí. Poslední zvažovaná varianta byla délka hrany 2 cm, nicméně tuto variantu jsem nechal v zásobě pro případ, že by bylo nutno zjemnit diskretizaci pole kvůli nedostatečné přesnosti algoritmů.

Brzy po začátku implementace dalších algoritmů jsem zavrhl variantu s řídkou sítí a pokračoval pouze s jedinou hustou sítí s poli o velikosti 2,5 $cm \times$ 2,5 cm. Důvodem byla nedostatečná přesnost pro správné aplikování vah grafu a hustá síť umožnila zjednodušit řešení viditelnosti objektů na hřišti.

Vlastní disretizaci pozic hráčů a míče jsem provedl následujícím výpočtem:

$$index_x = \lfloor x/a \rfloor \tag{4.1a}$$

$$index_y = \lfloor y/a \rfloor \tag{4.1b}$$

Konstanta a je délka hrany jednoho pole, tedy 2,5 cm. $[x,y]$ jsou reálné souřadnice na hřišti a $\lfloor\rfloor$ je operace zaokrouhlení dolů. Tímto se získají diskrétní souřadnice odpovídající indexům do dvourozměrného pole. Během zpracování diskrétních pozic v následných výpočtech se objevila potřeba, aby souřadnice byly ve spojité podobě, ale zároveň byly ve stejné soustavě jako ty diskrétní:

$$indexF_x = x/a - 0{,}5 \tag{4.2a}$$

$$indexF_y = y/a - 0{,}5 \tag{4.2b}$$

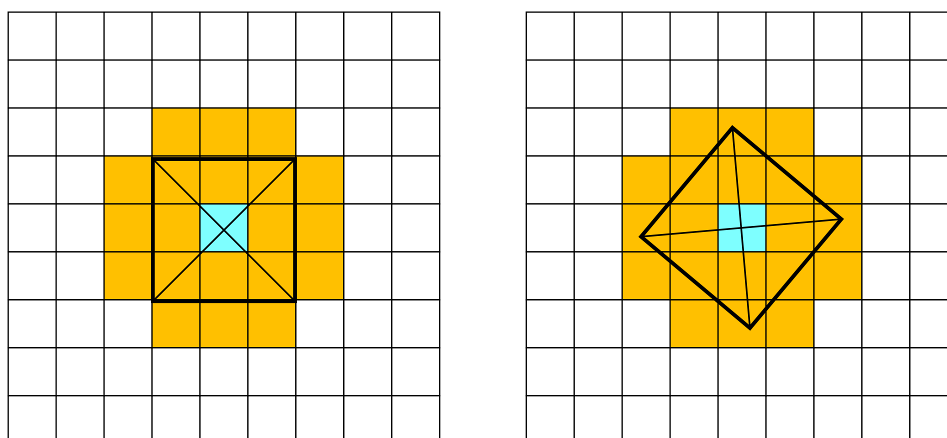
Proměnná $indexF$ je dvojice reálných čísel uložená jako datový typ float,

proto písmeno F. Tímto se získají spojité souřadnice, které pokud leží na určitém herním poli, jsou od jeho středu vzdáleny maximálně 0,5 v obou rozměrech. Platí tedy (za použití *round* jako funkce zaokrouhlení):

$$index_x = round(indexF_x) \tag{4.3a}$$

$$index_y = round(indexF_y) \tag{4.3b}$$

Nyní je pozice robota zdiskretizována jako jedno malé pole. Kvůli rozměrům robota je ale třeba určit, která pole pokrývá. Takřka vždy bude robot pokrývat 3×3 pole, nicméně při natočení zasahuje i do dalších polí. Zvažoval jsem, že pokrytá pole budu počítat podle natočení a pozice robota, takže by každý robot měl jiný tvar pokrytých polí. Nakonec jsem ale přistoupil k jednotnému schámatu pokrytí, jak ukazuje obrázek 4.1.



(a) Srovnání bez natoční

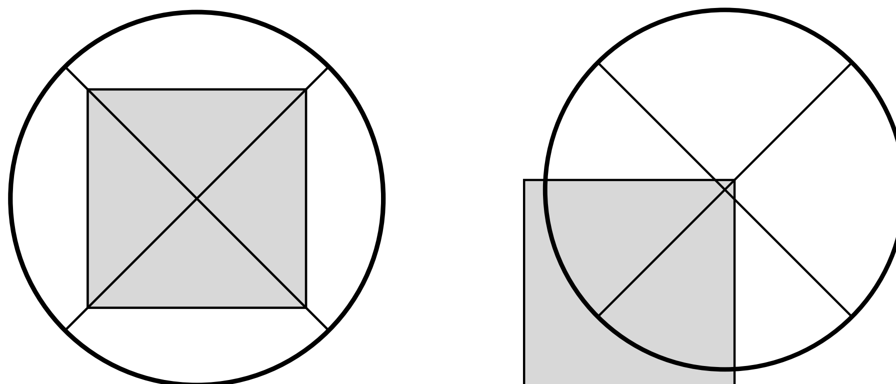
(b) Srovnání s natočením

Obrázek 4.1: Schéma určení polí pokrytých robotem

Obě části obrázku ukazují, že některá označená pole mohou zůstat skoro nebo úplně neobsazena. Rozhodl jsem se to takto ponechat a mít tak chybu spíše na straně obsazenosti pole, než naopak. Pokud je pole obsazeno, jedná se o významnou skutečnost, zatímco neobsazenost pole je neutrální sdělení, které poli nepřiřazuje nějaký vyšší význam. Naopak, zvažoval jsem rozšíření obsazené oblasti ještě o jednu vrstvu bodů kolem stávající oblasti, ale nakonec jsem zůstal u původní velikosti. V případě, že se ukáže vhodným rozšířit oblast, není problém tak v budoucnosti udělat.

Ve výsledku tak obsazená pole kolem robota tvoří čtverec o rozměrech 12,5 $cm \times 12{,}5$ cm s uřízlými rohy. Tento čtverec je sice větší než zvažované pole řídké sítě (10 $cm \times 10$ cm), nicméně má výhodu, že je víceméně zarovnaný kolem středu robota a tedy mnohem přesnější.

Pole v uvedeném okolí pozice robotů jsou tedy označena jako obsazená jedním z týmů. Protože ale má oblast obsazená robotem statickou velikost kolem jeho diskrétní polohy, může se stát, že některá pole připadnou dvěma robotům najednou. V případě, že jsou stejného týmu, se nic neděje, ale pokud pole sdílí oba týmy, je vhodné pole označit jako *sporné* (v angličtině *contested*). Tato označení obsazenosti polí zatím nemají další význam pro použité algoritmy, nicméně je dobré je do budoucna mít. Jediný postup, který zkoumá obsazenost polí, ale už ho nezajímá přesně kým, je zjištění viditelnosti, které popíše následující podkapitola.



(a) Pozice ve středu pole (b) Pozice na okraji pole

Obrázek 4.2: Srovnání velikosti míče a pole o délce hrany 2,5 cm

Posledním objektem na hřišti, kterému zbývá přiřadit diskrétní pozice je míč. Vlastní diskretizování pozice míče na jedno pole probíhá stejně, podle výše uvedených rovnic. Zbývá rozhodnout, zda míči nepřiřadit ještě nějaká další pole, která obsazuje. Míč má průměr 4,27 cm [4]. Obrázek 4.2 srovnává tuto velikost s jedním polem hřiště. Ukazuje, že míč dokáže spolehlivě pokrýt toto pole. Také demonstruje, že i při nejvyšším vychýlení pozice míče vůči středu pole míč stále obsazuje přibližně polovinu svého pole. Z této skutečnosti vyvozuji, že toto pole je pro určení pozice míče relevantní za každé situace. Pokud bude nějaký přístup vyžadovat znát přesnou pozici míče, lze

vždy využít spojitou pozici $indexF$, která je počítána pro takovéto případy.

4.2 Hodnocení hráčů

Jako statistiky pro hodnocení hráčů jsem nakonec zvolil a implementoval následující proměnné:

- Vzdálenost k míči
- Natočení k míči
- Vlastnictví míče
- Viditelnost míče
- Vzdálenost k domácí brance
- Vzdálenost k soupeřově brance
- Natočení k soupeřově brance
- Viditelnost spoluhráčů
- Inverzní útočné hodnocení

4.2.1 Vzdálenost k míči

Vzdálenost k míči byla realizována pomocí eukleidovské vzdálenosti mezi reálnými souřadnicemi míče a hráče:

$$d_b = \sqrt{(x - x_b)^2 + (y - y_b)^2} \tag{4.4}$$

$[x_b, y_b]$ značí reálné souřadnice míče. V tomto vzorci by bylo možno použít i vzdálenosti v diskretizované soustavě. Ať už eukleidovskou vzdálenost mezi diskretizovanými poli nebo třeba vzdálenost měřenou v rovných a diagonálních krocích mezi poli. Zatím žádné funkce nevyžadovaly takovouto změnu, statistika vzdálenosti k míči zatím slouží pouze ke srovnávání.

4.2.2 Natočení k míči

Natoční k míči je statistika značící, jak moc je robot odvrácen od směru k míči:

$$\alpha_b = |\phi_b - \phi| \tag{4.5}$$

ϕ je natočení robota a ϕ_b je natočení spojnice hráče s míčem, tedy směr vůči míči. Všechny úhly jsou počítány ve stupních, přičemž úhel 0 značí vodorovný směr zleva doprava.

4.2.3 Vlastnictví míče

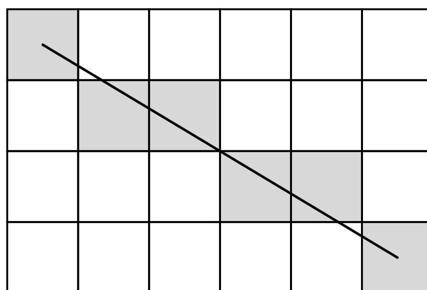
Vlastnictvní míče je binární proměnná. Hráč buď míč má nebo ne. Jak bylo zmíněno v teoretické části, je třeba zohlednit jak vzdálenost k míči, tak natočení hráče k míči. Určení vlastnictví míče není jednoznačné, takže bylo nutné stanovit určité hranice:

$$possession(d_b,\alpha_b) = \begin{cases} 0 & \text{pokud } d_b \geq 7{,}5cm \text{ nebo } \alpha_b \geq 45^\circ \\ 1 & \text{jinak} \end{cases} \tag{4.6}$$

Hodnota 45° značí, že robot má míč pouze, pokud je před přední stranou robota, tedy před stranou, kterou se pohybuje dopředu. Méně jednoznačné je to se vzdáleností míče. Mohlo by se zdát logické použít součet poloviny rozměru robota a poloviny rozměru míče $7{,}5\ cm/2 + 4{,}27\ cm/2 = 5{,}885\ cm$, neboli situace, kdy míč přiléhá na robota. V situaci, kdy míč je spíše směrem k rohu robota, je míč zákonitě dál. Maximální vzdálenost přiléhajícího míče u rohu robota je tedy $7{,}5\ cm/2 \cdot \sqrt{2} + 4{,}27\ cm/2 \doteq 7{,}44\ cm$. Zaokrouhlením na polovinu centimetru jsem získal hodnotu $7{,}5\ cm$. To sice znamená, že vlastněný míč nebude vždy přiléhat, avšak maximální mezera mezi míčem a hráčem přibližně $1{,}615\ cm$ je zanedbatelná, především vzhledem k tomu, že robot se směrem k míči pohybuje (podmínka natočení), takže míč v každém případě dožene.

4.2.4 Viditelnost míče

K určení viditelnosti míče jsem se rozhodl využít diskrétní pole hřiště a jejich informaci o obsazenosti. Na takto rozděleném poli lze přímku viditelnosti aproximovat posloupností polí a každé z nich zkontrolovat, zda není obsazené a tudíž brání ve výhledu. Pro aproximaci přímky viditelnosti jsem použil Bresenhamův algoritmus, používaný v počítačové grafice. Příklad ukazuje obrázek 4.3.



Obrázek 4.3: Příklad užití Bresenhamova algoritmu

Základním kamenem algoritmu je rozdíl souřadnic na obou osách d_x, d_y a pak jejich poměr. Pro osu, která má větší rozdíl se krok o jedna provede vždy, pro druhou osu pak pouze někdy a to v závislosti na poměru rozdílů. Poměr rozdílů menšího ku většímu se postupně sčítá jako chyba a pokaždé, když překročí hodnotu 0,5, udělá se krok i na druhé ose a chyba se sníží o jedna. Pseudokód algoritmu pro obrázek 4.3 ukazuje kód 4.1.

Tento kód je jednoduchý a platí pouze pro směry odpovídající obrázku. Jenže existuje celkem 8 směrů, které se musí řešit (poloviny všech čtyř kvadrantů s počátkem v prvním bodě přímky). Dále je potom kód optimalizovaný, aby nepoužíval desetinná čísla ani operaci dělení, čímž se výsledně urychlí výpočetní operace. Takový optimalizovaný kód, nejen pro přímky, lze nalézt například na [17]. Tato implementace využívá tuto optimalizovanou podobu, protože pokud lze ušetřit výpočetní čas bez obětování přesnosti, je chybou tak neudělat.

```
1  bresenham(A, B){
2    int dx = |A.x − B.x|;
3    int dy = |A.y − B.y|;
4    float slope = dy / dx;
5    Point temp = A;
6    float err = 0;
7    while (temp != B) {
8      draw(temp);
9      err = err + slope;
10     if (err >= 0,5) {
11       temp = new Point(temp.x + 1, temp.y + 1);
12       err = err − 1;
13     } else {
14       temp = new Point(temp.x + 1, temp.y);
15     }
16   }
17   draw(B);
18 }
```

Kód 4.1: Pseudokód pro Bresenhamům algoritmus na obrázku 4.3

4.2.5 Vzdálenosti k brankám

Vzdálenosti k oběma brankám jsou obdobou vzdálenosti k míči, takže i vzorec je téměř stejný:

$$d_{g0} = \sqrt{(x - x_{g0})^2 + (y - y_{g0})^2} \quad (4.7a)$$

$$d_{g1} = \sqrt{(x - x_{g1})^2 + (y - y_{g1})^2} \quad (4.7b)$$

[x_{g0},y_{g0}] jsou souřadnice středu domácí branky a [x_{g1},y_{g1}] obdobně souřadnice branky soupeře. Obě souřadnice jsou tedy jedním z bodů [0,90] a [220,90]. Která branka je která záleží na umístění týmů a toto umístění bude třeba nastavit při spuštění modulu. V současné době program předpokládá umístění domácí branky na [0,90], avšak není problém pozice prohodit.

Branky jsou prozatím reprezentovány středy z důvodů zjednodušení výpočtu. Vybrat vhodný bod pro počítání vzdálenosti totiž není tak jednoduché, jako vybrat nejbližší bod branky. Ve většině případů bude tento nejbližší bod roh branky. Roh branky ale už není vhodný bod jako cíl střelby míče, protože ten se zákonitě o roh odrazí. Vybrat bod blízký rohu také není ideální pozice, protože malá nepřesnost ve střele na tento bod může poslat míč

mimo branku. Protože tedy pozice, které bude chtít hráč dosáhnout budou spíše u středu branky, vybral jsem tento střed jako vhodný referenční bod pro počítání statistik vůči brance.

Dalším důvodem použití středů branek je ten, že nepřesnosti tohoto výpočtu se projeví až pro pozice hráčů blízko branky. Pokud by ze dvou hráčů funkce vzdálenosti vybrala jednoho z nich jako bližšího na základě vzdálenosti ke středu branky a druhého na základě vzdálenosti k nejbližšímu bodu branky, jsou oba hráči pravděpodobně v dobré útočné pozici. Pokud by se míra útočného potenciálu stanovovala pouze na vzálenosti k brance, výběr jednoho každého z nich by nebyl chybou.

4.2.6 Natočení k soupeřově bráně

Také výpočet natočení k bráně soupeře je variantou vzorce natočení k míči:

$$\alpha_{g1} = |\phi_{g1} - \phi| \tag{4.8}$$

Proměnná ϕ_{g1} je úhel směru k soupeřově brance, tedy k bodu $[x_{g1},y_{g1}]$, jakožto středu oné branky. Statistika natočení k brance se ukázala jako potřebná pro rozhodnutí situace, kdy dva hráči aspirují na roli útočníka, protože oba mají určeno vlastnictvní míče jako pravdivé. Oproti tomu pro statistiku natočení vůči domovské brance jsem nedokázal zjistit vhodné použití, a proto se tato statistika nepočítá.

4.2.7 Viditelnost spoluhráčů

Viditelnost spoluhráčů je statistika v současné době redundantní, nicméně v budoucích vylepšeních bude pravděpodobně užitečná. Má představa je, že až se bude řešit přihrávání spoluhráčům, bude viditelnost cílového spoluhráče jednou z podmínek provedení či neprovedení přihrávky.

Implementačně se zde opět používá Bresenhamův algoritmus mezi pozicemi hráčů a každé pole přímky, které je obsazené (okrajová pole obsazená testovanými dvěma hráči se nepočítají), způsobí nastavení viditelnosti jako

negativní. Každému hráči se spočítá viditelnost vůči všem ostatním spoluhráčům.

4.2.8 Inverzní útočné hodnocení

Inverzní útočné hodnocení (dále jen **IAR** z anglického *Inverse attack rating*) je statistika vypočítávaná z ostatních statistik hodnocení hráčů a slouží k lepšímu určení útočníka. Před implementací IAR bylo složité určit útočníka, když žádný hráč nevlastnil míč. Pro určení útočníka je pak třeba vzít v úvahu jeho vzdálenost od míče, viditelnost míče a natočení k míči. Bylo třeba uvažovat situace popsané v teoretické části. Například v situaci, kdy nejbližší hráč míč nevidí, je vhodné zvolit jako útočníka druhého hráče, který je jen o málo vzdálenější. Ovšem pokud je druhý hráč příliš vzdálen, je vhodné poslat k míči přeci jen prvního hráče.

Statistika má v názvu slovo inverzní, protože u tohoto hodnocení platí, že čím menší, tím lepší. IAR totiž vychází ze vzdálenosti, která je dále penalizována za nepříznivé hodnoty ostatních dvou statistik. První penalizace p_1 je aplikována, pokud hráč míč nevidí. Druhá penalizace p_2 je aplikována, pokud míč není před hráčem, tedy když natočení hráče k míči je vyšší než 45°. Zatímco p_1 se aplikuje vždy celá, p_2 vychází z nebinárního hodnocení, a proto se aplikuje v intevalu $p_2 \in (1, p_{2max}\rangle$.

Po testování výsledného programu jsem dospěl k hodnotám $p_1 = 1{,}75$ a $p_{2max} = 1{,}75$. To znamená, že hráč, který nevidí míč, je na tom stejně jako hráč, který je natočen plně směrem od míče, pokud jsou stejně vzdáleni od míče. Oba totiž musí provést velkou korekci cesty, aby se k míči dostali. Jeden hráč musí objet překážku a druhý se musí obloukem otočit. Zároveň platí, že každý hráč, který je méně než přibližně 3× vzdálenější než bližší hráč, je potenciálním kandidátem na útočníka. Tyto hodnoty bude možná ještě třeba upravit, ale na základě testování jsem si jist, že obě budou ležet v rozmezí 1,5 a 2. Při hodnotách 2 se někdy vybírali útočníci podle mého názoru příliš vzdálení od míče. Naopak pro penalizace 1,5 byl blízký, nevýhodně postavený hráč málo penalizován a někdy tak byl ignorován očividně výhodnější hráč. Pseudokód výpočetu IAR ukazuje Kód 4.2.

```
1  IAR(vzdálenost, vidí_míč, úhel){ //p1 a p2max jsou konstanty
2    float IAR = vzdálenost;
3    if (!vidí_míč) {
4      IAR = IAR * p1;
5    }
6    if (úhel > 45) {
7      float poměr = úhel / 180;
8      float p2 = 1 + (p2max - 1) * poměr;
9      IAR = IAR * p2;
10   }
11   return IAR;
12 }
```

Kód 4.2: Pseudokód pro výpočet inverzního útočného hodnocení

4.2.9 Časová náročnost

Časová náročnost diskretizace a následného výpočtu statistik byla na mém počítači v rozmezí 30 μs až 45 μs (0,03 ms až 0,045 ms). Použité výpočty jsou jednoduché vzorce a jediný složitý algoritmus, Bresenhamův algoritmus, je optimalizován pro rychlost výpočtu. Čas strávený diskretizací a počítáním statistik je vzhledem k 20 ms jedné iterace řídícího programu prakticky zanedbatelný. Je zde proto velký prostor pro implementaci dalších statistik či použití složitějších konstrukcí na výpočet stávajících.

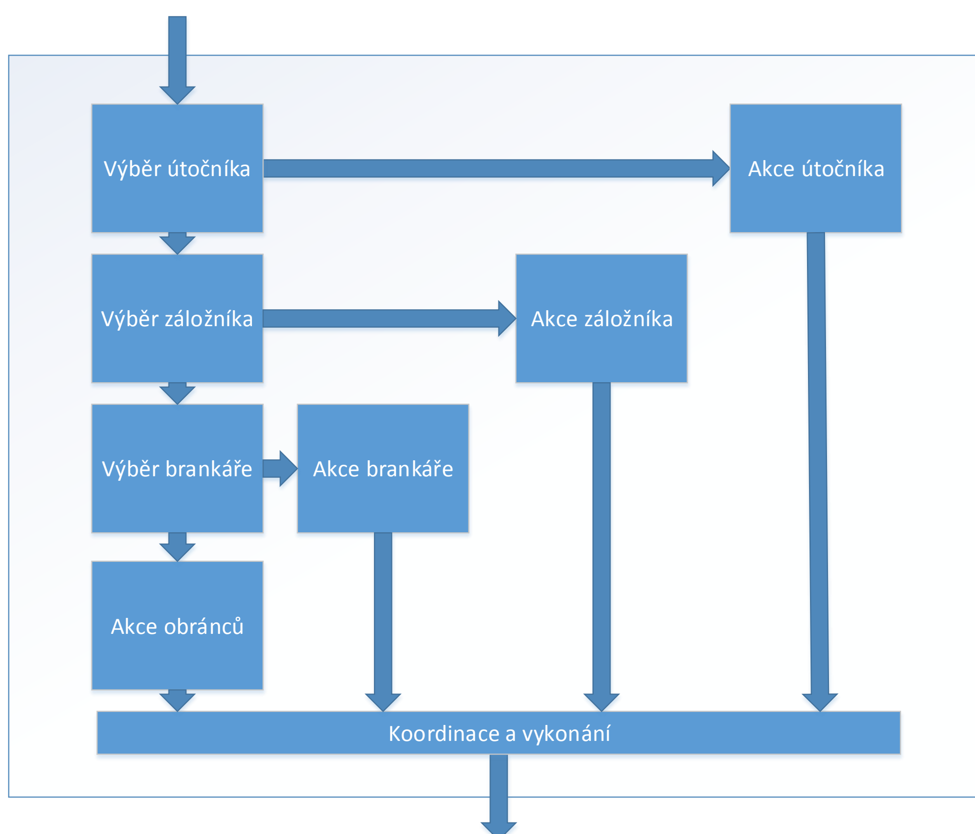
4.3 Výběr rolí a akcí

Pro výběr rolí a akcí jednotlivých hráčů jsem zvolil rozhodovací stromy. Důvodem byla jejich spolehlivost a jednoduchá implementace, tudíž jsem se mohl plně soustředit na problematiku a optimalizaci vážení grafu a jeho prohledávání. Tato část tedy slouží jako základní prototyp, který cílí na logický výběr útočníka. Ostatní role jsou zde zastoupeny symbolicky, protože veškeré algoritmy jsou testovány na roli útočníka. Tento fakt a také to, že útočník je definován jako hráč zaměřený na míč, dělají z útočníka nejdůležitější roli.

Původním návrhem bylo, že pro každého robota bude existovat identický strom, který mu určí nejvhodnější roli. Tyto role se na konci rozsoudí, protože může nastat situace, kdy více robotů bude aspirovat na určitou roli. Vzhledem k důležitosti role útočníka, jsem se princip rozhodovacích stromů rozhodl

obrátit. Místo toho, aby se hráčům přiřazovaly role, se nyní rolím přiřazují hráči. V případě iterativního přiřazování rolí podle jejich důležitosti se pak eliminuje nutnost řešit konflikty přidělených rolí. Důležitost rolí jsem stanovil takto:

1. Útočník
2. Záložník a brankář
3. Obránci



Obrázek 4.4: Schéma výběru rolí a akcí

O důležitosti role útočníka jsem již psal. Záložníkovi a brankáři přiřazuji stejnou důležitost, protože jsou to role s jasně vymezenými cíli - podpora útočníka a stráž domácí branky. Většinou by tyto role neměly při výběru

hráčů kolidovat, protože jde o role zajímající se o opačné části hřiště. Brankář je čistě defenzivní (více než obránce), zatímco záložník je čistě útočná role. Paradoxně může být záložník více útočný než útočník. Útočník má za cíl získat míč a zavést ho vpřed, zatímco záložník má za cíl čekat v útočné pozici na odražený míč či přihrávku.

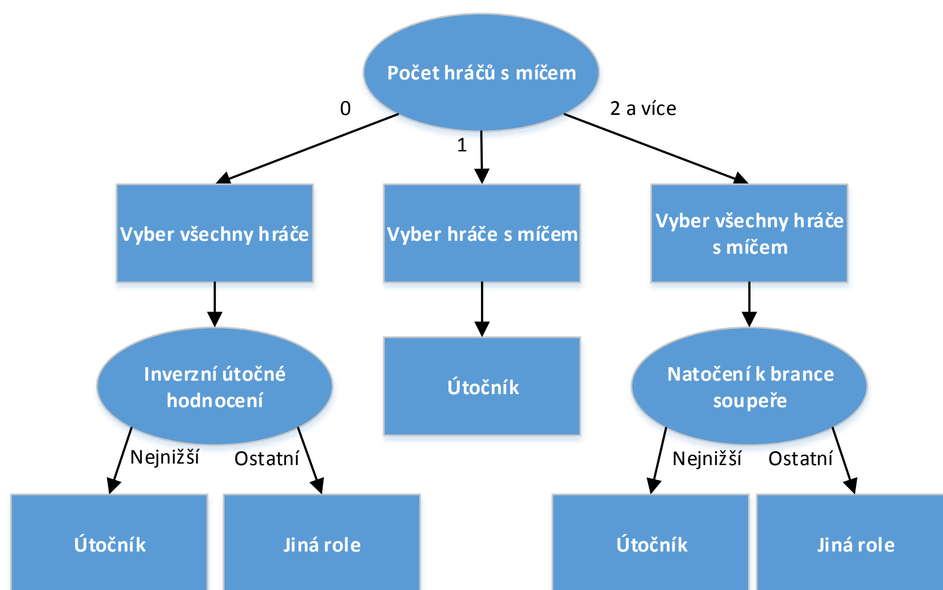
Nakonec jsou tu obránci, jejichž role je v rámci dynamického přiřazování rolí hůře definovatelná. Ve standardním fotbale jde o hráče, který má zbavit soupeře míče. Zde ale pokud robot má soupeře míče zbavit, stává se útočníkem. Obránce by tedy měl zaujímat určitou defenzivní pozici, z níž se snadno stane útočníkem, pokud má soupeř poblíž míč.

Co se týče vlastních akcí, akce jsou přiřazeny v podobě jednoho z řetězců datového typu výčtu (*enum*). Tyto řetězce značí abstraktní podstatu akce jako například *shoot* či *defendGoal*. Abstraktní akce jsou převedeny do podoby konkrétních příkazů až po přiřazení všech akcí. Důvodem je, že to umožňuje výslednou koordinaci akcí s cílem předejít konfliktům v příkazech či pozicích na hřišti. Koordinace akcí není součástí této práce, protože zde se konkrétní podoba akcí řeší pro roli útočníka, nicméně architektura je připravena pro budoucí rozšíření s touto funkcí. Schématické zobrazení výběru rolí a akcí tedy ukazuje obrázek 4.4.

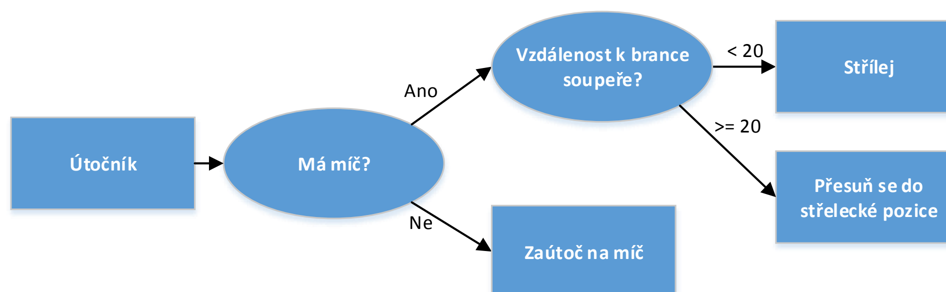
4.3.1 Útočník

Na obrázku 4.5 je zachycen rozhodovací strom pro výběr útočníka. Pokud má míč pouze jeden hráč, stává se útočníkem. Pokud má míč víc hráčů, stává se útočníkem ten, který je nejlépe natočen k brance soupeře. V případě, že ani jeden z hráčů nemá míč, útočníkem je pak ten, který má nejlepší statistiku inverzního útočného hodnocení.

Obrázek 4.6 znázorňuje výběr akcí pro útočníka. Pokud útočník nemá míč, snaží se jej získat. V opačném případě vyrazí hráč s míčem do střelecké pozice, případně vystřelí, když už je dostatečně blízko brance soupeře. Vzdálenost 20 (centimetrů) je zde pouze jako pracovní hodnota, která bude v budoucnu jistě upravena na základě schopností skutečných robotů. Stejně tak je i střelecká pozice určena prozatím staticky, aby byl k dispozici cílový bod pro přesun robota a tedy i pro prohledávání grafu. Budoucí implementace algoritmů do modulu herní strategie pravděpodobně časem bude generovat tento bod dynamicky nebo na základě funkčnosti modulu zvolí jiný bod. Protože útočníkem může být zvolen i hráč u vlastní branky, počítaná trasa může



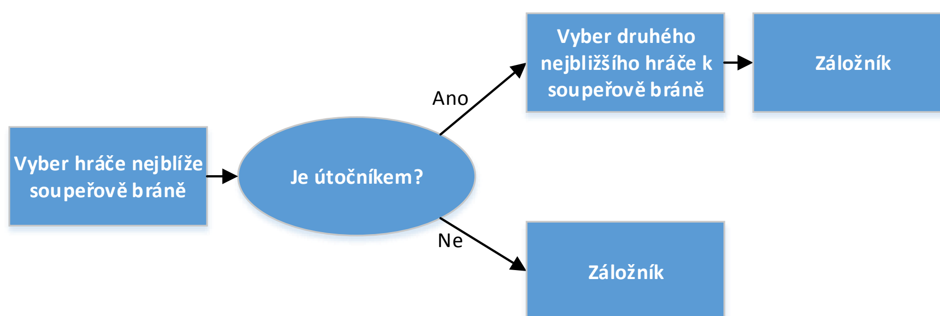
Obrázek 4.5: Schéma výběru útočníka



Obrázek 4.6: Schéma výběru akcí útočníka

být příliš dlouhá a tedy výpočetně příliš náročná. Proto pokud je hráč vzdálen více jak 150 *cm* (hodnota stanovená měřením výkonnosti prohledávání A*), přesune se na dočasný bod 100 *cm* před ním. Takto se přesunuje, dokud se nedostane pod 150 *cm* od branky, a až pak se přesune do skutečné střelecké pozice. Vzdálenost 100 *cm* před hráčem byla stanovena opět s ohledem na výpočetní rychlost. Objížďka by totiž musela být alespoň tvaru půlkružnice, aby měla větší délku než 150 *cm*, protože půlkružnice o průměru 100 *cm* má délku přibližně 157 *cm*.

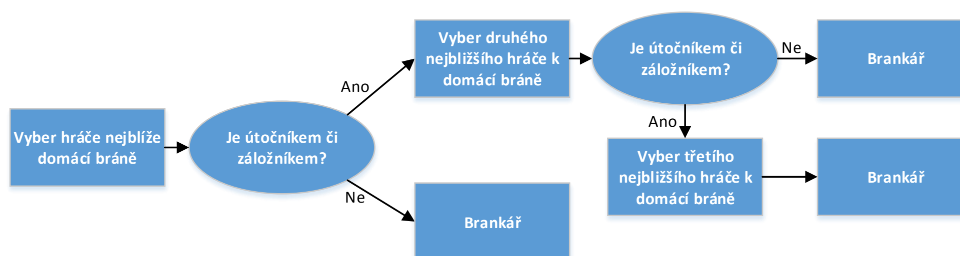
4.3.2 Záložník



Obrázek 4.7: Schéma výběru záložníka

Obrázek 4.7 ukazuje jednoduchý postup při výběru záložníka. V podstatě jde vždy o hráče nejblíže brance soupeře kromě situace, kdy je tento nejbližší hráč již útočníkem. V tom případě je záložníkem druhý nejbližší hráč. Co se týče akcí, je zatím záložníkovi vždy přiřazena abstraktní akce podpory útočníka, která značí přesun do pozice vhodné pro doražení odraženého míče nebo přijetí přihrávky.

4.3.3 Brankář



Obrázek 4.8: Schéma výběru brankáře

Jak ukazuje obrázek 4.8, brakář se vybírá podobně jako záložník. Zjednodušeně řečeno je brankář takový hráč, který je nejblíže domácí brance a není již útočníkem nebo záložníkem. Role brankáře vykonává zatím jedinou abstraktní akci a tou je bránění domácí branky.

4.3.4 Obránci

Pro roli obránců v podstatě neexistuje rozhodovací strom, protože tato role je přiřazena posledním dvěma hráčům bez role. Stejně jako u záložníka a brankáře vykonávají obránci jedinou abstraktní akci - bránit v pozici více vpředu než brankář. Ačkoliv jde o stejnou akci přiřazenou oběma obráncům, provedení každé z nich bude v budoucnu muset být logicky rozdílné v závislosti na pozicích hráčů.

4.3.5 Časová náročnost

Výběr rolí a akcí je časově velmi nenáročný, odpovídá tak představě o náročnosti rozhodovacích stromů. Implementačně jde pouze o soustavu podmínek a několik řazení seznamů o pěti objektech. Proto zde při výpočetním času kolem 2 μs (0,002 ms) neexistuje důvod pro pokusy o urychlení běhu této části programu. Naopak je zde velký prostor pro implementaci mnohem složitějších rozhodovacích stromů.

4.4 Vážení grafu

4.4.1 Pozice hráčů

Jak již bylo zmíněno, váhy v grafu musí zaprvé vycházet z pozice hráčů a pro tuto potřebu bude použita dvourozměrná gaussovská funkce:

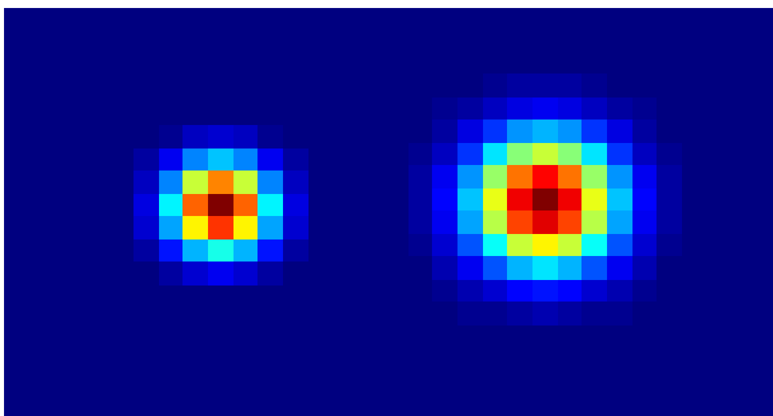
$$f(x,y) = Ae^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}} \tag{4.9}$$

Spojitost této funkce není problém, stačí pouze dosazovat souřadnice bodů, mezi kterými je třeba spočítat váhu. Větší obtíží je skutečnost, že funkce se nikdy nedotkne nuly, pouze se k ní asymptoticky blíží v obou nekonečnech. Je proto nutno omezit šířku funkce na takový rozměr, kdy většina relevantních hodnot zůstane zachována. Tuto skutečnost řeší statistické pravidlo tří sigma. To říká, že do vzdálenosti 3σ od středu gaussovské funkce leží 99,7% hodnot. Nejvzdálenější body, které budou ještě relevantní pro počítání vah, budou tedy ležet maximámlě 3σ daleko od bodu pozice robota.

Ovšem hodnota parametru σ je hůře představitelná než vlastní hodnota vzdálenosti, ve které se budou váhy počítat. Tato vzdálenost je v podstatě šířka a výška dvourozměrného gaussovského filtru, který přesně určuje, jakým polím bude v grafu přiřazena váha. Šířka filtru je tedy parametr, který snadno určuje, jak bude funkce ovlivňovat graf, a bylo by vhodné vypočítávat hodnotu σ z něj než naopak. Tento přepočet by měl ideálně aproximovat pravidlo tří sigma a pro účely této práce jsem převzal vzorec používaný pro tvorbu gaussovského filtru v knihovně *OpenCV* [18]:

$$\sigma = 0{,}3 \cdot (\frac{(N-1)}{2} - 1) + 0{,}8 \tag{4.10}$$

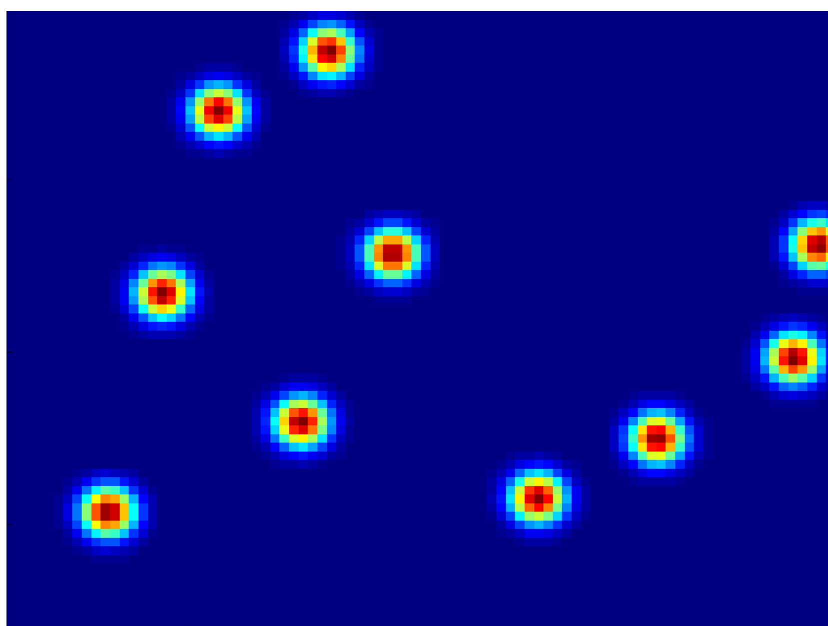
Zde N je zadaná velikost gaussovského filtru. Tato velikost byla původně nastavena na 7 (oproti šířce robotem obsazeného prostoru, který je široký 5 polí), nicméně při krajích filtru jsou hodnoty již velmi malé, takže jsem nakonec rozšířil velikost na 11. Jak ukazuje srovnání na obrázku 4.9, při velikosti filtru 11 a přibližném umístění hráče na střed prostředního pole vznikne ve filtru pole vysokých hodnot o rozměrech 3×3, které odpovídá rozměrům robota.



Obrázek 4.9: Srovnání gaussovských filtrů o šířce 7 a 11

Pozice hráče samozřejmě nebude vždy přesně odpovídat středu diskrétního pole a tím by mohly ve spočítaných vahách vzniknout drobné nepřesnosti, především by „kopce“ vytvořené váhami nebyly opticky vystředěné na pozici hráče. Řešením je ve vzorci 4.9 jako souřadnice středu $[x_0,y_0]$ použít neceločíselnou variantu diskrétních souřadnic (viz vzorec 4.2). Díky tomu bude funkce lépe respektovat pozici hráče, ačkoliv ve svém středu nemusí vždy dosáhnout přesně hodnoty A. Tento rozdíl je ale zanedbatelný.

Nakonec je ještě třeba vyřešit přiřazení více vah na jedno pole. Pokud jsou dva hráči blízko sebe, budou se ovlivněné oblasti v grafu překrývat. Jako řešení jsem se rozhodl v tomto případě vybrat maximální váhu z přiřazených, neboli nejvyšší možné ohrožení či znevýhodnění pole. Oproti možnosti sčítat váhy zde zůstanou zachovány tvary „kopců“ přiřazených vah. Naopak sčítání vah by mělo za výsledek mírně přesnější zhodnocení, protože pole omezeno dvěma hráči je o to více nevýhodné. Pro testování algoritmů jsem zvolil první možnost. Budoucí testování pak mohou prokázat, zda je druhá možnost nějak prospěšná.



Obrázek 4.10: Zobrazení vah pro pozice hráčů

Obrázek 4.10 ukazuje příklad takového přiřazení vah podle pozic hráčů. Jde o výstup ze skriptu pro matematické prostředí *Octave*. Modrá barva značí nulovou váhu a čím větší teplost barvy, tím větší váha. Z obrázku lze vidět, že váhy jsou přiřazeny podle pozic všech hráčů. To sice znamená, že počáteční bod prohledávání grafu bude začínat v centru oblasti vah, nicméně jsem nenašel rychlý způsob, jak toto obejít. Vytvářet síť vah pro každého z pěti robotů je časově příliš náročné, protože konstrukce jedné sítě potřebuje přibližně jednu milisekundu na mém počítači. Jiná možnost je navrhnout komplikovanou datovou strukturu, která by dokázala uchovat informace o více vahách a hráčích, kteří je způsobili. Avšak tato struktura by musela být náležitě zpracována i prohledávacím algoritmem, čímž by se prohledávání podstatně zkomplikovalo a stalo by se více závislým na konkrétní implemen-

taci datových struktur. Po testování se zdá, že váha v počátku prohledávání nemá velký vliv na podobu nalezené trasy. Trasa musí tak či tak opustit tuto oblast, a pokud je cena do všech stran stejná, nehraje vlastně žádnou roli. Jediným důsledkem tak je větší odchylka heuristiky od reálné trasy a tedy více prohledávaných možností.

4.4.2 Pohybové vektory soupeře

Pro váhování polí pod vektorem soupeřů jsem nakonec implementoval obě metody zmíněné v teoretické části. Pás kolem vektoru o konstatní šířce (stejné jako rozměr gaussovského filtru) i pás s rostoucí šířkou. Obě vycházejí ze stejného principu - podél vektoru pohybu se na něj po krocích aplikuje kolmo jednorozměrný gaussovský filtr a při těchto krocích se všechny hodnoty budou lineárně změnšovat. Pohybový vektor robota je zpracováván v metrech za sekundu. Při této volbě a nastavení jiných parametrů poskytovaly algoritmy uspokojivé výsledky, proto nebyla jiná varianta zpracována.

```
1  belt(počátek, vektor){
2    směr = vektor.y / vektor.x;
3    seznam_bodů = kolmé_body(počátek, směr);
4    if (Abs(směr) >= 1) {
5      krok.y = Sign(vektor.y) * K1;
6      krok.x = krok.y / směr;
7    } else {
8      krok.x = Sign(vektor.x) * K1;
9      krok.y = krok.x * směr;
10   }
11   for (int i = 0; i * krok.velikost < vektor.velikost; i++) {
12     posunutý_počátek = počátek + i * krok;
13     foreach (bod in seznam_bodů) {
14       posunutý_bod = bod + i * krok;
15       zaokrouhlený_bod = Round(posunutý_bod);
16       g = gauss(zaokrouhlený_bod, posunutý_střed);
17       vzdálenost = 1 - i * krok.velikost / vektor.velikost;
18       zaokrouhlený_bod.váha = g * vzdálenost;
19     }
20   }
21 }
```

Kód 4.3: Pseudokód pro výpočet vah podél vektoru

```
1  kolmé_body(počátek, směr) {
2    kolmý_směr = −1 / směr;
3    seznam_bodů = nový_seznam(K2); //délka závisí na K2
4    if (Abs(kolmý_směr) >= 1 {
5      kolmý_krok.y = K2;
6      kolmý_krok.x = kolmý_krok.y / kolmý_směr;
7    } else {
8      kolmý_krok.x = K2;
9      kolmý_krok.y = kolmý_krok.x ∗ kolmý_směr;
10   }
11   N = seznam_bodů.délka;
12   for (int i = −N/2; i <= N/2; i++) {
13     seznam_bodů[N/2 + i] = počátek + i ∗ kolmý_krok;
14   }
15 }
```

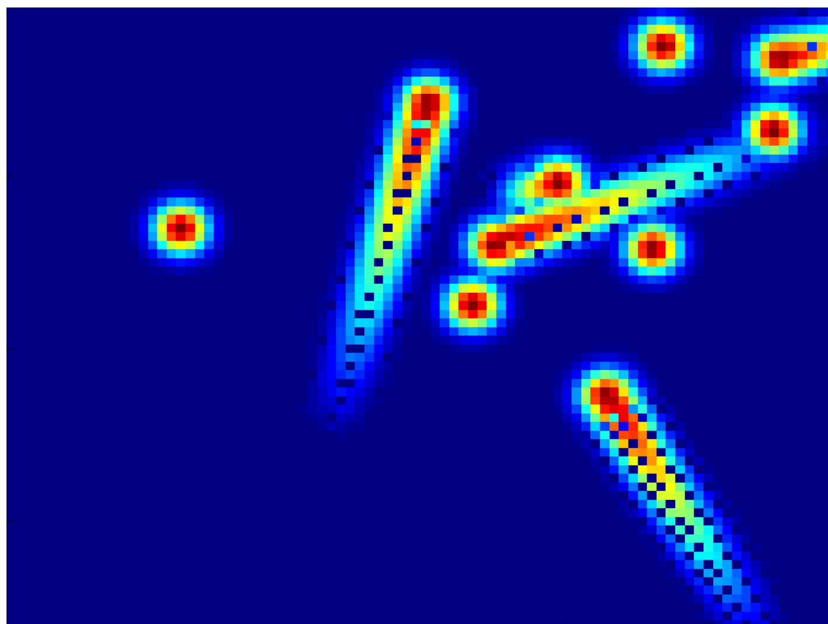
Kód 4.4: Pseudokód pro výpočet bodů v kolmém směru

Kód 4.3 je obecný pseudokód algoritmu pro výpočet vah v pásu kolem vektoru o konstantní šířce. Změna pro rostoucí šířku by znamenala generovat z metody *kolmé_body* více bodů a ve vnitřním cyklu hlavní metody vybírat ze seznamu bodů nejprve méně bodů kolem středu až postupně nakonec všechny. Tuto variantu zde nebudu kódem uvádět, protože zmíněný pseudokód je zde pro demonstraci principu algoritmu. V tomto algoritmu jsou dvě neznámé konstanty $K1$ a $K2$. Ty je potřeba správně nastavit, aby v pásu kolem vektoru byly nastaveny váhy všem bodům.

Algoritmus metody *kolmé_body* v kódu 4.4 generuje seznam bodů ležících na přímce, která prochází zadaným počátkem a je kolmá na zadaný směr. Tyto body jsou od sebe daleko vzdáleností, určenou konstantou $K2$. Tato konstanta zde funguje jako velikost kroku v té souřadnici, která je větší. Pokud je tedy velikost proměnné *kolmý_směr* větší než jedna, je ve vektoru tohoto směru souřadnice y větší než x. Proto se po souřadnici y bude algoritmus pohybovat kroky o velikosti $K2$, zatímco krok po x se vypočítá tak, aby body stále ležely na jedné přímce. Logickou prvotní volbou pro $K2$ je hodnota 1, kdy velikost seznamu kolmých bodů odpovídá přesně velikosti gaussovkého filtru.

Algoritmus hlavní metody *belt* je obdobný. Opět se zde kontroluje velikost směru a podle něj se nastaví krok ve větší souřadnici na konstantu $K1$ se znaménkem odpovídajícím znaménku stejné souřadnice vektoru pohybu. V okamžiku, kdy je známý vektor kroku, se ke každému z kolmých bodů přičte tento krok. Jeden z těchto bodů je střed posunutý o krok a tento bod se stává středem pro výpočet gaussovské funkce tak, jak je, zatímco vlastní body

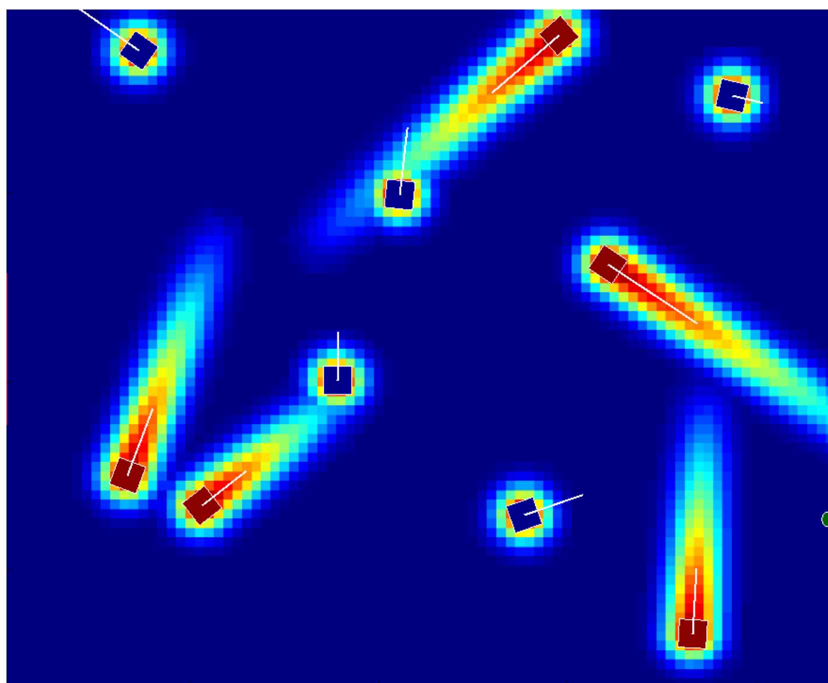
se předávají v zaokrouhlené podobě. Tyto zaokrouhlené body odpovídají souřadnicím v síti polí. Střed gaussovské funkce se používá nezaokrouhlený ze stejného důvodu jako u generování vah pro pozice hráčů, tedy pro zvýšení přesnosti. Původně byla hodnota $K1$ nastavena také na 1, kvůli zachování stejné velikosti kroku jako při generování kolmých bodů.



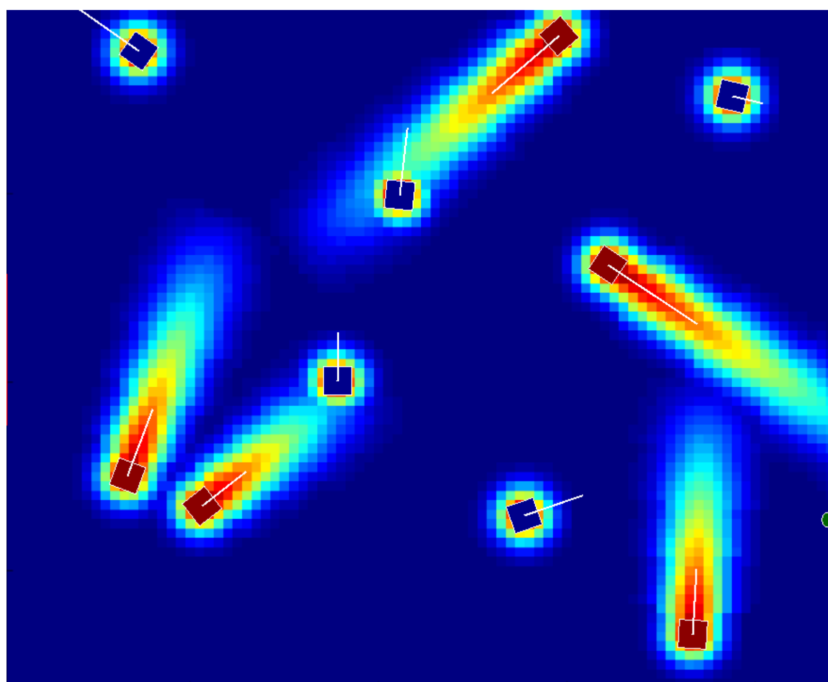
Obrázek 4.11: Chybějící váhy při $K1 = K2 = 1$

Bohužel, volby $K1$ a $K2$ jako 1 nebyly vhodné. Jak ukazuje obrázek 4.11, některé váhy zůstávají nepřiřazeny. Je to důsledkem toho, že krok 1 je příliš velký a přeskakuje některé důležité kolmé přímky. Představit si to lze na vektoru s úhlem přesně 45°, procházejícím středy polí. Kolmé přímky pak budou generovány v každém poli pod vektorem, ale každá druhá diagonální přímka v poli bude vynechána, protože přímky nestačí generovat ve středu polí, ale taky na jejich rozích, kde se pole potkávají. Z tohoto vyplývá, že vhodná hodnota $K1$ i $K2$ bude 0,5. Pro některá pole sice bude váha spočtena vícekrát, ale protože se přiřazuje maximální váha, nevnáší tyto výpočty do spočtených vah žádnou chybu. Příkladem takto správně přiřazených vah se zobrazením pozic a vektorů hráčů je obrázek 4.12. Nejsou zde žádné chybějící části a ani naopak žádné anomálie, které by mohly být způsobeny redundantními výpočty vah.

Na obrázku 4.12 se může zdát, že pás kolem vah se zužuje, ovšem ve skutečnosti je stále stejně široký, jen hodnoty po krajích klesají do méně



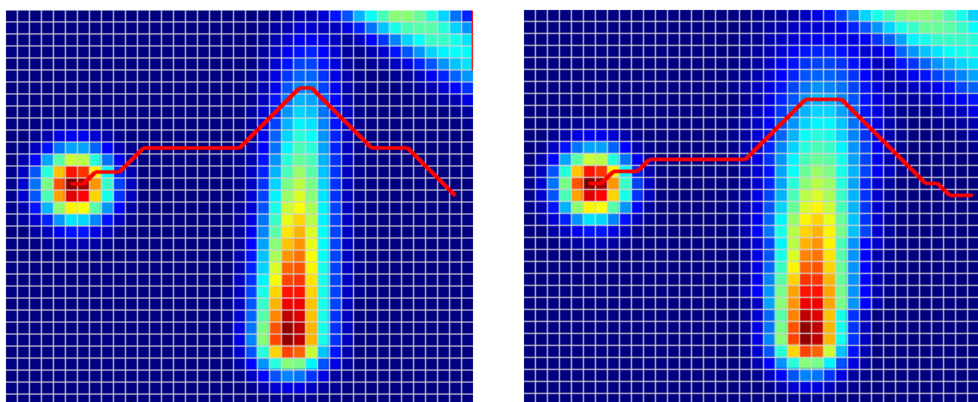
Obrázek 4.12: Příklad dobře přiřazených vah



Obrázek 4.13: Příklad vah s rozšiřujícím se pásem

viditelných hodnot. Příkladem vah v pásu, který se postupně rozšiřuje, je obrázek 4.13, kdy šířka filtru na konci vektoru hráče je dvojnásobná oproti začátku. Obrázek 4.12 i obrázek 4.13 vycházejí ze stejných dat, liší se pouze typem pásu vah kolem vektoru pohybu. Velikost vah v centru pásu, tedy přímo pod vektorem, zůstává stejná pro obě varianty. Ačkoliv je pravděpodobné, že soupeř v nějakém okamžiku zatočí, rovný směr je ale stále nejpravděpodobnější.

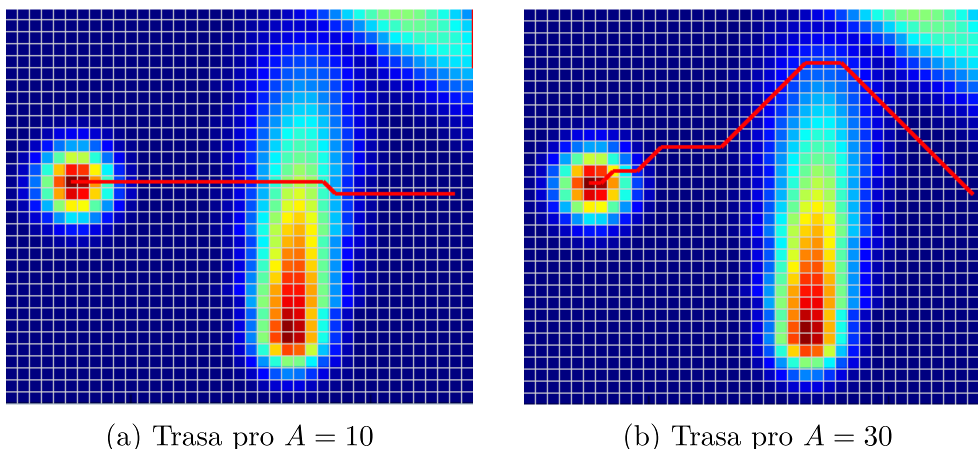
Volba, zda použít rozšiřující se pás nebo ne, závisí na výpočetní náročnosti a dopadech na vytvořenou trasu. Naměřené časy zpracování jsou zhodnoceny na konci této části, zbývá tedy prozkoumat dopad na strategii. Širší pás hodnot logicky znamená pro robota daleko složitější překonat tuto oblast. Jakákoliv trasa přes vektor soupeře bude totiž více penalizována. Obrázek 4.14 ukazuje srovnání tras nalezených algoritmem A* pro pás o konstatní šířce a pro pás o šířce rostoucí až na dvojnásobek. Trasa na druhém obrázku vypadá plynuleji, ale při bližším zkoumání lze vidět, že nejvyšší bod trasy je o jednu řadu níž. Robot tedy sice okraje pásu vah více objede, ale v centru se více odváží zkrátit si část cesty. Obrázek ukazuje, že není jednoznačné řešení pro výběr šířky pásu. Obě řešení jsou tedy v programu dostupná změnou jediné proměnné. Jako výchozí hodnotu jsem se rozhodl ponechat použití pásu se šířkou rostoucí až na dvojnásobek, tedy typ zobrazený obrázkem 4.14b. Rozšiřující se pás totiž více reflektuje možnost, že soupeř může zatočit, a tak trasa tomuto faktu více odpovídá. Rychleji se rozšiřující pásy nebyly testovány, ale jsou možností.



(a) Pás o konstantní šířce (b) Pás o rostoucí šířce

Obrázek 4.14: Srovnání důsledků šířky pásma

Zbývá ještě určit jeden parametr. A tím je konstanta A ve vzorci Gaussovy funkce. Ta určuje, jak moc zvětšeny budou všechny hodnoty vah a tedy



(a) Trasa pro $A = 10$ (b) Trasa pro $A = 30$

Obrázek 4.15: Srovnání důsledků velikosti parametru A

také maximální váhu, která se bude v centru vážené oblasti nacházet. Váhy musí být dost vysoké na to, aby hráč prakticky za všech okolností ignoroval pole s vysokou vahou pod pozicemi hráčů. Hráč také musí vzít dostatečně v úvahu váhy pod vektory pohybu soupeřů, avšak nesmí je kompletně objíždět. Varianty na obrázeku 4.14 pracovaly s hodnotou $A = 20$. Obrázek 4.15 ukazuje ostatní testované hodnoty, 10 a 30. Hodnoty vah promítnuté pod trasou mají stále stejnou intenzitu, protože skutečné hodnoty mají mezi sebou stále stejný poměr, takže výsledek jejich škálování je stejný. Při hodnotě $A = 10$ trasa očividně naprosto ignoruje veškeré váhy. Respektive jakákoliv odbočka by trasu penalizovala více než vlastní váhy. Naopak $A = 20$ není tak definitivně špatně. Trasa neobjíždí úplně všechny váhy, nicméně pravděpodobně je volena příliš opatrně. Protože se mi nepodařilo najít objektivní měřítko hodnocení nalezené trasy, zůstal jsem u hodnoty 20, ačkoliv vhodné hodnoty budou nejspíše ležet v rozmezí 20 až 30.

4.4.3 Časová náročnost

Časová náročnost počítání vah bude oproti předchozím částem programu složitější, protože zde probíhají výpočetně náročnější operace na desítkách až stovkách bodů. Zároveň ovšem hraje výpočet vah důležitou roli pro strategickou přesnost generování trasy v dalším kroku. Mezi přesností a složitostí je třeba najít vhodný kompromis. Protože parametr A neovlivňuje složitost výpočtů, lze jeho hodnotu libovolně nastavovat. Oproti tomu parametr velikosti gaussovského filtru a volba typu pásu vah kolem pohybového vektoru

podstatně ovlivňují složitost výpočtů. Naměřené složitosti se budou také částečně lišit podle velikosti vektorů a proto musí měření proběhnout několikrát. Měření zachycuje tabulka 4.1.

	Velikost filtru					
	Stálý pás			Rostoucí pás		
	9	11	13	9	11	13
Stroj 1 [ms]	0,550	0,657	0,797	0,821	1,012	1,213
Stroj 2 [ms]	0,449	0,535	0,659	0,674	0,839	1,008

Tabulka 4.1: Srovnání časové náročnosti generování vah

První měření bylo provedeno na původním stroji, kde se program implementoval. Druhé měření proběhlo na jiném, rychlejším stroji. Konfigurace obou strojů se nachází v příloze. Zatímco rozdíl mezi parametry typu šířky pásů je jasný z předchozích odstavců, parametr velikosti filtru byl měřen s cílem zjistit případné zrychlení či zpomalení při změně. Rostoucí šířka pásu vah je podle dat vždy přibližně o 50% náročnější, nicméně pravědodobně o trochu přesnější. Při zvolené šířce filtru 11 je pak časová náročnost prakticky jedna milisekunda, což je dobrá výchozí hodnota, jež se dá podle potřeby snížit.

Naopak posouvat hodnotou velikosti filtru se zdá být nevýhodné. Pokud by bylo třeba snížit složitost výpočtu vah, je vhodnější přejít na pás o stálé šířce, než snižovat velikost filtru. Zvětšování velikosti filtru pak zase přináší příliš velké zpomalení. Zvětšení na 13 zvýší složitost o 25% a zlepšení trasy bude pravděpodobně málo znatelné. Pro skutečný dopad by bylo třeba zvětšit filtr ještě víc, ale protože počet bodů roste s druhou mocninou velikosti filtru, byla by složitost pravděpodbně příliš vysoká.

4.5 Prohledávání grafu

Jako algoritmus prohledávání grafu jsem zvolil algoritmus A*, který se ukázal jako nejvhodnější ze zkoumaných obecných grafových algoritmů. Zároveň jsem při zkoumání algoritmů používaných ve variantách robofotbalu narazil na využívání tohoto algoritmu ve váženém grafu [19]. A* bude jakožto prohledávací algoritmus časově nejnáročnejší částí běhu programu, především pak, když bude spouštěn pro všech pět hráčů. Proto bude muset být částí nejvíce optimalizovanou. Jednak z hlediska kódu a použití konstrukcí a jednak

podle použitých algoritmů pro výpočet vzdálenosti a heuristiky.

Síť vah jsem se rozhodl chápat jako výškovou mapu, kde váha každého pole značí jeho výšku, přičemž výška ostatních polí je 0. Příklad sítě zobrazené ve 3D byl již dříve uveden na obrázku 3.11. V tomto pojetí vážené sítě lze přímo spočítat vzdálenost dvou sousedních bodů $d(A,B)$ jako eukleidovskou vzdálenost v prostoru. Není tedy nutné vzdálenosti ještě násobit vahami, protože ty jsou již součástí výpočtu vzdálenosti. Výchozí heuristika pak bude rovněž počítána jako eukleidovská vzdálenost v prostoru mezi zkoumaným a cílovým bodem. Využití této vzdálenosti pro oba výpočty má tu nevýhodu, že jejich součástí je operace odmocniny, která je výpočetně náročná. Na druhou stranu výhodou je, že jsou takto automaticky splněny podmínky přípustnosti i monotónnosti heuristiky.

4.5.1 Optimalizace kódu

První implementace prohledávání A* využívala generický seznam *List* pro seznamy *otevřeno* a *uzavřeno*, protože v knihovnách jazyka C# neexistuje třída pro prioritní frontu. Zároveň jsem pro uložení informací o f-skóre, g-skóre a předchůdcích zvolil pole. Tato implementace byla velmi pomalá. Hodnoty prohledávání byly typicky několik milisekund, ve výjimečných případech pak přes 10 *ms*. Ovšem bylo očekávané, že implementace pomocí třídy *List* bude pomalá. Nejčastější operací s oběma seznamy je test přítomnosti prvku, tedy metoda *Contains()*. Složitost této metody je totiž *O(n)*, stejně jako pro metodu odebrání prvku *Remove()* [20].

Jako další vylepšení jsem proto zvolil použití třídy *HashSet*. *HashSet* představuje množinu, která neuchovává zadané pořadí prvků, ale interně prvkům přiřazuje indexy podle jejich hash funkce. Díky tomu je složitost metod *Contains()* i *Remove()* pouze *O(1)* [21]. Použití *HashSetu* způsobilo velké urychlení běhu prohledávání. Pro prohledávání na krátkou vzdálenost bylo urychlení dvojnásobné, zatímco pro delší vzdálenosti došlo k urychlení více než desetinásobnému.

Ve srovnání s prioritní frontou má *HashSet* pomalé vybírání nejmenšího prvku, které má složitost *O(n)*, zatímco pro prioritní fronty jde o *O(1)*. Nicméně typická fronta implementovaná jako jednosměrný seznam má složitost hledání opět *O(n)*. Operací hledání prvku je v algoritmu A* podstatně více než výběru nejmenšího prvku. Proto se zdála být implementace za použití *HashSetu* finální úpravou. Ovšem při hledání možností optimalizace prioritní

fronty v jazyce C# jsem narazil na implementaci vysokorychlostní prioritní fronty pro C# od uživatele s přezdívkou BlueRaja [22].

Tato implementace má složitost vybrání nejmenšího prvku tradičně *O(1)*. Přidání nového prvku má složitost *O(log n)* a vyhledání prvku překvapivě opět *O(1)*. Autor toho dosáhl tím, že vlastní prvky prioritní fronty nejsou řazené objekty, nýbrž tyto objekty musí být samy nejprve obaleny objektem zděděným od třídy *PriorityQueueNode*. *PriorityQueueNode* uchovává index umístění v prioritní frontě, díky čemuž se test přítomnosti prvku zredukuje na jednoduchý test, kdy se prvek *node* srovnává s prvkem na pozici ve frontě určené indexem v prvku *node*.

Protože autor poskytuje svojí implementaci vysokorychlostní prioritní fronty volně k dispozici, rozhodl jsem se ji otestovat. Seznam *otevřeno* jsem nahradil touto prioritní frontou, ovšem pro seznam *uzavřeno* nemělo důvod nahrazovat *HashSet*, protože jediné dvě činnosti seznamu *uzavřeno* jsou vkládání a vyhledávání. Ve většině případů se takto implementace prioritní fronty ukázala jako výhodnější. Pro střední až dlouhé trasy bylo zlepšení mezi 20% až 30%. Jen pro extrémně krátké trasy (do 20 *cm*) byla prioritní fronta o málo náročnější než *HashSet*. Nicméně výpočty na takhle malé vzdálenosti trvají stejně jen několik desetin milisekundy, takže tento rozdíl nehraje důležitou roli. Naopak zlepšení pro výpočetně náročnější vzdálenosti důležité je, a proto je použití prioritní fronty rozhodně vhodnější.

Použití této prioritní fronty vynucuje vytvoření třídy *Node* zděděné od třídy *PriorityQueueNode*. Tato třída obsahuje oproti rodiči proměnnou bodu (*Point*), který se testuje. F-skóre je pak uloženo jako priorita objektu, jež je využívána pro řazení. Protože je takto vynuceno použití objektů, rozhodl jsem se dále otestovat, zda nedojde k urychlení přenesením informace o g-skóre a předchůdcích do těchto objektů. Díky této změně a následně další malé optimalizaci testování prvků v seznamu *uzavřeno* skutečně došlo k dalšímu urychlení. Toto urychlení se většinou pohybuje v rozmezí 5% až 10%, v závislosti na délce trasy. Na další možnosti urychlení v rámci optimalizace kódu se mi nepodařilo přijít.

Tabulka 4.2 ukazuje srovnání naměřených časů pro rostoucí délku výsledné trasy a pro rostoucí míru optimalizace kódu. *P. fronta 2* značí poslední krok optimalizace, tedy objektovou úpravu použití prioritní fronty. Délka trasy byla zvolena jako objektivnější měřítko oproti vzdálenosti k cíli, protože délka trasy závisí nejen na vzdálenosti, ale také na rozmístění vah podél trasy. Extrémně dlouhé trasy (přes jeden a půl metru) jsou zde uvedeny

	Průměrná délka nalezené trasy [cm]						
	10	20	30	40	50	60	70
List [ms]	0,061	0,174	0,425	0,876	1,53	2,433	3,626
HashSet [ms]	0,046	0,1	0,184	0,288	0,424	0,572	0,762
P. fronta [ms]	0,058	0,101	0,175	0,266	0,369	0,487	0,612
P. fronta 2 [ms]	0,052	0,098	0,172	0,255	0,348	0,462	0,576
	85	105	125	145	165	190	210+
List [ms]	7,054	14,48	25,94	40,94	43,94	170,3	217,4
HashSet [ms]	1,22	1,917	2,892	4,439	4,799	11,82	14,91
P. fronta [ms]	0,917	1,352	1,901	2,676	3,001	6,514	8,398
P. fronta 2 [ms]	0,843	1,219	1,696	2,313	2,522	4,928	6,591

Tabulka 4.2: Náročnost implementací A* měřená na stroji 1

orientačně, protože ve skutečné hře by nastat neměly. Útočník se totiž vždy snaží k míči přiblížit a ostatní role se stanou útočníkem, pokud jsou míči blíže. Všechny časy byly měřeny pro základní variantu heuristiky a měření vzdálenosti, tedy eukleidovskou vzdálenost. Časy v této tabulce byly změřeny na stroji, kde byl program vyvíjen. Měření na alternativním (rychlejším) stroji jsou uvedena v tabulce 4.3.

	Průměrná délka nalezené trasy [cm]						
	10	20	30	40	50	60	70
List [ms]	0,027	0,079	0,213	0,439	0,783	1,379	2,182
HashSet [ms]	0,016	0,034	0,069	0,111	0,165	0,238	0,321
P. fronta [ms]	0,017	0,031	0,057	0,087	0,125	0,172	0,224
P. fronta 2 [ms]	0,018	0,033	0,059	0,087	0,120	0,162	0,206
	85	105	125	145	165	190	210+
List [ms]	3,839	9,822	14,10	25,26	35,29	104,6	197,3
HashSet [ms]	0,469	0,871	1,105	1,620	2,171	4,325	8,566
P. fronta [ms]	0,313	0,546	0,662	0,915	1,241	2,345	4,687
P. fronta 2 [ms]	0,279	0,451	0,551	0,760	0,956	1,617	3,390

Tabulka 4.3: Náročnost implementací A* měřená na stroji 2

Z naměřených hodnot lze vyčíst, že složitost s použitím třídy *List* roste podstatně rychleji než u ostatních implementací. Tabulka také jasně ukazuje, že vylepšení s použitím prioritní fronty jsou nejvýraznější u delších a složitějších tras, zatímco u těch krátkých jsou zanedbatelně pomalejší. Výsledná podoba kódu s využitím základní heuristiky a měření vzdálenosti by ve vět-

šině případů měla trasu jednoho hráče najít do jedné milisekundy, extrémní případy pak do dvou milisekund.

4.5.2 Optimalizace heuristiky

Heuristika jako odhad ceny zbývající části cesty představuje nejdůležitější část algoritmu A*. Heuristika je vždy volena podle typu problému a proto neexistuje univerzální řešení. Pokud splňuje podmínku přípustnosti, tedy když heuristika podhodnocuje cenu výsledné cesty, najde heuristika optimální řešení. Vhodné zároveň ale je, aby toto podhodnocení nebylo příliš velké. Čím více se heuristika blíží skutečné vzdálenosti, tím rychleji najde řešení. Vhodným výchozím řešením heuristiky je eukleidovská vzdálenost, která podmínku přípustnosti splňuje:

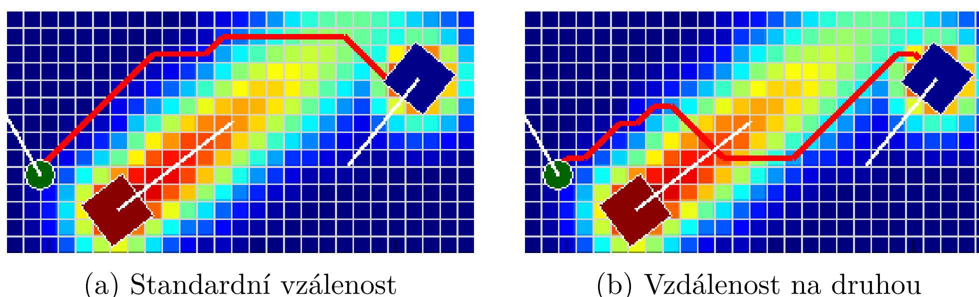
$$d(X) = \sqrt{(B.x - X.x)^2 + (B.y - X.y)^2 + (B.w - X.w)^2} \tag{4.11}$$

Proměnná B značí cílový bod a w je váha bodu. Nevýhodou je operace odmocniny, která je výpočetně podstatně náročnější než ostatní matematické operace. Při používání eukleidovské vzdálenosti a jejich srovnávání se někdy používá měření vzdáleností v druhé mocnině. Při výpočtu se tedy vynechá krok odmocnění. Protože při kladných vstupních číslech zachovává druhá mocnina vztah větší-menší, je srovnávání dvou vzdáleností v druhé mocnině validní možností. Pro použití v heuristikách je tento přístup nicméně nevhodný. Pokud by byla pouze heuristika v druhé mocnině, byla by nepřípustná, protože především na delší vzdálenosti by silně přeceňovala skutečnou trasu. Nalezená trasa by pak byla velmi neoptimální, jak ukazuje obrázek 4.16b. Tato heuristika není navíc ani monotónní, tudíž je možno znovuobjevit uzly ze seznamu *uzavřeno*. Změna kódu pro zohlednění tohoto faktu nicméně skoro vůbec nemění nic na neoptimálnosti výsledné trasy.

Možnost navrhovaná ve zdroji [23] je ve výpočtu f-skóre umocnit g-skóre na druhou, aby g-skóre $g(n)$ i heuristika druhé mocniny vzdálenosti $h^2(x)$ měly stejný rozměr:

$$f(x) = (g(x))^2 + h^2(x) \tag{4.12}$$

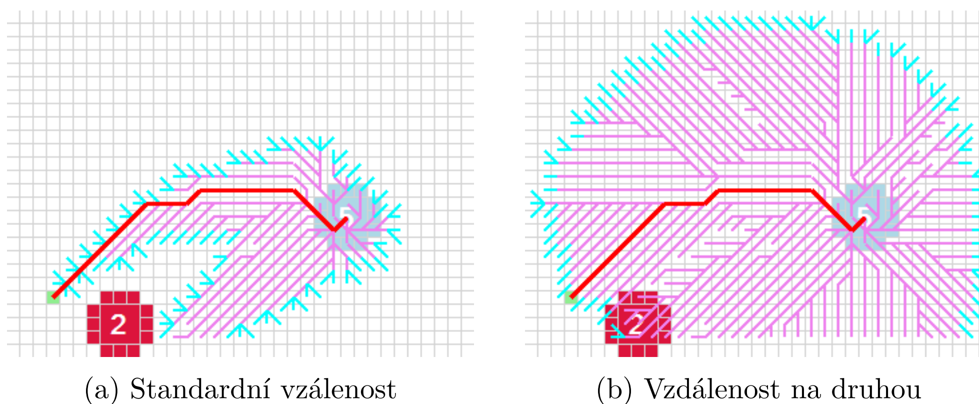
V této situaci je heuristika již konzistentní a najde optimální řešení. Jenže



(a) Standardní vzálenost (b) Vzdálenost na druhou

Obrázek 4.16: Důsledky použítí druhé mocniny vzdálenosti v heuristice

není monotónní a především jsou zde složky f-skóre nevyrovnané v různých částech trasy. Blízko počátku je složka g-skóre malá a složka standardní heuristiky velká. S druhou mocninou se tato velká hodnota heuristiky podstatně zvětší do té míry, že v počátku hledání trasy je složka g-skóre nepodstatná a prohledávání tak hledá stylem greedy best-first algoritmu. Naopak ke konci hledání je zase složka heuristiky čím dál zanedbatelnější a styl prohledávání tak sklouzne k Dijkstrově algoritmu. Prohledávání tak nakonec bude probíhat hlavně na základě uražené vzdálenosti a prozkoumá se mnohem víc možných tras. Jakákoliv výhoda ušetřená odstraněním operace odmocniny se ztratí nárůstem prohledávaných možností. Na základě testování jsem zjistil, že využití druhých mocnin v obou složkách f-skóre sice najde optimální řešení, ale 2× až 4× pomaleji. Rozdíl mezi prohledávanými trasami demonstruje obrázek 4.17.



(a) Standardní vzálenost (b) Vzdálenost na druhou

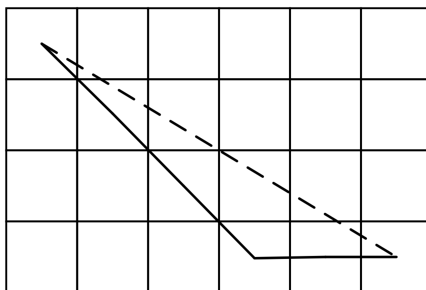
Obrázek 4.17: Použítí druhé mocniny vzdálenosti v heuristice a g-skóre

Poslední možná heuristika, již jsem testoval, byla heuristika postavená na diagonální vzdálenosti v mřížce [16; 23]. Tato vzdálenost je nejmenší možná vzdálenost, která se ve dvourozměrné mřížce dá urazit pomocí rovných a

diagonálních kroků. Rovné kroky mají vždy délku 1 a diagonální $\sqrt{2}$, nicméně tato odmocnina není počítána, ale je použita konstatní číslená hodnota jí přibližně odpovídající. Pseudokód této heuristiky zobrazuje kód 4.5 a příklad je uveden na obrázku 4.18.

```
1  h2(A, B){
2    dx = Abs(A.x - B.x);
3    dy = Abs(A.y - B.y);
4    diff = Abs(dx - dy);
5    max = Max(dx,dy);
6    return diff * 1,4142 + max - diff;
7  }
```

Kód 4.5: Pseudokód pro výpočet diagonální vzdálenosti



Obrázek 4.18: Diagonální vzdálenost a pro srování čárkovaně eukleidovská

Tato heuristika je extrémně výhodná na dvourozměrné mřížce, protože přesně odpovídá uražené vzdálenosti bez překážek. Tato vzdálenost je tedy vždy větší nebo rovna eukleidovské vzdálenosti, ale nikdy ne větší než skutečná. Proto A* pak prohledává méně potenciálních cest. Na vážené mřížce, respektive trojrozměrné mapě, není tato heuristika jednoznačně výhodnější. Pouze v situaci, kdy rozdíl sousedních vah by byl vždy 1 nebo 0 a váhy tak byly pouze celočíselné, by šel algoritmus rozšířit o další rozměr s diagonálním krokem v prostoru o délce $\sqrt{3}$. Bohužel takto moje vážená mapa nevypadá.

Zbývá tedy použít heuristiku v její dvourozměrné podobě, protože jsem nepřišel na žádnou možnost, jak zohlednit třetí rozměr a přitom zachovat heuristiku přípustnou. Vlastní výpočet této heuristiky je sice přibližně 2x rychlejší než výpočet eukleidovské vzdálenosti, avšak kvůli ignoraci třetího rozměru je ve výsledku často větším podhodnocením skutečné vzdálenosti než její eukleidovský protějšek. Výrazný rozdíl v podhodnocení pak vede k

většímu množství prohledaných cest, což někdy může i přes levnější výpočet způsobit dražší běh heuristiky. Prakticky jsem ovšem pozoroval zhoršení výpočtu jen u malého množství testovaných příkladů.

	Heuristika		Zlepšení
	Eukleidovská	Diagonální	
Stroj 1 [ms]	0,505	0,428	15,3%
Stroj 2 [ms]	0.196	0.171	12,8%

Tabulka 4.4: Srovnání průměrné časové náročnosti heuristik

Tabulka 4.4 srovnává časovou náročnost heuristiky využívající eukleidovskou vzdálenost a heuristiky využívající diagonální vzdálenost. Každé měření proběhlo zprůměrováním času běhu A* na 1000 náhodných konfiguracích hráčů a míče. Průměrný výsledek měření ukazuje, že diagonální vzdálenost je o přibližně 15% levnější na časovou náročnost, což je užitečné zrychlení. Důležité také je zjistit, pro jaké vzdálenosti dojde k jakým zrychlením. Tato měření zachycuje tabulka 4.5.

	Průměrná délka nalezené trasy [cm]						
	10	20	30	40	50	60	70
Eukleidovská [ms]	0,052	0,098	0,172	0,255	0,348	0,462	0,576
Diagonální [ms]	0,106	0,144	0,194	0,261	0,334	0,418	0,451
	85	105	125	145	165	190	210+
Eukleidovská [ms]	0,843	1,219	1,696	2,313	2,522	4,928	6,591
Diagonální [ms]	0,692	0,948	1,255	1,827	1,914	3,361	3,866

Tabulka 4.5: Srovnání časové náročnosti heuristik na stroji 1

Tabulka 4.5 používá stejné vstupní konfigurace jako tabulka 4.2 výše. Je zřejmé, že diagonální vzdálenost je pro heuristiku výhodnější zejména na středních a větších vzdálenostech. Naopak na kratších vzdálenostech je zpomalení až několikanásobné, za což pravděpodobně může fakt, že počáteční bod prohledávání začíná na „vrcholu“ v mapě vah, takže diagonální vzdálenost je mnohem větší podhodnocení reálné vzdálenosti než ta eukleidovská. Přesto je zpomalení na krátkých vzdálenostech zanedbatelné, protože zapomalení je v rámci setin či desetin milisekundy. Oproti tomu na delších vzdálenostech je zrychlení i přes půl milisekundy, což je již podstatná hodnota. Zrychlení na středních až delších trasách je dáno tím, že na nich je podstatně více plochých úseků o váze 0, kde diagonální vzdálenost získává na

výhodnosti, protože je menším podhodnocením. Obdobná měření provedená na alternativním stroji zachycuje tabulka 4.6.

	Průměrná délka nalezené trasy [cm]						
	10	20	30	40	50	60	70
Eukleidovská [ms]	0,018	0,033	0,059	0,087	0,12	0,162	0,206
Diagonální [ms]	0,039	0,053	0,069	0,09	0,113	0,135	0,168
	85	105	125	145	165	190	210+
Eukleidovská [ms]	0,279	0,451	0,551	0,76	0,956	1,617	3,39
Diagonální [ms]	0,216	0,351	0,391	0,639	0,619	1,318	2,983

Tabulka 4.6: Srovnání časové náročnosti heuristik na stroji 2

Podle výše uvedených měření jsem se rozhodl použít v kódu primárně heuristiku založenou na diagonální vzdálenosti, protože dosažené zrychlení je podstatným vylepšením prohledávacího algoritmu. I pro velmi špatné scénáře se dá očekávat doba výpočtu kolem jedné až dvou milisekund (v závislosti na stroji), přičemž počítání na extrémní vzdálenosti by nemělo v reálné hře nastat.

4.5.3 Vzdálenost sousedních bodů

První implementace počítání vzdálenosti pracovala stejně jako heuristika s eukleidovskou vzdáleností v trojrozměrném prostoru podle vzorce 4.11. Bylo již zmíněno, že operace odmocniny je oproti ostatním aritmetickým operacím pomalá. Pokusil jsem se nahradit odmocninu některou její aproximací. Ačkoliv jsem použil celá čísla (původní hodnota vynásobená stem a oříznutá na celé číslo), nedokázal jsem dosáhnout urychlení. Použití předpočítaných tabulek bylo stále o 5% pomalejší a implementace Newtonova algoritmu pro výpočet odmocniny dokonce o 15% horší [24]. Zdá se, že funkce odmocniny je hardwarově vysoce optimalizovaná a je velmi složité, možná dokonce nemožné, ji softwarově urychlit ani za cenu ztráty přesnosti.

Pro zlepšní přesnosti algoritmu jsem při počítání vzdáleností z počátku a do cíle použil spojitou (nezaokrouhlenou) verzi souřadnic. Vylepšení nalezené trasy by mělo být minimální, nicméně proběhne při prakticky stejné výpočetní složitosti. Posledním zlepšením výpočtu vzdálenosti bylo zohlednění natočení robota při zkoumání sousedů počátečního bodu. Kód 4.6 ukazuje tento výpočet.

```
1 vzdálenost = vzdálenost(A,B);
2 p3_natočení = (Abs(hráč.natočení − natočení(A,B)) / 180;
3 p3_vzdálenosti = Max(hráč.vektor.velikost / p3Scale, 1) − 1;
4 p3 = 1 + p3_natočení ∗ p3_vzdálenosti;
5 vzdálenost = p3 ∗ vzdálenost;
```

Kód 4.6: Pseudokód pro výpočet vzdálenosti s natočením

Natočení je bráno v potaz v míře, kterou určuje délka vektoru natočení. Pro robota s velkým pohybovým vektorem a tedy velkou rychlostí je složitější prudce změnit směr, tudíž se algoritmus snaží zachovat směr jeho pohybu penalizovaním ostatních směrů. A tato penalizace je tím větší, čím je rozdíl od směru vektoru větší, aby malé změny směru jízdy byly možné. Naopak pro pomalu či skoro vůbec se nepohybující hráče není problém se na místě otočit, proto to kód dovoluje. Vše toto je řízeno proměnnou *p3Scale*. Ta je nyní nastavena na 15 a je to číslo, kterým se dělí velikost vektoru, aby se získala složka pro výpočet penalizace vzdálenosti. Pokud je vektor nižší než *p3Scale*, je tato složka omezena na nulu, aby se zápornými hodnotami nedosáhlo nežádoucích chyb ve výpočtech. Při nule je pak celá penalizace 1, neboli není vzdálenost bodů natočením nijak penalizována.

4.5.4 Zhodnocení

Po všech optimalizacích kódu a heuristiky bylo dosaženo uspokojivých výsledků prohledávání grafu algoritmem A*. Tabulka 4.7 ukazuje výsledky s použitím konečné optimalizace kódu, heuristiky s diagonální vzdáleností a vzdáleností bodů zohledňující natočení v počátku. Jde o srovnání výsledků z tabulek 4.5 a 4.6. Podrobné specifikace obou strojů jsou v příloze, nicméně zde mohu podotknout, že oba stroje nejsou starší dvou let. Ve výkonnosti stroje 1 může hrát roli to, že jde o notebook s procesorem optimalizovaným pro spotřebu. Přibližně trojnásobné zrychlení mezi těmito stroji ukazuje, že výkonný procesor dokáže podstatně urychlit běh prohledávání.

Naměřené časy na stroji 2 jsou pro většinu vzdáleností pod jednu milisekundu. Jak jsem již psal dříve, extrémně dlouhé vzdálosti by ve výpočtu v podstatě nastat neměly, protože role hráčů jsou u tří rolí z pěti (mimo obránců) voleny na základě vzdáleností k objektům na hřišti a cíle trasy hráčů leží poblíž těchto objektů. Navíc na základě předchozích iterací řídícího softwaru by měl být hráč většinou blízko těchto objektů a cíle trasy by se s dalšími iteracemi měly na hřišti pohnout pouze o malou vzdálenost. Tudíž

	Průměrná délka nalezené trasy [cm]						
	10	20	30	40	50	60	70
Stroj 1 [ms]	0,106	0,144	0,194	0,261	0,334	0,418	0,451
Stroj 2 [ms]	0,039	0,053	0,069	0,09	0,113	0,135	0,168
	85	105	125	145	165	190	210+
Stroj 1 [ms]	0,692	0,948	1,255	1,827	1,914	3,361	3,866
Stroj 2 [ms]	0,216	0,351	0,391	0,639	0,619	1,318	2,983

Tabulka 4.7: Srovnání časové náročnosti A* na dvou strojích

považuji trasy o délce větší jak 150 cm za vzácné. Pokud se jako horní hranice výpočtu A* pro jednoho hráče stanoví přibližně 0,65 ms, bude výpočet pro všech pět hráčů trvat maximálně 3,25 ms. Trvání výpočtu vah bylo pro stroj 2 přibližně 0,85 ms. Protože předchozí kroky měly složitost zanedbatelnou, je odhadovaná maximální složitost jednoho kroku herní strategie s algoritmy uvedenými v této práci 4,1 ms. Z 20 ms dostupných pro tři moduly řídicího softwaru to není ani celá třetina tohoto času. Proto zde uvedené algoritmy splňují požadavek rychlosti výpočtu.

Obrázky ukazující příklady nalezených tras spolu s prohledávanými uzly se nachází v příloze.

4.6 Budoucí rozšíření

Další rozšíření této práce by mělo mít za cíl implementovat tyto algoritmy v modulu herní strategie. Nejprve bude třeba rozšířit výběr akcí a jejich převedení na příkazy pro ostatní role. Díky tomu bude možné změřit celkovou výkonnost algoritmů a ověřit, zda mé předpoklady o časové náročnosti byly správné.

Dále bude nutné implementovat rozhraní k sousedním modulům. Převést formát posílaný z modulu rozpoznávání na formát používaný v tomto modulu a nastavit případné proměnné (pozice týmů na hřišti). V závislosti na dohodě s autorem modulu rozpoznávání pak případně ještě naimplementovat výpočet pohybových vektorů. Na opačné straně modulu pak bude třeba data posílat v takovém formátu, který se alespoň trochu blíží požadavkům modulu elementární inteligence.

Po zakomponování modulu do řídícího softwaru bude možné ověřit strategický dopad algoritmů při skutečné či simulované hře. Tím se otevře možnost vypozorovat případné chyby v chování hráčů a podle toho upravovat parametry herní strategie.

Po dokončení modulu herní strategie a ověření jeho funkčnosti se bude nabízet mnoho způsobů rozšíření, především v závislosti na nedostatcích implementované strategie. Například by bylo vhodné otestovat algoritmy strojového učení pro výběr hráčů i rolí. Nebo otestovat jiné velikosti diskretizovaných polí a dopad na rychlost a spolehlivost strategie.

5 Implementace

Zde uváděné algoritmy byly implementovány a testovány v jazyce C#, který byl použit i pro ostatní části řídícího softwaru. Díky tomu bude snazší začlenit implementované algoritmy do modulu herní strategie a ten pak do celého řídícího softwaru. Kód byl postaven na *.NET Framework 4.5*, v době psaní práce poslední verzi platformy pro *.NET*. Díky verzi 4.5 dochází v kódu k urychlení, protože třídy vysokorychlostní prioritní fronty použité v algoritmu A* využívají optimalizační přístup označovaný jako *aggresive inlining*, který je dostupný pouze v této verzi. Implementace nevyužívá žádné externí knihovny kromě těch dostupných v *.NET Framework*. Třídy vysokorychlostní fronty jsou připojeny přímo ve formě zdrojových souborů díky autorově svolení k volnému využití jeho kódu. Zdrojové soubory této práce tvoří šest tříd:

Program.cs: Třída automaticky generovaná prostředním *Visual Studio*, jež je spuštěna jako první a která pouze obstarává spuštění grafického rozhraní v podobě třídy *GUI*.

GUI.cs: Tato třída je grafickým rozhraním implementovaného programu. Obsahuje metody pro ovládání algoritmů umělé inteligence a zobrazení jejich výsledků v testovací aplikaci. Třída *GUI* volá metody z třídy *Processor*.

Processor.cs: Třída *Processor* obsahuje metody pro veškeré algoritmy herní strategie implementované v této práci. Třída byla navržena tak, aby byla nezávislá na třídě grafického rozhraní. Díky tomu je možné tuto třídu snadno použít v jiné práci, například v budoucím kompletním modulu herní strategie. Třída je závislá pouze na zbylých třídách vysokorychlostní prioritní fronty.

IPriorityQueue.cs: Toto je interface pro metody prioritní fronty.

PriorityQueueNode.cs: Tato třída je třídou pro uzel v prioritní frontě. Od této třídy jsou zděděné třídy používané v algoritmu A*.

HeapPriorityQueue.cs: Hlavní třída prioritní fronty poskytující hlavní metody pro práci s prvky prioritní fronty. Pro řazení využívá princip datové struktury haldy.

Kromě tříd v jazyce C# byly také vytvořeny dva skripty v jazyce *Octave* pro zobrazení váženého herního pole. Skripty se jmenují *plotPointField.m* a *plotPointField2.m*.

5.1 GUI.cs

Soubor *GUI.cs* obsahuje tři třídy používáné pro zobrazení a ovládání algoritmů herní strategie. Třída *Player* je potomkem třídy *ProcessedPlayer*. *ProcessedPlayer* se nachází v souboru *Processor* a uchovává informace o hráčích podstatné pro algoritmy herní strategie. Potomek *Player* si navíc přidává informace nutné pro vykreslování hráčů v testovací aplikaci. Druhou vedlejší třídou je třída *Section*, která reprezentuje části herního pole obsazené hráči.

Hlavní třída v tomto souboru je *PlannerForm*, potomek třídy *Form* z knihoven jazyka C#. Zde se obstarává veškerá komunikace s uživatelem. Ať už nastavení parametrů či zobrazení spočítaných výsledků. V této třídě jde v kódu nastavit několik proměnných ovlivňujících zobrazený výsledek. Proměnná *vectorScale* určuje násobek skutečného pohybového vektoru, který je nakonec zobrazen. Pro lepší přehlednost je nyní tato proměnná nastavena na 0,25. Vektory se tedy zobrazují jako čtrtinové. Dalším nastavením jsou proměnné, jejichž název je zakončen slovy *Brush* či *Pen*. Ty nastavují veškeré barvy zobrazených čar a výplní, stejně tak, jako šířky čar. Poslední přenastavitelnou proměnnou je *playerNumberFont*, která určuje font, jímž se na schématu diskretizovaného hřiště vykreslují čísla hráčů. Ostatní proměnné této třídy jsou globální proměnné nastavované v průběhu výpočtů.

Co se týče podstatných metod, první z nich je *randomStart*. Tato metoda vytváří náhodné pozice a pohybové vektory pro všechny hráče a míč. Tato metoda je volána při spuštění aplikace a může být vyvolána i uživatelem pro nové náhodné rozmístění. Metoda *onPaint* vykreslí nediskretizované pozice hráčů a míče v popředí. Kreslí také jejich pohybové vektory. Metoda *addPlayers* přidá na hřiště nového hráče či míč. Pokud je hráčů již maximální počet nebo je zadaná pozice nevhodná, nepřidá se. V opačném případě se spočtou potřebné parametry a vytvoří nový objekt *Player*. Metoda může být volána automaticky při vytváření náhodných pozic nebo na příkaz uživatele, chce-li přidat hráče na konkrétní pozici. Metoda *RemovePlayers* odstraňuje určitého hráče, pokud byla lokace předaného parametru klepnutí myší uvnitř čtverce, který představuje hráče (nebo míč). Tato metoda je volána po klepnutí pravým tlačítkem myši.

Metoda *drawGameField* obstarává kreslení mřížky na herním poli a je volána z metody *onBackgroundPaint*, která se stará o kreslení na pozadí herního pole. Kreslí diskretizované pozice hráčů a míče, spojnice viditelnosti, nalezenou trasu a všechny prohledávané trasy. Zobrazení těchto informací je podmíněno checkboxy v grafickém rozhraní. Dále třída obsahuje metody pro obsluhu událostí klepnutí myší do herního pole (přidání/odebrání hráče) a obsluhu klepnutí na tlačítka pro vygenerování náhodných pozic (*btnRandomize_Click*), smazání všech hráčů (*btnDeleteAll_Click*), spuštění výpočtů (*btnDiscretize_Click*), uložení pozic hráčů do souboru (*btnSave_Click*) a načtení pozic hráčů ze souboru (*btnLoad_Click*).

5.2 Processor.cs

První vedlejší třídou v tomto souboru je třída *ProcessedPlayer*. Ta uchovává informace o hráči či míči. Tyto informace zahrnují jeho pozici, vektor, natočení, hodnocení, číslo hráče, tým a přiřazenou akci. Jen některé proměnné jsou označeny jako ukladatelné do souboru, ostatní se znova dopočítávají, aby bylo možno snadno provádět změny editací uloženého souboru. Další třídou je *Vector*, jenž pomáhá vrátit informace o natočení i vzdálenosti z výpočtu určitých metod. Třída *Statistics* uchovává informace o hodnocení hráče. Třída *GameFieldSection* reprezentuje disretizované pole hřiště, které má informace o přiřazené váze a o obsazení určitým hráčem a týmem. Výčet *Team* obsahuje proměnné reprezentující oba týmy, míč, pole bez týmu a pole sdílené oběma týmy. Druhý výčet *Action* obsahuje proměnné abstraktních akcí hráčů. Nakonec třída *Node* odděděná od *PriorityQueueNode* reprezentuje uzel použitý v prohledávacím algoritmu A*.

Hlavní třída se jmenuje *Processor*. Obsahuje globální proměnné a konstanty nutné pro běh výpočtů. Pak obsahuje velké množství paramterů ovlivňujících podobu těchto výpočtů. Proměnné *p1*, *p2Max* a *p3Scale* představují hodnoty penalizací zmíněných v této práci. *kernelSize* je velikost gaussovského filtru a *K1* a *K2* jsou proměnné ovlivňující hustotu výpočtu pásu vah kolem pohybového vektoru. Booleovská proměnná *wideVectorWeights* ovládá použití pásu vah s rostoucí šířkou a proměnná *wideVectorRatio* určuje násobek původní šířky pásu, na který rozšiřující se pás naroste. Proměnné *teamGoalCenter* a *opponentGoalCenter* určují pozice branek a je třeba je nastavit opačně, pokud tým útočí zprava doleva. Nakonec seznam *occupationOffsets* obsahuje rozdíly indexů vůči pozici hráče. Pole na těchto indexech jsou pak označená jako obsazená hráčem.

Metody, které jsou důležité pro běh výpočtu herní strategie, jsou následující a měly by být volány v daném pořadí. Právě tyto metody dohromady obstarají celý výpočet herní strategie:

setPlayers: Nastaví vlastní hráče, hráče soupeře a míč pro další výpočty herní strategie. Všichni hráči a míč musí být instancí třídy *ProcessedPlayer*.

discretize: Spustí diskretizaci pozic hráčů a míče. Hráčům určí obsazená pole.

calculateStats: Spočítá všechna hodnocení hráčů pro výběr rolí a akcí.

selectRoles: Vybere role na základě hodnocení.

selectAction: Určí abstraktní akce na základě role a hodnocení.

weightPointField: Spočítá váhy diskretizovaných polí na hřišti.

execute: Provede abstraktní akce přiřazené hráčům. Tato metoda spouští prohledávání algoritmem A*.

Metodu *printPointField* lze volitelně spustit po těchto metodách. Jejím úkolem je uložit váhy herních polí do souboru *pointField.txt*, aby je bylo možno vizualizovat ve skriptech jazyka *Octave*. Ve třídě jsou tři metody pro výpočet heuristiky. Metoda *h* je založena na eukleidovské vzdálenosti, metoda *h2* na diagonální vzdálenosti a metoda *hSquare* používá neodmocněnou vzdálenost. Podobným způsobem existují i tři metody pro výpočet vzdálenosti dvou sousedních bodů. Základní metoda *d* počítá eukleidovskou vzdálenost pomocí odmocniny, metoda *d2* se pokouší odmocniny aproximovat Newtonovou metodou a metoda *d3* zase aproximuje pomocí vyhledávacích tabulek.

Kromě různých metod pro výpočet vzdáleností, natočení a dalších skutečností jsou ve třídě přítomny metody pro algoritmus A* ve všech podobách, jak byly v této práci zmíněny. *findPathUseList* je první implementací se seznamem *List*, *findPathUseSet* seznam nahrazuje množinou *HashSet*, *findPathUsePQueue* nahrazuje množinu prioritní frontou a nakonec *findPathUsePQueueObjects* optimalizuje využití objektů s prioritní frontou. Této poslední metodě se předávají jako parametry navíc i funkce heuristiky a vzdálenosti, aby bylo možno srovnávat jejich výkonnost.

5.3 Skripty prostředí Octave

V práci jsem několikrát zmínil, že pole s přiřazenými váhami lze chápat jako výškovou mapu. Pro prostředí *Octave* [25] jsem implementoval dva skripty, jejichž výsledky byly již v této práci ukázány.

Skript *plotPointField* vykreslí váhy jako skutečnou 3D síť a uloží ji do souboru *hills.svg*. Mnohem užitečnější je druhý skript *plotPointField2*. Ten váhy zobrazí jako 2D mapu intenzit. Čím vyšší je váha, tím intenzivnější (teplejší) je barva. Nulová váha je pak modrá. Takto zobrazené váhy jsou uloženy do souboru *background.png*.

6 Závěr

Modul herní strategie je nejdůležitější částí řídícího softwaru robotického fotbalu. Určuje chování hráčů i týmu jako celku. Správně zvolená strategie znamená rozdíl mezi vítězstvím a porážkou, zatímco špatná strategie je vinou pouze tohoto modulu. Proto musí být algoritmy použité v herní strategii naprosto spolehlivé. I špatná strategie je lepší než žádná strategie, a proto musí být algoritmy připraveny na všechny možnosti. Dále nesmí spolehlivost ani robustnost implementace jít na úkor rychlosti výpočtů, která je omezena 20 ms pro tři moduly řídícího softwaru. Kvůli zachování časové rezervy by modul herní strategie neměl plně vyčerpat svou kvótu.

V rámci této práce byly připraveny algoritmy pro budoucí implementaci v herní strategii. Byla také vytvořena testovací aplikace pro vytváření různých situací na hřišti a zobrazení výsledků z algoritmů herní strategie. Kvůli cílení na optimalizaci algoritmů byla implementace omezena na výpočet strategie útočníka.

Pro volbu rolí a akcí hráčů byly použity rozhodovací stromy, především kvůli své jednoduchosti. V těchto stromech jsou testována určitá hodnocení hráčů, která berou v potaz jak pozici na hřišti, tak pozici míče i soupeře. Na hřišti byla vytvořena diskretizovaná pole o rozměrech 2,5 $cm \times$ 2,5 cm kterým byla přiřazena váha v závislosti na pozicích a pohybových vektorech hráčů. Tyto váhy značí nevýhodnost pole, ať už z důvodu obsazení hráčem nebo kvůli ohrožení pole soupeřem.

Váhovaná pole pak tvoří graf pro prohledávácí algoritmus A*. Ten byl optimalizován použitím vysokorychlostní prioritní fronty a heuristiky založené na diagonální vzdálenosti. Algoritmy i grafické rozhraní byly naimplementovány v jazyce C#, čímž je zaručena kompatibilita se zbytkem řídícího softwaru. Navržené algoritmy byly implementovány pomocí globálně nastavovaných proměnných a díky tomu lze výrazně měnit podobu herní strategie změnou několika čísel. Vzniklo tak dobré prostředí pro ladění výpočtů herní strategie.

Rychlost výpočtů byla měřena na dvou strojích, kdy ani jeden z nich není vyloženě drahý přístroj. Ačkoliv byly algoritmy implementovány, a tedy i měřeny, pouze pro útočníka, byla odhadnuta horní hranice doby výpočtu 4,1 ms na výkonějším z obou strojů. To není ani třetina z dostupných 20 ms pro tři moduly a je zde tedy i určitý prostor pro chybu.

Literatura

[1] LEPIČ, Jaromír. *Multiagentní systém pro plánování herní strategie robotického fotbalu.* Bakalářská práce, Západočeská univerzita v Plzni, 2011. Vedoucí práce Kamil Ekštein.

[2] Federation of International Robot-soccer Association. *Overview*, cit. 6.4.2014. Dostupné z `http://www.fira.net/contents/sub01/sub01_2.asp`.

[3] Federation of International Robot-soccer Association. *Robo Soccer*, cit. 6.4.2014. Dostupné z `http://www.fira.net/contents/sub03/sub03_1.asp`.

[4] Federation of International Robot-soccer Association. *FIRA Middle League MiroSot Game Rules*, cit. 6.4.2014. Dostupné z `http://www.fira.net/contents/data/MiroSot_Rules_Middle_League.doc`.

[5] ALTMAN, Petr. *Jádro řídicího systému a virtualizační modul pro robotický fotbal.* Bakalářská práce, Západočeská univerzita v Plzni, 2011. Vedoucí práce Kamil Ekštein.

[6] FRIČ, Vojtěch. *Robofotbal: modul počítačového vidění.* Bakalářská práce, Západočeská univerzita v Plzni, 2012. Vedoucí práce Kamil Ekštein.

[7] ECKSTEIN, Robert. *Pokročilé algoritmy počítačového vidění pro robotický fotbal.* Diplomová práce, Západočeská univerzita v Plzni, 2013. Vedoucí práce Kamil Ekštein.

[8] VALIŠ, Jakub. *Aplikace metod strojového učení v robotickém fotbalu.* Diplomová práce, Západočeská univerzita v Plzni, 2013. Vedoucí práce Kamil Ekštein.

[9] KAEHLER, Steven D. *Fuzzy Logic Tutorial*, cit. 9.4.2014. Dostupné z `http://www.seattlerobotics.org/encoder/mar98/fuz/flindex.html`.

[10] SCHETTER, Chris. *Hexagon grids: coordinate systems and distance calculations*, cit. 10.4.2014. Dostupné z http://keekerdc.com/2011/03/hexagon-grids-coordinate-systems-and-distance-calculations/.

[11] ROKACH, Lior. *Decision Trees*. Tel-Aviv University, cit. 13.4.2014. Dostupné z http://www.ise.bgu.ac.il/faculty/liorr/hbchap9.pdf.

[12] NG, Andrew. *Machine Learning*. Stanford University, cit. 13.4.2014. Dostupné z https://www.coursera.org/course/ml.

[13] BERTEIG, Ross. *Basic Concepts for Neural Networks*. Cheshire Engineering Corporation, cit. 13.4.2014. Dostupné z http://www.cheshireeng.com/Neuralyst/nnbg.htm.

[14] GeeksforGeek team. *Dynamic Programming | Set 23 (Bellman–Ford Algorithm)*, cit. 15.4.2014. Dostupné z http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/.

[15] PATEL, Amid. *Game Programming: Introduction to A**. Stanford University, cit. 15.4.2014. Dostupné z http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[16] PATEL, Amid. *Game Programming: Heuristics*. Stanford University, cit. 15.4.2014. Dostupné z http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[17] ZINGL, Alois. *The Beauty of Bresenham's Algorithm*, cit. 17.4.2014. Dostupné z http://members.chello.at/~easyfilter/bresenham.html.

[18] OpenCV development team. *OpenCV documentation - Image Filtering*, cit. 23.4.2014. Dostupné z http://docs.opencv.org/modules/imgproc/doc/filtering.html.

[19] KADEN, S., MELLMANN, H., SCHEUNeEMANN, M., a BURKHARD, H.-D. *Voronoi Based Strategic Positioning for Robot Soccer*. Humboldt-Universität zu Berlin, Německo, cit. 26.4.2014. Dostupné z http://csp2013.mimuw.edu.pl/proceedings/PDF/paper-23.pdf.

[20] Microsoft Corporation. *List<T> Class*, cit. 27.4.2014. Dostupné z http://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx.

[21] Microsoft Corporation. *HashSet<T> Class*, cit. 27.4.2014. Dostupné z http://msdn.microsoft.com/en-us/library/bb359438(v=vs.110).aspx.

[22] BlueRaja. *A Heap-Based C# Priority Queue Optimized for A* Pathfinding*, cit. 27.4.2014. Dostupné z http://www.blueraja.com/blog/356/a-heap-based-c-priority-queue-optimized-for-a-pathfinding.

[23] IMMS, Daniel. *A* pathfinding algorithm*, cit. 29.4.2014. Dostupné z http://www.growingwiththeweb.com/2012/06/a-pathfinding-algorithm.html.

[24] KURTIS, Ron. *Newton's Square Root Approximation*, cit. 1.5.2014. Dostupné z http://www.school-for-champions.com/algebra/square_root_approx.htm.

[25] EATON, John W. *GNU Octave*, cit. 6.5.2014. Dostupné z https://www.gnu.org/software/octave/.

A Uživatelská dokumentace

A.1 Soubory a překlad

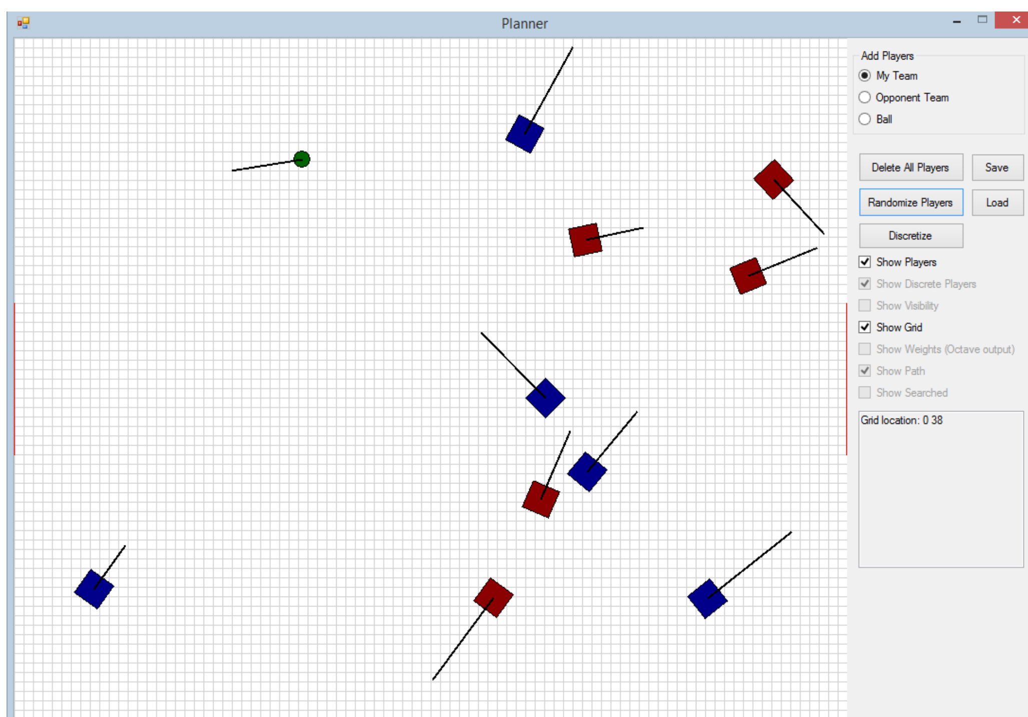
Na přiloženém CD jsou ve složce *src* v podsložce *Planner* uloženy zdrojové soubory testovací aplikace a skriptů prostředí *Octave*. Pro vlastní překlad je nutné importovat zdrojové soubory do prostředí *Visual Studio* (použita byla verze 2013). Pro takový import je nejsnazší použít soubor *Robosoccer Strategy Planner.sln* ve složce *src*, který uchovává informace o projektu. Dále složka *bin* obsahuje již přeložený program *Planner.exe* a oba skripty jazyka *Octave*.

A.2 Testovací aplikace

Pro spuštění testovací aplikace je nutné mít nainstalován *.NET Framework 4.5*. Pro spuštění scriptů jazyka *Octave* je třeba mít nainstalované jeho běhové prostředí. Spouštění těchto skriptů není nutné pro běh implementovaných algoritmů, nicméně pomáhají s vizualizací vah a výstup z nich lze zpětně zobrazit v aplikaci. Spustitelným souborem aplikace je *Planner.exe*. Grafické rozhraní nově spuštěného programu ukazuje obrázek A.1.

Většinu plochy okna programu zabírá znázornění hřiště s hráči a míčem. Modré čtverce jsou hráči vlastního týmu a červení jsou hráči týmu soupeře. Menší zelený kruh je míč. Černá čára, která směřuje ze středu hráče či míče značí pohybový vektor. Ten je vykreslen v délce odpovídající jeho čtvrtinové velikosti v metrech za sekundu. Vektory jsou zmenšeny z důvodu zlepšení přehlednosti, poměr rychlostí různých objektů je stále zachován. Veškeré míry zobrazených objektů jsou také zobrazeny v poměru ke skutečným velikostem. To se týká nejen poměrů stran hřiště, ale také velikosti hráčů, míče, branek a mřížky. Branky jsou znázorněny černě po stranách hřiště a mřížka na pozadí odpovídá diskretizační mřížce s poli o rozměrech $2{,}5\ cm \times 2{,}5\ cm$. Při pohybu myší po této mřížce se v textovém poli vpravo dole zobrazuje diskretizovaná pozice odpovídající indexu pole, v němž se myš nachází.

Pokud si uživatel přeje změnit pozice hráčů na hřišti, má několik možností. Tlačíkem **Randomize Players** vytvoří nové náhodné pozice hráčů na hřišti. Pro pouze smazání všech hráčů je zde tlačítko **Delete All Pla-**



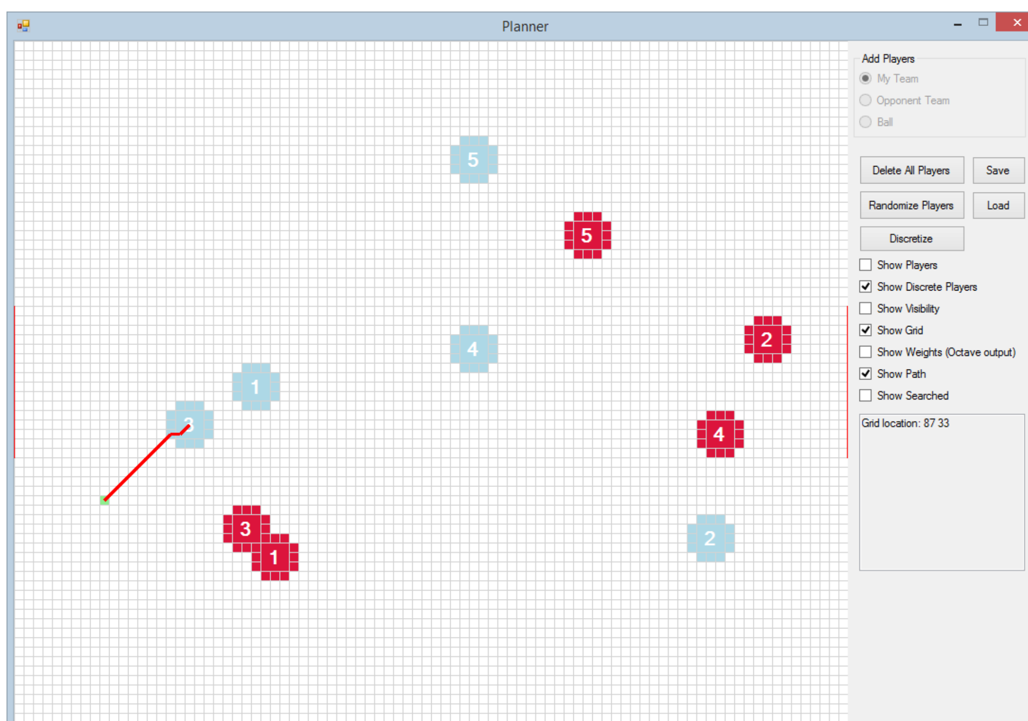
Obrázek A.1: Grafické rozhraní programu

yers. Případně pokud chce uživatel smazat jen některé z hráčů či míč, stačí na ně klepnout pravým tlačítkem myši. Před přidáním hráče či míče zpět na hřiště je třeba zvolit v kolonce *Add Players* jednu ze tří možností: **My Team** pro vlastní tým, **Opponent Team** pro tým soupeře nebo **Ball** pro míč. Při klepnutí levým tlačítkem myši na hřiště se potom objekt zvolené kategorie přidá na hřiště. Hráče či míč nelze přidat tak, aby se překrývali s jiným hráčem nebo okrajem hřiště. Kontrola překrývání bere v potaz nenatočené hráče, proto se natočení hráči mohou rohy drobně překrývat nebo mohou rohy zasahovat mimo hřiště.

Pouhé klepnutí levým tlačítkem přidá do pole hráče či míč nenatočené a s nulovým vektorem pohybu. Pro stanovené natočení a délky vektoru je třeba levé tlačítko stisknout a držet. Hráč se vykreslí a následným pohybem myši (při stálém držení tlačítka) se stanoví cílový bod vektoru. Hráč natočením a vektorem okamžitě reflektuje pohyb myši. Je třeba opět brát v potaz, že vektory jsou znázorněny čtvrtinově oproti skutečnosti, proto je v textovém poli vypisována přesná hodnota vektoru tak, jak s ním bude následně počítáno. Délka vektoru je uváděna v centimetrech, odpovídá tedy rychlosti v centimetrech za sekundu. Na hřiště nelze přidat více hráčů než je v týmech, tedy

pět, ani víc než jeden míč. Když bude naopak nějaký z objektů do správného počtu chybět, bude tlačítko pro diskretizaci neaktivní.

Pozice hráčů a míče lze uložit do souboru pomocí tlačítka **Save**. Uložený soubor se jmenuje *save.xml* a objeví se ve stejné složce jako spustitelný soubor. Do souboru se uloží pouze pozice, natočení, délka vektoru (skutečná) a příslušnost k týmu. Veškeré proměnné počítané v průběhu algoritmů se neukládají, stejně jako se neukládají proměnné objektů pomáhajících s vykreslováním. Důvodem je to, aby bylo možno soubor snadno editovat a znovupoužít pro nové výpočty. Načtení z tohoto souboru proběhne stisknutím tlačítka **Load**. Údaje uvedené v souboru jsou validovány stejným způsobem jako při ručním přidávání hráčů. V případě konfliktů (špatné pozice nebo počet hráčů v týmech) může dojít k situaci, kdy po načtení není na hřišti plný stav hráčů. Toto samozřejmě může nastat i v situaci, kdy jsou uloženy pozice hráčů při několika chybějících hráčích.



Obrázek A.2: Grafické rozhraní programu po provedení výpočtů

Stiknutím tlačítka **Discretize** se spustí diskretizace a veškeré další algoritmy vedoucí k výpočtu trasy pro útočníka. Stav programu po provedení výpočtů ukazuje obrázek A.2. V této chvíli již nelze měnit pozice hráčů na hřišti. Ovšem stále lze použít tlačítka *Delete All Players*, *Randomize Players*

a *Load*, která způsobí návrat stavu programu zpět před diskretizaci a dovolí tak měnit pozice. Po diskretizaci je stále možné ukládat pozice tlačítkem *Save*.

Po stisknutí tlačítka *Discretize* se změní zobrazené objekty na hřišti. Reálné pozice hráčů nahradí diskrétní pozice hráčů určená středem čtverce s číslem hráče. Zároveň je znázorněna celá oblast polí, kterou hráč obsazuje. Modrá barva opět značí vlastní tým, červená tým soupeře. Zelené pole je diskrétní pozice míče a pokud je nějaké pole obsazené oběma týmy najednou, je označeno fialově. Silná červená čára značí nalezenou trasu k míči či jinému určenému cílovému bodu. Při najetí myší na vlastního hráče (respektive čtverec s jeho číslem) se ukáží v textovém okně kromě pozice kurzoru v mřížce také hodnocení daného hráče:

Distance to ball: Vzdálenost k míči v centimetrech

Has ball: Značí, zda hráč má míč v držení

Can see ball: Značí, zda hráč vidí míč

Angle to ball: Natočení vzhledem k míči

Distance to goal: Vzdálenost k bráně soupeře

Angle to goal: Natočení vzhledem k bráně soupeře

Distance to home: Vzdálenost k vlastní bráně

Can see players: Čísla spoluhráčů, které hráč vidí

IAR: Spočtené inverzní útočné hodnocení

Pod tlačítky pro ovládání aplikace a nad textovým polem se dále nachází sada přepínačů, která ovlivňuje zobrazení informací na nákresu hřiště. Většina z nich je použitelná pouze po dokončení výpočtů:

Show Players: Ovládá zobrazování reálných pozic hráčů tak, jak jsou dostupné v módu přidávání a odebírání hráčů. Reálné pozice jsou vždy v popředí a zakrývají tak většinu informací prezentovaných po dokončení výpočtů. Z tohoto důvodu se tento přepínač vždy po provedení výpočtů vypne, avšak vždy je možné ho ručně zapnout. Užitečné pokud uživatel chce srovnat nalezenou trasu s vektory protihráčů.

Show Discrete Players: Ovládá zobrazení diskretizovaných pozic hráčů a jimi obsazených polí. Vhodné vypnout například pokud uživatel zapne reálné pozice hráčů, aby dobře viděl dopad pozice a vektorů na spočítané váhy. V základu jsou disretizované pozice zobrazovány.

Show Visibility: Zobrazí spojnice hráčů vlastního týmu mezi sebou a mezi míčem. Tyto spojnice značí viditelnost objektu na druhé straně. Pokud je spojnice přerušena jiným objektem, je cílový objekt neviděn a hodnocení hráčů by to měla reflektovat. Slouží především k ověření funkčnosti zjišťování viditelnosti, a proto je v základu tato možnost vypnuta.

Show Grid: Ovládá zobrazení mřížky diskrétních polí. Užitečné především při zobrazování vah polí, kdy někteří uživatelé mohou považovat mřížku za rušivou. V základu je mřížka zobrazována.

Show Weights: Zobrazí spočítané váhy na pozadí hřiště. Pro zobrazení potřebuje soubor *background.png*. Poznámka *Octave output* složí jako připomínka, že tato funkce je závislá na spuštění externího skriptu jazyka *Octave*, který zmíněný soubor vygeneruje. Bez spuštění skriptu *plotPointField2.m* mohou být zobrazeny špatné váhy z předchozích výpočtů nebo nebudou zobrazeny vůbec, pokud soubor *background.png* vůbec neexistuje. V základu váhy proto zobrazovány nejsou. Existence souboru *background.png* je kontrolována vždy při zapnutí přepínače, lze tedy soubor kdykoliv správně vygenerovat.

Show Path: Ovládá zobrazení nalezené trasy útočníka k cíli. V základu je trasa zobrazována.

Show Searched: Zobrazí všechny trasy, které algoritmus A* prohledával. Tato možnost byla použita v kapitole Realizace v obrázku 4.17. Fialově se zobrazují trasy přes body ze seznamu *uzavřeno*, tedy body, do nichž byla již spočítána nejvýhodnější trasa. Tyrkysově se značí části tras do bodů ze seznamu *otevřeno*. Tyto body ještě nemají stanovenou nejvýhodnější trasu kromě toho s nejnižším f-skóre. Pokud by se algoritmus A* nezastavil nalezením cíle, bod s tyrkysovou trasou s nejnižším f-skóre by byl vybrán jako další krok prohledávání. Tato možnost je v základu vypnutá, ale je velmi vhodná pro zjišťování optimalizace heuristiky, protože větší množství prohledávaných cest značí větší výpočetní náročnost.

Výše uvedené přepínače lze libovolně kombinovat. Vypnutím všech s výjimkou zobrazení vah se váhy polí zobrazí čistě bez jakékoliv překrývající

informace. Při zapnutí prvního i druhého přepínače lze kontrolovat správnost diskretizace a vhodnost volby obsazených polí. Možností je mnoho a cílem bylo vytvořit snadno ovladatelné prostředí pro ladění různých aspektů výpočtu herní strategie, především pak prohledávání grafu.

Po proběhnutí výpočtů program vygeneruje soubor *pointField.txt*. Tento soubor obsahuje uložené váhy polí hřiště a je potřebným souborem pro skripty jazyka *Octave*.

A.3 Skripty jazyka *Octave*

Poslední částí patřící k aplikaci jsou skripty v jazyce *Octave* pro zobrazení vah polí. Skripty byly naimplementovány ve verzi *Octave 3.6.4* určené pro operační systém *Windows*. V jiných verzích a pod jinými systémy nebyly skripty testovány. Součástí instalace *Octave* je rozhraní s příkazovou řádkou. Spuštěním programu (v mém případě jménem *Octave 3.6.4*) se spustí právě toto rozhraní. Ve verzi 3.6.4 se rozhraní spouští v neinteraktivní podobě, kdy jakákoliv chyba způsobí vypnutí programu. Proto je vhodné si spuštění rozhraní *Octave* v interaktivním režimu vynutit přepínačem **-i**.

Spuštění rozhraní *Octave* v interaktivním režimu lze dosáhnout například tím, že se na ploše vytvoří zástupce programu (klepnout pravým tlačítkem na soubor *Octave 3.6.4* > *Odeslat* > *Plocha (Vytvořit zástupce)*). Následně se ve vlatnostech (klepnout pravým tlačítkem na zástupce > *Vlastnosti*) v položce *cíl* přidá na konec cesty výše zmíněné „ -i". Po spuštění rozhraní s příkazovou rádkou je třeba změnit pracovní adresář na složku se soubory testovací aplikace pomocí následujícho příkazu:

cd "cesta k adresáři testovací aplikace"

Uvozovky kolem cesty nejsou nutné, pokud cesta neobsahuje žádné mezery. Narozdíl od příkazu *cd* v příkazové řádce operačního systému *Windows*, nepotřebuje *cd* v jazyce *Octave* přepínač „ /d" při změně adresáře mezi disky. Oveření správnosti změněného adresáře lze provést příkazem **pwd**, který vypíše současný pracovní adresář. Pokud jsou skripty *plotPointField.m* a *plotPointField2.m* uživatelem přesunuty mimo adresář testovací aplikace, je třeba změnit adresář příkazem *cd* do složky s těmito skripty. Nicméně není doporučeno mít tyto skripty mimo adresář s programem *Planner.exe*. V tom případě by totiž vždy po dokončení výpočtů testovací aplikace bylo nutné ko-

pírovat soubor *pointField.txt* do adresáře se skripty a naopak výstup skriptů *background.png* zpět k testovací aplikaci.

Je-li pracovní adresář správně nastaven na složku se skripty a souborem *pointField.txt*, lze spustit první skript příkazem **plotPointField**. Skript zobrazí váhy jako 3D síť (viz obrázek 3.11) a zároveň síť uloží do souboru *hills.svg*. *Svg* soubory lze otevřít například v prohlížeči *Internet Explorer* či ostatních moderních prohlížečích. Druhý skript se spouští příkazem **plotPointField2**, který zobrazí váhy jako plochu barevných intenzit, kde modrá značí nulu a červená nejvyšší možnou váhu. Zároveň se tento obrázek uloží do souboru *background.png*. Právě tento soubor je nutný pro správné zobrazní vah v testovací aplikaci přepínačem *Show Weights*.

Zatímco skript *plotPointField2* je nutný pro zobrazení vah v testovací aplikaci, první skript *plotPointField* pouze složí k zobrazení vah v 3D perspektivě. Celkově je zobrazení plochou intenzit vhodnější a srozumitelnější řešení.

B Konfigurace testovacích strojů

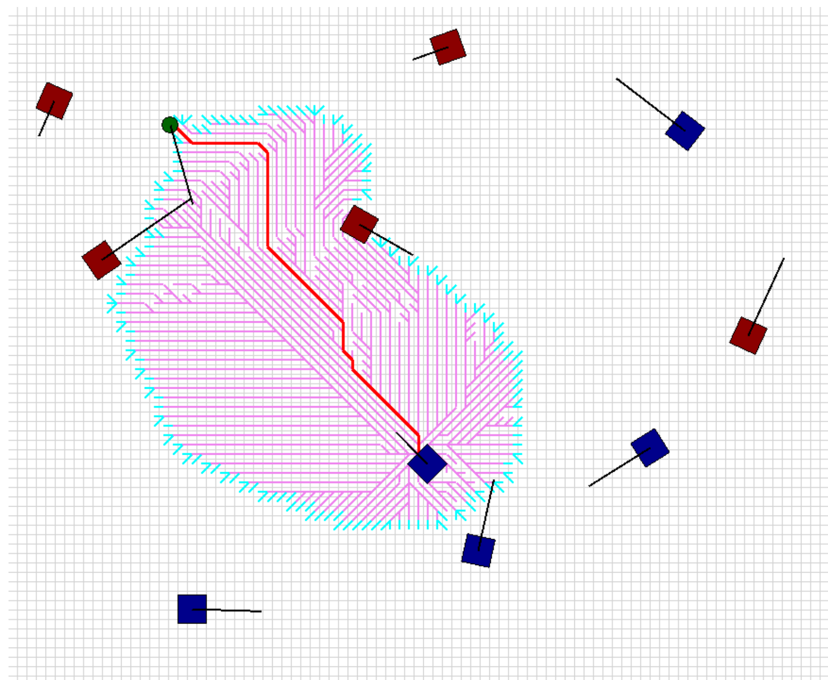
B.1 Stroj 1

Typ:	Notebook
Operační systém:	Windows 8.1 64-bit
Processor:	Intel Core i5-3210M, 2,50 GHz
Operační paměť:	8,00 GB RAM
Grafická karta:	Intel HD Graphics 4000 (integrovaná)
Grafická karta:	NVIDIA GeForce GTX 660M (dedikovaná)
Pevný disk:	Kingston 120 GB (SSD)
Pevný disk:	1 TB (mechanický)
Pořizovací cena:	22 000 Kč (červen 2013)

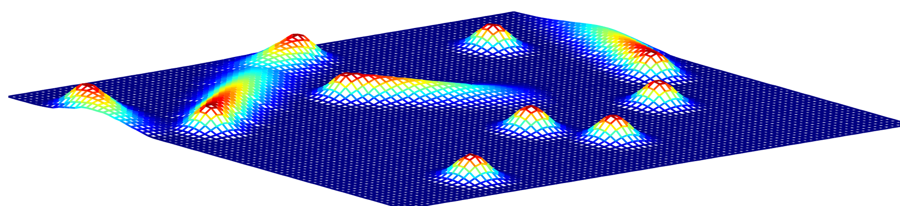
B.2 Stroj 2

Typ:	Stolní počítač
Operační systém:	Windows 7 SP1 64-bit
Processor:	Intel Core i5-2500K, 3,70 GHz (nepřetaktováno)
Operační paměť:	8,00 GB RAM
Grafická karta:	NVIDIA GeForce GTX 670 (dedikovaná)
Pevný disk:	Kingston 120 GB (SSD)
Pevný disk:	2,5 TB (mechanický)
Pořizovací cena	32 000 Kč (květen 2012)

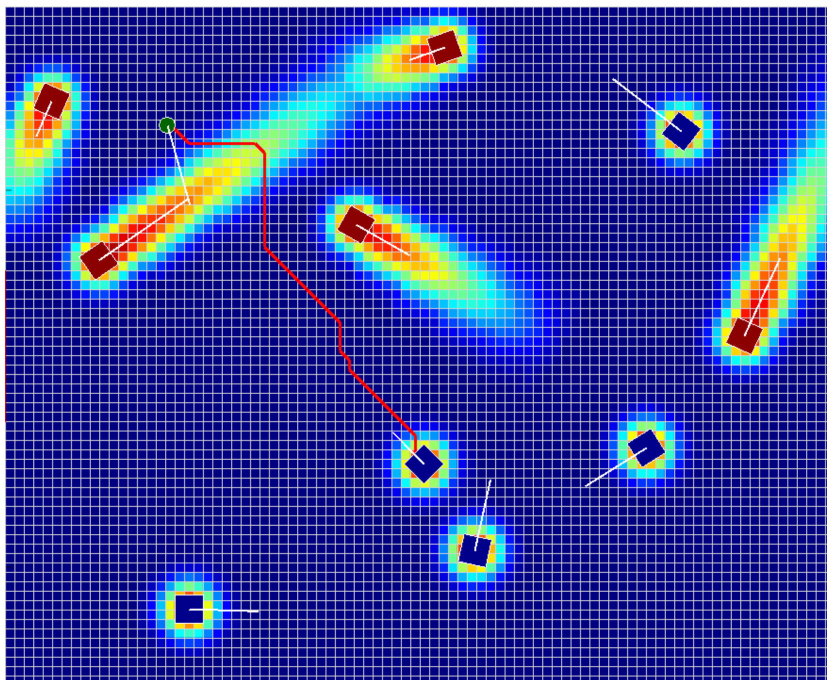
C Obrázkové přílohy



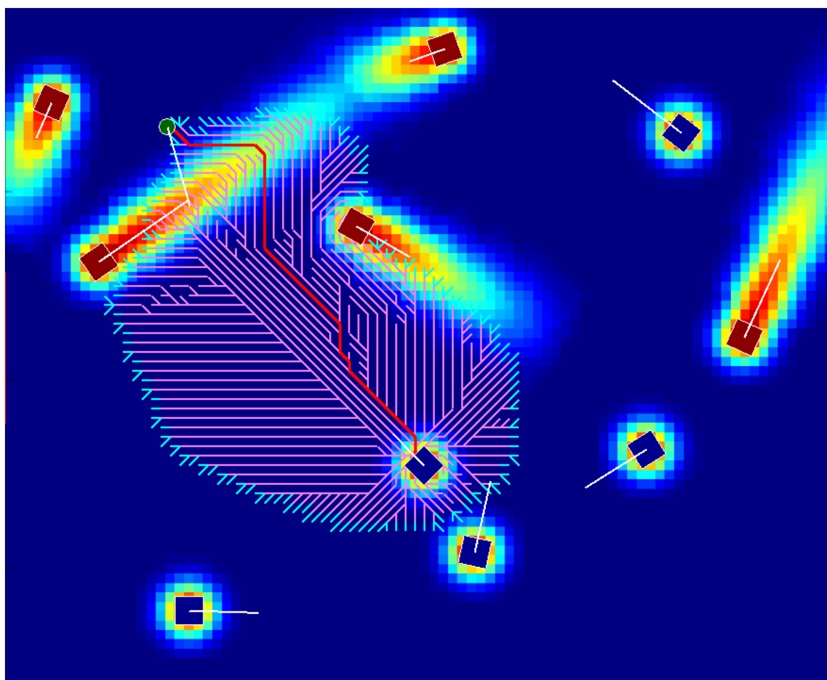
Obrázek C.1: Příklad spočtené trasy a prohledávaní



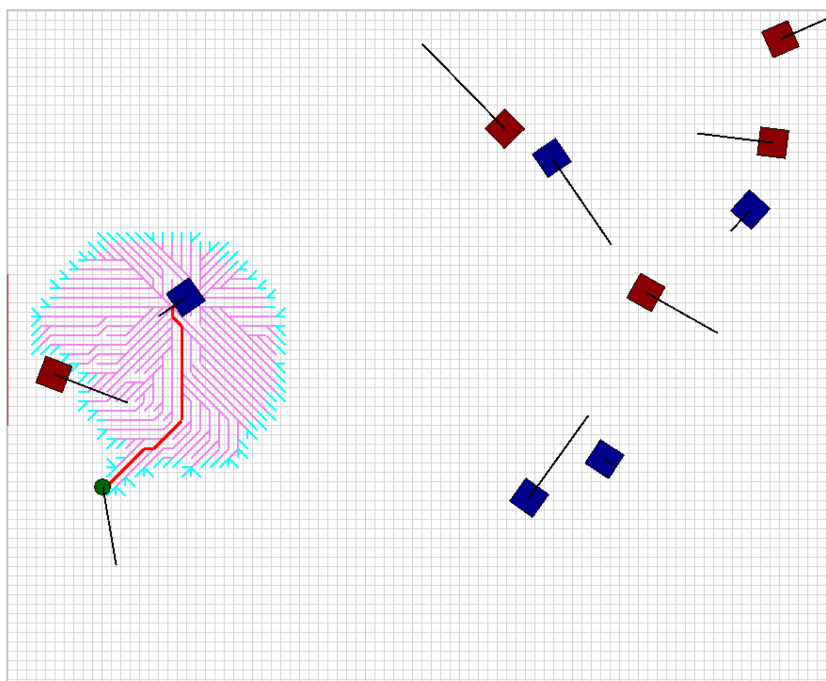
Obrázek C.2: 3D mapa pro váhy odpovídající obrázku C.1



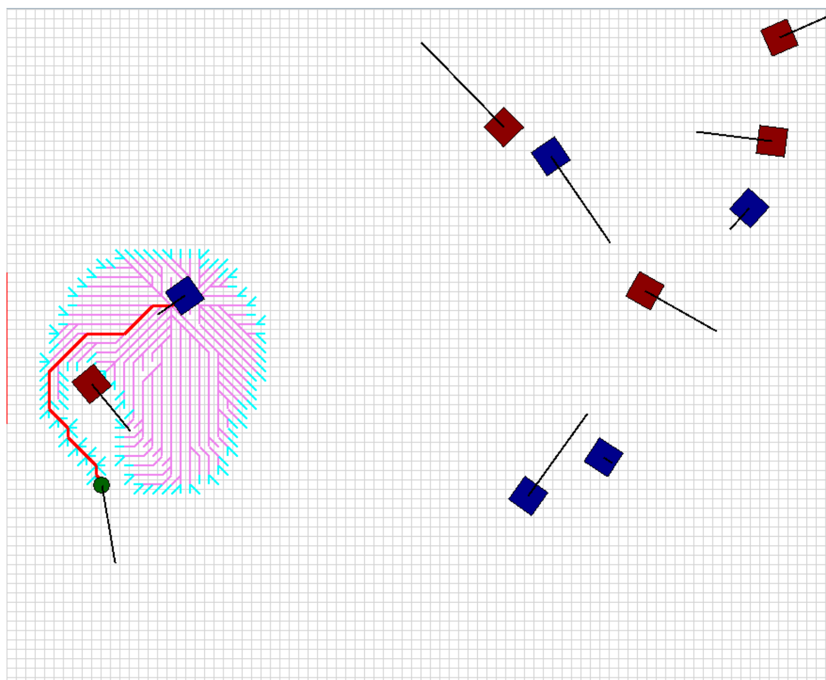
Obrázek C.3: Obrázek C.1 se zobrazenými vahami v mřížce



Obrázek C.4: Obrázek C.1 se zobrazenými vahami a prohledáváním



Obrázek C.5: Příklad nalezené trasy při určité pozici soupeře



Obrázek C.6: Změna nalezené trasy při jen trochu jiné pozici soupeře