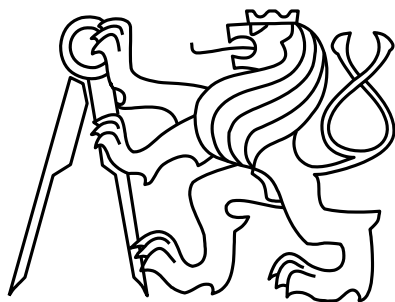


akČeské vysoké učení technické
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Umělá inteligence v tahových strategiích

Lukáš Beran

Vedoucí: Ing. Michal Hapala

Studijní program: Softwarové technologie a management
Studijní obor: Softwarové inženýrství

Leden 2011

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Michalu Hapalovi za věcné připomínky a cenné rady během zpracování této práce. Dále bych rád poděkoval Aleně Varkočkové a Pavle Balíkové za jejich spolupráci při tvorbě základů Expanze. A nakonec všem, kteří hru ochotně vyzkoušeli a vznesli hodnotné námitky a návrhy pro její vylepšení.

Declaration – Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v příloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 15. května 2011

.....

Lukáš Beran

Abstract

The bachelor thesis summarizes the techniques of AI in computer games. It gives a theoretical basis for the decision trees, finite automata, fuzzy sets, goal-driven architecture, genetic algorithms, neural networks, artificial life, pathfinding and scripting.

It describes the implementation of decision trees and goal-driven architecture using genetic algorithm in my own turn based strategy game based on the principles of expansion board game Settlers of Catan. AI is based on plugins architecture which enables a number of experiments when comparing different AIs.

Abstrakt

Tato bakalářská práce shrnuje techniky z oblasti AI v počítačových hrách. Podává teoretický základ o rozhodovacích stromech, konečných automatech, fuzzy množinách, goal-driven architektuře, genetických algoritmech, neuronových sítích, umělém životě, vyhledávání cest a o skriptování.

Popisuje realizaci rozhodovacích stromů a goal-driven architektury s využitím genetického algoritmu ve vlastní tahové strategické hře Expanze založené na principech deskové hry Osadníci z Katanu. AI je založena na pluginové architektuře, která umožňuje množství experimentů při porovnávání jednotlivých AI.



Obsah

SEZNAM OBRÁZKŮ	XI
SEZNAM TABULEK.....	XI
SEZNAM UKÁZEK KÓDU	XI
1 ÚVOD.....	1
1.1 HISTORIE.....	1
1.2 ILUZE INTELIGENCE.....	2
1.3 NEJEN SOUPEŘ JE INTELIGENTNÍ.....	CHYBA! ZÁLOŽKA NENÍ DEFINOVÁNA.
2 AI ALGORITMY A TECHNIKY	3
3 ROZHODOVÁNÍ.....	3
3.1 ROZHODOVACÍ STROMY	3
3.1.1 Vnitřní uzly.....	4
3.1.2 Zlepšování výkonu.....	6
3.1.3 Stromy chování.....	7
3.2 STAVOVÝ AUTOMAT.....	8
3.2.1 Konečný stavový automat.....	8
3.2.2 Implementace	9
3.2.3 Vylepšení FSM	10
3.3 GOAL – DRIVEN ARCHITEKTURA	11
3.3.1 Fáze rozhodování.....	12
3.3.2 Analýza hry	12
3.3.3 Vytváření cílů a jejich evaluace	13
3.3.4 Priorita cílů.....	13
3.3.5 Plánování	14
3.3.6 Akce a chování	14
3.4 FUZZY LOGIKA	14
3.4.1 Fuzifikace.....	16
3.4.2 Fuzzy pravidla.....	18
3.4.3 Defuzifikace	19
4 INSPIROVÁNO PŘÍRODOU	19
4.1 GENETICKÉ ALGORITMY.....	19
4.1.1 Evoluce v přírodě.....	19
4.1.2 Hledání cesty pomocí genetického algoritmu	20
4.1.3 Selektce párů.....	20
4.1.4 Kombinace křížením.....	21
4.1.5 Mutace	21
4.2 NEURONOVÉ SÍTĚ	22
4.2.1 Umělý neuron.....	23
4.2.2 Pracovní fáze umělé neuronové sítě.....	25
4.2.3 Back-propagation	25
4.3 UMĚLÝ ŽIVOT.....	26
4.3.1 Mazlíčci.....	27
4.3.2 Společenské simulace.....	27
4.3.3 Hry na Boha.....	27
4.3.4 Evoluční hry.....	27
4.3.5 A-Life a tahové strategie.....	27
5 SPECIFICKÉ METODY.....	28
5.1 HLEDÁNÍ CEST.....	28
5.1.1 Prohledávání do šířky	28
5.1.2 Pár slov k A*.....	29
5.2 SKRIPTOVÁNÍ	30
5.2.1 Interpretace vs. kompilace	31
5.2.2 Skriptovací jazyky	32
6 FRAMEWORKY A ENGINY	32

7	REALIZACE	33
7.1	ROZDĚLENÍ ÚLOH	33
7.2	STRUČNÝ POPIS HRY	33
7.3	VÝVOJOVÉ PROSTŘEDÍ	34
7.4	REPREZENTACE SVĚTA	35
8	PLUGINOVÁ ARCHITEKTURA.....	36
8.1	KOMPONENTY	36
8.2	ZPŮSOB OVLÁDÁNÍ.....	37
8.2.1	<i>Myš.....</i>	37
8.2.2	<i>Klávesnice.....</i>	38
8.3	SKRIPTOVACÍ JAZYK.....	39
8.4	PODVÁDĚNÍ ZAKÁZÁNO	40
9	ZÁKLADNÍ AI.....	41
10	ROZHODOVACÍ STROMY	43
10.1	ZÁKLADNÍ STRUKTURA „STROMU“	43
10.2	DRUHY UZLŮ	45
10.2.1	<i>Akce.....</i>	45
	ActionNode	45
	ChangeSourcesActionNode	45
	FakeActionNode	45
10.2.2	<i>Rozhodovací uzly.....</i>	46
	DecisionBinaryNode	46
	SerialNode	46
	RandomNode	46
	SourcesNode	46
	ForBestNode	47
10.3	SHRNUTÍ.....	47
11	GOAL DRIVEN ARCHITEKTURA	48
11.1	STRUKTURA CÍLŮ	48
11.1.1	<i>AtomGoal</i>	49
11.1.2	<i>CompositeGoal.....</i>	49
	RaiseSources	50
	BuildSourceBuilding.....	50
	BuildMarket	51
	FortCaptureHexa.....	51
	BuildTown	51
11.2	GENETICKÝ ALGORITMUS.....	53
11.2.1	<i>Proč říct Ne evoluci.....</i>	53
	Zaměřeno na AI	53
	Fitness	53
	Determinismus	54
11.2.2	<i>Nastavení GA</i>	54
11.2.3	<i>Výsledky</i>	54
11.3	SHRNUTÍ.....	55
12	TESTOVÁNÍ.....	56
12.1	TESTOVÁNÍ PRAVIDEL	56
12.1.1	<i>Závislost na pořadí hráčů</i>	57
12.1.2	<i>Jak ovlivňují náhodné události hru</i>	57
12.1.3	<i>Využití jednotlivých surovin</i>	57
12.1.4	<i>Vhodnost všech kombinací nastavení mapy</i>	58
12.1.5	<i>Evoluce testuje</i>	58
12.2	MONKEY TESTOVÁNÍ.....	58
12.3	TESTOVÁNÍ AI MODULŮ	59
12.3.1	<i>DT-AI vs. GD-AI</i>	59
12.3.2	<i>Grafy ve hře.....</i>	60
12.4	UŽIVATELSKÉ TESTY	61
12.4.1	<i>Nastavení testu</i>	61
12.4.2	<i>Dotazník</i>	62
12.4.3	<i>Výsledky</i>	62

13	ZÁVĚR.....	62
14	CITOVANÁ LITERATURA.....	65
A	OBSAH CD	67

Seznam obrázků

OBR. 1 ROZHODOVACÍ STROM PRO CHOVÁNÍ HOUBY Z MARIA.....	4
OBR. 2 DEKOMPOZICE AND POMOCÍ DVOU ROZHODOVACÍCH UZLŮ	5
OBR. 3 DEKOMPOZICE OR POMOCÍ DVOU ROZHODOVACÍCH UZLŮ	5
OBR. 4 VÝČTOVÝ TYP V BINÁRNÍM STROMU.....	6
OBR. 5 VÝČTOVÝ TYP V OBEČNÉM STROMU	6
OBR. 6 VYVÁŽENÝ STAM S 8 AKCEMI.....	7
OBR. 7 KONEČNÝ STAVOVÝ AUTOMAT CHOVÁNÍ GOBLINA	9
OBR. 8 STAVOVÝ AUTOMAT ARCHITEKTURY PLÁNOVÁNÍ	12
OBR. 9 ZÁKLADNÍ TYPY FUNKCÍ PŘÍSLUŠNOSTI. ZDROJ [10].	16
OBR. 10 MANIFOLD FUNKCÍ PŘÍSLUŠNOSTI FLV VZDALENOST ARMADY. ZDROJ [10].	16
OBR. 11 GRAFICKÉ ZNÁZORNĚNÍ OPERACÍ AND, OR, NOT.....	17
OBR. 12 SCHÉMA NEURONU. ZDROJ [14].	22
OBR. 13 SCHÉMA UMĚLÉHO NEURONU. ZDROJ [14].....	23
OBR. 14 VRSTEVNATÁ NEURONOVÁ SÍŤ. ZDROJ [14].....	24
OBR. 15 TVAR SIGMOIDY. ZDROJ [14].	24
OBR. 16 PROHLÉDÁVÁNÍ DO ŠÍŘKY 5. KROK OBR. 17 PROHLÉDÁVÁNÍ DO ŠÍŘKY 15. KROK	29
OBR. 18 PRŮBĚH ALGORITMU A-STAR, 5. KROK. (MANHATTONKÁ METRIKA)	30
OBR. 19 SCREENSHOT Z ROZEHRANÉ HRY. JMÉNA HRÁČŮ JSOU INSPIROVÁNY PRŮZKUMNÍKY A DOBYVATELI.	34
OBR. 20 SOUŘADNICE JEDNOTLIVÝCH HEX	35
OBR. 21 ZJEDNODUŠENÉ SCHÉMA NÁVRHU HRY.....	37
OBR. 22 VÝBĚR MÍST PRO STAVBU JE PROVÁDĚN TECHNIKOU PICKINGU. KAŽDÁ CESTA MÁ SVŮJ ODSŤÍN ZELENÉ, MĚSTO ODSŤÍN MODRÉ A HEXA ODSŤÍN ČERVENÉ.	38
OBR. 23 OKNO, VE KTERÉM HRÁČ VYBÍRÁ, KTEROU Z BUDOV NA POUŠTI POSTAVÍ.	40
OBR. 24 VÝŘEZ ROZHODOVACÍHO STROMU.....	44
OBR. 25 DETAIL NA BUY LICENCE Z OBR. 24	44
OBR. 26 CLASS DIAGRAM CÍLŮ	48
OBR. 27 PO SKONČENÍ HRY SI LZE ZOBRAZIT NĚKOLIK GRAFŮ, KTERÉ MOHOU ODKRÝT SLABINY NĚKTERÉHO Z HRÁČŮ. ZDE JE VIDĚT, ŽE ŽLUTÝ HRÁČ HROMADIL SUROVINY A BYL VHODNÝM CÍLEM PRO KRÁDEŽ.....	60

Seznam tabulek

TABULKA 1 PŘÍKLADY DATOVÝCH TYPŮ DLE [6]	4
TABULKA 2 FUZZY PRAVIDLA PRO FLV VZDALENOST ARMADY A VELIKOST ARMADY.....	18
TABULKA 3 PŘÍSLUŠNOSTI KE KONSEKVENTŮM VYTVOŘ VOJÁKY	18
TABULKA 4 ZÁVISLOST VÝHRY NA POŘADÍ HRÁČŮ GOAL DRIVEN AI.....	57
TABULKA 5 PRŮMĚRNÉ HODNOTY DOTAZNÍKU K AI ROZHODOVACÍCH STROMŮ.....	62

Seznam ukázek kódu

KÓD 1 PSEUDOKÓD JEDNODUCHÉHO STAVOVÉHO AUTOMATU PRO CHOVÁNÍ GOBLINA.	10
KÓD 2 PSEUDOKÓD UPDATE FUNKCE FSM AUTOMATU DLE [8].....	10
KÓD 3 ROZHODOVÁNÍ V BOOLEOVSKÉ ALGEBŘE.	15
KÓD 4 UKÁZKA NĚKOLIKA FUZZY PRAVIDEL.	18
KÓD 5 JEDNODUCHÝ XML SKRIPT	31
KÓD 6 PŘÍKLAD SKRIPTU DEFINUJÍCÍ ÚKOL VE HŘE.	31
KÓD 7 TŘI FÁZE NASTAVENÍ ZAVŘENÍ OKNA NA KLÁVESY M A ESCAPE VE STAVU GAMEMARKET. 1. VYTVOŘÍ SE AKCE A PŘÍŘADÍ SE JÍ KLÁVESY. 2. ZMĚNÍ SE STAV INPUT MANAGERU NA GAMEMARKET. 3) ZJISTÍ SE AKTIVACE AKCE CLOSE.	39
KÓD 8 UKÁZKA KÓDU AI LEHKÁ.....	42
KÓD 9 VYUŽITÍ LAMBDA FUNKCÍ JAKO PODMÍNKY V ROZHODOVACÍCH UZLECH.	46
KÓD 10 UKÁZKA LOGU SLOŽENÉHO CÍLE	49
KÓD 11 UKÁZKA ATOMICKÉHO CÍLE PRO STAVBU CEST.	49
KÓD 12 ČÁST METODY PROCESS Z COMPOSITEGOAL.....	50

1 Úvod

Cílem této bakalářské práce je seznámit se s technikami využívanými ve speciální oblasti herní umělé inteligence, navrhnout a vytvořit vlastní tahovou strategickou počítačovou hru a implementovat do ní umělou inteligenci. Udělat rešerši o existujících řešeních a frameworkcích a zhodnotit jejich vhodnost pro použití v tahové strategii. Na závěr otestovat vlastní řešení.

Mým osobním cílem bylo vytvořit protivníka, který by byl výzvou i pro mě jako autora hry. Vymyslet AI, která mě dokáže porazit.

Ve zbytku první kapitoly popisují počátky umělé inteligence v počítačových hrách a porovnávám výzkum umělé inteligence na akademické úrovni s umělou inteligencí využívanou v zábavném průmyslu.

Následující čtyři kapitoly seznamují s devíti technikami, které se ve hrách používají. V druhé kapitole jsou uvedeny a roztříděny do následujících tří kapitol. 3. kapitola se zabývá metodami používanými při plánování, mezi jež patří rozhodovací stromy, konečné automaty, goal-driven architektura a fuzzy množiny. Ve 4. kapitole se zaměřuji na algoritmy inspirované přírodou, uvedu zde základy genetických algoritmů, neuronových sítí a zmíním se o hrách známých pod názvem umělý život. Tuto část uzavírá 5. kapitola s technikami, které nelze opominout v práci o umělé inteligenci ve hrách. Jedná se o hledání cest a skriptování.

V poslední 6. kapitole teoretické části uvádím existující frameworky a enginy, které jsou více či méně vhodné pro využití v počítačových hrách.

Na počátku 7. kapitoly popíši zvolenou hru pro implementaci umělé inteligence, vymezím, které části hry jsem vytvářel já a které vytvořil zbytek týmů v rámci semestrální práce do předmětu Počítačové hry a animace.

8. kapitola nese název Pluginová architektura a zabývá se způsobem, jakým je ve hře AI vytvořena, jakým způsobem komunikuje se zbytkem hry.

Další tři kapitoly popisují konkrétní řešení umělé inteligence. První jednoduchá založená na sérii podmínek, druhá vycházející ze stromů chování a poslední založená na Goal-driven architektuře využívající genetický algoritmus.

Praktickou část uzavírá 12. kapitola o testování a Závěr jako kapitola 13.

1.1 Historie

Vývoj umělé inteligence do počítačových her započal se startem her samotných. Již první grafická hra z roku 1952 Tic Tac Toe, piškvorky 3 x 3 pole, měla zabudovanou umělou inteligenci protihráče. [1] Totéž platí i pro následovníka „Tenis pro dva“, nyní známějšího pod jménem Pong.

Přestože v počátcích video her byly hry značně jednoduché včetně umělé inteligence, mnohé by mohla překvapit propracovanost a složitost umělé inteligence ve hře Pac Man. Jedná se o jednoduchou hru odehrávající se na jedné obrazovce, kde máme za úkol sbírat po bludišti kolečka představující jídlo a zároveň nás nesmí chytit jeden ze čtyř duchů. Duchové nejen mají různé barvy, jména a přezdívky, ale také odlišná chování. Málokterý hráč si toho všimne. [2]

Složitější umělé inteligence bylo potřeba vyvíjet právě s rozvojem strategických her (Civilization, Heroes of Might and Magic), kde bylo a je nutností udělat poměrně složitou AI, aby hra byla vůbec hratelná.

V minulosti byl vývoj umělé inteligence v ústraní. Mnohem důležitější bylo vyvíjet grafickou stránku hry, která zároveň znatelně vytěžovala CPU počítačů a tedy ani výpočetní výkon nezbyval pro AI. Vývoj umělé inteligence často probíhá až v posledních pár měsících tvorby hry, a tedy AI nemůže být dokonalá.

V současné době, kdy už je hráč nasycen téměř dokonalou fotorealistickou grafikou, kdy výkon počítačů je mnohem dál, je již dostatek prostoru pro rozvoj umělé inteligence do počítačových her. [3]

1.2 Iluze inteligence

Umělá inteligence v počítačových hrách má s akademickou umělou inteligencí mnoho společného. Metody jako rozhodovací stromy, konečné stavové automaty nebo neuronové sítě můžeme najít v obou oborech.

Například AI ve hrách se v určitých aspektech liší od umělé inteligence řízení leteckého provozu. Nemusí být co nejchytřejší, co nejlepší. Již dávno není problém udělat inteligenci bota ve FPS střelce neomylného. Bot, který by zastřelil hráče jednou ranou do hlavy. Podobně lze udělat nepřekonatelného střelce v basketbalu, který trefí koš přes celé hřiště. Oba dva případy by potencionální hráče brzy odradily.

Existují hráči, kteří až absurdní obtížnost ocení, ale většina hráčů ne. Vhodnější model je, když hráči polovinu času vyhrávají a polovinu času prohrávají. Vůbec nemusí být špatným nápadem korigovat obtížnost hry dle počtu výher a proher, a tím jejich poměr zachovávat přibližně roven jedna ku jedné.

Je důležité zachovat určitou reálnost chování NPC, hráč by neměl mít pocit toho, že soupeř podvádí. Při vývoji hry Empire Earth měli testéři hlásit jakékoliv podezřelé chování soupeře, měli zapsat kdykoliv si mysleli, že jejich soupeř podvádí. Přestože se vždy nejednalo o podvod, vývojáři nahlášené problémy vždy prodiskutovali a případně umělou inteligenci poté upravili. [4]

Podvádění soupeřů někdy nemusí být na škodu, ale nesmí to hráč poznat. NPC z pohledu vývojáře podvádí, pokud využívá věcí ve hře, které nemůže využít lidský hráč. Např. když od počátku hry soupeř ví, kde má hráč základnu a vysílá tam své vojáky bez předchozího průzkumu. Ve strategických hrách není neobvyklým jevem, když soupeř vytvoří pro obranu své základny jednotky z ničeho, ze surovin, které nemohl během hry získat. Oba ze zmíněných podvodů mohou být přípustné a zlepšit celkovou hratelnost hry, pokud je hráč neodhalí.

Z pohledu laika může být podvádění i to, když se soupeř chová až moc dokonale, nejedná realisticky. Sem patří již zmíněná dokonalá střelba v FPS. Realističnost je z pohledu hráče hodně důležitá, ale jsou případy, kdy by mu mohla přijít otravná. Adventura, kde jsou dvě cesty. Na konci každé z nich je truhla. V jedné je klíč od té druhé. Hráč může zvolit jakoukoli z těchto cest jako první. Pokud by zvolil jako první tu, jež ho vede k zamčené truhlici, musel by se vracet. Jít druhou cestou, získat klíč a opět jít zpět k první truhle. Tento nedostatek můžeme elegantně vyřešit. První truhla, kterou hráč otevře, bude obsahovat klíč od té druhé truhly. [5]

Můžeme se na problém tvorby AI podívat z jiného pohledu. Nevytváříme NPC přehnaně komplexní a inteligentní, ale vložíme inteligenci do světa kolem ní. Známým příkladem je hra The Sims, simulátor lidí, kteří mají své potřeby jako je hlad, jež je potřeba uspokojit. Pokud bychom se na to podívali z pohledu reálného světa, postava, když dostane hlad, najde v domě nejbližší jídlo (lednička, hotové jídlo na stole), dojde k němu a začne jíst. V The Sims je to

dělané jinak. Lednička do určitého poloměru vysílá zprávu „Můžu uspokojit tvůj hlad“. Pokud se simik dostane do blízkosti ledničky, vyhodnotí své aktuální potřeby a jestliže hlad má, vezme si z ledničky jídlo. [5]

Do prostředí můžeme zanést rozmanité informace. Např. bodliny v 2D plošinovce, které ubírají životy všemu, co se jich dotkne. Nebudeme logiku ubírání životů bodlinami a měnění hráčovi animaci programovat v rámci logiky hráče, ale v rámci logiky bodlin. Bodliny budou vědět, že mají hráči ubrat životy, změnit mu animaci, odstrčit ho. Tímto přístupem zjednodušíme nejen komplexnost logiky hráče, ale zjednodušíme přidávání nových herních prvků do hry.

Musíme si zapamatovat, že hra nemusí být dokonale realistická, férová, neporazitelná. Pořád se pohybujeme v herním, v zábavním průmyslu a tedy dobrá AI má za úkol hlavně pobavit a být výzvou pro hráče. Musí umět bavit naprostého nováčka i zkušeného hráče hrajícího online turnaje.

2 AI algoritmy a techniky

Algoritmy a techniky používané pro tvorbu herní umělé inteligence jsou stejné jako pro akademickou AI. Popíšeme si srozumitelnou formou několik nejznámějších a nepoužívanějších technik umělé inteligence v počítačových hrách.

U každé techniky si uvedeme názorné příklady, jež mají napovědět, v kterých situacích je daná technika vhodná. Vždy se seznámíme pouze se základními principy každé z technik. Další informace nalezneme v knihách [6], [7], [8], [9].

Jednotlivé techniky nemusíme používat samostatně. Naopak, mnohdy můžeme dosáhnout skvělého řešení kombinací dvou a více technik. Neuronové sítě můžeme učit pomocí genetických algoritmů [8], fuzzy logiku můžeme kombinovat se stavovými automaty [10] nebo s rozhodovacími stromy. Skripty rozšíříme libovolnou z ostatních technik atd.

Techniky rozdělíme do tří kategorií: Rozhodování, Inspirováno přírodou, Specifické metody.

Důležitá poznámka: v následujícím textu není popsán algoritmus Min-Max, který je často používán u klasických strategických her¹. Neuvedl jsem ho zde pro jeho velkou známost a poměrně specifické využití především u her s malým počtem možných akcí každé kolo, her bez náhody.

3 Rozhodování

Zaútočit, zlepšovat obranu, přihrát spoluhráči, střílet na bránu, umělá inteligence musí stejně jako hrát dělat různá rozhodnutí ve hře. Musí analyticky dobře vyhodnotit danou situaci a dle toho si zvolit další akci, kterou provede.

Existuje několik metod, jak ve hře rozhodování provádět. Mezi některé patří níže popsané Rozhodovací stromy, Stavové automaty, Goal-driven architektura a Fuzzy logika.

3.1 Rozhodovací stromy

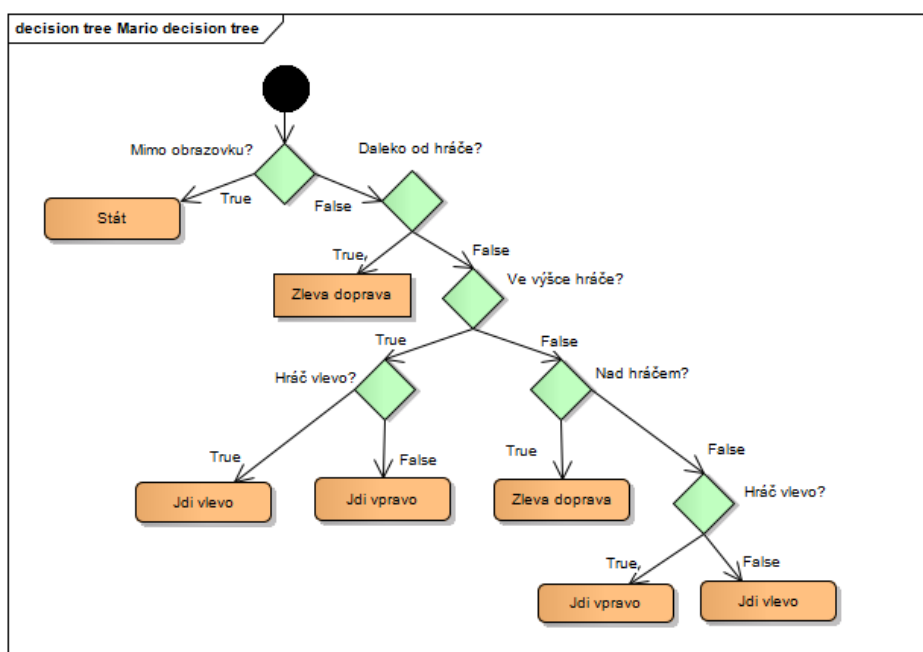
Rozhodovací stromy (decision trees) jsou jednou z oblíbených technik využívaných nejen v umělé inteligenci, ale také ve vytěžování dat (data mining). Rozhodovací stromy si svojí

¹ Mezi klasické strategické hry patří např. šachy, piškvorky, dáma.

popularitu získaly především díky své jednoduchosti. Snadně se interpretují, chápou, ale i implementují.

Představme si, že tvoříme umělou inteligenci pro nebezpečné houby z 2D plošinové hry Mario. Houby ve hře zabíjíme skokem na ně. V případě jiného dotyku houby zraní vás. Jak by se takové houby měly chovat? Když nejsou vidět, hráč je příliš daleko, neměly by dělat nic. Pokud jsou již na obrazovce, ale hráč je stále daleko, budou chodit z jednoho okraje plošinky k druhému. Jinak zkontrolujeme, jestli je hráč ve výšce houby. Když ano, houba jde směrem k hráči. Když je hráč nad houbou, houba se cítí ohrožena a jde proti směru pohybu hráče v naději, že na ni hráč nedopadne. Poslední případ je, že houba je na plošině, která je nad hráčem. V takovém případě houba chodí od okraje ke okraji.

Cíleného chování bychom mohli dosáhnout pomocí hierarchie if-then podmínek. Takové řešení je sice možné, ale obtížně bychom ho upravovali, debutovali a rozšiřovali. Popsané chování přímo navádí k využití rozhodovacích stromů. Strom odpovídající příkladu s Mariem můžeme vidět na následujícím obrázku Obr. 1.



Obr. 1 Rozhodovací strom pro chování houby z Maria

Uzly v rozhodovacím stromě jsou dvojího druhu. Vnitřní uzly představují podmínky, v listech jsou umístěny akce, vzory chování. Můžeme si všimnout, že některé akce jsou umístěny na více místech. Můžeme jich dosáhnout průchodem více různých cest. To je zcela v pořádku.

Rozhodování, jaká akce se vykoná, začíná v kořeni stromu. Dle splnění podmínky se přejde do levého, či pravého potomka kořene a opět se vyhodnotí podmínka v něm. Tento postup se rekurzivně opakuje dokud se nedosáhne akce v některém z listů. Tato akce se vykoná.

3.1.1 Vnitřní uzly

Každý vnitřní uzel by měl kontrolovat jednoduchou podmínku, zpravidla závislou na typu proměnné. Některé druhy podmínek shrnuje následující tabulka Tabulka 1.

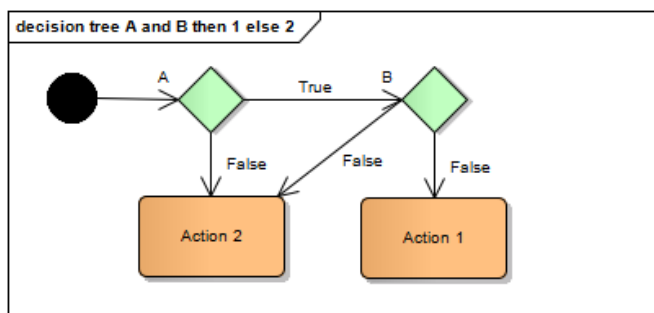
Tabulka 1 Příklady datových typů dle [6]

DATOVÝ TYP	ROZHODNUTÍ
------------	------------

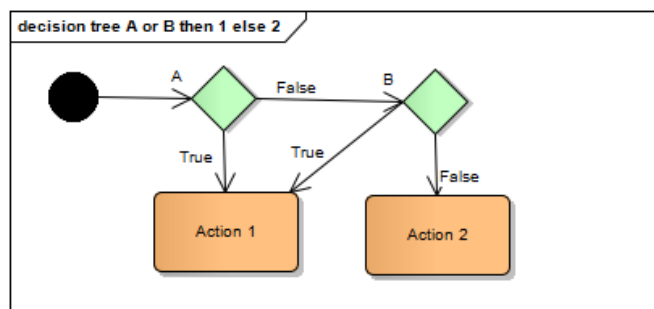
boolean	hodnota je true
výčet hodnot (právě jedna z nich je možná v jednom okamžiku)	shoda s jednou z množiny hodnot
číselné	hodnota je v daném intervalu
vektor	vektor má délku v daném rozsahu (např. vzdálenost hráče a nepřítele)

Měli bychom využívat pouze atomární podmínky bez spojek jako jsou AND, OR, XOR a jiné. Složené podmínky můžeme nahradit posloupností více atomárních podmínek jdoucích za sebou v rozhodovacím stromu.

Příklady dekompozice podmínek AND a OR si pohlédneme na Obr. 2 a Obr. 3.



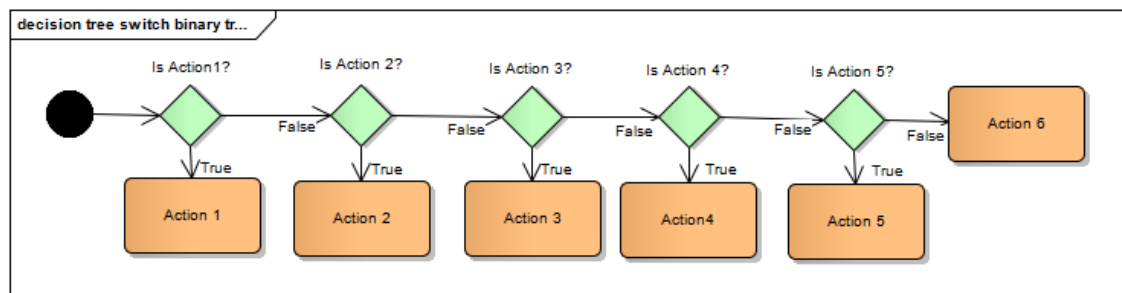
Obr. 2 Dekompozice AND pomocí dvou rozhodovacích uzlů



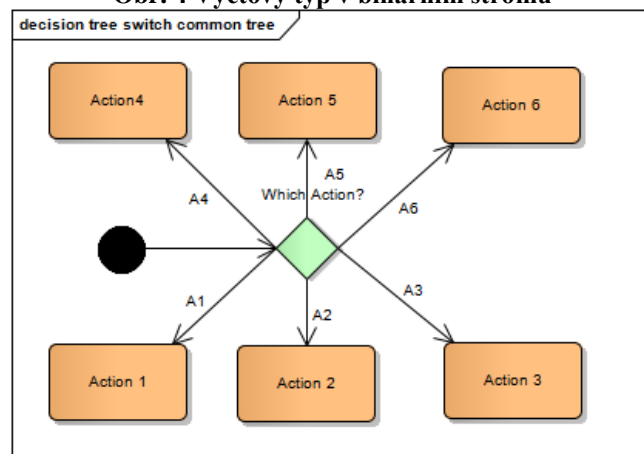
Obr. 3 Dekompozice OR pomocí dvou rozhodovacích uzlů

Rozhodovací stromy jsou často stromy binární. Každý rodičovský uzel má dva potomky. Použití binárních rozhodovacích stromů umožňuje použít na stromy různé algoritmy učení a ponechávají jednoduchost řešení.

Binární stromy ale mají svá úskalí. Mějme podmínku na výčtový typ o 6 možnostech. V případě binárního stromu bychom museli testovat až 5 podmínek, jestliže by měla být správná možnost 5, či 6. Pokud bychom si zvolili strom bez omezení počtu potomků, mohli bychom pokaždé testovat pouze jednu podmínku.



Obr. 4 Výčtový typ v binárním stromu



Obr. 5 Výčtový typ v obecném stromu

O rozhodovací strom se může z hlediska AI jednat, přestože z hlediska teorie grafů graf podmínek stromem není. Můžeme toho dosáhnout spojením více větví grafu do společné rozhodovací podmínky. Obdobně jako na různých koncích stromu můžeme mít stejné akce. Můžeme mít i stejné podmínky u potomků různých uzlů. Pokud se pustíme do takového rozšíření rozhodovacího stromu, musíme si dát pozor na vznik smyček, které by byly nežádoucí.

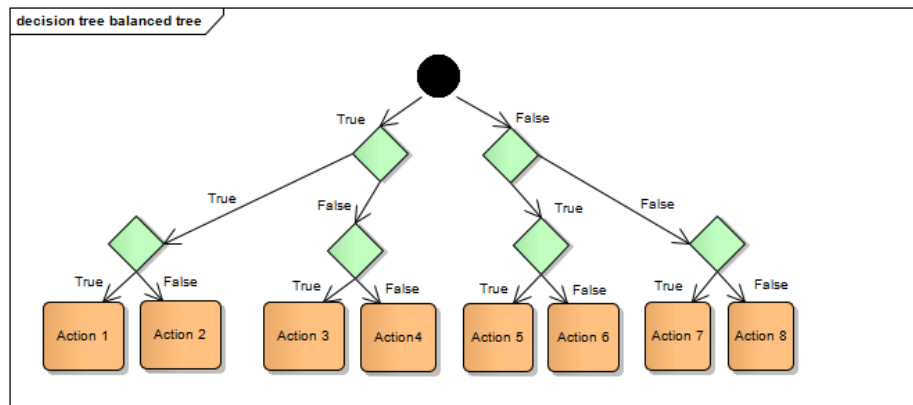
Někdy můžeme do rozhodovacího stromu zanést náhodu. Opakování stejných akcí dokola může časem působit nudně. Ozvláštňeme rozhodovací strom přidáním vnitřních uzlů, které náhodně vybírají ze svých potomků. Když se houba z našeho příkladu dostane na obrazovku, vydá se vlevo. Lepší by bylo, kdyby náhoda rozhodla, jakým směrem se vydá.

Problém může nastat, pokud voláme funkci rozhodovacího stromu každý frame. Jestliže se podmínky světa nezmění, při rozhodování se dostaneme do stejného uzlu s náhodným výběrem potomka. Pokud tento stav neošetříme, bude naše houba oscilovat. Jednou se začne hýbat vlevo, podruhé vpravo. Možné řešení nalezneme v [6].

3.1.2 Zlepšování výkonu

Rozhodovací stromy jsou oproti jiným technikám poměrně nenáročné na paměť a procesor počítače. Přesto bychom se měli při konstrukci stromu zamyslet, jestli by nešel vystavět lépe. Mějme strom, v kterém je v jeho spodní části celkem 8 akcí. Pokud bychom vytvořili nevyvážený strom o hloubce 7 se sedmi rozhodovacími podmínkami obdobný stromu z Obr. 4 a každá akce by měla stejnou pravděpodobnost, průměrně bychom prošli 4,5 podmínkami.

Po vyvážení by mohl mít strom hloubku rovnou 3. Každá z akcí by byla dosažena po průchodu tří podmínek.



Obr. 6 Vyvážený strom s 8 akcemi

Dosáhli jsme výrazného zlepšení o $1/3$. Původní strom měl složitost $O(n)$, vyvážený strom má složitost $O(\log_2 n)$.

Bohužel ke zlepšení mohlo dojít pouze na papíře a ve skutečnosti se mohlo rozhodování naopak zpomalit. Abychom mohli spočítat průměr 4,5 rozhodnutí, museli jsme uvést předpoklad, že každá akce je stejně pravděpodobná. To se děje velmi zřídka, často jsou některé akce pravděpodobnější než jiné. Pokud by se akce 1 a 2 projevily v 90% času, pravděpodobně by byl výhodnější strom první.

Také musíme zvážit, kolik času zaberou jednotlivá rozhodování. Podmínka na vzdálenost dvou bodů v 3D světě je několikanásobně náročnější než jednoduchá podmínka na true/false, a proto bychom ji měli kontrolovat blíže listům stromu.

3.1.3 Stromy chování

Stromy chování (behavior trees) jsou obdobné rozhodovacím stromům. Uzly se nedělí na rozhodovací a akce. Bude jeden kořenový uzel a každý další uzel bude obsahovat podmínku a akci.

Uzly budou tří typů: prioritní, sekvenční, stochastické.

Akci, která se provede, vyhodnocujeme trochu jinak než v rozhodovacích stromech. Opět vyhodnocujeme podmínky od kořene směrem dolů k listům. Pro aktuální uzel (první aktuální uzel je kořen) vezmeme všechny jeho potomky a vyhodnocujeme se jejich podmínky (validujeme je). Další aktivní uzel určíme dle typu rodiče.

U prioritních uzlů validujeme jejich potomky v pořadí dle priority. První uzel, jehož podmínka je splněna, zvolíme uzlem aktivním. Provedeme pokračujeme ve vyhodnocování jeho potomků.

U sekvenčních uzlů postupně validujeme všechny potomky. Každý z validních uzlů zvolíme uzlem aktivním.

U stochastických uzlů nejdříve validujeme potomky. Poté náhodně vybereme jeden z validních potomků, a ten se stane aktivním.

Více se můžeme dozvědět ze záznamu přednášky Behavior trees: Three way of Cultivating Game AI, která zazněla na konferenci GDC 2010. Volně přístupné slidy můžeme nalézt na [11].

3.2 Stavový automat

V mnohých hrách můžeme pozorovat následující chování jednotek. Z dálky pozorujeme skupinku nepřátelských goblinů stojících v hloučku. Když se k nim přiblížíme na dostatečnou vzdálenost, goblini nás zpozorují a zaútočí. Zaútočíme na jednoho z nich. Když mu ubereme většinu jeho života, začne od nás utíkat. Druhého stihneme zabít rychleji než se může dát na útěk. Souboj pokračuje, ale náš hrdina zeslábné a my se musíme dát na útěk. Z dálky pozorujeme, jak se ke skupince goblinů vrací ten, jež utekl a jemuž se postupně doplňuje zdraví.

3.2.1 Konečný stavový automat

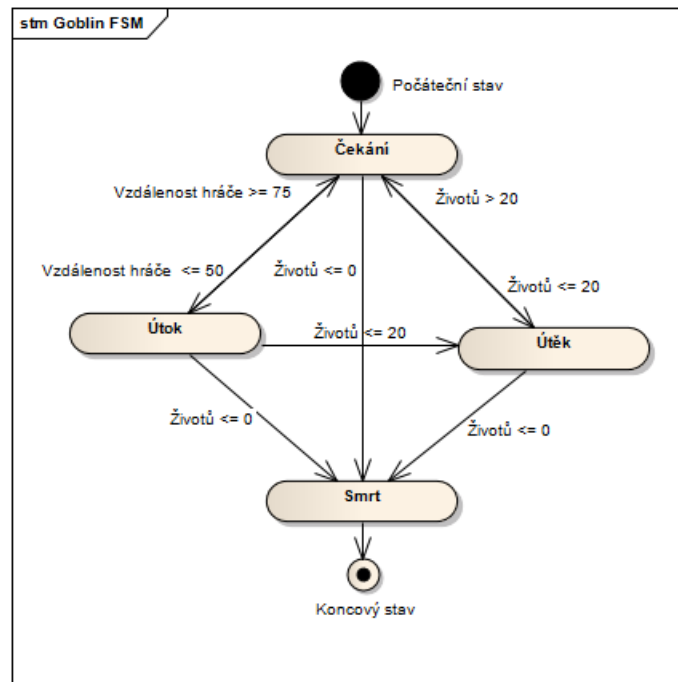
Pro zachycení tohoto chování je vhodné využít konečné stavové automaty. Konečný stavový automat (*Finite State Machine, FSM*) lze definovat jako uspořádanou pětiici $(Q, \Sigma, \delta, q_0, F)$, kde Q je konečná neprázdná množina stavů. Σ je konečná množina vstupních symbolů. δ je přechodová funkce, která lze definovat $\delta : Q \times \Sigma \rightarrow Q$. Počáteční stav $q_0 \in Q$. Množina koncových stavů $F \subseteq Q$.

Definice může být matoucí, především nemusí být jasné, co se zde míní konečnou množinou vstupních symbolů Σ . Vysvětlíme si to na příkladu AI goblinů. Množina všech stavů Q obsahuje stavy čekání, útok, útěk, smrt. Počátečním stavem q_0 je čekání. Každý goblin se po svém vytvoření nachází v tomto stavu. Konečný stav je zde pouze jediný, a to je stav smrt. Symboly Σ si lze představit jako podmínky, které mohou vést ke změně z jednoho stavu do druhého. Zde např. podmínky životů ≤ 0 , životů ≤ 20 , životů > 20 , vzdálenost hráče ≤ 50 , vzdálenost hráče ≥ 75 .

δ je dle definice kartézský součin množiny všech stavů a množiny symbolů. Jinak řečeno musíme definovat pro každý stav, co se stane při splnění každé z podmínek. Máme 4 stavy a 5 podmínek, což je dohromady $4 * 5$, 20 možných případů přechodu. U stavových automatů použitých pro tvorbu AI nás nebudou zajímat všechny možné případy. V jednotlivých stavech bude kontrolovat jen ty podmínky, které mění jeden stav na druhý. Např. ve stavu čekání nás budou zajímat pouze podmínky životů ≤ 0 , životů ≤ 20 , vzdálenost hráče ≤ 50 . Pokud ve stavu čekání bude splněna podmínka životů ≤ 0 , nový stav bude smrt, podmínka životů ≤ 20 , nový stav bude útěk, podmínka vzdálenost hráče ≤ 50 , nový stav bude útok. Poslední 4. podmínka vzdálenost hráče ≥ 75 a 5. podmínka životů > 20 nemají v aktuálním stavu vliv na změnu stavu, a nebudou nás zajímat. Pokud bychom počítali pouze se zbraněmi s dosahem nižším než je 50, stačila by nám reakce pouze na podmínku vzdálenost hráče ≤ 50 , jelikož by nemohlo dojít ke snížení životů goblina.

Při navrhování stavových automatů je vhodné si nakreslit diagram obdobný tomu na následujícím obrázku Obr. 7, který znázorňuje kompletní stavový automat pro chování goblinů.

K vytváření diagramu můžeme využít některý z CASE nástrojů jako je např. Enterprise Architect, ale pro začátek si vystačíme s tužkou a papírem.



Obr. 7 Konečný stavový automat chování goblina

3.2.2 Implementace

Způsobů, jak implementovat FSM je mnoho. Mezi nejjednodušší varianty patří využití if-then-else podmínek pro změnu stavu a switch/case pro výběr aktivní akce dle stavu automatu.

Tato varianta je jednoduše naprogramovatelná, snadno pochopitelná a i rychlá v provozu. Přestože všechny zmíněné výhody zní úžasně, pro pokročilé programátory lze doporučit některé ze složitějších řešení založených na polymorfismu.

Můžeme si všimnout copy-paste vady. Ve všech stavech jsme zkopírovali podmínku pro změnu stavu na DEATH. V případě složitějšího automatu s více stavy bychom stejnou podmínku museli kopírovat i do nově přidávaných stavů, což dělá přidávání nových stavů komplikované. Mohli bychom při rozšíření množiny stavů tuto podmínku zapomenout zkopírovat. Zde se jedná pouze o jednu podmínku, kterou můžeme ještě ohlídat. V případě více podmínek a složitějších bychom ale mohli snadno udělat chybu.

```

switch(state)
case DEATH :
case IDLE :
    if(hitpoints < 0)           state = DEATH;
    elseif(hitpoints < 20)      state = FLEE;
    elseif(playerDistance < 50) state = ATTACK;
case ATTACK :
    if(hitpoints < 0)           state = DEATH;
    elseif(hitpoints < 20)      state = FLEE;
    elseif(playerDistance > 70) state = IDLE;
case FLEE :
    if(hitpoints < 0)           state = DEATH;
    elseif(playerDistance > 70) state = IDLE;

switch(state)
case DEATH : deathAction();
case IDLE :  idleAction();
case ATTACK : attackAction();
case FLEE :  fleeAction();
  
```

Kód 1 Pseudokód jednoduchého stavového automatu pro chování goblina.

Zkopírovaný kód má i další nevýhody. Např. podmínka pro smrt se může rozrůst o kontrolu, jestli se goblin neocitl v lávě. To se mohlo stát při zmateném útěku i při bezhlavém útoku na vás. Chyby se v takovém kódu špatně hledají.

Můžeme chtít mít ve hře různé nepřátele s rozdílnými chováními. Rozšíříme hru o hraničáře, který by měl rozdělen útok na stavy útok z dálky lukem a na útok zblízka. Opět bychom kopírovali kód, protože např. stavy a přechody na IDLE a FLEE by zůstaly i pro hraničáře stejné. Takový kód bychom špatně udržovali a rozšiřovali. [12]

Obvyklé rozšíření stavového automatu přidává akce, které se mají vykonat pouze jednou při přechodu z jednoho stavu do druhého. Akce můžou být potřeba jak při vstupu do nového stavu, i při výstupu z něj. Příkladem může být potřeba při smrti goblina změnit jeho animaci a přičíst zkušenosti hráči.

Lepší řešení využívá polymorfismu a návrhového vzoru State pattern [13], jehož název napovídá, že zde nalezne využití.

```

triggeredTransition = null;
for transition in currentState.getTransitions()
    if(transition.isTriggered())
        triggeredTransition = transition
        break;
if (triggeredTransition)
    targetState = triggeredTransition.getTargetState();
    actions = currentState.getExitActions();
    actions += triggeredTransition.getActions();
    actions += targetState.getEntryActions();
    currentState = targetState;
    return actions;
else return currentState.getActions()

```

Kód 2 Pseudokód update funkce FSM automatu dle [8]

Každý stav má určené akce, které se vyvolají při přechodu do něj (getEntryActions), při odchodu z něj (getExitActions) a akce v případě, že se stav nemění (getActions). Dále má definovanou množinu přechodů do jiných stavů (getTransitions), které mají metodu isTriggered(), jež určí, jestli přechod má nastat. Pokud ano, tak pomocí metody getTargetState se určí následující stav. Navíc stejně jako v ukázce můžou být definovány akce s spojené s přechodem (getActions).

V update funkci se nejdříve zkontrolují všechny přechody, jestli některý z nich nastal. Pokud ano, zavolají se výstupní akce současného stavu, akce přechodu a vstupní akce nového stavu. Jestliže nedojde ke změně stavu, zavolají se akce hlavní, jako je např. idleAction.

3.2.3 Vylepšení FSM

Druhé řešení je již o mnoho pružnější než první. Jednoduše lze přidávat další stavy a přechody mezi nimi, starat se o různá chování jednotlivých druhů monster. Rozšíření na hraničáře se dvěma druhy útoku, či na bereserkeru, který nezná strach a nemá akci FLEE, je již jednoduché.

Problematická může být metoda isTriggered(), která je u přechodů. Někdy v ní může být pouze jednoduchá podmínka, jindy složená z mnoha podmínek provázaných spojkami OR a AND. Mohou vzniknout přechody s podmínkami, které se od sebe moc neliší. Např. jednou se kontroluje počet životů ≤ 0 , u nelétavých nepřátel se navíc kontroluje, jestli nespadli do lávy.

Nabízené řešení vede ke kopírování první z podmínek do vstupní podmínky druhého přechodu. Tento problém lze napravit pomocí návrhového vzoru Composite [13]. Konkrétní řešení se nachází v [8].

Problém s kopírováním podmínky smrti do více stavů přetrvává i do druhé verze, i když zde to již není tak špatné jako ve switch verzi FSM. Zůstává zde nutnost registrovat přechod kontrolující zdraví goblina do všech jeho stavů. Jedno nabízené řešení je implementovat globální stavy, které se kontrolují každý frame nezávisle na stavu lokálním. [9] Druhou možností je implementovat hierarchické stavové automaty, kde každá jednotka AI může být ve více stavech zároveň. Je zde implementováno více stavových automatů, které se liší svou prioritou. Kontrola smrti by byla využita v FSM s vyšší prioritou než FSM obsahující stavy IDLE, ATTACK, FLEE. Více informací o hierarchických FSM lze získat z [8].

3.3 Goal – driven architektura

Vzpomeňme si, jak přemýšlíme při hraní počítačové hry. Mějme příklad tahové strategie Heroes of Might and Magic (HoMaM). Při hraní uvažujeme nad cíli, dle toho, jaké jsou naše aktuální potřeby.

Hlavní cíl je vyhrát. Tento cíl je sám o sobě poměrně složitý, a proto vyžaduje rozklad na nižší cíle. Na začátku mise bylo prozrazeno, kde se nachází soupeř. Naše cíle jsou najít cestu k němu, vytvořit dostatečně velkou armádu, zaútočit. Cíl vytvořit armádu bude zpočátku důležitější než cíl průzkumu, protože při průzkumu můžeme narazit na rozdílně obtížné nepřátele, kteří by nás mohli zničit. Jako dostatečně velkou armádu považujeme, když je v ní několik nejsilnějších příšer, draků. Abychom dokázali vytvářet draky, potřebujeme vybudovat v našem dungeonu dračí jeskyni. Cíl vybudování jeskyně lze rozložit na vytvoření budov, které je nutné mít před stavbou dračí jeskyně, a na zisk určitého počtu surovin.

Takto bychom mohli pokračovat dále. Při hraní myslíme úkolově. Vybereme jaký úkol je dle situace nejdůležitější. Rozdělíme si ho na menší podúkoly, které dále dělíme, pokud si to jejich složitost vyžaduje.

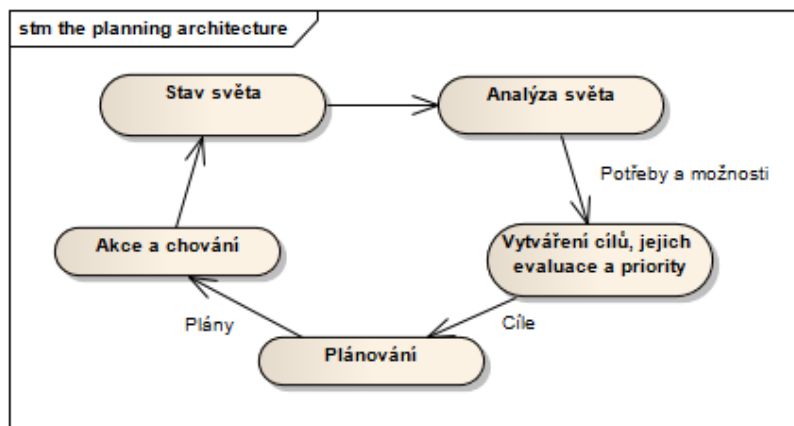
Úkoly jsou umístěny v určité hierarchii a buď jsou složené z jiných úkolů, nebo jsou již atomické. Atomickým úkolem by mohl být přesun hrdiny z jednoho místa do druhého, výměna jedné suroviny za druhou, přesun jednotek mezi hrdiny. Jsou to úkoly, které po důkladném naplánování děláme ve hře jako hráč. Tyto úkoly jsou ještě dělitelné na akce, o které se jako hráči nestaráme. Často jedna akce odpovídá jednomu úkolu. Například akce přesunu hrdiny z místa na místo vyžaduje vypočet nejrychlejší cesty a plynulý pohyb po ní. (Přestože se hráč pohybuje pouze z políčka na políčko, pohyb mezi políčky není dán skokem, ale plynulým přesunem.)

Když hrajeme, musíme často své naplánované úlohy měnit. Při delší cestě po mapě se na ní může objevit se začátkem nového měsíce příšera. To vede k přeplánování naší strategie. Pokud nás zaskočila příšera slabá, pravděpodobně pouze odložíme aktuální úkol na později. Zaútočíme na příšeru v cestě, zabijeme ji a vrátíme se k původnímu úkolu. V případě, že je příšera příliš silná a nemůžeme zamýšlený úkol splnit, zrušíme zcela naplánovaný úkol a na základě aktuální situace naplánujeme úkoly nové.

Metoda AI řízená úkoly (Goal driven) funguje přesně jako bylo výše popsáno. Máme jednotlivé cíle v hierarchické struktuře. K uložení se využívá návrhového vzoru Composite [13].

3.3.1 Fáze rozhodování

Rozhodování lze rozdělit do několika fází. Analýza světa, jejíž výstupem jsou potřeby a možnosti, vytváření cílů a jejich evaluace a priority, jejíž výsledkem jsou cíle (goals), plánování a z nich plány a nakonec akce a chování, jež vedou ve vytvoření nového stavu světa, hry.



Obr. 8 Stavový automat architektury plánování

3.3.2 Analýza hry

Při analýze hry získáme informace o světě (naše velikost a rozmístění armád, množství surovin, vyspělost jednotlivých hradů, doba, kdy se objeví nové příšery k najmutí atd.) a na základě nich určíme aktuální potřeby. Cílem této fáze je vždy sjednotit několik proměnných o světě a z nich určit, jak je velká potřeba. Na nás na vývojářích AI je určit, v jaké škále budeme ohodnocovat jednotlivé potřeby. Od 0 do 100, kde 100 je nejvyšší potřeba. Škálu můžeme mít také od 0,0 do 1,0, či od -50 do 50, kde záporné hodnoty jsou pro zápornou potřebu, kladné, pro kladnou. Ohodnocení potřeby je čistě na nás.

Je důležité si dát pozor na dvě věci. Konzistentní stupnice a absolutní výsledky. (consistent scale a absolute results) [10]

Konzistentní stupnice. Pokud se určí stupnice od 0,0 do 1,0, nesmí se stát, že nejvyšší hodnota potřeby, kterou bude vracet příslušná funkce, bude maximálně 0,6. Všechny potřeby musí být vyhodnocovány tak, aby se v některých případech potřeba rovnala minimální hodnotě na stupnici a za jiných podmínek maximální hodnotě.

Výsledky musí být absolutní. Pokud budou dva hráči ovládané AI v totožné situaci, tak v této fázi musí funkce potřeby vždy vracet pro oba stejné číslo. Zde by neměly být promítnuty individuální vlastnosti AI, např. defenzivní, či ofenzivní povaha hráče. Povaha hráče může být určena až v následující fázi této architektury.

Tato podkapitola byla zatím abstraktní, nyní je zde prostor pro konkrétní případ potřeby. Jedna z potřeb v HoMaM by mohla být potřeba Zlepšit obranu hradu. Pokud se jedná o hrad, který máme v rohu obrazovky, mezi ním a soupeřovými hrady máme hrad další a hrad je vyzbrojen mnoha silnými jednotkami, potřeba Zlepšit obranu hradu bude rovna 0,0. Naopak, jestliže je hrad strategickým místem, které bylo nově dobyto a nemá v sobě žádnou armádu, potřeba bude 1,0. Jak bude tato potřeba uspokojena, je na dalších fázích.

3.3.3 Vytváření cílů a jejich evaluace

V této fázi máme za úkol přehodnotit aktuální cíle (goals) AI na základě výsledků předchozí fáze a případně je doplnit o cíle nové.

První případ je poměrně přímočarý. U každého aktuálního úkolu přehodnotíme, jestli je opravdu ještě aktuální. Pokud hráč měl jako aktuální úkol Bránit hrad a dle funkce vyhodnocující potřebu Zlepšit obranu hradu, je tato potřeba už nízká, tak již úkol Bránit hrad není aktuální. Hrad už byl dostatečně posílen nebo byl dobyt další hrad v řadě. Cíl Bránit hrad je tedy splněn.

Dalším krokem je přidat cíle nové. Při jejich přidávání je třeba mít na paměti pár věcí. Ne každý nový cíl je založen jen na jedné potřebě. Cíl Bránit hrad není pouze závislý na potřebě Zlepšit obranu hradu, ale také na potřebě Zabít nepřátelského hrdinu v blízkosti hradu, jež je ovlivněna silou nepřátelského hrdiny. Při rozhodování se o přidání cíle Bránit hrad se budeme rozhodovat dle obou potřeb a na základě nich vytvoříme jedno číslo ve stejném rozsahu jako můžou být jednotlivé potřeby. Jedno číslo můžeme získat např. aritmetickým průměrem, či váženým průměrem, kde dáme váhu jednotlivým potřebám dle toho, jak ovlivňují daný cíl.

Poté bychom měli ořezat příliš nízké hodnoty. (Senzible cutoffs) Při vyhodnocování potřeby cíle potřebujeme určit minimální hodnotu, která musí být splněna, aby AI takový cíl brala vůbec v úvahu. Ořezávat musíme s rozvahou. Nemůžeme oříznout každý cíl, který má celkovou potřebu nižší než 0,5. Po této fázi bychom měli mít vyloučené pouze ty cíle, které bychom nebrali v úvahu v žádné ze strategií. (útočná, farmářská apod.)

Na konci této fáze máme seznam všech cílů, které stojí za to zvážit při dalším rozhodování se. V další fázi vybereme ten, který je nejvíce vhodný v danou chvíli.

3.3.4 Priorita cílů

V této fázi již rozhodujeme, jaké akce budeme dál vykonávat. Tato fáze je nejdůležitější, je srdcem rozhodovacího procesu. Zde zjišťujeme, co budeme dělat. Dále už jen zařídíme, jak to provedeme.

Cíle, které prošly minimalizačním sítím seřadíme podle ohodnocení jejich potřeb od nejvyšší hodnoty po nejnižší a zvolíme několik nejdůležitějších cílů, které přidáme k těm aktivním.

Ohodnocení cílů můžeme vyvážit pomocí multiplikačních konstant a tím upřednostňovat jednu akci před druhými. Zde máme místo pro vytváření různých AI s různými povahovými rysy. Velice jednoduše můžeme vytvořit hráče, který upřednostňuje defenzivní strategii, kdy nechce přijít o žádný z jeho hradů, a tak si ohlíká minimální velikost armády v každém hradu, nebo naopak chceme mít AI, která hrady nechává téměř prázdné, přenechává veškerou armádu hrdinům, kteří mají za úkol obsazovat další a další hrady. Mezitím jiné hrady ztratí.

K tomuto chování stačí hodnoty útočných cílů násobit konstantou větší než jedna a naopak hodnoty cílů související s obranou a farmařením násobit konstantou menší než jedna.

Pokud máme ve hře pouze několik málo cílů, je možné nastavovat pro každý cíl konstantu zvlášť. U větších her může být praktičtější seskupovat podobné cíle související se stejnou strategií a nastavovat jim hromadně jeden společný koeficient.

Jinou sadou koeficientů docílíme toho, že dva hráči se budou ve stejných podmínkách rozhodovat odlišně a bude to vést k větší zábavě při hraní. Jednotlivé mise kampaně můžeme rozlišovat nejen různým prostředím a jinými podkreslujícími úkoly, ale taky pokaždé můžeme nasadit hráče s jiným chováním, a tak prodloužit celkovou dobu hratelnosti.

Kromě různých strategií můžeme zakomponovat do hry rozdílné obtížnosti. Víme, že v naší hře je určité chování výhodnější než jiné, tak mu dáme větší priority u hráče, kterého využijeme pro nejtěžší úroveň hry. Naopak lehká úroveň bude více využívat cíle, které většinou nezajistí úspěch.

Popsané chování vybudované v této architektuře velice připomíná, jak se chová hráč. Hráč se chová úkolově. Pokud se chceme ještě více přiblížit simulaci lidského hraní, můžeme zde ke koeficientům přidat určitou náhodnost. Když vám jako hráči je dána stejná situace, také pokaždé nehrajeme stejně. Představme si např. hru šachy, kde by soupeř dělal prvních několik tahů pokaždé zcela stejně. Taková hra by nás brzy omrzela.

3.3.5 Plánování

Když AI ví, co bude dělat, ještě se musí rozhodnout, jak toho má dosáhnout. Náš příklad cíle Bránit město lze rozdělit do několika dílčích úkolů Postavit hradby, Najmout novou posádku, Stáhnout zpět nejbližšího hrdinu.

Tahle fáze by měla být jednodušší než vyhodnocování cílů. Často je již poměrně jasné, jak daného cíle dosáhnout.

U každého podúkolů musíme sledovat jeho stav. Když dojde k jeho splnění, můžeme ho ze seznamu aktivních úkolů vyřadit. Po splnění všech podúkolů cíle můžeme přidávat cíl další.

U některých úkolů se může stát, že již je nadále není možné splnit. Při vracení se nejbližšího hrdiny mu vstoupí do cesty jiný hrdina nebo neutrální příšera a on již nemůže posílit obranu. Nesplnitelnost úkolů musíme hlídat. U reálné FPS se často nepřátelé zaseknou do stěny a neustále se pokoušejí ji projít. Každý z úkolů by měl mít daný limit, po kterém se vyhodnotí jako neúspěšný a vymyslí se jiný plán.

Jak již bylo naznačeno, dosáhnoutí daného cíle lze často mnoha způsoby. Bude stačit stáhnout hrdinu, nebo musí postavit i hradby? I zde se opět rozhoduje obdobně jako v předchozí fázi. Každá z možností se ohodnotí a vybere se ta možnost, která má největší ohodnocení. Hodnotí se na základě aktuálních zdrojů, ceny možností, ale i dle osobnosti hráče.

3.3.6 Akce a chování

Poslední fáze je nejjednodušší ze všech. Už máme naplánováno, co uděláme, jakým způsobem toho chceme dosáhnout. Už zbývá pouze „fyzická“ vrstva, která je daná, u níž již AI nemusí „přemýšlet“.

Když přemístíme jednotku, tak už víme, že se s ní chceme dostat do hradu, už máme i naplánovanou cestu zpět. Zbývá zajistit samotný pohyb. Naprogramovat plynulý pohyb hrdiny mezi jednotlivými dvěma políčky, které jsou na jeho cestě. Se zatáčkami měnit natočení hrdiny, promítat jeho animaci, zrychlovat pohyb, když jede po kamenité cestě, zpomalovat při jízdě ve sněhu.

Uvedli jsme si jednotlivé fáze úkoly řízené AI dle [10]. Jak takovou architekturu implementovat, se dozvíme např. v [9].

3.4 Fuzzy logika

Ve svém životě často používáme vágní, nepřesné výrazy. Vezmeme-li si do ruky kuchařku, pravděpodobně bude plná nepřesných výrazů. Použit špetku soli, hrnek mléka, chvíli

smažíme dozlatova. Špetka, hrnek, chvilka, dozlatova jsou nejasně definované výrazy, přesto dle nich dokážeme udělat chutnou večeři.

Pokud hrajeme FPS, často volíme, kterou zbraň zrovna použijeme. Pokud bude soupeř blízko a my budeme mít brokovnici a dostatek nábojů do ní, pravděpodobně využijeme ji. Naopak v takové situaci nezvolíme raketomet, protože bychom zabili nejen soupeře, ale i sebe.

Ve strategii objevíme soupeřovu armádu pěšáků, která je středně velká a nachází se daleko od vaší základny. Začneme vyrábět svojí armádu takovou, aby obstála útoku soupeře.

V klasické booleovské logice bychom mohli mít následující sadu intervalových podmínek.

```

if(pocetVojaku < 20) then      velikostArmady = MALA;
else if(pocetVojaku < 50) then velikostArmady = STREDNI;
else                           velikostArmady = VELKA;

if(vzdalenost < 30) then      vzdalenostArmady = BLIZKO;
else if(vzdalenost < 70) then vzdalenostArmady = DAL;
else                           vzdalenostArmady = DALEKO;

if(velikostArmady = MALA AND vzdalenostArmady = BLIZKO) then VytvorVojaky(20);
if(velikostArmady = STREDNI AND vzdalenostArmady = BLIZKO) then VytvorVojaky(40);
if(velikostArmady = VELKA AND vzdalenostArmady = BLIZKO) then VytvorVojaky(60);
// Dalších šest pravidel. Všechny kombinace velikostArmady a vzdalenostArmady.
    
```

Kód 3 Rozhodování v booleovské algebře.

Jestliže se bude k naší základně pohybovat armáda o 20 vojáků a bude ve vzdálenosti 29, situace bude vyhodnocena jako středně velká armáda, která je blízko, a tedy se začne vytvářet 40 nových vojáků. Kdyby v soupeřově armádě bylo o vojáka méně, začalo by se rekrutovat o polovinu méně vojáků. Je zde přesné ohraničení, co je malá a co velká armáda. Člověk by situace o 19, či 20 vojáků vyhodnotil stejně.

Danou situaci lze lépe vyřešit fuzzy logikou. Fuzzy lze do češtiny přeložit jako nejasný, neurčitý, neostrý. V našem příkladu jsme měli dvě množiny. Jednu pro velikost armády obsahující prvky MALA, STREDNI, VELKA a druhou množinu pro její vzdálenost též o třech prvcích BLIZKO, DAL, DALEKO. V booleovské algebře jsme situaci o 20 vojáků vyhodnotili tak, že situace je ze 100% STREDNI armáda, z 0% MALA a VELKA armády.

Jsou situace, kdy nám sdělení, že prvek plně patří do skupiny, či vůbec nepatří, přijde přirozené. Např. číslo 7 patří ze 100% do množiny lichých čísel a z 0% do množiny čísel sudých.

Proces fuzzy logiky použité v umělé inteligenci lze rozdělit do tří fází. Fuzifikace, použití Fuzzy pravidel a defuzifikace (*fuzzification, fuzzy rules, defuzzification*).

Fuzifikace nám připraví data z ostrých množin (*crisp set*) do fuzzy množin (*fuzzy set*). Nad fuzzy množinami se provedou fuzzy pravidla. Po jejich aplikaci máme stále fuzzy množiny a z nich je potřeba opačným postupem získat ostrá data.

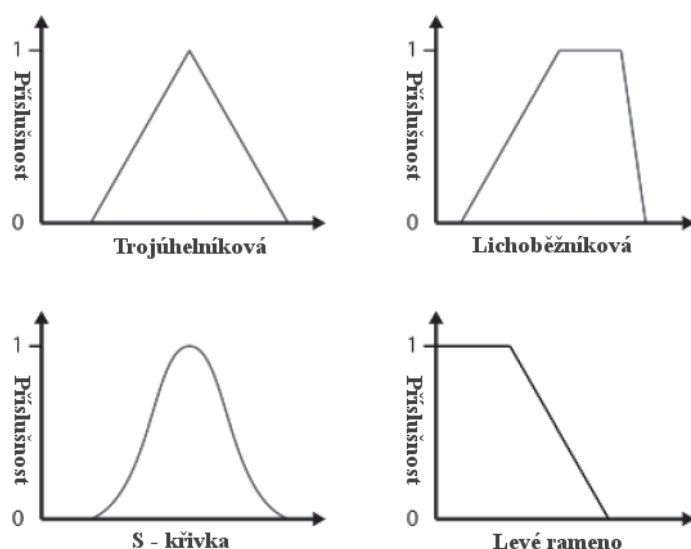
3.4.1 Fuzifikace

U fuzzy množin je důležitá příslušnost k prvku množině. To je přesně to, co potřebujeme. Chceme říct, že armáda o 20 vojácích je napůl MALA a napůl STREDNI. Příslušnost k prvku v množině vyjadřujeme číslem od 0,0 do 1,0. Proměnné velikostArmady a vzdálenostArmady se ve fuzzy logice označují jako *fuzzy linguistic variable* (FLV).

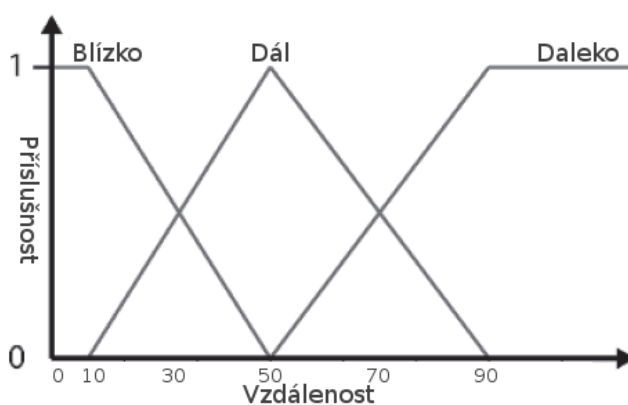
$$velikostArmady = \{MALA, STREDNI, VELKA\}$$

$$vzdálenostArmady = \{BLIZKO, DAL, DALEKO\}$$

Příslušnost k proměnné určuje funkce příslušnosti (*membership function*). Několik základních druhů si lze prohlédnout na Obr. 9. Po spojení všech funkcí příslušnosti jednotlivých prvků vznikne funkční manifold. Pro FLV vzdálenostArmady by mohl vypadat obdobně jako na Obr. 10.



Obr. 9 Základní typy funkcí příslušnosti. Zdroj [10].



Obr. 10 Manifold funkcí příslušnosti FLV vzdálenostArmady. Zdroj [10].

Na základě grafu z Obr. 10 probíhá fuzifikace ostré množiny. Např. pokud máme armádu ve vzdálenosti 80 (např. v počtu hex), uděláme svislou přímku na ose x v 80 a zaznamenáme, které prvky fuzzy množiny v jaké výšce přímka protne. Zde můžeme říci, že příslušnost vzdálenosti armády k BLIZKO je 0,0 (její graf neprotíná), DAL 0,25 a DALEKO 0,75.

Tvorba manifoldů je na citu a umění vývoje. Každý problém vyžaduje jiné funkce příslušnosti, jinak strmé apod. Při jejich tvorbě je nutné dodržet dvě důležitá pravidla. Každá svislá přímka by měla protnout maximálně dvě funkce příslušnosti. Graf z Obr. 10 by nesměl být navržen tak, že by aktuální vzdálenost armády mohla mít příslušnost k BLIZKO i k DALEKO větší než 0,0. Druhé pravidlo, součet příslušností k jednotlivým prvkům musí být vždy alespoň přibližně roven 1. Špatně by bylo, kdyby v jednu chvíli byla armáda ze 0,75 DAL a zároveň ze 0,80 DALEKO.

Pokud chceme zjistit, jestli je armáda zároveň BLIZKO a DAL, či jestli je jedno z DAL nebo DALEKO, musíme použít spojky AND a OR známé z booleovské logiky. Je více způsobů, jak určit operace AND či OR. Operace AND musí být t-norma. Binární operace na intervalu $< 0, 1 >$ je t-norma, pokud splňuje pravidla komutativity, asociativity, je neklesající a 1 je její jednotkový element. Jedním z příkladů t-normy je obyčejné minimum.

$$A \text{ AND } B = \text{MIN}(A, B)$$

$$0,5 \text{ AND } 0,2 = 0,2.$$

Obdobně operace OR musí být t-konorma. Nejznámějším t-konormou je maximum.

$$A \text{ OR } B = \text{MAX}(A, B)$$

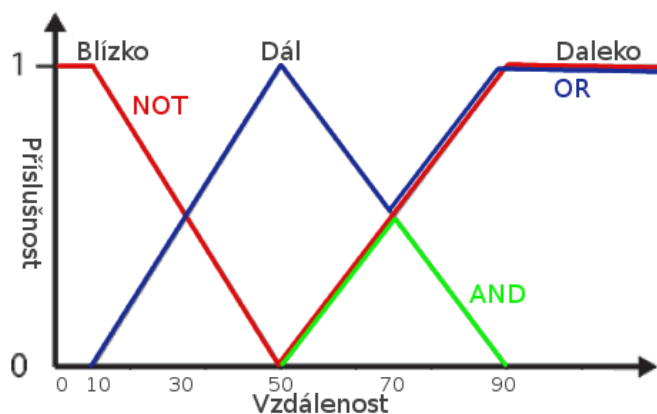
$$0,5 \text{ OR } 0,2 = 0,5$$

Unární operace NOT, negace je doplňkem do jedné.

$$\text{NOT } A = 1 - A$$

$$\text{NOT } 0,3 = 1 - 0,3 = 0,7$$

Vizuální znázornění operací AND, OR, NOT si můžeme prohlédnout na Obr. 11. Červenou barvou je zvýrazněno NOT DAL, modrou barvou DAL OR DALEKO a nakonec zelenou barvou DAL AND DALEKO.



Obr. 11 Grafické znázornění operací AND, OR, NOT.

Oproti booleovské logice se zde objevují nové unární operace (*hedges*) VERY a FAIRLY. Unární operaci VERY použijeme, chceme-li říct, že něco velmi patří do konkrétní skupiny.

$$\text{VERY } A = A^2$$

Opakem VERY je FAIRLY. Patří nějaká ostrá hodnota téměř do fuzzy hodnoty, je v její blízkosti, použijeme právě operaci FAIRLY.

$$\text{FAIRLY } A = \sqrt{A}$$

3.4.2 Fuzzy pravidla

Zápis fuzzy pravidel je velice obdobný lidské řeči. Pokud budeme spolupracovat při vývoji umělé inteligence s expertem na naši hru a budeme zapisovat, podle jakých pravidel se on rozhoduje, nemusí se tato pravidla lišit od těch, které zapíšeme do počítače.

Od experta bychom mohli získat pravidla obdobná následujícím :

POKUD je nepřítel hodně napravo, POTOM se otočím rychle vpravo. POKUD je soupeř velmi zraněn, POTOM na něj bezhlavě zaútočím. POKUD se soupeřova armáda dostane blízko mé základny A je středně velká, POTOM začnu vyrábět hodně vojáků. Na základě podobných informací si zvolíme fuzzy proměnné, vytvoříme funkční manifoldy a především fuzzy pravidla obdobná z Kód 4.

IF enemy_farRight THEN turn_quicklyRight
 IF very(enemy_badlyInjured) THEN attack
 IF velikostArmady_stredni AND vzdalenostArmady_blizko) THEN vytvorVojaky_hromadu

Kód 4 Ukázka několika fuzzy pravidel.

Poslední z pravidel se shoduje s pravidlem z Kód 3. Liší se pouze částí za THEN. U fuzzy pravidel se za THEN neobjevuje přesné číslo (VytvorVojaky(40)), ale též fuzzy proměnná FLV.

$$\text{vytvorVojaky} = \{PAR, SKUPINU, HROMADU\}$$

Všechna pravidla se skládají ze dvou částí, předchůdců (*antecedent*), jež je před částí THEN a následovníků (*consequent*), který je za THEN. Zde jsou antecedenty FLV velikostArmady a vzdalenostArmady, konsekvent vytvorVojaky. Při tvorbě pravidel dobře poslouží obyčejná tabulka, viz Tabulka 2.

Tabulka 2 Fuzzy pravidla pro FLV vzdalenostArmady a velikostArmady

VelikostA\VzdalenostA	BLIZKO	DAL	DALEKO
MALA	SKUPINU	PAR	PAR
STREDNI	HROMADU	SKUPINU	PAR
VELKA	HROMADU	HROMADU	SKUPINU

Při vyhodnocování příslušnosti ke konsekventům vyhodnotíme všechna pravidla. Ve fázi fuzifikace bylo popsáno, jak získat příslušnosti k antecedentům. Spočteme příslušnost velikosti armády k hodnotě STREDNI 0,27 a vzdálenosti armády k hodnotě BLIZKO 0,21.

IF velikostArmady_stredni AND vzdalenostArmady_blizko) THEN vytvorVojaky_hromadu

Po dosazení vyjde příslušnost k HROMADU 0,21. (spojka AND funguje jako minimum z hodnot)

Tabulka 3 Příslušnosti ke konsekventům vytvorVojaky

VelikostA\VzdalenostA	BLIZKO	DAL	DALEKO
MALA	0,21	0,73	0,0
STREDNI	0,21	0,27	0,0
VELKA	0,0	0,0	0,0

Příslušnost k PAR je 0,73, k HROMADU 0,21. Prvek SKUPINU má v tabulce dvě nenulová zastoupení. Jednou z možností je udělat OR nenulových hodnot, který se ve fuzzy logice rovná maximu z hodnot. Příslušnost ke SKUPINU je 0,27.

3.4.3 Defuzifikace

Posledním krokem je defuzifikace. V našem příkladu jsme získali následující hodnoty :

PAR	0,73
SKUPINU	0,27
HROMADU	0,21

Z těchto dat musíme získat hodnotu z ostrých množin. Zde bude mít získaná hodnota význam počet vojáků, kteří se mají vytvořit.

Existuje více možných postupů, více nalezneme např. v [10] nebo v [15]. Jednou z nejjednodušších variant je každou z hodnot FLV konsekventu vážit předem zvolenými hodnotami. Zvolme si koeficienty 5, 15, 40, kterými budeme vážit PAR, SKUPINU, HROMADU.

$$pocetVojaku = 5 * kPAR + 15 * kSKUPINU + 40 * kHROMADU$$

V našem ukázkovém příkladu vyjde $5 * 0,73 + 15 * 0,27 + 40 * 0,21 = 16$. V aktuální situaci potřebujeme vytvořit 16 vojáků.

4 Inspirováno přírodou

V mnohých odvětvích se člověk učí z přírody. Při vytváření prvních letadel se zkoumal pohyb ptáků. Při vynalézání supersilných tenkých vláken se analyzují pavoučí sítě. Stavitelé budov hledají vzor v přírodě. Jinak tomu není u počítačů a algoritmů umělé inteligenci.

Genetické algoritmy a neuronové sítě jsou jedněmi z nejpoužívanějších zástupců těchto algoritmů. První ze zmíněných mají základ v evoluci, neuronové sítě v nervové soustavě živočichů. Zvláštní kapitolou je Umělý život, hry založené na simulace života a společnosti.

4.1 Genetické algoritmy

4.1.1 Evoluce v přírodě

Genetické algoritmy patří k algoritmům inspirovaných přírodou, zde konkrétně evolucí. V přírodě přežijí pouze nejsilnější živočichové, obecněji živočichové lépe přizpůsobení prostředí, v kterém žijí. Je-li myš rychlejší než ostatní, má o něco větší šanci přežít, spářit se a přenést své geny na potomstvo. Geny obou myších rodičů se zkříží a vznikne nový potomek, který zdědí vlastnosti úspěšných rodičů.

Samotné křížení genů vybraných lepších jedinců není jediným důležitým prvkem v evoluci. Je dobře známo, že nejdříve žili živočichové pouze ve vodě a až později se přesunuli na souš a byli schopni dýchat atmosférický vzduch. Kdyby bylo v přírodě pouze křížení, tak by kombinací genů zodpovědných za tvorbu žaber nikdy nevznikly geny pro vznik plic. K tomu je zapotřebí mutace, která vznáší do DNA potomka geny, jež neměl ani jeden z jeho rodičů.

Často vlivem mutace vzniknou jedinci, kteří nejsou schopni v přírodě dlouho přežít. Příkladem mohou být albíni, tedy živočichové, kteří místo maskující barvy srsti mají srst

bílou. Pro člověka to působí jako nádhera, ale v přírodě je to spíše na obtíž. Albín se mnohem hůře schovává před predátory.

Avšak čas od času mutace dokáže vytvořit nového jedince schopnějšího přežít v daném prostředí, a tedy tím přenášet nově vzniklý gen do dalších generací.

Tímto ukončíme krátké připomenutí biologie ze střední školy. Genetické algoritmy můžeme využít k mnohým účelům, nejsou úzce spjaty s umělou inteligencí.

Pomáhají nalézt řešení daného problému, ale nezaručují nalezení nejlepšího řešení, ani nalezení nějakého řešení.

4.1.2 Hledání cesty pomocí genetického algoritmu

Zkusme vyřešit problém hledání cesty mezi dvěma místy pomocí genetického algoritmu. Budeme hledat cestu v 2D mřížce mezi dvěma čtverci skrz bludiště. Máme povoleny 4 směry pohybu, tedy nemáme povolen diagonální pohyb.

Hledané řešení genetickým algoritmem bude posloupnost příkazů nahoru U, vpravo R, dolů D, vlevo L, která dovede hráče ze startu do cíle. Hledaná posloupnost bude vyšlechtěným potomkem vzniklým křížením a mutací jiných posloupností příkazů, které sice nevedly k cíli, ale postupně se mu přibližovaly.

Zbývá určit, podle čeho vybírat úspěšné a neúspěšné jedince. Kterým dát možnost se křížit a přiblížit se hledanému řešení, a které nekompromisně zahodit. Pro tento specifický problém bude úspěšnost organismu dána vzdáleností od cíle, kam by se hráč dostal, kdyby se pohyboval dle posloupnosti instrukcí. Čím menší vzdálenost, tím lepší. Pokud nulová, našli jsme řešení. Zde nutno podotknout, že nalezená cesta nemusí být nejkratší, tímto způsobem se může nalézt nějaká cesta mezi startem a cílem.

Algoritmus bude fungovat následovně. Na začátku si určíme, s jak velkým potomstvem budeme pracovat. Mějme např. populaci o 100 kusech. Každý kus na začátku inicializujeme náhodnou posloupností URDL a vypočítáme mu hodnotu *fitness* (jak moc je úspěšný při hledání cíle). Z těchto 100 kusů vybereme dvojice, které se budou křížit a mutovat, a tak vytvářet novou generaci potomků, jimž se určí nová hodnota *fitness*. Následně budeme kroky selekce dvojic, křížení a mutace, vznik nových potomků opakovat v jednotlivých generacích dokud nevznikne potomek s ideální hodnotou *fitness*, tedy ten, který řeší úlohu.

4.1.3 Selektce párů

Je více způsobů, jak vybírat vhodné dvojice pro páření. První, co by asi každého napadlo, kombinovat pouze ty nejlepší. Což na první pohled může vypadat jako skvělý nápad, ale trpí nedostatkem, že může nalézat pouze lokálně nejlepší řešení, ne globálně. V našem případě to znamená, že algoritmus se ztratí ve slepé uličce, která končí blízko cíle, ale před cílem je zeď. Nebere to vůbec v úvahu možnost, že hledaná cesta k cíli směřuje od startu nejdříve směrem od cíle. K tomuto typu selekce patří elitářství (*elitism*), kde je zaručeno, že n nejlepších kusů bude zachováno do další generace, či selekce setrvalého stavu (*steady state selection*), kdy se do další generace zanechá např. 4/5 populace a zbylá pětina se vytvoří křížením.

Druhým způsobem je výběr proporcionálně k úspěšnosti jedince. Čím úspěšnější jedinec (větší *fitness*), tím má větší šanci, že bude vybrán ke křížení se. K této metodě patří selekce ruletou (*roulette wheel selection*). Kolo rulety je rozděleno na n výřezů dle velikosti populace, v našem případě 100. Velikost výřezu je dána velikostí *fitness* daného potomka. Pokud má jeden potomek *fitness* 2 a druhý 6, tak ten s *fitness* 6 má třikrát větší výřez na kole rulety než

ten s fitness 2 a tím i třikrát větší šanci, že bude vybrán pro křížení se. Nevýhodou tohoto řešení je, že nemáte jistotu výběru nejlepších jedinců. Může se s malou pravděpodobností stát, že řešení blízké cíli bude zahazeno. Dobré je tento přístup kombinovat s předchozím, vybrat např. 5 nejlepších, kteří mají jistotu přežití do další generace populace a se zbytkem provést selekci ruletou.

Dalším typem selekce je selekce turnajem (*tournament selection*), která poměrně úspěšně eliminuje nevýhody předchozích dvou. Při výběru potomka ke křížení se náhodně vybere ze všech potomků n jedinců a z nich se vyberou dva nejlepší(s největším fitness), kteří se zkříží.

4.1.4 Kombinace křížením

Ke křížení nedochází vždy po výběru dvou jedinců. Jestli ke křížení dojde, určíme zvolenou hodnotou. Příkladem může být 0,7. Ze 70% je šance, že dojde ke křížení, ze 30% dojde pouze k zachování rodičů ze současné generace do generace následující.

I zde můžeme vymyslet mnoho způsobů, jak křížit potomstvo. Křížení i mutace se odvíjí od toho, jak máme zakódované jednotlivé jedince. U hledání cesty jsem zvolil výčet 4 prvků URDL, které mohou být reprezentovány v programu celými čísly.

Mějme dvě zkrácené trasy RDDLRULRU a LUDDLRLUR. Na těchto trasách ukážeme několik možností jejich křížení.

Křížení jedním bodem (*single-point crossover*). Zvolíme náhodně pozici v řešení. Od té pozice řešení dvou křížených jedinců roztrhneme a konce řešení zaměníme mezi sebou. Např. pro pozici 5 v našem příkladě :

RDDLRLURU a **LUDDLRLUR**

X

RDDL**RLUR** a **LUDD**RULRU

Křížení dvěma body (*two-point crossover*) je obdobné. Zvolíme v řetězci dvě pozice, místo mezi nimi vystříháme a zaměníme mezi sebou. Např. pro náhodně zvolené pozice 2 a 5.

RDDLRLURU a **LUDDLRLUR**

X

LUDDLULRU a **LDDLRLUR**

Nic nebrání v tom si zvolit bodů více (*multi-point crossover*), a tak zaměňovat libovolné kousky řetězců.

RDDLRLURU a **LUDDLRLUR**

X

R**DDL**UL**UU** a **LU**DL**RLRR**

4.1.5 Mutace

Stejně jako v přírodě i v genetických algoritmech nedochází k mutacím při každém křížení. Pravděpodobnost mutace se nastavuje obdobně jako pravděpodobnost křížení, ale zde na mnohem menší hodnotu. Pravděpodobnost může být v jednotkách promile, tedy např. 0,005.

Nejjednodušším typem mutace může být záměna jednoho či více příkazů za jiné.

RDDLRLURU > **UL**DLRLURU.

Jinou mutací může být prohození jednoho příkazu s jiným, skupiny příkazů s jinou. RDDLRULRU > RUDLRDLRU. Můžeme i otočit pořadí několika prvků RDDLRULRU > RDDURLRU.

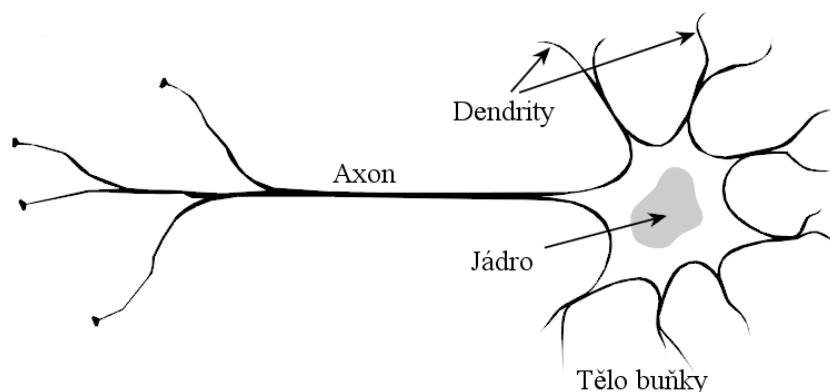
Způsob mutace, či křížení je ponechán pouze naší představivosti. Můžeme si vymyslet jiné způsoby, jež jsme neuvedli.

Mat Buckland v [8] tvrdí, že být dobrý v genetickém programování není jen věda, ale také umění. Mimo obdobného příkladu hledání cesty můžeme v jeho knize nalézt příklad využití genetického algoritmu pro řešení problému obchodního cestujícího, či pro ovládání lunárního vozítka ze známé hry Moon Lander.

Tento příklad byl pouze ukázkovým vhodný pro vysvětlení základních principů genetických algoritmů. Genetické algoritmy se nevyužívají pro hledání cesty, proto existují jiné algoritmy, jež zmíním v kapitole 5.1 Hledání cest.

4.2 Neuronové sítě

Neuronové sítě se stejně jako genetické algoritmy inspirovaly přírodou, konkrétně nervovou soustavou, jak již název obsahující slovo neuron napovídá. Opět začneme tuto kapitolu s krátkým připomenutím biologie.



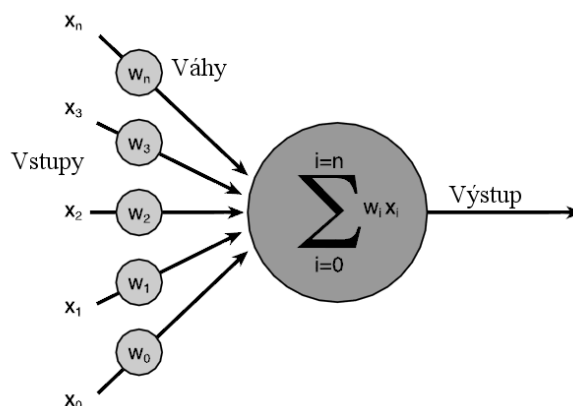
Obr. 12 Schéma neuronu. Zdroj [14].

Základní jednotkou v nervové soustavě je neuron. Člověk má ve svém těle přibližně 10^{10} neuronů, oproti němu šnek jich má pouze 10^4 . [14] Každý neuron lze rozdělit na tělo (soma), do něhož vstupují krátké dostředivé dendrity a axon, který z neuronu vystupuje. Neuron je propojen s dalšími neurony právě přes dendrity a axon. Axon se na svém konci větví a vstupuje do jednoho či více neuronů spojen s jejich dendrity. V lidském těle je každý neuron spojen s průměrně 10 000 jinými neurony.

Když se v těle šíří signál, přichází do těla buňky přes jednotlivé dendrity. V neuronu vzniká akční potenciál a neuron rozhodne, jestli signál přepošle po axonu do dalších neuronů, nebo ne. Nehraje zde roli velikost vyslaného signálu, jde pouze o binární informaci, buď se signál přepošle dál, nebo nikoli. Co rozhoduje o šíření signálu je již nad rámec tohoto textu. Takto zjednodušené fungování nervové soustavy bude pro nás dostačující.

4.2.1 Umělý neuron

Umělý neuron je dosti podobný tomu organickému. Model umělého neuronu lze graficky znázornit jako je tomu na Obr. 13.



Obr. 13 Schéma umělého neuronu. Zdroj [14]

Neuron (od teď neuronem máme namysli umělý neuron) má vstupy, které můžeme označit x_1 až x_n , kde n je počet vstupů neuronu. Každý vstup neuronu má svou váhu w_i . Akční potenciál se počítá jako skalární součin vektoru vstupů a vektoru vah.

$$a = \sum_{i=1}^n x_i w_i$$

Nyní musíme určit, zda-li neuron pošle signál dál, či nikoliv. Jestli na jeho výstupu bude hodnota nula, nebo jedna.

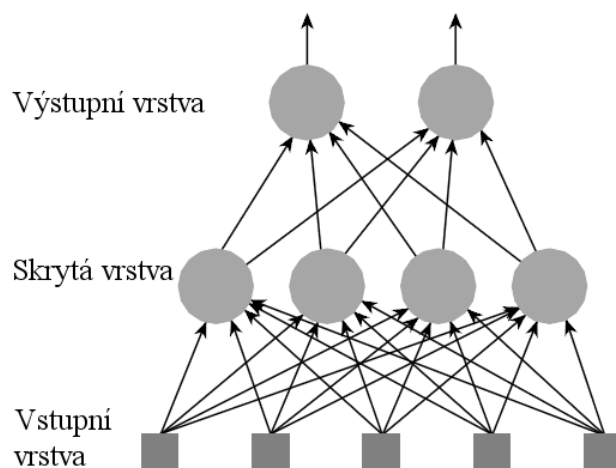
Akční potenciál je vstupem nelineární přenosové funkce. Pokud chceme na výstupu pouze binární hodnotu 0, či 1 a nic mezitím, použijeme skokovou Heavisidovu funkci, jež pro kladné vstupy vrací jedničku, pro záporné nulu.

Výstup neuronu y lze definovat jako $y = H(a)$. K neuronům můžeme přiřadit aktivační práh, mnohdy nazývaný bias, který se odečte od hodnoty akčního potenciálu a . $y = H(a - \Theta)$.

Spojíme-li všechny uvedené vzorce do jednoho, místo Heavisideovy funkce použijeme obecnou funkci S , získáme následující matematický vzorec neuronu.

$$y = S\left(\sum_{i=1}^n x_i w_i - \Theta\right)$$

V umělé neuronové síti se často neurony skupí do vrstev. Vždy máme minimálně jednu vstupní vrstvu, jednu výstupní vrstvu. Mezi těmito základními vrstvami se můžou nalézat další vrstvy skryté. Každý neuron je spojen se všemi neurony z následující vrstvy. V rámci jedné vrstvy nejsou neurony spolu spojené.



Obr. 14 Vrstevnatá neuronová síť. Zdroj [14]

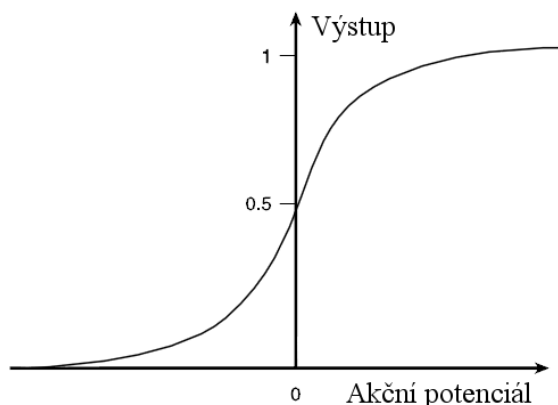
Vstupy a výstupy neuronové sítě jsou specifické dle jejich použití.

Jedno z možných využití neuronových sítí je oblast robotiky. Mějme roboty, jež se pohybují pomocí dvou pásů. Pokud se oba pásy točí stejným směrem, robot se pohybuje kupředu, či dozadu. Když se pohybují proti sobě s různou rychlostí, robot zatačí do jedné ze stran. Tito roboti mají za úkol sbírat mince. Neuronové sítě budou jejich mozkem.

Požadovaným výstupem budou vektory rychlostí pro jednotlivé pásy. Můžeme si všimnout, že vektory rychlostí nemohou nabývat pouze binárních hodnot. V takové síti nemůžeme využít skokovou Heavisideovu funkci. Jinou alternativou je logistická funkce (sigmoida), jež se velice často používá právě v neuronových sítích.

$$y = \frac{1}{1 + e^{-\frac{a}{p}}}$$

Hodnota parametru p určuje strmost funkce. Jestliže se p přibližuje číslu 0, tvar sigmoidy je obdobný skokové funkci.



Obr. 15 Tvar sigmoidy. Zdroj [14].

Vstupními hodnotami mohou být vektory směru robota a vektor k nejbližší minci. Velikost vektoru směru je 1, velikost vektoru k nejbližší minci může být mnohonásobně větší. U neuronových sítí je potřeba, aby se vstupy pohybovaly ve stejném měřítku. V tomto případě stačí druhý vektor normalizovat.

4.2.2 Pracovní fáze umělé neuronové sítě

Neuronová síť pracuje ve dvou fázích. Tyto fáze jsou dvě, adaptivní a aktivní. V adaptivní fázi se síť učí a v aktivní se vybavuje naučená činnost. Síť se učí upravováním vah na vstupech neuronů a případně upravováním aktivačního prahu, biasu. Síť je naučená, jestliže jsou váhy a bias nastaveny tak, že pro jednotlivé vstupy získáváme požadované výstupy.

Učení lze rozdělit na dva typy. Učení se s učitelem (*supervised training*) a učení se bez učitele (*unsupervised training*).

Při učení s učitelem máme vždy k dispozici sadu dat, vstupů a očekávaných výstupů. Pro náš příklad s roboty si můžeme připravit sadu vstupů, základní vektory směrů robota a směrů k mincím a k nim určit ideální rychlosti pásů jako výstupy. Při učení se vezmeme připravené vektory, předložíme je síti a na základě rozdílu očekávané a získané hodnoty se upravíme váhy. Jednou z metod učení je učení zpětné, jež popíšeme v další podkapitole.

U učení se bez učitele nemáme žádnou sadu vstupů a očekávaných výstupů. Nemáme zde vnější kritérium správnosti. Algoritmus hledá ve vstupních datech vzorky se společnými vlastnostmi. Jiný název pro učení se bez učitele je samoorganizace. Učení se bez učitele se nebudeme dále věnovat.

Ve fázi vybavování se váhy neupravují. Využívá se naučené neuronové sítě. Neuronové sítě fungují na základě generalizace. Dokážou se rozhodovat na základě vstupů, jež jsou podobné známým vstupům.

Musíme si dát pozor, abychom fázi učení ukončili včas a síť se nám nepřeučila. Přeučení znamená, že pro známé vstupy a výstupy fáze dosahuje ideálních výsledků, ale pro vstupy od nich lehce odlišné, vrací výstupy zcela chybné. Schopnost generalizace by byla poškozena.

4.2.3 Back-propagation

Zpětné učení (*back-propagation*) je známou metodou pro učení se s učitelem. Algoritmus lze rozepsat do několika kroků. [15]

1. Připrav sady vstupních dat a jejich požadovaných výstupů
2. Inicializuj všechny váhy v síti na malé náhodné hodnoty
3. Každou sadu využij jako vstup sítě a spočti výstup
4. Porovnej výstup s požadovaným výstupem a spočti chybu
5. Uprav váhy tak, aby se chyba snížila a opakuj kroky 3 – 5

Jednomu cyklu kroků 3 – 5 se zpravidla říká epocha. Kroky 1 – 3 jsou poměrně jednoduché, věnujme pozornost posledním dvěma krokům.

Chyba, jindy nazývaná jako energetická funkce, se počítá jako průměr čtverců odchylek získaných dat a dat očekávaných. Někdy je srozumitelnější vzorec než jeho popis :

$$\varepsilon = \frac{1}{2} \frac{\sum (n_v - n_o)^2}{m}$$

Počet výstupů je m , n_v je hodnota vypočítaná u jednoho vstupu, n_o je hodnota očekávaná. Čím nižší ε , tím je chyba menší a síť je naučenější.

Dalším krokem je vypočítat chybu pro jednotlivé výstupy.

$$\delta_i^0 = \Delta n_i^0 f'(n_{vi})$$

Zde je δ_i^0 chyba na i-tém výstupu neuronu. Δn_i^0 je rozdíl požadovaného a získaného i-tého výstupu, $f'(n_{vi})$ je derivace přenosové funkce. Zde je důvod, proč přenosová funkce nesmí být lineární, protože její derivace by byla konstanta a nebyla by zde závislost na vypočteném i-tém výstupu n_{vi} .

Pokud za přenosovou funkci dosadíme sigmoidu, výpočet chyby i-tého výstupu bude vypadat následovně :

$$\delta_i^0 = (n_{oi}^0 - n_{vi}^0) n_{vi}^0 (1 - n_{vi}^0)$$

Chyba výstupu neuronů ze skrytých vrstev se počítá jiným vzorcem.

$$\delta_i^h = \left(\sum \delta_j^{h-1} w_{ij} \right) f'(n_{vi}^h)$$

Opět dosadíme za f sigmoidu.

$$\delta_i^h = \left(\sum \delta_j^{h-1} w_{ij} \right) n_{vi}^h (1 - n_{vi}^h)$$

Tento vzorec si zaslouží vysvětlení. δ_i^h je chyba i-tého neuronu v h-té vrstvě. $\sum \delta_j^{h-1} w_{ij}$ je suma chyb z následující vrstvy vážená jejich váhami. Pokud se jedná o vrstvu těsně před vrstvou výstupní, spočítá se zde suma vážených chyb z výstupní vrstvy. n_{vi}^h je vypočítaný výstup i-tého neuronu ve vrstvě h.

Z tohoto vzorečku můžeme vyčíst, že při výpočtu chyb na jednotlivých neuronech, postupujeme od výstupní vrstvy ke vstupní. Od toho je odvozen název techniky učení zpětná propagace.

Poslední částí pátého kroku je spočítat, o kolik se změní vstupní váhy na jednotlivých neuronech.

$$\Delta w_i = \rho \delta_i n_{vi}$$

δ_i je vypočtená chyba na i-tém neuronu, n_{vi} vypočítaný výstup na daném neuronu a ρ je koeficient učení. Nová váha $w_i = w_i + \Delta w_i$.

Vzorec pro výpočet rozdílu vah může být rozšířen o momentum, které zamezuje získání lokálních extrémů před globálními. Více v [15].

4.3 Umělý život

Umělý život (*Artificial Life, A life*) je dalším ze způsobů, který můžeme využít pro tvorbu AI, ale i jako základní princip hratelnosti. Jedná se o simulaci života, kde hráč přímo, či nepřímo ovládá jednu, nebo několik životních forem (lidí, zvířat, příšer), které bez zásahu hráče dokážou ve hře samostatně myslet, pohybovat se, žít.

Prapůvod umělého života je v Conwayově simulaci Game of Life. Jde o svět tvořený 2D mřížkou. Každá buňka mřížky představuje buď živou, nebo mrtvou buňku. Život se v tomto světě rozvíjí v tazích, kolech, kde několik pravidel určuje, jestli v následujícím tahu vznikne nová buňka na místě, kde žádná není, nebo zanikne tam, kde je, nebo jen stávající buňka bude pokračovat ve svém životě. Pravidla pro život v dalším kole jsou celkem čtyři a odvíjí se pouze podle počtu živých buněk v okolí buňky v kole současném.

Přestože se jedná o jednoduchou simulaci, bylo v tomto světě o 4 pravidlech objeveno několik stovek vzorů skupin buněk se specifickým chováním rozdělených do několika kategorií. Patří

jsem např. glider, skupina buněk, která vlivem zanikání a vzniku nových buněk působí dojmem, že se celá skupina pohybuje jedním směrem. [6] Game of life není stále zcela prozkoumána, každým rokem vědci z celého světa objevují nové vzory živočichů.

Simulaci života můžeme vidět v mnohých hrách, které rozdělíme do několika skupin. [7]

4.3.1 Mazlíčci

Žánr, kde se staráme o své virtuální zvířátko, reálné, či smyšlené. Musíte ho krmit, cvičit, za prohřešky trestat. Zvíře je schopné se od hráče učit. Mazlíčci jsou většinou roztomilí a svými emocemi, chováním dávají hráči najevo, co s nimi má dělat.

Oproti jiným A-Life hrám se zvířátka často nemůžou rozmnožovat, umírat. Neznamená to ale, že nemůžete v takových hrách dělat špatná rozhodnutí. Pokud se chováte špatně ke svěřenému zvířeti, může od vás utéct.

Mezi známé zástupce tohoto žánru patří např. Tamagotchi, Nintendogs nebo Neopets.

4.3.2 Společenské simulace

Kdo by neznal hru The Sims, která odstartovala tento žánr. Úkolem je starat se o jednu, či více osob, které mají několik základních potřeb (hygiena, hlad, zábava, potřeba wc, kamarádi apod.), jež je třeba udržovat. Na rozdíl od starání se o mazlíčky jsou v těchto hrách důležité sociální vazby s ostatními osobami v rodině a v sousedství, které je třeba udržovat a rozvíjet pro lepší pokrok ve hře.

4.3.3 Hry na Boha

Populous, Black and White, Dungeon Keeper jsou zástupci tohoto žánru. V Dungeon Keeperu se impové sami od sebe starají o zpevnování stěn, těžbu drahokamů, či zabírání nových území. My je jako představitelé „Boha“ můžeme trestat, obětovávat, dávat jim nepřímé rozkazy, nebo je jen sledovat. I ostatní příšerky žijí svým životem, chodí spát, jíst, cvičit se a nechávají si od nás zaplatit za obranu našeho království. My je můžeme vzít a přesunout tam, kde jsou zrovna potřeba. Např. poslat je objevovat nová kouzla, nebo zdokonalovat se v útoku.

4.3.4 Evoluční hry

Hry, v nichž se staráme o populace několika generací, které se množí, vylepšují, přenášejí nové lepší vlastnosti do dalších generací. V těchto hrách můžeme nechat organismy se vyvíjet vlastní cestou, nebo můžeme do vývoje zasahovat změnou prostředí, či přidáním nově navržených vlastních forem života.

Populárními hrami jsou např. Spore, či Evolution : The Game of Intelligent Life .

4.3.5 A-Life a tahové strategie

Původní experiment Game of Life se odehrává v jednotlivých krocích, ale stěží to lze označit za hru, jde pouze o matematický experiment.

Nenalezneme žádnou tahovou strategii spojovanou s umělým životem. Důvodem může být, že ve všech zmíněných hrách bylo podstatné, aby se bez hráčova zásahu vše hezky hýbalo, žilo vlastním životem a aby to bylo zábavné jen sledovat. Sledovat v reálném čase.

A to může být hlavní překážkou pro zdárné využití této myšlenky v tahových strategiích, kde by čekání na odkliknutí dalšího tahu narušovalo plynulý život.

5 Specifické metody

Tato kapitola by mohla mít méně honosný název Ostatní. Jsou zde kapitoly Hledání cest a Skriptování, které nelze opomenout při zmínce o umělé inteligenci v počítačových hrách.

Hledání cest se zabývá konkrétním problémem nalezení cesty v mapě, kde je jedno, jestli je svět dvourozměrný, nebo 3D, jestli je složen z pravidelných objektů, nebo ne.

Naopak Skriptování není řešením žádného konkrétního problému. Jedná se o velice obecné řešení, jež se dá kombinovat s ostatními metodami nejen AI.

5.1 Hledání cest

Jedná se o speciální problém, který spadá do problematiky umělé inteligence. Hráč chce vyslat jednotky do soupeřovy základny. Neurčuje jednotkám jednotlivé kroky, určí jen požadovanou pozici a zbytek se udělá automaticky.

Úkol je zřejmý, nalézt nejoptimálnější a zároveň věrohodnou cestu mezi dvěma místy na mapě. Nejoptimálnější často znamená nejkratší cestu, ale nemusí tomu být tak. Např. pokud v nejkratší cestě bude minové pole, tak by se mu měly jednotky umět vyhnout. Minimálně jednotky ovládané počítačem.

Zachování věrohodnosti chování a přirozenosti je také důležité. Nemusí působit dobře, pokud se jednotky s naprostou přesností k cíli a obcházejí horu, kterou ještě hráč neobjevil, a tedy jednotky by o ní neměly vědět.

Dalším problémem, který je třeba v této kategorii řešit, je přesouvání více jednotek naráz. Působilo by nepřirozeně, kdyby se všechny jednotky snažily jít nejkratší možnou cestou, a tedy by šly jeden za druhým jako kachničky za svou matkou.

S hledáním cest je nemálo problémů, ale pro řešení základní úlohy, hledání nejkratších cest, bylo již vymyšleno několik funkčních algoritmů.

Hledání (nejkratších) cest je jedním ze stěžejních problémů teorie grafů. V tomto případě grafem se nemíní grafické zobrazení tabulkových hodnot na osách x a y , ale graf je množina uzlů a hran mezi nimi. Příkladem grafu může být bitevní pole složené ze šestiúhelníků z tahové strategie HoMaM 3, kde co jeden šestiúhelník, to jeden uzel grafu. Každý uzel má až šest hran, které znázorňují možnost pohybu na sousední šestiúhelníky.

Mezi algoritmy řešící tuto úlohu patří prohledávání do šířky, prohledávání do hloubky, heuristické prohledávání (sem patří ve hrách velice populární algoritmus A^*), Dijkstrův algoritmus, Floyd-Warshallův algoritmus, Bellman-Fordův algoritmus.

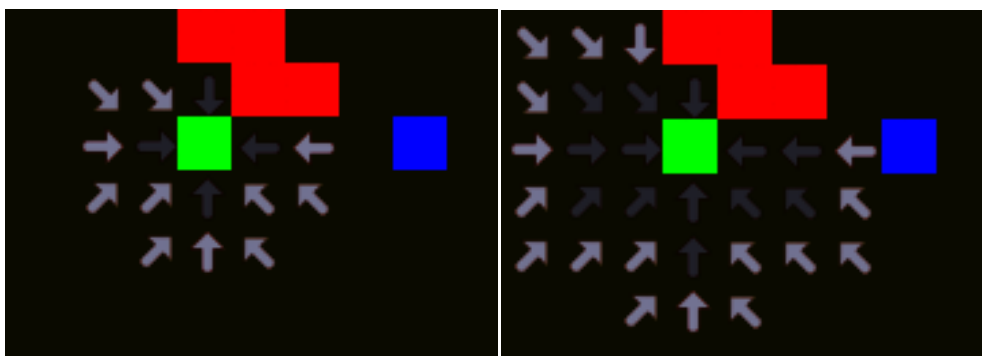
5.1.1 Prohledávání do šířky

Prohledávání do šířky je poměrně jednoduchý algoritmus snadno použitelný v počítačových hrách. Průběh vyhledávání připomíná vlnu rozšiřující se na hladině vody, když na její povrch dopadne kapka vody.

Základní princip algoritmu vysvětlíme na příkladě čtvercové 2D mapy, kde je označen jeden čtverec jako začátek (např. by to mohla být aktuální pozice jednotky), jeden čtverec jako cíl (např. místo, kam hráč klikl myší a kam chce poslat jednotku) a některé ze čtverců jsou

označené jako neprůchozí zdi. V naší imaginární hře s 2D mapou se můžou jednotky pohybovat do 8 směrů.

Vyhledávání probíhá v několika krocích. Prvně zkontrolujeme, jestli se začátek neshoduje s cílem, pokud ano, „cesta“ nalezena. Pokud ne, podíváme se postupně na jednotlivé sousedy startovního pole, jestli ony nejsou cílem. Až zkontrolujeme všechny a nenacházíme-li mezi nimi cíl, zkontrolujeme sousedy sousedů startovního pole. Když ani poté cíl nenalezneme pokračují stejným způsobem. Otestujeme všechny čtverce vzdálené na tři kroky od startu. Tento postup opakujeme dokud nenarazíme na cíl, nebo nezjistíme, že cíl je nedosažitelný (např. se nachází na ostrově, na který nevede most).



Obr. 16 Prohledávání do šířky 5. krok

Obr. 17 Prohledávání do šířky 15. krok

Jak poznáme, že se k cíli ze startu nemůžeme dostat? Při procházení mapy „vlnou“ od startu si označujeme čtverce, na kterých už jsme byli a hledali cíl. Pokud se dostaneme do fáze, kdy všechna pole dostupná ze startu jsou označená a ani jedno z nich nebylo cílem, značí to, že cíl je nedostupný.

Tento algoritmus s jistotou najde nejkratší cestu do cíle, pokud existuje a umí i zjistit, že taková cesta neexistuje. Jeho nevýhodou je, že hledá cíl ve všech směrech a prohledává velké množství prostoru, což se negativně projeví především ve velkých prázdných plochách bez zdí. Pokud start a cíl budou vzdálené deset polí, tak než se cíl nalezne, projdou se všechna pole do vzdálenosti devět čtverců od startu. Druhý problém je ještě výraznější. Pokud na mapě velké tisíc krát tisíc polí neexistuje cesta mezi startem a cílem, prohledá se až milión polí než se zjistí, že cesta neexistuje.

První problém lze částečně vyřešit vysláním vln zároveň z cíle a startu proti sobě. U příkladu polí vzdálených deset polí od sebe můžeme porovnat obsahy dvou kruhů o poloměru pět a jednoho o poloměru deset. Konstantu π můžeme vynechat a máme to $2 * 5 * 5 = 50$ ku $10 * 10 = 100$, tedy u dvou vln v tomto případě prohledáme polovinu polí oproti vlně jedné.

Zamezit prohledávání miliónu polí lze jednoduchou podmínkou. Pokud nenalezneme cíl do vzdálenosti 30, 50, ... vyhledávání ukončíme jako neúspěšné. Toto vylepšení není ideální, protože teď algoritmus nezaručuje, že cestu nalezne, přestože může existovat. Pokud zvolíte limit 50. Co když by byl cíl ve vzdálenosti 51? Hranici, kdy by mělo vyhledávání skončit, není jednoduché najít.

5.1.2 Pár slov k A*

A* patří k nejpoužívanějším vyhledávacím algoritmům v počítačových hrách. A* patří mezi heuristické algoritmy informovaného prohledávání. Využívají znalosti směru mezi startem a cílem. S algoritmem prohledávání do šířky má více společného než se na první pohled může zdát. Prohledávání do šířky je vlastně speciálním případem A*.

U prohledávání do šířky jsme postupně kontrolovali nejdříve všechny čtverce vzdálené jedna od startu, poté 2, 3, 4, atd. V jakém pořadí se budou kontrolovat čtverce, bylo dáno pouze jejich vzdáleností od startu. U algoritmu A* k tomu přibude druhý parametr, a to předpokládaná vzdálenost k cíli. Čtverce prohledáváme v pořadí dané součtem vzdálenosti od počátku a předpokládané vzdálenosti do cíle. Čtverec, který má tento součet menší než jiný, bude testován před ním.

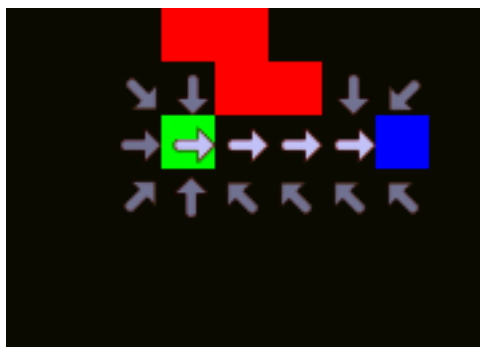
Pod předpokládanou vzdáleností si lze představit odhad vzdálenosti se zanedbáním terénů, zdí mezi kontrolovaným čtvercem a cílem. Např. u čtvercové mapy dobře funguje jako odhad tzv. Manhattanová metoda. Odhad touto metodou se spočítá jako součet absolutních hodnot rozdílu souřadnic zkoumaného čtverce $S[x_1, y_1]$ a cíle $C[x_2, y_2]$.

$$distM(S, C) = |x_1 - x_2| + |y_1 - y_2|$$

Příklad: čtverce se souřadnicemi $[3, 4]$ a $[0, 7]$ mají vzdálenost $|3 - 0| + |4 - 7| = 6$. Mezi další metody patří např. euklidovská vzdálenost.

$$distE(S, C) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Algoritmus lze snadno vylepšit, aby podporoval různé typy povrchů (cesta, bažina), aby znemožňoval dokonalou navigaci v neprozkoumaném prostředí apod.



Obr. 18 Průběh algoritmu A-Star, 5. krok. (Manhattanová metrika)

5.2 Skriptování

Představme si, že tvoříme klasickou strategickou hru jako je Heroes of Might and Magic. Zjistíme, že jednotka rytíře je příliš slabá, zvětšíme jí tedy sílu. Poté musíme hru znovu zkompileovat a slinkovat a znovu spustit. Objevíme v textech překlep, gramatickou chybu, opravíme ji, opět kompilujeme, linkujeme a spouštíme hru. Všechno trvá nějakou dobu a zdržuje to vývoj. Zvlášť např. u vyvažování jednotek, kdy budeme upravovat jejich atributy poměrně často než dosáhneme kýženého výsledku.

Herní data zakomponovaná ve zdrojových kódech mají i jiné nevýhody. Scénárista a pisatel textů, či designér hry nepotřebují a někdy i nesmí mít přístup ke zdrojovým kódům. Nemusí to být vůbec technicky znalí lidé, které by kód mohl rozptylovat a mohli by do něj nepozorností zanést chyby.

Řešením je herní data vytáhnout ze zdrojových souborů do externích textových souborů, skriptů. Skript může sloužit k jednoduchému ukládání páru vlastnost – hodnota, ale lze je využít pro mnohem komplikovanější účely jako je vytváření nových herních událostí, či dokonce lze pomocí skriptů řídit chování programu.

```

<monster>
<name>Ogre</name>
<hitpoints>50</hitpoints>
<firststrike />
</monster>

```

Kód 5 Jednoduchý XML skript

Při použití skriptu stačí restartovat hru a změna je v ní aktivní. Skripty můžeme načítat nejen na začátku spuštění programu se hrou, ale i průběžně během hraní. Pak již nemusíme restartovat hru. Změna se projeví okamžitě a můžeme vyvíjet „on the fly“. Nevýhoda tohoto přístupu je zřejmá, opakované načítání z disku zbytečně zatěžuje disk a procesor.

Jazyk, ve kterém budou psány skripty do naší hry, si určíme sami. Může být jednoduchý stejně jako ukázkový kus skriptu definující obra. Jiný druh skriptu můžeme využít pro definici úkolů ve hře. Takový skript může být posloupnost kroků, které musí hráč udělat, aby získal svou odměnu.

```

ODMENAGP=500;
ODMENAXP=1500;
DOJIT_NA(80,40); // Strom poznání
DOJIT_NA(100,42); // Strom vsevedení
PRISERA = VYTVORIT_MONSTRUM(DRAK, 90, 41);
ZABIT(PRISERA); // Zabit draka

```

Kód 6 Příklad skriptu definující úkol ve hře.

Definujeme si vlastní jazyk DSL (Domain specific language). Jeho slova (DOJIT_NA, VYTVORIT_MONSTRUM) a gramatiku (jeden příkaz na řádku, příkaz zakončen středníkem, cokoliv za středníkem je komentář). Takový jazyk je dobře srozumitelný ne technicky vzdělaným lidem, a tak se může zapojit do projektu mnohem více lidí, uvidí problém zcela z jiného úhlu a tím přispějí ke vzniku dobré hry.

Takový jazyk můžeme dále rozšiřovat. Přidávat příkazy větvení, smyček, či dát možnost definovat vlastní funkce. Větší složitost dává možnost vytvářet více zajímavých a rozličných úkolů. Nevýhodou je, že již ne každý může vytvářet dané skripty. Pokud posadí neprogramátora k takto rozšířeným možnostem skriptování, je pravděpodobné, že většinu jeho nabízených možností ani nikdy nevyužije. Lépe je příkazy a konstrukce postupně přidávat, aby si na ně všichni zvyknuli.

5.2.1 Interpretace vs. kompilace

Jazyky DSL se můžou buď interpretovat, nebo kompilovat. Při interpretaci se v programu čte textový soubor řádek po řádku a rozeznávají se textové příkazy. Tento způsob je rychlejší při vývoji. Stačí uložit upravený textový soubor a je vše hotovo. Dalším plusem je, že hry využívající interpretové skripty jsou snadno využitelné modery, fanoušky, kteří si chtějí hru upravit dle vlastních potřeb a tím prodlouží životnost hry. Může to prodloužit prodejnost hry za její původní plnou cenu až o 8 týdnů. [9] Mínusem je větší objem dat a rychlost interpretace, která je oproti využití předkompilovaných skriptů pomalejší.

Předkompilování urychlí běh skriptů ve hře a skripty poté zaberou méně místa na disku, což je stále aktuální problém při vývoji her na mobilní zařízení. Nevýhodou je pomalejší vývoj, kdy při každé změně skriptu se musí překompilovat a také to zhorší přístup modářům. Zamezení přístupu hráčů ke změně dat může být občas i záměrné, např. u her, které jsou určeny pro online hraní a turnaje. Zamezí se tak možnosti podvádění hráčem.

Před vytvářením vlastního skriptovacího jazyka se musíme zamyslet, jestli by nešel využít nějaký již existující volně dostupný jazyk. Takový jazyk bude odzkoušen širokou základnou uživatelů, bude bez chyb a pravděpodobně bude mít stránku, kde budeme moci napsat své připomínky na vylepšení jazyka. Nevýhodou využití cizího skriptovacího jazyka je, že časem může nastat situace, kdy budeme potřebovat nějakou vlastnost, jež jazyk nepodporuje a nepůjde si ji dodělat. Pak můžeme přednést svoji prosbu, ale při vývoji hry nemůžeme spoléhat, že někdo třetí naší prosbu vyslyší a vyplní.

5.2.2 Skriptovací jazyky

Mezi nejznámější skriptovací jazyk používaný v počítačových hrách určitě patří jazyk Lua, který je populární také svou syntaxí podobnou jazykům z rodiny C. Dalšími možnostmi jsou např. Python, či Scheme. Stručný přehled skriptovacích jazyků naleznete v [9], základy jazyku Lua můžeme pochytit z knihy [10].

6 Frameworky a enginey

Herních frameworků a engineů existuje spousta. Jsou poměrně známé obecné frameworky jako je Unity, nebo frameworky zaměřené například na grafiku Ogre3D, fyziku PhysX apod. Nabízí se otázka, jestli existuje nějaký framework určený pro tvorbu AI do her, který by byl obecně použitelný, nebo alespoň použitelný pro tahové strategie.

Existují open source projekty implementující jednu ze zmíněných metod z předchozí kapitoly, např. pro fuzzy logiku, neuronové sítě, či rozhodovací stromy, ale tyto projekty nebyly zamýšlené pro využití v herním průmyslu, mají sloužit hlavně k vědeckým účelům, data miningu a statistice. Seznam takových projektů můžeme nalézt na [9], kde jsou jejich stručné popisy, licence a odkazy na jejich domovské stránky. Pro Case Base Reasoning mohu zmínit jColibri [18], který je napsán v Javě a není tedy moc vhodný pro využití ve hrách.

Jako obecně použitelné řešení se může jevit SOAR [19]. SOAR je architektura pro vývoj obecných systémů, které vykazují inteligentní chování. Je ve vývoji již od roku 1983 a nyní je ve verzi 9. Na stránce projektu jsou k dispozici ke stažení jednoduché hry využívající tuto architekturu. Bohužel se jeví pouze jako ukázka toho, k čemu lze SOAR přizpůsobit, ale není zde reference na nějakou konkrétní, nejlépe komerční hru využívající tuto architekturu. A pokud taková hra nevznikla za téměř třicetileté trvání SOARu, tak asi není k tomuto účelu využitelná.

Mezi ukázkami je využití této architektury pro řešení hry hanojských věží, pro logickou hádanku obdobné převozníkovi, jedné loďce, s dvěma místy, kozou, vlkem a zelím. Najdeme zde i akční 2D hru viděnou z ptáčích perspektivy, tanky, které po sobě stílí. Pokud si spustíme ukázkou TankSOAR, po chvíli si můžeme všimnout jedné věci. Souboje tanků nejsou vůbec zábavné. Na mnoho kol se dokážou zaseknout na jednom místě, protože ani jeden nechce vyjet ze svého výhodného úkrytu. Taková strategie je inteligentní. Je pochopitelné, že se tank nesnaží vyjet před hlaveň druhého, ale sledovat minutu vyčkávající tanky na jednom místě dokáže unudit.

Jinou možností je využít kompletního enginu, který mimo herního, grafického enginu obsahuje i umělou inteligenci. Sem patří např. Cipher engine [20], jenž pro své rozmanité využití (FPS, RPG, závodní hry) obsahuje pouze základní jednoduchý framework pro budování AI a chování. Visual 3D Game engine [21] obsahuje mimo dalšího editor stromů chování.

V případě volby konkrétního žánru hry jako je FPS nebo RTS, je možné využít kompletní řešení pro daný typ hry, které má v sobě zabudované hotové řešení pro umělou inteligenci. V základu jsou, bohužel, všechny FPS sobě podobné jako vejce vejci a podobně je tomu i u RTS. Pro tvorbu FPS lze zmínit od roku 2005 open source Quake engine, který lze stahovat z [22]. Obdobně pro real-time strategie existuje OpenRTS engine [23].

Jako poslední zmíníme frameworky vyvíjené pro použití ve hrách, které jsou ale použitelné pouze k nějakému konkrétnímu účelu. Patří sem Recast, jež umožňuje implementaci pathfindingu ve 3D světě na základě dat jeho geometrie. Dále jsou zde mohutnější systémy, které umožňují simulace lidí a dopravy. Jmenovitě AI implant a Alive!.

Žádné z nalezených řešení není vhodné pro implementaci do tahové strategie. Buď se jedná o příliš specifická řešení pro konkrétní žánry a problémy, nebo o řešení nevhodné pro vývoj her. Nenajdeme žádnou strategii úspěšně využívající nějaký obecný framework, a tedy je stále potřeba si napsat vlastní AI využívající některou z metod předchozí kapitoly.

7 Realizace

Pro praktické osvojení si metod umělé inteligence jsem zvolil vlastní hru Expanzi, kterou jsem začal vytvářet v rámci předmětu Počítačové hry a animace Y36PHA. Expanze je originální hrou. Záměrně jsem nevytvořil kopii některé z existujících her, abych se nemohl příliš inspirovat existujícími řešeními. Zvolená hra vyžaduje nový, vlastní přístup k vývoji AI.

7.1 Rozdělení úloh

Základ hry jsem vytvářel v rámci tříčlenného týmu. Grafický návrh uživatelského rozhraní a veškeré 3D modely dodala Pavla Balíková. S Alenou Varkočkovou jsme si rozdělili programátorskou část. Alena si navíc vzala na starost tvorbu uživatelské dokumentaci a finální prezentaci, já jsem přišel s konceptem a designem dokumentem a zhostil se role vedoucího týmu.

Alena programovala systém jednotlivých menu nabídek, navrhla .xml soubory pro data map a naprogramovala jejich načítání. Postarala se o HUD a o okno měnění surovin.

Veškerou herní logiku, zobrazení 3D světa a umělou inteligenci jsem programoval sám.

7.2 Stručný popis hry

Expanze je budovatelská 2D tahová strategie ve 3D prostředí. Pravidla a svět byly inspirovány populární společenskou hrou Osadníci z Katanu. Herní plán se skládá z pravidelných šestiúhelníků. Šestiúhelníků je sedm druhů, po jednom druhu ke každé surovině, poušť a voda, jež obklopuje ostrov, kde se hra odehrává. Hexy se liší mimo druh i ve výnosnosti.

Hráč zakládá města na rozích mezi hexami a expanduje po ostrově stavěním cest mezi hrany hex. Každé město sousedí až se třemi hexami, kde na každé z nich může hráč postavit jednu

budovu. Na výběr jsou buď budovy těžby, které zajistí přísun surovin z dané hexy dle její výnosnosti, nebo speciální budovy, které mohou urychlit expanzi hráče.

Cílem hry je jako první získat předem známý počet bodů. Body hráč dostává za stavbu měst, cest a speciálních budov. Navíc může získat body za medaile. Medaile jsou speciální ocenění, pokud nějaký hráč někde vyniká. Medaili cestovatele získá, jestliže má postaveno více cest než jakýkoliv jiný hráč a má jich alespoň 10. Poslední možností k získání bodů je akce Předvedení vojenské přehlídky, kterou si hráč může opakovaně kupovat, pokud má postavenou speciální budovu pevnosti. Při velkém počtu hráčů na malém území může být tato akce jediným způsobem, jak hráč může hru vyhrát.



Obr. 19 Screenshot z rozehrané hry. Jména hráčů jsou inspirovány průzkumníky a dobyteli.

V pevnosti lze zakoupit ještě dvě další akce. Hráč může soupeřům ukrást část jejich zásob nebo může zabrat některou z hex a tím si přivlastňovat vytěžené suroviny soupeřem. Dalšími speciálními budovami jsou klášter a tržiště.

Suroviny ve hře si nejsou rovné. Opakovaným hraním a testováním jsem zjistil, že stavební kámen a ruda je důležitější než zbylé tři suroviny. Přesto hráč potřebuje všechny druhy surovin. Ne vždy je možné hned od začátku hry získávat alespoň nějaké množství od každé suroviny, proto je zde možnost suroviny měnit za jiné. Je zde ale nevýhodný poměr 4 ku 1. Na tržišti si hráč může kopit listiny, které zajišťují výhodnější směnný kurz.

Klášter je centrem vzdělanosti a lze v něm vynalézat nové pokroky. Po vynalezení pokroku všechny hráčovi těžební budovy daného typu začnou vynášet více surovin.

7.3 Vývojové prostředí

Hru jsme programovali dva, a tudíž volba programovacího jazyka a knihoven nebylo jen na mně. Poměrně rychle jsme se shodli na využití jazyka C# a XNA.

Alternativou byla Java, nebo C++ a OpenGL, s kterými jsme měli zkušenosti. Java není příliš vhodná pro vývoj 3D her, a proto byla zavrhnuta. C++ s OpenGL je vhodnou kombinací pro vývoj her. Nevýhodou této kombinace je, že by se nejdříve musel tvořit grafický engine hry a my jsme se chtěli zaměřit na hru samotnou a k tomu je vhodnější XNA. Navíc jsme se oba chtěli naučit novou technologii, což bylo další plus pro XNA.

Další volba byla mezi XNA 3.1 a novější a aktuální verzi 4.0. Pro verze 3.1 byla velká množství tutoriálů na internetu, cvičení předmětu Počítačové hry a animace, které probíhalo též pro verzi 3.1. Přesto jsme raději zvolili naučit se aktuální verzi 4.0. S tím bylo dáno i vývojové prostředí Visual Studia 2010.

Vývoj ve dvou lidech potřeboval správu zdrojového kódu. Oba jsme měli dobrou zkušenost s SCM Gitem. Založili jsme repozitář na GitHubu, kde jsme využívali i místní issue tracking system.

7.4 Reprezentace světa

Svět se skládá z šestiúhelníku, hex. Hexy jsou uloženy v řádcích a sloupcích. Každá hexa má své unikátní souřadnice. Viz. Obr. 20 Souřadnice jednotlivých hex. Každý nový řádek hex je posunut na ose x o polovinu šířky hexy v kladném směru. Tím je zaručena návaznost hex. Aby mohl být zachován tvar ostrova ve tvaru velkého šestiúhelníku, musí být mapa doplněna o neviditelné hexy na krajích. Ten jev je vidět vedle vodní hexy se souřadnicemi [4; 1] z Obr. 20, kde po pravé straně od ní se nachází neviditelná hexa se souřadnicemi [4; 0].

Hexy jsou uloženy v dvourozměrném poli, kde řádky a sloupce odpovídají popsané struktuře. Tato reprezentace je dostačující pro zjišťování sousednosti jednotlivých hex. Přesto jsem doplnil do všech hex reference na sousední hexy. (lze se dotazovat pomocí výčtového typu *RoadPos* s hodnotami *UpLeft*, *UpRight*, *MiddleLeft*, *MiddleRight*, *BottomLeft*, *BottomRight*) Usnadňuje to práci mnohým algoritmům. Třetí možností přístupu k jednotlivým hexám je přes unikátní číslo ID, které má každá hexa.



Obr. 20 Souřadnice jednotlivých hex

Všechny hexy mimo okrajových vodních hex v sobě obsahují reference na města v rozích a cesty na hranách. Města jsou odkazovatelná přes enum *TownPos*, cesty stejně jako hexy jsou přístupné pomocí *RoadPos*. Města a cesty společné pro více hex jsou vytvořeny pouze jednou, ostatní ukazují na již vytvořené instance. Každá hexa si pamatuje, které instance měst a cest

vytvořila a na které jen odkazuje. O vykreslování, zpracovávání událostí apod. se stará hexa, jež dané prvky vytvořila. Podobně jako hexy mají města a cesty vlastní unikátní číslo ID.

Největší význam ID mají města. Pomocí *TownID* lze velice jednoduše jedním for cyklem projít všechna města a najít nejvhodnější místo pro stavbu. První dotaz na město dle ID je vždy zdlouhavý. Časová složitost je lineární dle počtu hex. (ptá se každé hexy, jestli obsahuje město s daným ID) Jelikož se vyhledávají města podle ID velice často, výsledek vyhledávání není zapomenut, ale uložen do jednorozměrného pole o velikosti počtu měst. Následující dotazy na město jsou vyřízeny v konstantním čase přes index do pole.

AI využívá algoritmy na hledání cest mezi dvěma městy, zjišťování vzdálenosti dvou měst apod. Pro tyto účely jsou v uloženy ve hře redundantní informace, které by šlo získávat i za běhu programu, ale nebylo by to efektivní a pravděpodobně ani ne moc přehledné. Každé město má reference na sousední tři města a i na tři cesty z něj vycházející. Cesty mají reference na krajní města, s nimiž jsou spojeny.

8 Pluginová architektura

Rozhodl jsem se pro implementaci AI použít pluginovou architekturu. AI se do hry načte z dynamické knihovny .dll. Tento přístup vyžadoval složitější návrh a časově náročnější přípravu před samotným vývojem AI, ale přineslo to nepopiratelné klady.

Nyní je AI striktně oddělena od samotné hry, všech herních mechanismů. Nejsou zde přílišné provázanosti a AI nesplývá se zbytkem kódu. Tento aspekt přináší větší čistotu a srozumitelnost zdrojového kódu a také usnadňuje debutování. Lze jasně určit, jestli se chyba nachází v AI, nebo ve hře samotné.

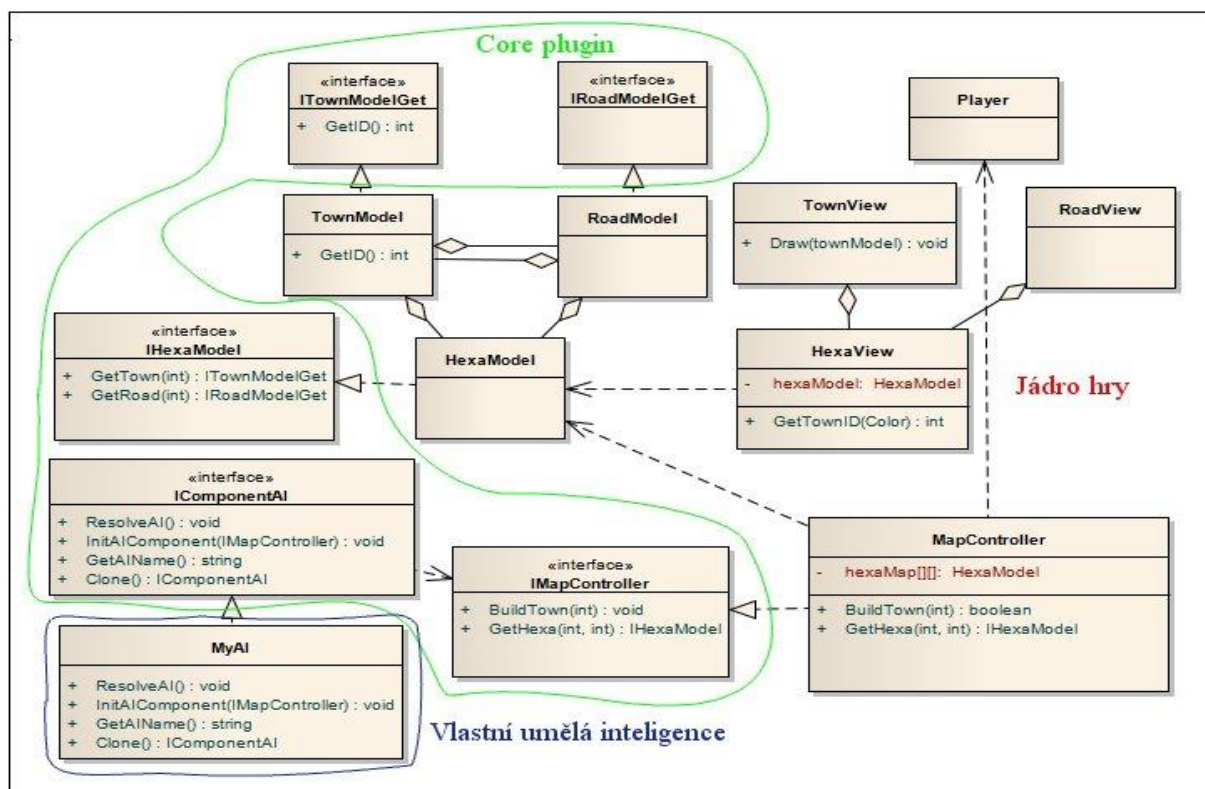
Tento přístup urychluje znatelně vývoj, při každé změně se překládá pouze dynamická knihovna a spustí se s nezměněným jádrem hry.

V neposlední řadě to umožňuje vyvinout více naprosto rozdílných umělých inteligencí. Pro přidání nové umělé inteligence stačí přidat .dll, v kterém je třída implementující rozhraní *IComponentAI*. Vlastní umělou inteligenci si může naprogramovat každý.

8.1 Komponenty

Hra se skládá ze tří částí. Jádro hry, Core plugin a jednotlivá AI. Core plugin obsahuje skupinu rozhraní, přes které knihovna AI komunikuje se hrou. Tato rozhraní implementují modely v jádře hry (*TownModel*, *HexaModel*, atd.), které je založeno na architektonickém stylu Model-View-Controller.

Nejdůležitější třídou v jádru hry je *MapController*. V ní jsou metody odpovídající veškerým akcím, které může ve hře hráč provádět. Namátkou jsou zde metody *BuildTown*, *ChangeSources*, *BuyLicence*. Tuto třídu využívá View z MVC, tedy pokud lidský hráč klikne na obrazovce a chce např. postavit město, zavolá se metoda *BuildTown* z *MapControlleru*, ale je též využívána umělou inteligencí.



Obr. 21 Zjednodušené schéma návrhu hry

Při vytváření umělé inteligence musí vývoář vytvořit pouze jednu třídu, jež implementuje rozhraní *IComponentAI*. Rozhraní obsahuje 4 metody : *InitAIComponent(IMapController)*, *ResolveAI()*, *GetAIName()*, *Clone()*. Při založení nové hry se pomocí *Clone* naklonují protihráči ovládané počítačem. Umožňuje to mít ve hře více hráčů stejné umělé inteligenci, kdy každý má vlastní proměnné. Pokud žádné proměnné AI nemá, stačí do těla metody napsat `return this`. Poté každé AI získá přístup *MapControlleru* pomocí *InitAIComponent(IMapController)*, kde se může provést veškerá inicializace. Když přijde AI hráč na řadu, spustí se jeho metoda *ResolveAI()* v novém vláknu. V ní přes *MapController* provádí veškeré herní akce.

Poslední komponentou je Core plugin, která spojuje AI knihovny se hrou. Kód je založen na [24].

8.2 Způsob ovládání

Než přejdu k popisu, jak hru ovládá AI, vysvětlím implementaci ovládání hry živým hráčem. Celá hra lze ovládat pouze pomocí myši. Navíc jsem do hry přidal mnoho klávesových zkratk, které mohou výrazně urychlit hru.

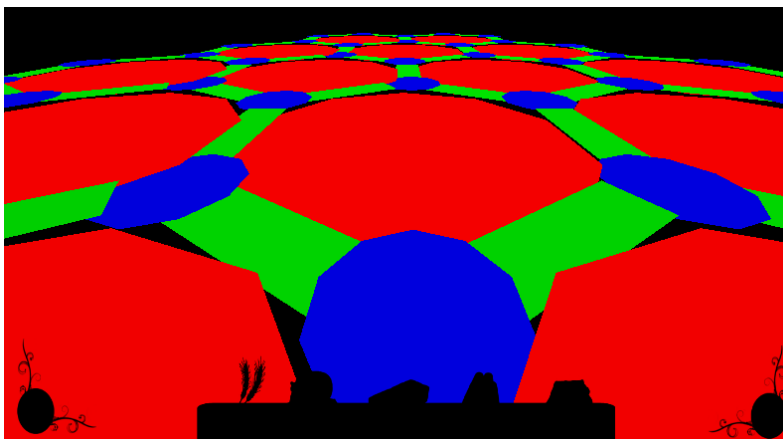
8.2.1 Myš

Existují dva běžně používané principy pro označování objektů v 3D scéně pomocí myši. První způsob je využití analytické geometrie a počítání průsečíku paprsku a obalových těles. Druhou možností je využití pickingu.

V prvním případě se „klikatelné“ objekty obalí jednoduchými tělesy, u nichž je jednoduchý výpočet průsečíku paprsku (polopřímky) a daného tělesa. Vhodnými obalovými tělesy jsou

např. koule, či kvádry zarovnané se souřadnicovými tělesy. Polopřímka směřuje od pozorovatele, hráče, skrz bod na obrazovce, kde je myš, dále do herní scény. Záporem tohoto řešení je nepřesnost kvůli aproximaci objektů ve scéně pomocí jednoduchých těles.

V Expanzi jsem využil druhou variantu, picking. K ovládání hry je nutné mít možnost klikat na města, cesty a hexy. Pokud chce hráč využít speciální budovu, klikne nejdříve na město, ke kterému se budova váže, poté na hexu, na níž je budova postavena. Budovy tedy nemusí být klikatelné.



Obr. 22 Výběr míst pro stavbu je prováděn technikou pickingu. Každá cesta má svůj odstín zelené, město odstín modré a hexa odstín červené.

Při pickingu se klikatelné objekty vykreslují dvakrát. Jednou se všemi efekty, osvětlením na obrazovku, podruhé pouze s konstantním ambientním světlem do textury. Každý objekt musí být vyobrazen s jiným odstínem RGB barvy, aby jednotlivé objekty byly od sebe rozeznatelné. Po vykreslení scény do textury stačí vzít barvu pixelu na souřadnicích myši a určit, kterému objektu barva patří. Se shodujícím se objektem se provede onHover, či onClick efekt.

Pro urychlení vykreslování do textury nevykresluji originální objekty, ale pouze 2D obrazce, mnohoúhelníky pro města a hexy, obdélníky pro cesty. V 99% času se hra hraje pouze z ptací perspektivy, a tedy tato aproximace je zcela postačující. Klikatelné objekty navíc překreslím HUD obrazovkou v černé barvě, aby hráč nemohl kliknout na objekt, jež nevidí. Výslednou texturu si lze prohlédnout na Obr. 22.

Hráč je dopředu informován o akcích, které nelze provést. Např. mu není zobrazeno potvrzovací tlačítko pro stavbu města, když nemá suroviny, nebo se snaží stavět příliš blízko jinému městu. V okně s výběrem stavby se mu zobrazuje červeným nápisem jeden z důvodů, proč akci nemůže provést. Ukázalo se to jako vhodnější varianta než hráči dovolit stisknout tlačítko pro stavbu a až poté ho upozornit pop-up oknem, že akci nemůže provést.

Samotná akce se provádí přes MapController, který využívají i umělé inteligence, a tedy i v rámci kontroléru musí docházet k ověření platnosti akcí, aby AI nemohla podvádět. Výsledkem je dvojitá kontrola akcí, když je používá hráč. První proběhne při zjišťování, jestli se má tlačítko pro akci zobrazit, podruhé po stisknutí tlačítka.

8.2.2 Klávesnice

Přesuny myši po obrazovce bývají mnohdy zdlouhavé. Proto je dobrým nápadem hru doplnit o klávesové zkratky, jež není nutné používat, ale po naučení se jich hraní zpříjemní.

Jako základ své utility pro ovládání hry jsem využil koncept popsany v [25]. Tento koncept se snaží oddělit akce od kláves. V Expanzi mám akci zobrazit okno pro výměnu surovin. Chci, aby se okno zobrazilo po stisknutí klávesy m. V kódu ale nemám natvrdo napsáno Pokud hráč stiskne ‘m’, zobraz tržiště. Místo toho Pokud hráč aktivuje akci Zobraz tržiště, tržiště se zobrazí. Jinde v kódu se nachází spoj akci Zobraz tržiště s klávesou ‘m’.

Takto lze snadno na jednom místě zaměňovat klávesy pro jednotlivé akce, dokonce je možné jedné akci přiřadit více klávesy. Poté stačí stisknout jednu z nich a akce se provede. Navíc to umožňuje jednoduché rozšíření na další herní zařízení. Stačí umožnit spoj akce Zobraz tržiště s tlačítkem gamepadu.

Koncept jsem rozšířil o jednoduchý stavový automat. Vytvořenou akci spojuji nejen s klávesou, ale také se stavem, jehož se týká. Chtěl jsem odlišit klávesové zkratky používané na herním plánu od zkratk používaných při měnění surovin, či v okně se stavbou budov. Pokud jsem ve stavu s oknem výměny surovin, nefungují zkratky, které jsou definované v jiných stavech.

Příkladem může být využití klávesy Esc pro více účelů. Ve stavu Výměny surovin stisknutí klávesy zavře okno se surovinami, v hlavním stavu Esc otevře menu s možností ukončení hry. Kdybych neměl ve hře stavy, musel bych pro každý z účelů použít jinou klávesu, protože jinak by mi stisk Esc zavřel okno s výměnou surovin a zároveň vyjel menu pro ukončení hry.

```
GameAction close = new GameAction("close", GameAction.ActionKind.OnlyInitialPress);
im.MapToKey("gamemarket", close, Keys.Escape);
im.MapToKey("gamemarket", close, Keys.M);
-----
public void SetIsActive(bool active) {
    if (active) InputManager.Inst().SetActiveState("gamemarket");
    else InputManager.Inst().ClearActiveState("gamemarket");
}
-----
if (InputManager.Inst().GetGameAction("gamemarket", "close").IsPressed())
    MarketComponent.Inst().SetIsActive(false);
```

Kód 7 Tři fáze nastavení zavření okna na klávesy M a Escape ve stavu gamemarket. 1. vytvoří se akce a přiřadí se jí klávesy. 2. Změní se stav input manageru na gamemarket. 3) Zjistí se aktivace akce close.

8.3 Skriptovací jazyk

Způsob komunikace AI se hrou lze považovat za skriptovací jazyk založený na C#. Tento jazyk se před použitím ve hře musí zkompileovat, neinterpretuje se. Skript se načte vždy jednou při spuštění hry. Při změně skriptu se musí hra znovu spustit.

Slova jazyka jsou založená na krátkých výstižných anglických větách. Jejich názvy kopírují akce ve hře. Jsou zde slova jako *BuyLicence* a *InventUpgrade*, kde v prvním případě je parametrem výčtový typ *SourceKind* a v druhém případě *SourceBuildingKind*. V jazyce jsou tyto akce takto rozlišeny, přestože v jádře hry se z nich volá společná metoda *BuyUpgradeInSpecialBuilding*, kde se pod pojmem Upgrade schovává i Licence.

Většina slov jazyka je ve dvou verzích. V druhé verzi mají slova předponu Can. Tedy jsou zde slova *CanBuyLicence* a *CanInventUpgrade*. Metody Can vrací jednu z možností výčtového typu přiřazenému dané metodě. Vždy je zde možnost OK, která značí, že všechny podmínky pro využití dané akce byly splněny. Např. metoda *CanInventUpgrade* vrací *MonasteryError*, která obsahuje *OK*, *MaxUpgrades*, *NoSources*, *HaveSecondUpgrade*. První možnost značí, že je vše v pořádku. Druhá možnost znamená, že hráč nemá postavený klášter, nebo již nemá volný slot pro další upgrade (je zde omezení 3 upgrady na klášter). Poslední

dvě možnosti oznamují nedostatek surovin, či již koupený druhý stupeň upgradu pro daný typ budovy.



Obr. 23 Okno, ve kterém hráč vybírá, kterou z budov na poušti postaví.

Can metody slouží pouze pro rozlišení příčin neuskutečnění akce. Navíc umožňují zjistit aktuální možnosti aniž by se některá akce provedla. Pro pouhé zjištění, jestli se hráčova akce povedla, stačí kontrolovat návratovou hodnotu. Jestliže je návratová hodnota false, nebo null, akce se nepovedla (false/null závisí na zvolené metodě).

Některých akcích lze dosáhnout více způsoby. Jeden způsob zpravidla kopíruje chování hráče. Druhý způsob zjednodušuje používání některých akcí. Příkladem je zmíněné vynalézání pokroků. Metoda *InventUpgrade* se nachází v rozhraní *IMapController*, ale také v *IMonastery*. V prvním případě může hráč pokrok vynalézat přímo, jádro hry se již postará o nalezení klášteru s volným slotem pro pokrok. V druhém případě může AI zavolat *InventUpgrade* v konkrétním klášteře stejně jako by to dělal lidský hráč ve hře.

8.4 Podvádění zakázáno

Podvádění může v některých situacích navodit lepší dojem z umělé inteligence. Osobně jsem tuto možnost ve své hře zakázal a neumožnil ji. Umělá inteligence má stejné znalosti a prostředky jako jakýkoliv jiný hráč.

Nemůže kupovat budovy a pokroky, na které nemá suroviny. Nemá informace navíc, nemá možnost zjistit, kdy skončí dopad katastrof a zázraků, či jaké nové se kdy objeví. Musí též počítat s tímto minimem náhody. A ani nemá možnost některé budovy zbořit a postavit místo nich nové. (boření budov není ve hře povoleno)

U všech pokusů o provedení akce se kontroluje, jestli není daná akce v rozporu s pravidly. V této hře by to nebylo možné provádět jinak, protože celý herní svět je pro hráče viditelný. Každý hráč vidí, jaké budovy jsou na plánu rozestavěny, jaké suroviny, pokroky nebo medaile mají jednotliví hráči.

Kromě zamezení podvodů bylo zapotřebí se postarat o zamezení snahy shodit hru v případě, že vyhrává někdo jiný. Tento problém není podstatný při běžném hraní. Projevil by se v případě možné soutěže více různých umělých inteligencí proti sobě. Hráči mají přístupná data o aktuálních bodech ostatních hráčů. Někdo by mohl do své umělé inteligence umístit

úmyslně chybu, která by se zavolala těsně před prohrou a tím by se zamezilo zaznamenání výsledku hry. Mimo vyvolání chyby by se mohla umělá inteligence zacyklit².

První problém řeší jednoduché ošetření všech neošetřených výjimek z AI jedním blokem try/catch. Pokud v „přemýšlení“ nastane kritická chyba, která by způsobila pád celého programu, zde se odchytí a hráč ovládaný danou umělou inteligencí se odstříhne od hry. Lidskému hráči se zobrazí, že se jeho protihráč vzdal. Jeho města a cesty ve hře zůstávají. Nemůže ale dělat další akce, jeho metoda *ResolveAI* již není znovu volána.

Existuje v C# jedna výjimka, která tímto způsobem nelze ošetřit. Vývojáři se od verze Net 2.0 rozhodli znemožnit odchycení výjimky Stack Overflow Exception. Objekt výjimky se vytvoří, ale i přes snahu ji odchytit, program se ukončí. Vyvolat tuto výjimku je snadné, rekurzivní funkcí, která má ve svém těle pouze volání sama sebe. Byl jsem bohužel při hledání řešení neúspěšný. Nenašel jsem způsob, jak odchytit stav blížící se Stack Overflow Exception. Pokud by se mi to podařilo, stačilo by vlákno ukončit před vyčerpáním paměti, jež využívá stack.

Napadlo mě nepřímé řešení. Interface pro AI ji nedává možnost zjistit, jestli hraje proti lidskému hráči, jiné AI, či jejím druhu. Před samotnou soutěží by mohla proběhnout kvalifikace. Spustilo by se 100 her, kde by hráči byly ovládané počítačem jednoho druhu. Jestliže by AI vyhodila výjimku, neprošla by kvalifikací.

Druhý problém má své řešení. Nemohu přímo zachytit zacyklení v cizím kódu. Ve hře je pro AI časové omezení dané na několik vteřin. Pokud je tento limit překročen, tah AI se ukončí, ale AI hráč stále setrvává ve hře. Jeho metoda *ResolveAI* bude zavolána až se hráč příště dostane na řadu.

9 Základní AI

Ve hře jsou nyní tři obtížnosti soupeřů. První obtížnost – Lehká je velice jednoduchá a sloužila především jako referenční AI pro testování následujících dvou umělých inteligencí.

S touto obtížností bylo vyvíjeno samotné rozhraní pro umělou inteligenci, aby bylo připraveno pro další vývoj.

Tato umělá inteligence dokáže dosáhnout vítězství, ale zabere jí to většinou mnohem více herních kol než člověku.

Pouze rozestavování prvních dvou měst lze nazvat inteligentním chováním a není příliš odlišné od dalších AI. Výběr pozice pro město je na základě sumy výnosností okolních hex daného města. Dále je zohledněna rozmanitost druhů surovin a také jednotlivé druhy surovin mají své koeficienty důležitosti, kterými se násobí výnosnost hex.

² Nekonečný for, while cyklus se špatně nastavenou ukončovací podmínkou.

```

turn++;
TryChangeSources();
BuildAllPossibleSourceBuilding();
BuildRandomTown();

if (turn % 4 == 0) {
    for (int loop1 = 0; loop1 < 3; loop1++) {
        if (BuildRandomRoad())
        {
            TryChangeSources();
            if (BuildRandomTown()) {
                TryChangeSources();
                BuildAllPossibleSourceBuilding();
            }
        }
    }
}
if (turn % 7 == 0)
    if (hasFort)
        myFort.ShowParade();

```

Kód 8 Ukázka kódu AI lehká

Chování v hlavní fázi hry je již dosti chaotické. AI nezohledňuje aktuální potřebu, akce protihráče. Rozvoj probíhá pseudonáhodným směrem.

Kód 8 Ukázka kódu AI lehká je veškerý kód, který se provádí v každém kole hlavní fáze. Metoda *TryChangeSources* zajišťuje měnění surovin. Neohlíží se vůbec, jaké suroviny hráč potřebuje. Pokud má některé ze surovin více jak 200 kusů a jiné méně jak 100, vymění polovinu nadbytečné suroviny za nedostatkovou. *BuildAllPossibleSourceBuilding* projde všechna postavená města a zkusí v nich postavit na každou hexu těžební budovu. Pouze v případě, že je 10. a pozdější kolo, pokusí se postavit jednu ze speciálních budov. Každou speciální budovu má maximálně 1. Umí ale využít pouze Vojenskou přehlídku z pevnosti. *BuildRandomTown* zkusí na všech možných i nemožných místech postavit město.

Další kus kódu se provádí pouze každé 4. kolo hry. Pokud by se hráč pokoušel stavět cesty každé kolo, pravděpodobně by nikdy nepostavil žádné město. Vždy by měl dříve dostatek surovin na cestu než na stavbu města, a tedy by stavěl pouze cesty. Když se podaří postavit cestu, zkusí se vyměnit suroviny pro případ, že by se stavbou cesty vypotřebovala nedostatková surovina a existovala by stále nějaká přebývající. Pokud se podaří postavit i město, zkusí se znovu vyměnit suroviny, a pak v něm postavit jednotlivé těžební budovy. Tato část kódu se provede 3x. Důvodem je metoda *BuildRandomRoad*, která v závislosti na pořadí kola zkouší stavět cesty z různé startovní pozice na mapě. S možností stavět více jak jednu cestu za kolo je pravděpodobnější, že AI vytvoří dvě cesty za sebou, a tím vytvoří nové volné místo pro město.

Na závěr se každé sedmé kolo pokusí získat tři body za armádní přehlídku. Důvod podobný jako u stavění cest. Zkoušet vyvolat přehlídku každé kolo by snižovalo pravděpodobnost stavby měst, které jsou důležitější a navíc je za ně 5 bodů. Nevýhodou tohoto omezení je prodlužování konce hry, kdy už je celý herní plán zastavěný a jediná možnost výhry je využívání vojenských přehlídek.

Tohoto hráče není těžké porazit ani pro začínajícího hráče. Je vhodný pouze do tutoriálu, či jako doplňkový protihráč k jiným AI na větších mapách.

10 Rozhodovací stromy

S rozhodovacími stromy je spojována jejich jednoduchost, a mimo jiné proto jsem se rozhodl je zkusit využít pro první důmyslnější umělou inteligenci. Cílem bylo zjednodušit a uspořádat do grafu rozhodovací podmínky, kterými se řídí hráč expert. Převést myšlenky „Pokud těžím hodně jedné suroviny, postav tržiště. Pokud mám u města volné místo pro těžební budovu, postav ji. Atd.“ do počítačem pochopitelné formy.

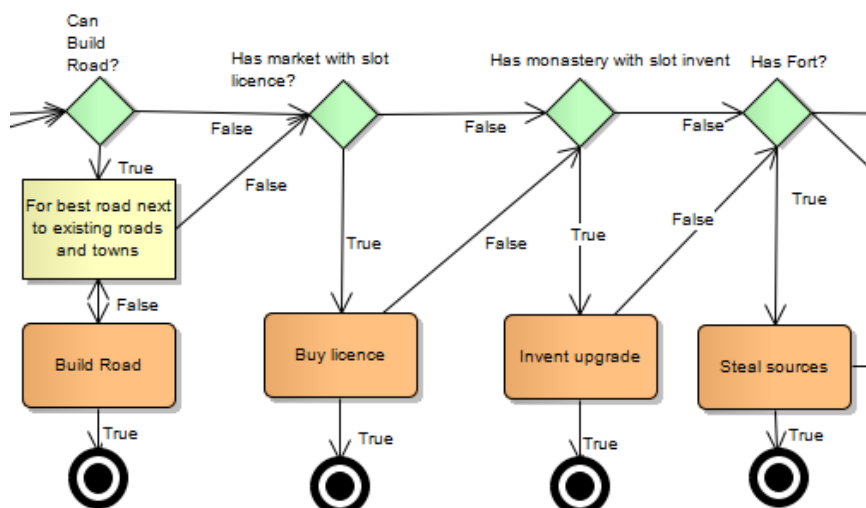
Musel jsem vyřešit čtyři obecné problémy, které jsou společné i pro následující AI.

- I. Jak zajistit, aby AI mohla konat více akcí během jednoho kola a zároveň věděla, kdy má pustit protivníky ke svému tahu?
- II. Druhým problémem bylo vyřešit, kdy a jak má hráč měnit suroviny. Je lepší postavit těžební budovu s výnosem 16 surovin za kolo bez měnění surovin nebo postavit budovu s výnosem 24 surovin za kolo s nutností měnit suroviny?
- III. Je lepší vyměnit suroviny a provést jednu z akcí, nebo počkat jedno a více kol, kdy se potřebné suroviny vytěží?
- IV. Jak se vyrovnat s rozšiřujícím stavovým prostor. Nestačí vyřešit, jakou akci provést, jestli postavit cestu, koupit pokrok. AI se musí rozhodnout, jakou z možných cest postavit, který z pokroků vynalézt.

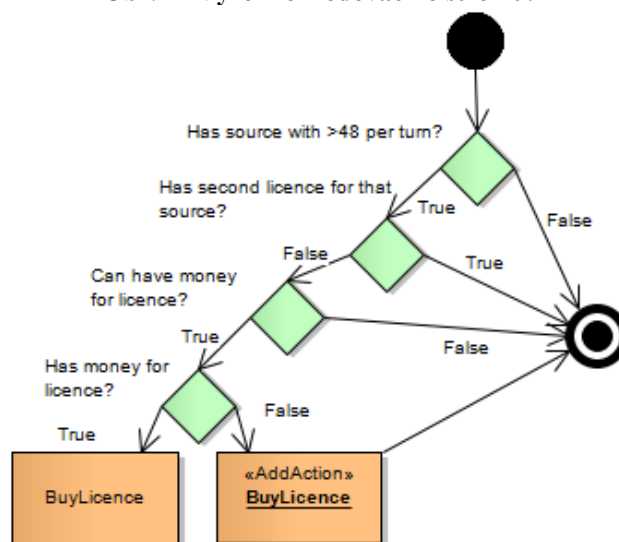
10.1 Základní struktura „stromu“

Přestože dle názvu by se mohlo zdát, že graf podmínek musí tvořit strom, není tomu tak. Zmínil jsem to již v teoretické části, že různé variace nemusí striktně splňovat stromovou strukturu, a tedy hran vstupujících do uzlu může být více. Této varianty jsem se držel i já, plus jsem se inspiroval speciálními vnitřními uzly z 3.1.3 Stromy chování. Obrázky Obr. 24 a Obr. 25 znázorňují základní strukturu stromu. Za každým obdélníkem s oblými hranami se skrývá ještě malý podstrom. Rozdělil jsem to pro větší přehlednost. Každá z akcí³ má vlastní podstrom a odpovídá na otázku, jestli je vhodná daná akce v konkrétní chvíli. Vhodnost je vždy testována několika podmínkami, které musí být všechny pravdivé, aby se akce provedla. Na operátor AND mezi podmínkami jsem použil Obr. 2 Dekompozice AND pomocí dvou rozhodovacích uzlů. Není-li jedna z podmínek pravdivá, zkouší se vhodnost jiné akce. V hlavním diagramu je to značeno false větví ze zaobleného obdélníku. Zde je vidět porušení hierarchie stromové struktury, kde jeden uzel má více vstupních hran.

³ Akce jsou postavit cestu, město, ve městě těžební budovu, klášter, pevnost, tržiště. V klášteře vynalézt pokrok, na tržišti koupit licenci, z pevnosti předvést vojenskou přehlídku, obsadit okolní hexu, okrást soupeře.



Obr. 24 Výřez rozhodovacího stromu.



Obr. 25 Detail na Buy licence z Obr. 24

Každá z akcí má rozhodovací uzel ve tvaru *Can have money* a uzel *Has money for*. První druh uzle slouží ke zjištění, jestli si hráč může dovolit provést toto kolo danou akcí. Jinak řečeno je zde dotaz, jestli hráč má všechny potřebné suroviny, nebo jestli může suroviny vyměnit tak, aby za akci mohl zaplatit. V některých případech je tento dotaz uveden v hlavní části diagramu stromu jako první rozhodovací podmínka, jestli vůbec má smysl vyhodnocovat vhodnost akce. Takto je umístěn u akcí, které mají dány pevnou cenu. Mezi ně patří postavit cestu, město. Akce koupit licenci nemá pevnou cenu, jelikož různé licence k jednotlivým surovinám stojí různé typy surovin. Druhá podmínka *Has money for* zjišťuje, jestli si může hráč akci dovolit bez měnění surovin. Jestliže ano, akce se provede. Pokud ne, akce se přidá na konec seznamu akcí *actionSource*, které hráč může provést toto kolo, ale musel by měnit suroviny, a následně se pokračuje vyhodnocováním vhodnosti dalších akcí.

Vyřešen II. problém. Tato AI dává přednost vykonáním akce bez měnění surovin před jakoukoli jinou akcí. Není to nejvhodnější řešení. Teoreticky je možné, že AI dá přednost předvedení přehlídky, za kterou má 3 body před postavením města, které kromě dalších možností stavby a rozvoje dává bodů 5. Obojí stojí přibližně stejné množství surovin. Na

město mohl chybět jeden kus dřeva, přesto se nepostaví. Testy⁴ ukázaly, že tento nedostatek není podstatný. Nikdo nezaznamenal, že by AI vykonala akci vyloženě hloupou.

Dosažení konce rozhodovacího stromu znamená, že se nevykonala žádná akce. Tedy neexistuje v danou chvíli vhodná akce, na kterou by AI mělo suroviny aniž by je musela měnit. Poté se prohlédne seznam *actionSource*, který obsahuje akce a počet surovin, jež je potřeba vyměnit.

Nyní přecházím k řešení I. problému. Jestliže je seznam prázdný, není žádná vhodná akce, která by se mohla vykonat v daném kole a pustí se další hráč na řadu. V druhém případě se vybere jedna z akcí seznamu, vykoná se a spustí se znovu rozhodovací podmínka v kořenu stromu. Každá z akcí stojí suroviny. V tu chvíli je jiná situace a vyhodnocování podmínek skončí jinak. Výběr akce ze seznamu závisí jak na pořadí vložení akce do stromu, tak i na množství surovin, jež je třeba vyměnit.

Bystřejší čtenář si mohl povšimnout řešení III. problému. AI pokud může vykonat akci, vykoná ji. Nikdy nečeká, aby mohla koupit akce bez nevýhodného měnění surovin.

10.2 Druhy uzlů

Uzli jsou dvou základních typů. Rozhodovací podmínky uvnitř stromu a akce v jeho listech.

10.2.1 Akce

Začnu s vysvětlením listů stromu, protože jsou jednodušší a jsou pouze tří typů.

ActionNode

Hlavní typ listu. Důležitým parametrem při jeho vytváření je delegate bool *DelAction()*. Delegát na metodu, která se má vykonat, pokud se dostane vyhodnocování stromu až do listu. Metoda vrací hodnotu typu bool udávající, jestli se akce vykonala. Hodnota měla především význam při debugování, jelikož podmínky nutné pro vykonání každé akce se kontrolují v rámci decision tree.

ChangeSourcesActionNode

Jak jsem již uvedl v předchozí kapitole, v případě, že hráč musí kvůli provedení akce měnit suroviny, nemůže se přímo zavolat akce. Namísto toho se po návratu false z podmínky *HasMoneyFor* dostane vyhodnocování do *ChangeSourcesActionNode* uzlu, který má za úkol přidat akci s její cenou do seznamu *actionSource*. Poté zavolá vyhodnocování vhodnosti další akce. Uzel je zděděný z obecného *ActionNode*.

FakeActionNode

Uzel pro falešné akce. Byl často využit na konci větví stromu, které ještě nebyly dokončeny. Nyní se nachází za všemi kontrolami vhodnosti jednotlivých akcí a značí, že se má přejít k výměně surovin a provedení akce z *actionSource*. Také ho využívají některé rozhodovací uzly.

⁴ Viz kapitola **Chyba! Nenalezen zdroj odkazů.** Chyba! Nenalezen zdroj odkazů.

10.2.2 Rozhodovací uzly

DecisionBinaryNode

Základní z rozhodovacích uzlů. Parametry jsou uzly true a false a podmínka, jež je předávána jako delegát na metodu s nula parametry a návratovou hodnotou bool.

Často jsou potřeba metody, které mají různý počet různých parametrů. K tomu se ideálně hodí anonymní lambda funkce v jazyce C#.

```
DecisionBinaryNode hasSecondLicence =
    new DecisionBinaryNode(trueNode, falseNode,
        () =>
        {
            me = map.GetPlayerMe();
            activeLicenceKind = me.GetMarketLicence(activeSourceKind);
            return activeLicenceKind == LicenceKind.SecondLicence;
        });
```

Kód 9 Využití lambda funkcí jako podmínky v rozhodovacích uzlech.

SerialNode

Jeho parametry jsou seznam uzlů a uzel návratový. Postupně se provedou podstromy z každého ze seznamu uzlů v pořadí, jakém byly přidány do seznamu. Pokud se v jednom z podstromů dosáhne ActionNodu, akce se zavolá a strom se znovu prochází od kořene. Jestliže se nedosáhne žádné z pravých akcí (nikoliv FakeActionNode), pokračuje se uzlem návratovým.

Chování uzlu odpovídá chování stromu z Obr. 24 Výřez rozhodovacího stromu. Místo řetězení několika DecisionNode v sérii by mohl být použit právě SerialNode, který by dostal na vstupu uzly CanBuildRoad, HasMarketWithFreeSlot, HasMonasterzWithFreeSlot, HasFort.

RandomNode

Jsou dvě varianty tohoto typu uzlu. Multiple a One. V obou případech je parametrem konstruktoru seznam uzlů. V prvním případě se vytvoří náhodná permutace těchto uzlů, a poté se postupně volají jednotlivé podstromy nově seřazených uzlů. Ve verzi One se provede právě jeden z podstromů.

Tento uzel jsem zařadil, aby se AI nechovala příliš předvídatelně a nebyla pouze deterministická. Obtížnost hry přidáním náhodného uzlu spíše utrpí, jelikož je možné, že se vlivem náhody provede méně dobrý tah.

RandomNodeMultiple jsem využil pro akce postavení speciálních budov. Jako autorovi hry mi nepřijde žádná ze speciálních budov lepší než jiná, tedy mi přišlo správné kontrolovat vhodnost postavení speciálních budov v náhodném pořadí.

SourcesNode

CanHaveSourcesNode a *HasSourcesNode* byly vysvětlovány dříve. První se ptá, jestli může mít hráč suroviny na zaplacení akce v daném kole. Druhá se ptá, jestli je už má aniž by musel suroviny měnit.

HasSourcesNode má vždy jako *trueNode* *actionNode*, který akci vykoná, a jako *falseNode* *ChangeSourcesActionNode*. Tento uzel vždy musí následovat po *CanHaveSourcesNode*, avšak nemusí být jeho přímým potomkem.

ForBestNode

Nyní se dostáváme k řešení IV. problému nastíněného v úvodu kapitoly o Decision Tree AI. Jak vyřešit velký stavový prostor? Inspiroval jsem fitness funkcí z genetických algoritmů.

ForBestNode existuje v pěti variantách : 1) *ForBestFortHexaNeighbour*, 2) *ForBestPlayerNode*, 3) *ForBestPlaceNode*, 4) *ForBestRodeNode* a speciální verze 5) *ForEachFreeHexaInTownNode*. Parametry konstruktorů jsou *trueNode* a *falseNode*. Zde jsou to nepřesné názvy. *TrueNode* se vykoná po nalezení nejvhodnějšího objektu. Pokud se průchodem *trueNode* nedosáhne akce, pokračuje se *falseNode* dále.

- 1) Využito při rozhodování se o akci Obsadit pole. Vezme všechny okolní hexy všech pevností hráče a vybere tu, kde získá za kolo nejvíce surovin. Zohledňuje pokroky soupeřů. Vybere nejlepší variantu a dále se pokračuje v rozhodování, jestli ta nejlepší varianta je vhodná pro vykonání.
- 2) Hledá hráče s nejvíce surovinami. Více si cení surovin, které hráč sám nemá. Důležité pro akci Ukrást suroviny.
- 3) Jestli hráč může postavit na více koncích svých cest města, vybere to místo, kde získá nejvíce surovin z okolních hex.
- 4) Vybere ze všech cest, které sousedí s hráčovou infrastrukturou, nejvhodnější cestu pro stavbu. Fitness se počítá na základě fitness města na konci cesty a na fitness měst, které jsou na konci cest vedoucích z dané cesty.
- 5) Oproti předchozím čtyřem uzlům zde ohodnotí všechny volné hexy ve městech, seřadí je dle jejich fitness a postupně zkouší jejich vhodnost. Teď je vhodná otázka, proč pouze tento uzel je *ForEach* a naopak, proč ostatní jsou pouze *ForBest*? Důvodem jsou ceny surovin. Předchozí 4 uzly se týkají akcí, která pokaždé stojí stejně. Zde tomu tak není. Cena každé z budov se skládá z jiných surovin. Tento uzel je použit pro těžební i speciální budovy. Liší se pouze předáním parametru, který říká, jestli čím vyšší fitness hexy, tím líp, nebo opačně. Fitness hexy je dáno množstvím a druhem suroviny, jež lze z ní těžit. Speciální budovy je nejlepší stavět na poušti.

10.3 Shrnutí

Rozhodovací stromy se ukázaly jako dobrou volbou pro vytvoření AI. Navrhnout ji nebylo nijak složité. Lze ji jednoduše rozšiřovat, přidávat nové větve, spravovat.

Rozdíly oproti základní AI :

- 1) Cesty nestaví náhodným směrem. Vybírá se vždy cesta s největší hodnotou fitness.
- 2) Nestaví další cestu, jestliže může postavit město na konci jedné z cest.
- 3) Mění suroviny účelně za akce, které chce využít.
- 4) Pokud má dostatek surovin pro jakoukoli akci, suroviny vymění a akci provede. Bez ohledu na to, že by další kolo měla suroviny na mnohem vhodnější akci.
- 5) Umí používat speciální budovy.

Mezi největší slabiny mnou navržené DT-AI patří neschopnost myslet na více tahů dopředu. Neumí si našetřit potřebné suroviny, vždy vše utratí. Při rozhodování se o stavbě cest nebere v úvahu, že jiný hráč může postavit dané město dříve. I přes tyto nedostatky dokáže porazit lidského hráče⁵.

11 Goal driven architektura

Během tvorby Decision Trees AI mě napadala spousta věcí, jak by šla udělat v Goal driven architektuře, a proto jsem se rozhodl vyzkoušet vytvořit AI založenou na ní.

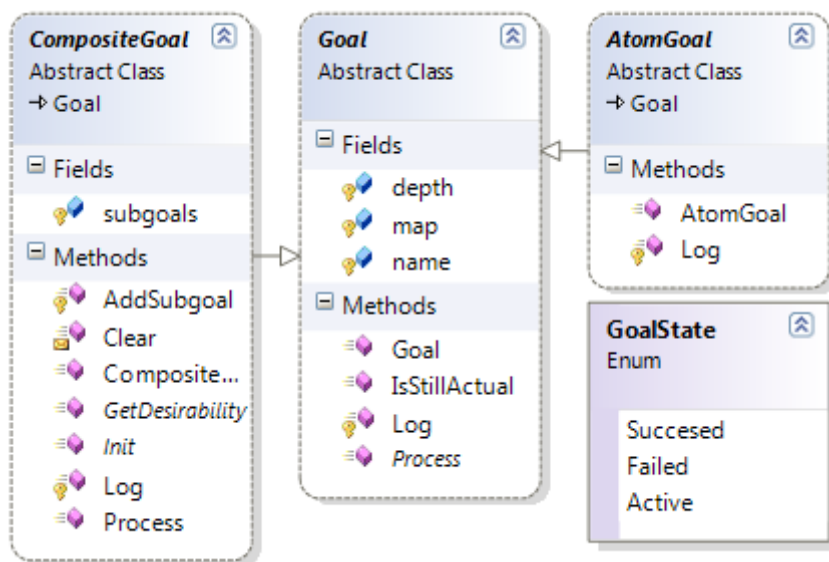
Ve hře se hráč může rozhodnout pro jednu z 13 akcí, v terminologii Goal driven cílů. Do kompletního seznamu patří postavit město, cestu, těžební budovu, pevnost, klášter, tržnici, obsadit pole hexy, okrást soupeře, předvést vojenskou přehlídku, vynalézt pokrok, koupit licenci, vyměnit suroviny, počkat kolo a pustit dalšího hráče k jeho tahu. Jednotlivým akcím odpovídají atomické cíle.

Složených cílů existuje 12, z toho 10 má definovanou GetDesirability metodu, která určuje vhodnost cíle v daném stavu. Zbýlé dva cíle do 12 jsou hlavní cíl ThinkGoal a cíl získání suroviny RaiseSources.

Většina složených cílů je složena pouze z RaiseSources cíle a atomického cíle stejného jména jako složený cíl. Příklad BuildMarket se skládá z RaiseSources a BuildMarketAtom.

11.1 Struktura cílů

Pro implementaci cílů používám Composite pattern, který se pro ně hodí. Cíl je abstraktní třídou. Class diagram je znázorněn na Obr. 26.



Obr. 26 Class diagram cílů

Proměnné name a depth slouží pouze k zalogování aktuálně prováděného cíle, viz Kód 10.

Každý cíl má definovanou metodu IsStillActual, která defaultně vrací true. Má za úkol zjistit, jestli všechny podcíle aktuálního cíle jsou stále proveditelné. Např. AI se rozhodne postavit

⁵ Viz kapitola 12.4

město, cíl postavení města se skládá z cílů postavit N cest a na závěr postavit město. Může se stát, že soupeř postaví na daném místě město dříve než se k tomu dostane naše AI. V tom případě se musí zrušit aktuálně prováděný cíl a naplánovat cíl jiný. Jinak by došlo k nelogickému stavění cest k místu, kde by AI zjistila, že už město postavit nemůže a pozdě by přeplánovala svůj cíl.

Druhou sdílenou metodou je Process, která vrací GoalState. GoalState je výčtový typ o třech hodnotách Succeded, Failed, Active. Stav Active vrací pouze atomární cíl WaitTurnAtom a BuildTown v první fázi hry. Pokud je vráceno Active, plánování v daném kole je ukončeno, pustí se další hráč k svému tahu. Při návratu Succeded a Failed se dle vhodnosti vybere nový cíl, jež se bude plnit.

```
New Plan
Fitness > 0,446001038309747
Think {
  Build Source Building {
    Raise sources {
      @ Change sources > succes
    }
    @ Build source building > succes
  }
}
```

Kód 10 Ukázka logu složeného cíle

11.1.1 AtomGoal

AtomGoal definuje způsob logování atomických cílů. Zděděné konkrétní cíle mají implementovanou metodu Process, jež zpravidla obsahuje jedno volání metody z MapControlleru a dle úspěšnosti vrací GoalState. Některé cíle mají definovanou dříve zmíněnou metodu IsStillActual.

Kvůli jednoduchosti atomických cílů přidávám pouze jednu ukázkou BuildRoadAtom.

```
public override bool IsStillActual() { return !road.GetIsBuild(); }
public override GoalState Process() {
  if (road.Build() == null) {
    Log(GoalState.Failed);
    return GoalState.Failed;
  } else {
    Log(GoalState.Succeded);
    return GoalState.Succeded;
  }
}
```

Kód 11 Ukázka atomického cíle pro stavbu cest.

11.1.2 CompositeGoal

Složené cíle lze rozřadit do tří skupin. ThinkGoal je hlavní cíl, který má za úkol zvítězit. Obsahuje v sobě deset párů dvojic cíl – koeficient vhodnosti. Každý z těchto deseti cílů má definovanou GetDesirability, která vrací desetinné číslo v intervalu 0,0 – 1,0. Do třetí skupiny patří RaiseSources a BuildRoad, jež nemají GetDesirability, a tedy při plánování nemohou být zvoleny přímo v ThinkGoal.

Plánování ve hře funguje následovně. Jestliže ThinkGoal nemá žádný plán, neobsahuje podcíle, rozhodne se vybrat nový cíl z druhé skupiny. Zavolá jejich funkci GetDesirability, výsledek vynásobí zadaným koeficientem a vybere největší z těchto součinů.

Poté zavolá funkci Init daného cíle, která rozplánuje cíl na podcíle a přejde k její Process metodě.

Má-li v sobě WaitTurnAtom, cíl bude plněn skrz více kol, jinak se provede v rámci aktuálního kola. Po splnění kompletního cíle, či jeho selhání, ThinkGoal naplánuje další postup. Metoda Process sdílená všemi CompositeGoal je na Kód 12.

```
while (subgoals.Count > 0) {
    foreach (Goal goal in subgoals)
        if (!goal.IsStillActual())
            return GoalState.Failed;

    Goal actualGoal = subgoals.Peek();
    GoalState state = actualGoal.Process();
    switch (state) {
        case GoalState.Failed: subgoals.Clear();
                                return GoalState.Failed;
        case GoalState.Succeded: subgoals.Dequeue();
                                break;
        case GoalState.Active: return GoalState.Active;
    }
}
```

Kód 12 Část metody Process z CompositeGoal

Nyní bude následovat bližší popis několika vybraných složených cílů.

RaiseSources

Tento cíl má úkolem získat suroviny pro konkrétní účel, a proto bývá použit před většinou atomických cílů.

Při inicializaci se rozhodne, jestli má zařadit do cíle čekání pár kol WaitTurnAtom, nebo pouze vyměnění surovin ChangeSourcesAtom. Tím se liší od DTAI, která vymění suroviny, jakmile může, přestože další kolo by mohla danou věc pořídit bez nevýhodného měnění později důležité suroviny.

Čekání mezi své podcíle nezahrne, jestliže alespoň jeden ze soupeřů má pevnost a on sám má hodně surovin ($\frac{3}{2}$ součtu surovin ceny za akci Kradení surovin), nebo pokud sám má součet surovin větší než 500.

Druhou z podmínek jsem musel přidat, protože jinak se stávalo, že hráčovi často chyběl jeden druh suroviny, na kterou se snažil čekat, přestože jiných surovin měl po stovkách.

BuildSourceBuilding

Cíl, který má definovanou jednoduchou vhodnostní funkci. Pokud nemá hráč žádné město s volnou hexou, vhodnost je nula. Rozhodl jsem se pro toto zjednodušení a nezahrnovat podcíl BuildTown, v případě, když by hráč neměl volné město.

Podcíle má pouze dva, RaiseSources a BuildSourceBuildingAtom. Podobně se skládají ze dvou podcílů i další složené cíle, proto u dalších cílů nebudu uvádět jejich podcíle, jestliže budou stejné jako zde. (jen s jiným odpovídajícím atomickým cílem)

Vhodnost je tvořena ze základní výnosnosti budovy (bez vlivu pokroků) a dle aktuálních surovin hráč a ceny budovy jednoduchým vztahem $(S / 24) * kS + (1 - \frac{C}{100}) * kC$. S – výnosnost hexy bez náhodných událostí, 24 – maximální výnosnost ve hře, C – kolik hráč musí vyměnit

surovin pro koupi budovy , kS a kC koeficienty určující z čeho se skládá desirability více kS + kC = 1.

Pokud může být postavena těžební budova na více místech, spočítá se vhodnost pro všechny a vybere se budova s nejlepším výsledkem.

BuildMarket

Speciální budovy stejně jako těžební mají vhodnost rovnou nule, když není kde je postavit. Vhodnost tvoří 4 části, které jsou váženy koeficienty. Koeficienty se v součtu rovnají 1, aby výsledná vhodnost nabývala hodnot od 0 do 1. Vhodnost ovlivňuje –

- 1) Výnosnost hexy – u speciálních budov, čím vyšší výnosnost, tím hůře
- 2) Suroviny potřebné ke stavbě – čím více by hráč musel měnit surovin, tím menší vhodnost
- 3) Celkový výnos jedné suroviny za kolo – jestliže hráč získává od jedné suroviny velké množství, vyplatí se mu tržiště a koupit licenci v něm
- 4) Má někdo jiný tržiště – jestliže ne, postavením tržiště hráč získá medaili a s ní potřebné body

Výsledek je násoben číslem menším než jedna, jestliže hráč už jedno tržiště postavil. Prakticky stejně je tvořen cíl BuildMonastery a ne příliš odlišně i cíl pro stavbu pevnosti.

Vhodnosti cílů koupit licenci a vynaleznout pokrok tvoří pouze bod 3).

FortCaptureHexa

Opět, pokud hráč nemá pevnost, vhodnost je nulová. AI se v tom případě nesnaží postavit pevnost, či dokonce postavit město, v kterém by mohla pevnost postavit.

Vhodnost se počítá pro každou ze sousedních hex všech pevností a vybere se ta maximální, jež se vrátí jako vhodnost celé akce. Spočítá se jako suma surovin, jež by hráč navíc získával každé kolo děleno 144. Na každé hexe můžou být maximálně 3 budovy⁶, v základu nejvyšší výnosnost je 24, s druhým pokrokem je dvojnásobná, $2 * 24 * 3 = 144$.

BuildTown

Idea stavění měst se zde liší od DT-AI. V ní bylo hlavní stavění cest. Vždy se zvolila cesta, která vedla směrem, kde by se případně dalo postavit výhodně město. Když se na jejím konci mohlo postavit město, v nejbližších tazích se postavilo. Vybírá pouze z cest, které může postavit, z těsně navazujících na již postavenou infrastrukturu.

GD-AI přistupuje k problému odlišně. Nejdříve nalezne na mapě místo pro stavbu města, které se líbí nejvíce, poté zjišťuje jaké cesty musí nejdříve postavit, aby mohl vybudovat město.

DT-AI při zakládání cest nebrala v úvahu, že už na jejím konci nestihne postavit město. Brzy postavené cesty prozrazovaly další úmysly počítače. Zde jsem to trochu napravil. AI nejdříve vydělá dostatek surovin na cestu + město, a poté postaví poslední cestu z městem zároveň v jednom tahu.

⁶ Kolem hexy lze postavit maximálně 3 města, protože spolu nemohou přímo sousedit.

Výpočet vhodnosti se částečně liší pro první fázi hry, rozestavování budov, a pro hlavní fázi. Nejdříve popíšeme společný základ pro obě fáze. Podobně jako u stavby budov se i zde spočte vhodnost pro každé z měst, poté se vybere město s největší hodnotou, která je i výsledná pro cíl BuildTown.

Vhodnost města se spočte jako součet vhodností tří sousedních hex. Záleží na jejich absolutní výnosnosti, ale také i na druhu suroviny, které produkují. Suroviny jsou váženy koeficienty dle zjištěných hodnot v 12.1.3. Pokud se jedná o surovinu, kterou hráč prozatím netěží, je vhodnost vynásobena konstantou 3,5. Pokud hráč nezískává některou ze surovin, musí za ní jiné suroviny měnit, z počátku v poměru 4 ku 1, později lépe – kompromis 3,5.

Navíc AI upřednostňuje poušť před vodou, protože na poušti může stavět speciální budovy.

První fáze

Při rozestavování první dvou měst AI zohledňuje počet možných měst, které jsou právě vybíranému město nejbližší. Algoritmus je založen na prohledávání do šířky. Z každého již postaveného města se pošle „černá“ vlna. Z místa, kde bychom chtěli postavit město se pošle vlna „bílá“. Algoritmus spočítá, kolik volných měst zasáhne „bílá“ vlna dříve než vlny „černé“. Nutné zdůraznit, že i vlastní dříve postavená města vysílají vlnu „černou“. Do seznamu Open jsou nejdříve vložena „černá“ města, tedy pokud je některé město stejně daleko od černého i bílého města, nezapočítá se.

Hlavní fáze

Vhodnost jednotlivých měst musím vážit jejich vzdáleností na počet nepostavených cest od již zastaveného území. K výpočtu opět využívám prohledávání do šířky. Při inicializaci algoritmu se vloží do seznamu Open všechny zastavené uzly grafu se vzdáleností 0. Zastavené uzly jsou postavená města a konce postavených cest. Poté proběhne klasické prohledávání do šířky, prohledává se pouze skrz nepostavené cesty.

Algoritmus si zároveň uchovává u každého uzlu grafu uzel předchozí, z kterého se do něj dostal, aby bylo možné později rekonstruovat cestu⁷.

Přišlo mi vhodné uložit algoritmus na hledání cest do MapControlleru, pro využití v jiných AI. Metoda GetDistanceToTown kromě parametru cílového města přijímá hráče, ke kterému se dotaz váže. Umožňuje to porovnat vzdálenost cílového města i od ostatních hráčů. Ptá-li se AI opakovaně na vzdálenost různých měst k jednomu hráči, vzdálenosti se znovu nepočítají, jsou ve hře uloženy dokud jsou platné. Platné přestanou být, pokud některý z hráčů postaví město, či cestu, nebo pokud se AI začne ptát na vzdálenost k jinému hráči.

Pomocí GetRoadsToTown získá AI posloupnost cest končící cestou vedoucí do tázaného města. S metodou GetDistanceToTown sdílí data i algoritmus. Metody se pouze liší v návratové hodnotě.

Pro první fázi hry jsem naprogramoval jiný algoritmus, protože by bylo zdlouhavé využívat GetDistanceToTown pro všechna města a jednotlivé hráče. Pro každého hráče by musel algoritmus projít znovu celou mapu.

⁷ Zde se cestou myslí grafový pojem, uspořádaná množina hran.

11.2 Genetický algoritmus

V Goal Driven architektuře jsou důležité koeficienty, které ovlivňují vhodnosti jednotlivých cílů. Koeficienty zde figurují na dvou úrovních. Vhodnosti některých cílů jsou složeny z více vhodnostních funkcí, které jsou váženy koeficienty nižší úroveň jejichž součet je roven jedné. Potom jsou zde koeficienty na vyšší úrovni, které ovlivňují, které akce hráč bude upřednostňovat. Dále koeficient 1,0 pro cíl FortBuild a 0,001 pro cíl MarketBuild, pravděpodobně tržiště hráč nikdy nepostaví, naopak pevnost ano.

Určit tyto dvě sady koeficientů je složité. Podle svého uvážení jsem koeficienty natvrdo určil a spustil jsem mnoho her, jak si GD-AI s mými koeficienty vede oproti DT-AI. Při některých nastaveních měla více vítězství, ale většinou spíše pokulhávala za DT-AI.

Napadl mě experiment, vyzkoušet sady koeficientů vyšlechtit pomocí genetického algoritmu. Od počátku jsem věděl, že v tomto případě existuje spousta zádrhelů, ale přesto mě zajímalo, jestli je reálné zde genetický algoritmus využít. Můj cíl byl omezen na vyšlechtění jedince, který bude porážet DT-AI při jednom nastavení mapy. V případě, že by se mi podařilo takového jedince získat, šlo by předpokládat, že i pro jiná nastavení mapy lze vyvinout jiného jedince. Jedinec určený specificky pro danou situaci bude vždy lepší než jedinec univerzální.

Navíc by to vedlo k větší hratelnosti. Kdybych do hry přidal kampaň, nemuseli by se soupeře na jednotlivých mapách lišit pouze svým jménem a popiskem, ale každý by mohl mít odlišné chování, každý by mohl být vycvičen na svou konkrétní mapu.

11.2.1 Proč říct Ne evoluci

Předchozí řádky mohou vyznívat skvěle, ale v úvodu této podkapitoly jsem uvedl, že zde bude spousta problémů.

Zaměřeno na AI

Jedním z nich je, že vyšlechtěná AI oproti DT-AI může být dobrá pouze proti ní, neznamená to, že porazí hráče. Je tu reálná možnost, že pouze rozpozná slabinu druhého hráče a toho začne využívat. Bohužel naopak je nereálné ji vyvíjet proti lidským hráčům, jelikož by museli odehrát tisíce her.

Fitness

U popisu genetických algoritmů je často uváděno, že fitness funkce musí být navržena tak, aby byla rychlá. Populace musí být v krátkém čase ohodnocena, aby mohla projít mnoha generacemi.

V mém případě fitness musí být založena na výsledcích z odehrané hry. Jedinec bude ohodnocen dle toho, jestli hru vyhraje a s jakým výsledkem, jinak rozumně definovat fitness nelze. Navíc odehrání jedné hry nestačí. V kapitole 12.1 ukazují, že výsledné pořadí hráčů je ovlivněno mnoha faktory, mimo jiné pořadím hráčů, vygenerováním mapy. Fitness tedy nelze založit pouze na jedné hře, ale musí být na několika.

Když jsem prováděl pokus o sto členné populaci, kdy fitness byla počítána na základě šesti her, ohodnocení jedné populace trvalo na mém notebooku téměř hodinu čistého času. Jedna hra s běžným nastavením trvá v řádu sekund, kolem 5.

Determinismus

Expanze se liší od her jako jsou šachy tím, že není deterministická. Stejně kroky neznamenají stejný výsledek. Jestliže jedinec bude hrát dvě hry na zcela stejné mapě, může pokaždé dopadnout jinak díky přítomnosti zázraků.

Vzniká zde problém, že i dobrý jedinec kvůli nepříznivým podmínkám může dopadnout špatně. Což je sice v přírodě zcela běžné, ale u genetického algoritmu je více než vhodné, když je fitness určena už tím, jak je jedinec definován.

11.2.2 Nastavení GA

Nastavení nebylo jednoduché, zvolit správně kódování chromozomu, druh selekce, křížení a mutace, výpočet fitness. Nejdříve jsem používal křížení jedním bodem, posléze třemi body. Při křížení jedním bodem se mi stávalo, že již kolem 10. generace se jednotliví jedinci v populaci od sebe téměř nelišili.

Při mutaci se náhodně přičetla, či odečetla hodnota od -0,1 - +0,1 k genu a posléze ořezala do 0,0 – 1,0.

První pokusy jsem využíval elitářství a část stálé populace. Později jsem přešel k selekci pomocí rulety.

Fitness je součtem fitness za jednotlivé hry. Fitness jedné hry je dána poměrem bodů vítězné a poražené AI s důrazem na vítěze. Chtěl jsem, aby vítězství bylo vždy víc než prohra a zároveň, aby se příliš nelišila fitness jedince, který prohrál se 74 body a který vyhrál se 75 body.

Více možností jsem nevyzkoušel, většinu pokusů jsem provedl při křížení více body a selekci pomocí rulety. Hlavní důvodem byla již dříve zmíněná dlouhá doba vývoje populace.

Chromosome

Chromosome se zde skládá z celkem 19 čísel typu double. Koeficientů je celkem 31, ale koeficienty nižší úrovně týkající se jednoho cíle jsem zakódoval do čísla jednoho.

Chtěl jsem snížit poměrně velké množství genů chromozomu a zároveň umožnit zanechat význam křížení jaký má být. Kdybych nezakódoval více souvisejících koeficientů do jednoho, mohlo by se stát, že by při křížení si nový jedinec vzal část z těchto koeficientů z jednoho Chromozomu a druhou z jiného. Tím by narušil to, co již bylo dříve vyšlechtěno a zanesl by do populace novou genetickou informaci, což má být úloha mutace, ne křížení.

Kódování jsem použil velice jednoduché. V případě, že cíl má pouze dva koeficienty, tak jsem ho nahradil jedním k , druhý se snadno dopočte $1 - k$. V ostatních případech oželím přesnost a z prvních 3 desetinných míst udělám první koeficient, z dalších 3 míst druhý koeficient atd.

11.2.3 Výsledky

Zaměřil jsem se na hru pouze dvou hráčů, kdy jedna je GD-AI, druhá DT-AI a na nastavení mapy malá, normální, střední výnosnost na 75 bodů. Tyto omezení jsem zvolil z důvodu rychlosti výpočtu fitness. Čím více hráčů, tím déle hra trvá.

Nejdříve jsem funkčnost ověřil při vývoji populace na stejné mapě, s vypnutými náhodnými událostmi, bez měnění pořadí hráčů. Kdyby v sobě DT-AI neměla zakódován podíl náhody

pro rozhodování, vývoj hry by byl zcela deterministický. Do 20. generace fitness prudce rostla, poté začala oscilovat kolem konstantní hodnoty.

Poté jsem chtěl přejít rovnou k počítání fitness na různých mapách s měněním hráčů se zapnutými náhodnými událostmi. Vývoj generací jsem nechal provádět na svém PC více jak 24 hodin. Rostoucí trend byl velice nevýrazný. Neviděl jsem žádný pokrok a algoritmus jsem ukončil. Náhodné události znemožňovaly vývoj. Kromě fitness všech jedinců v populaci a průměrného fitness jsem si každou generaci logoval i nejlepšího jedince s kompletní sadou chromozomů. Mimo jiné pro případ, kdyby hra z neočekávaných důvodů spadla. Vybral jsem jedince, který dosáhl ze všech populací nejlepšího výsledku a spustil s ním 2 tisíce her na stejném typu mapy, co byl vyšlechtěn. Bohužel dopadl hůře než dřívější jedinec, kterému jsem určil koeficienty ručně.

Nakonec jsem se rozhodl pro kompromis. Oproti předchozímu pokusu jsem vypnul pouze náhodné události. Mapy se měnily každou hru, i pořadí hráčů. Rozhodnutí se nakonec ukázalo správné. Po dalších cca 24 hodinách evoluce vznikl jedinec, který byl otestován při 2 tisíc hrách na stejném typu mapy, ale se zapnutými náhodnými událostmi. GD-AI zvítězila ve 1441 her a prohrála v 559 hrách. Poté jsem změnil nastavení na Rychlou hru, která se lišila pouze počtem bodů nutných k dosažení vítězství, místo 75 se nyní hrálo na bodů 50.

Opět se GD-AI předvedla s 1113 vítězstvími a 887 prohrami. Dříve byl na stejné mapě poměr opačný, 1221 výher pro DT-AI a 788 proher. Vyzkoušel jsem ji i na více odlišných nastaveních hry, ale výsledky už nebyly tak dobré. Konkrétně u hry všech tří AI dopadly GD a DT stejně.

11.3 Shrnutí

Vytvořit umělou inteligenci založenou na hierarchickém plánování nebylo příliš složité. Největším oříškem bylo určit, na základě čeho se mají určit vhodnosti všech cílů. S mnou zvolenými koeficienty si GD-AI nevedla příliš dobře, což dokazují výsledky dále v 12.3.1 DT-AI vs. GD-AI.

Po využití genetického algoritmu se situace obrátila, ale pouze při nastaveních hry obdobných tomu, při kterém byla AI vyšlechtěna.

Pokud bych opět shrnul rozdíly oproti DT :

- 1) Odlišný přístup ke stavbě měst. Nepostaví cestu bez účelu postavit město na jejím konci.
- 2) V některých případech vyčkává a vykoná více akcí v jednom kole naráz.
- 3) Upřednostní čekání na vytěžení potřebných surovin před okamžitým měněním.
- 4) Rozhoduje se více dle toho, co je potřeba, ne dle aktuálních zásob surovin.
- 5) Při rozestavování prvních měst přihlíží k pozicím ostatních měst. Upřednostňuje části mapy, které nejsou využité na úkor počáteční těžbě z města.
- 6) Bere v potaz konání jiných hráčů. Pokud nikdo nepostavil některou ze speciálních budov, postaví ji aby získala body za medaili.
- 7) S vyšlechtěnými koeficienty je agresivnější než DT, využívá mnohem více pevnosti a obsazování hex.

Obecně bych chování označil za více lidské, přirozené. S postavením cesty a města zároveň dokáže soupeře překvapit. Šetření surovin také působí strategicky.

12 Testování

Plán testů jsem zaměřil na systémové a validační testy. Nevyužil jsem unit testů a integračních testů, protože by se v počítačové hře na mnoha místech špatně implementovaly a neměly by valné výsledky. Hry patří mezi event driven aplikace. Navíc díky náhodně vytvářeným mapám a přítomností zázraků a pohrom program není deterministický.

U každého testu je třeba zvážit, jestli se „zaplatí“. V tomto případě jsem došel k závěru, že nikoliv.

Zvláště u her nelze vynechat uživatelské testy. Vlivem tématu bakalářky jsem testoval na uživateli obtížnost AI. Mimo to jsem provedl automatické testy pravidel, monkey testy a otestoval jsem obtížnost jednotlivých AI mezi sebou.

12.1 Testování pravidel

Základ pravidel pochází ze hry Osadníci z Katanu, ale značně jsem ho pozměnil. Bylo důležité zjistit, jestli jsou pravidla hry vyvážená a umožňují více různých strategií vedoucích k vítězství.

Prvotními testy byly alfa testy v rámci týmu. Při hraní byly odhaleny mnohé nedostatky hry. Přání a námitky jsem zaznamenával do trackovacího systému na GitHubu, kde jsem je nechal nějakou dobu uležet, zvážil jejich pro a proti, a poté je buď zavedl do hry, nebo z trackovacího systému vymazal.

Od prvotní verze pravidel k současné došlo k mnoha změnám. Mezi hlavní patří vylepšení speciálních budov kláštera a pevnosti, které předtím nikdo nevyužíval.

Nyní stačí v klášteře vynaleznout pokrok a všechny budovy daného typu poté vynášejí více surovin. Dříve bylo nutné po vynalezení pokroku vylepšit každou budovu zvlášť za další suroviny. Než se vrátily investice, hra byla u konce.

Pevnost obsahovala 4 akce, kde se tři ukázaly jako příliš slabé a dvě z nich jako téměř redundantní. Akce poničení hexy, jež snižovala výnosnost z ní, byla zrušena. Akce obsazení hexy byla rozšířena. Dříve obsazená hexa nevynášela žádnému hráči suroviny, nyní vynáší suroviny pouze hráči, který ji obsadil. Nikdo také nevyužíval akci Zničení zásob soupeře. Především ve hře více hráčů útočící hráč pouze ztrácel, nic nezískával. Akci zničení zásob nahradila akce Ukradení zásob. Jak už název napovídá, jeden hráč suroviny ztratí, druhý je získá.

Dlouho diskutovaným tématem bylo, jestli umožnit ve hře boření budov. V pozdější fázi hry by se vyplatilo místo méně výnosné těžební budovy postavit jednu ze speciálních. V případě hry mnoha hráčů na malé mapě je nutností k výhře vlastnit pevnost a v ní kupovat Armádní přehlídku za 3 body. Pokud hráč včas neodhadne, kdy má postavit pevnost, nemá již šanci hru vyhrát. Přesto jsem se rozhodl boření budov do hry nezavádět a vylepšit umělé inteligence natolik, aby se jim jen zřídka stávalo, že prohrají z popsání důvodu.

Také ceny jednotlivých budov a akcí nebyly od počátku stejné. Především došlo k jejich zjednodušení a částečnému vyvážení. Když jsem si ani já jako autor nepomatoval po mnoha hrách, která budova stojí které suroviny, něco bylo špatně. Nyní už není ve hře budova, jež by stále od každé suroviny jiné množství.

Zajímavým experimentem bylo využít umělou inteligenci k otestování některých pravidel.

Využil jsem možnosti hry, kdy lze nechat hrát proti sobě pouze hráče ovládané umělou inteligencí. Nechal jsem běžet tisíce her s různými nastaveními a logoval jsem do .csv souboru závěrečné statistiky.

U každé hry jsem zaznamenával počet kol hry a jednotlivé hráče v pořadí, v kterém skončili. U hráčů jsem logoval druh AI, pořadí, v kterém začínali hru, počet získaných bodů, postavených měst, cest, medailí na konci hry, postavených tržišť, koupených licencí, postavených klášterů, vynalezených pokroků, postavených pevností, provedených akcí.

Dále rozebrané záznamy her lze naléznout na přiloženém CD.

12.1.1 Závislost na pořadí hráčů

První mě zajímalo, jestli nejsou hráči ve výhodě, když začínají hru, uzavírají. K testu jsem využil střední normální mapy se středními výnosnostmi na 75 bodů. Aby výsledek byl vypovídající, pustil jsem 1000 her pro trojici stejných hráčů.

Tabulka 4 Závislost výhry na pořadí hráčů Goal Driven AI

Začínal\skončil	První	Druhý	Třetí
První	471	296	233
Druhý	273	398	329
Třetí	256	306	438

Tabulka 4 dokazuje, že na pořadí hráčů závisí. AI, která začínala, téměř v polovině případů vyhrála. Podobných hodnot nabývaly i ostatní dvě AI.

12.1.2 Jak ovlivňují náhodné události hru

Pro tento pokus jsem musel vygenerovat na počátku jednu mapu a tu jsem beze změny pustil 1000x. Jednotlivé hry se lišily pouze tím, kdy a jaké suroviny postihl zázrak, či katastrofa. Vyzkoušel jsem hru tří Goal Driven, a poté jsem opakoval pokus pro tři hráče Decision Tree, Goal Driven, Základní v tomto pořadí. Zaměřím se na výsledky druhé trojice.

Na stejné mapě Decision Tree vyhrála 626x, Goal Driven 374x, základní AI zázraky nepomohly a nezvítězila ani jednou. Hra skončila průměrně v 31. kole, nejméně k výhře stačilo 22 kol, nejvíce 37. Nejnižší bodový zisk výherce 75 bodů, nejvyšší 85, průměrně 77. Vítěz průměrně utratil 4556 všech surovin, nejméně 3780, nejvíce 5800.

Je vidět, že náhodné události hru podstatně ovlivňují, přestože se projeví průměrně každý desátý tah. Z tohoto důvodu jsem opakoval předchozí pokus s vypnutými zázraky. Začínající hráč vyhrál 425x z 1000 her, což je podobný výsledek jako při měření s náhodnými událostmi.

12.1.3 Využití jednotlivých surovin

Bylo potřeba zodpovědět otázku, které suroviny jsou nejdůležitější. Opět jsem pustil 1000 her se třemi různými AI a udělal výslednou statistiku.

Vítěz průměrně spotřeboval 821 kusů obilí, 848 ovcí, 1087 kamene, 737 dřeva, 1105 rudy. Průměrně jakýkoliv hráč využil 645 obilí, 680 ovcí, 945 kamene, 582 dřeva, 978 rudy.

Do spotřebovaných surovin se počítají pouze suroviny nutné přímo ke koupi, nereflektuje to výměnu surovin. Když hráč postavil cestu za 50 kamene a 50 rudy, započítalo se do statistiky po 50 kusech u těchto dvou surovin, ať už je měl původně, nebo pro zaplacení 50 kamene využil 200 obilí.

Kámen a ruda jsou nejdůležitějšími surovinami ve hře, naopak dřevo je využito nejméně. U něho nelze s jistotou říct, že není tak důležité jako ovce, či obilí. Strategii AI jsem mohl nastavit tak, že upřednostňuje akce, jež dřevo nevyužívají, ale u živého hráče to tak být nemusí.

Na základě této statistiky jsem upravil vhodnost jednotlivých surovin při výběru města u GD-AI i DT-AI.

12.1.4 Vhodnost všech kombinací nastavení mapy

Vyzkoušel jsem, jak budou úspěšné jednotlivé AI na mapě, na níž je velice obtížné vyhrát. Dosadil jsem 6 stejných hráčů do malé mapy typu nížina (bez pouští, minimum kamene a rudy) s nízkou výnosností surovin na 100 bodů a sledoval, jaká AI si s tím poradí. Zde jsem spustil hru s každou AI pouze 20x.

Goal Driven se svým nastavením koeficientů pro desirability jednotlivých akcí zcela selhala. Každý hráč mohl obsadit velice malé území a získal maximálně 5 měst. Potřebné body šlo získat pouze pomocí pevnosti a akce Armádní přehlídka. Ani v jedné z 20 her nevyhodnotila Goal Driven situaci natolik, aby postavila pevnost, a tedy ji nikdy nestačil maximální limit 100 kol k dosažení vítězství. Nejlepší hráč získal průměrně pouhých 29 bodů, nejméně 23.

Decision Tree AI pokaždé postavila pevnost, ale přesto pouze ve dvou případech dosáhla 100 bodů. Průměrně získala 71 bodů, minimálně 49. Průměrně provedla 27 akcí.

Základní AI také pokaždé postavila pevnost, ale neumí využívat jiné speciální budovy a neměla nikdy dostatek surovin, a tedy stejně jako Goal Driven ani jednou nedosáhla vítězství. Průměrně bodů 67, minimálně 57, což je kupodivu největší minimum mezi jednotlivými AI.

Závěrem tohoto pokusu lze říci, že existují kombinace nastavení mapy, která nejsou vhodná a je u nich téměř nerealné dosáhnout vítězství. Z tohoto důvodu jsem do hry zabudoval limit 100 kol na jednu hru. (pro 6 hráčů 600 tahů) V game designu bylo potřeba zvážit, jestli uživateli dovolit nastavovat „nesmyslné“ hry, nebo některé kombinace zakázat. Rozhodl jsem se ponechat volnost.

12.1.5 Evoluce testuje

Myšlenka, jak by šlo testovat pravidla pomocí genetického algoritmu. Mějme AI založenou na Goal Driven architektuře a pomocí evolučního algoritmu vyšlechtíme koeficienty, které upřednostňují jedny akce před druhými. Jestliže nejlepší jedinci budou mít koeficient váhy akce rovný, nebo blízký nule, znamená to, že akce není vhodná při postupu za vítězstvím. Zjištěný fakt by se dal do hry reflektovat snížením ceny nepoužívané akce, či jejím zesílením.

12.2 Monkey testování

Možnost spouštění tisíců her vybízí k využití Monkey testování. Při každé hře se generuje nová mapa, a to umožňuje vyzkoušet obrovské množství situací.

V případě, že umělá inteligence selže a vyvolá výjimku, kterou v sobě neodchytí, tuto výjimku zachytí hlavní program. Zaznamenaná o ní všechny dostupné informace do souboru.

Odstřihne hráče od konkrétní hry, a ostatní ve hře pokračují. V další hře může být opět testována, jestli nevyvolá stejnou chybu, nebo novou.

Pomocí Monkey testů jsem odchytil jednu závažnou chybu ve správě vláken. Při běžném nastavení se provedená akce okamžitě projeví v Modelu mapy, ale ve View je aktualizována se zpožděním. Chtěl jsem, aby hráč měl možnost sledovat stavbu jednotlivých budov soupeřem a zároveň, aby AI mohla nepřerušena pokračovat v plánování akcí, jako by se budova postavila okamžitě.

V případě, že si hráč nepřeje čekat a chce vidět všechny akce v instantním čase, nestačí pouze nastavit prodlevu mezi akcemi na 0ms. Kontrola, jestli má View zobrazit výsledek akce probíhá jednou za frame, a tedy i s nulovou prodlevou byl čekala. Původní řešení bylo zavolat kontrolu na prodlevu rovnou bez čekání na Update dané metody.

V 8432. hře nastala nepravděpodobná událost, kdy byla zavolána kontrola přímo z vlákna s AI a probíhalo zobrazení nově postavené budovy. Ve stejné části kódu bylo také hlavní vlákno. Obě vlákna prošla kontrolou na neprázdnost zásobníku s akcemi, které ještě nebyly zobrazeny. První vlákno z něj vyndalo poslední akci, pro druhé vlákno tam akce nezbyla a hra spadla.

12.3 Testování AI modulů

Při vývoji Decision Tree i Goal Driven AI bylo nutné otestovat jednotlivé změny a pokroky. Jestli provedená změna vede k vylepšení AI, nebo naopak. Stěžejní bylo, jestli se zlepšuje proti živým protihráčům. Bohužel mít při ruce někoho živého, kdo AI otestuje při každé změně, je nereálné. Navíc u testování pravidel jsem zjistil, že pouhá existence náhodných událostí může vést na stejné mapě k různým výsledkům, tedy by správně bylo nutné každou změnu otestovat několika hrami, což se nedalo provést.

Podobně jako u testování pravidel jsem i zde využil možnosti logování a spuštění stovek her za sebou. Na mém notebooku jedna hra tří AI trvá v řádu několika sekund, spuštění stovky her zabere snesitelných 10 minut. Obě AI jsem nejdříve testoval oproti Základní AI. Přestože ona se chová poměrně náhodně, dlouho první pokusy „chytrých AI“ byly poráženy. Goal Driven jsem mohl později testovat i proti hotové Decision Trees.

12.3.1 DT-AI vs. GD-AI

Součástí zadání bakalářské práce je vytvořit umělou inteligenci škálovatelnou dle obtížnosti. K tomuto problému jsem se rozhodl přistoupit vytvořením třech rozdílných AI.

Základní AI porazí bez problémů začínající hráč. Neměla by být pro nikoho výzvou, a proto ve hře vystupuje jako obtížnost Lehká.

K rozlišení obtížnosti DT-AI a GD-AI jsem využil možnost spustit tisíce her. Mělo se jednat o objektivní zhodnocení umělých inteligencí, ale má to pár háčeků. Za prvé hra není deterministická, v předchozí kapitole jsem uvedl, jak hru ovlivňuje počáteční pořadí hráčů, náhodná generace map a událostí ve hře. Za druhé vítězství jedné AI nad druhou neznamená, že proti lidskému hráči si nebudou vézt opačně. Ideální by bylo, kdyby obtížnost zhodnotilo dostatečné množství dobrovolníků, kteří by si postupně zahráli oproti oběma soupeřům. Bohužel se mi k tomu nepodařilo získat dostatek dobrovolníků.

Před využitím genetického algoritmu ke zjištění koeficientů GD-AI více vítězila DT-AI. Při hře tří hráčů, každý jiného typu, s nastavením mapy střední velikost, normální typ, střední

výnosnost a 75 bodů se v 1500⁸ hrách objevila na prvním místě 10 krát Základní, 810 DT a 680 krát GD.

Výsledky 4 hráčů, po dvou od GD a DT, na velké mapě, normální typ, vysoká výnosnost, 75 bodů byla jedna z DT 5058 první a 2942 krát se na 1. místě objevila GD.

Při boji dvou hráčů na malé mapě, normální se střední výnosností a na 50 bodů 1212 vítězí DT, 788 GD. Po vylepšení GD o podmínku, že nemá čekat na vytěžení surovin a měnit vždy, když má více jak 500 surovin v součtu, výsledek byl 1104 ku 896.

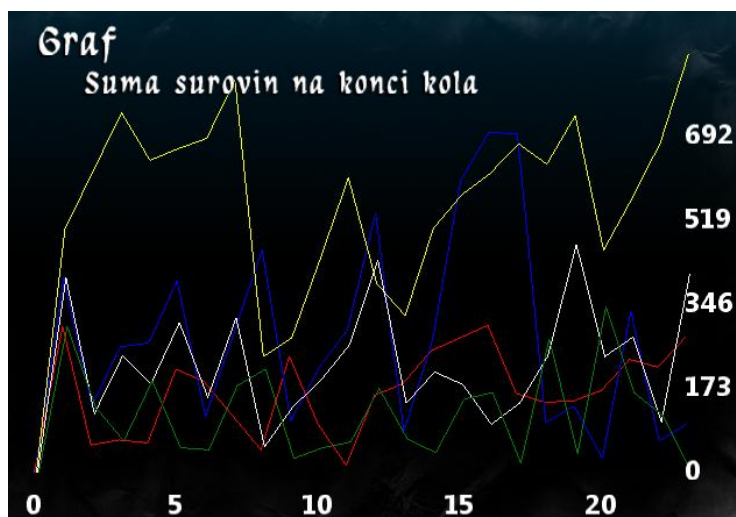
Ve všech zmíněných případech ze soubojů vychází lépe DT. Po nasazení genetického algoritmu se výsledky změnily. 2 hráči, malá mapa, normální, střední výnosnost, 75 bodů 1441 z 2000 her pro GD. Za těchto podmínek byla GD vyšlechtěna. Při souboji na stejné mapě pouze na 50 bodů 1113 z 2000 výher pro GD. Hra tří hráčů všech typů AI dopadla v poměru 13:744:743 pro Z:DT:GD. Zde si byly rovnocennými soupeři.

DOPSAT DOPSAT

12.3.2 Grafy ve hře

K odladování umělé inteligence jsem si vytvořil sadu grafů, která se vygeneruje v jádru hry a zobrazí se vedle tabulky vítězů. Inspirací mi byly jiné strategické hry, které po skončení také ukazují různé statistiky hráčům.

Sledoval jsem několik veličin vždy v závislosti na čísle kola. Počítal jsem počet bodů, měst, cest, medailí, tržišť, licencí, klášterů, pokroků, pevností, akcí a sumu surovin, které zůstali hráči na konci daného kola. Data o počtu speciálních budov spíše sloužila jako údaj o kole, kdy hráč postavil konkrétní speciální budovu. Málokdy někdo postaví více jak jednu speciální budovu stejného druhu.



Obr. 27 Po skončení hry si lze zobrazit několik grafů, které mohou odkrýt slabiny některého z hráčů. Zde je vidět, že žlutý hráč hromadil suroviny a byl vhodným cílem pro krádež.

Grafy odhalily velkou slabinu některých hráčů. U statistiky počtu bodů bylo vidět, že někdy někteří hráči už v půlce hry začali stagnovat a nezískali další body. Znamená to, že už se nemohli rozvíjet a ani si nestihli pořídit pevnost.

⁸ Hráči hráli ve všech možných pořadí po 250 hrách. Permutace 3 hráčů se rovná $6 \cdot 250 = 1500$

Dále prozradily, že Goal Driven AI příliš hromadila suroviny. Měla v sobě podmínku, že jestliže může získat konkrétní surovinu počkáním pár kol, má vždy počkat. Platilo to ale i v případě, že měla pouze jednu surovinu nedostatečnou, ostatní ji přebývali a měla jich na konci v součtu více jak 1000, a tedy byla velice vhodným kandidátem pro krádež.

Graf počtu medailí ukázal, že medaile hry vyhrávají. Jen zřídka kdy hru vyhrál hráč, který neměl nejvíce medailí ze všech hráčů.

Vedle statistik z tisíců her byly grafy dobrým pomocníkem pro vývoj AI. Především protože jednu hru uměli zhodnotit téměř okamžitě.

12.4 Uživatelské testy

Uživatelské testy jsem zaměřil na zjišťování, jestli mnou vytvořené umělé inteligence jsou schopné porazit alespoň začínajícího hráče. Cílem testů nebylo zjistit uživatelskou přívětivost ovládání, či vyváženost pravidel.

Testující uživatel měl za úkol se seznámit s pravidly hry, a poté si zahrát proti přednastaveným konfiguračním soupeřům. Test se vždy zaměřil na jednu umělou inteligenci.⁹ Ostatní hráč ignoroval. Na konci vyplnil tester krátký dotazník na webu.

Nevýhodou tohoto testu je jeho časová náročnost. I přes jednoduchost Expanze trvá nějakou dobu seznámení se se všemi možnostmi hry. Délka jednoho testu je minimálně 30 minut. Druhým problémem se stala nízká popularita tahových strategií.

Podařilo se mi otestovat hru čtyřmi lidmi. Není to vysoký počet, ale pro základní představu o úspěšnosti práce to stačí.

12.4.1 Nastavení testu

Test probíhal na počítačích uživatelů. Hra byla přístupná ke stažení na webu, stejně tak i dotazník, jež se měl poté vyplnit.

K vytvoření dotazníku jsem použil aplikaci googlu, která výsledky vyplněných dotazníků ukládá přehledně do tabulky.

Uživatel byl seznámen se základními pravidly hry a stáhl si archiv se hrou a uživatelským manuálem podrobně popisujícím pravidla hry. Seznámit se hrou se mohl pomocí manuálu, formou pokus-omyl, či sledováním hry dvou a více počítačů proti sobě.

Po vyzkoušení několika her byl dotazníkem instruován ke spuštění alespoň jedné hry od každé z přednastavených konfigurací.

První byla hra dvou hráčů na malé mapě s normálním rozložením surovin, se střední výnosností. Cíl získat 50 bodů. Počítač začínal. Tato konfigurace se nachází pod volbou Rychlá hra.

Druhá konfigurace byla přednastavena v Hot Seat menu. Hra tří hráčů, uživatel prostřední. Mapa středně velká se střední výnosností surovin s běžnými surovinami. Vítězem byl hráč se 75 body.

⁹ Goal Driven, či Decision Trees

12.4.2 Dotazník

Dotazník obsahoval 6 otázek. První dvě se týkali znalosti a oblíbenosti tahových strategií. Zbýlé 4 otázky se již zabývali jednou z umělých inteligencí.

- 1) Připadal vám soupeř inteligentní?
 - a. Váš subjektivní pocit. Prováděl akce, přestože vám přišlo, že v dané chvíli je výhodné udělat něco jiného?
 - b. Stupnice od 1 – Naprosto hloupý po 5 – Působil jako člověk
- 2) Byl postup soupeře odhadnutelný?
 - a. Věděli jste před každým dalším kolem, jak se váš soupeř zachová?
 - b. Stupnice od 1 - Naprosto předvídatelný po 5 – Nepředvídatelný
- 3) Bylo těžké soupeře porazit?
 - a. Oproti otázce, jestli se vám zdál inteligentní, tato otázka je objektivní.
 - b. Stupnice od 1 – Všechny hry jsem vyhrál po 5 – Pokaždé jsem byl poslední
- 4) Přišlo vám, že soupeř podváděl?
 - a. Věděl, co udělá jiný AI hráč, předpovídal katastrofy/zázraky?
 - b. Možnosti Ano/Ne

12.4.3 Výsledky

První vytvořená AI byla založena na rozhodovacích stromech, a proto byla také první otestována. Výsledné shrnutí znázorňuje Tabulka 5.

Tabulka 5 Průměrné hodnoty dotazníku k AI rozhodovacích stromů

Inteligentní	Odhadnutelný	Porazitelný	Podváděl
3,75	4	3,75	4 Ne, 0 Ano

Z tabulky lze vyčíst, že AI je důstojným soupeřem začínajících hráčů, kteří ji hodnotili jako nadprůměrně obtížnou. Nikomu nepřišlo, že by soupeř podváděl. V této hře to není možné ani provést, jelikož je celý herní plán viditelný všem hráčům, a kdykoli lze zkontrolovat množství surovin soupeřů.

Tento výsledek byl postačující pro ukončení vývoje AI založené na rozhodovacích stromech.

13 Závěr

Umělá inteligence v počítačových hrách je velice rozsáhlé téma. Přesto se domnívám, že začátečníkovi poslouží první část mé bakalářky více než dobře. Dává základní přehled o nejčastěji používaných technikách včetně příkladů, kde a jak se dají použít. Čtenář se v případě hlubšího zájmu o konkrétní techniku může obrátit na některou z citovaných knih.

Oproti fyzikálním a grafickým enginům neexistuje žádný univerzální nástroj pro tvorbu AI už jen z její podstaty. Mezi použitelná řešení patří enginy, které umožňují na nich založit celou hru daného žánru včetně umělé inteligence. Mezi ně patří OpenRTS, Quake Engine.

V druhé části jsem měl za úkol nabyté vědomosti z teoretické části bakalářky využít při tvorbě vlastní hry. Dříve jsem již vytvořil několik her a i jako semestrální práce z jiných předmětů¹⁰, avšak vždy buď umělá inteligence chyběla (logické hry), nebo stačila pouze založená na sérii IF-THEN-ELSE podmínek (Mario).

U Expanze jsem si vyzkoušel být v roli vedoucího tříčlenného týmu a s tím se pojila nutnost mé myšlenky převádět do podoby Design Dokumentu. Dalším přínosem je práce na jednom programu ve více programátorech, kde jsme museli s Alenou V. zanést určitý řád, protože přestože jsme měli na programu odlišné úlohy, některé zdrojové soubory jsme spravovali oba a museli jsme navzájem rozumět cizímu kódu. Tím se to lišilo od spolupráce např. v předmětech Y36SIN, Y36SI2, kde nás více programátorů dělalo na jednom projektu, ale buď ne paralelně, nebo každý vytvářel a spravoval vlastní soubory.

Mé vedení i volba hry s její rozsáhlostí bylo úspěšné a završeno vítězstvím mezi semestrálními pracemi z Y39PHA. Dále jsem pokračoval na hře sám.

Při tvorbě jsem se naučil pracovat v novém prostředí XNA ve verzi 4.0 a s tím spojeným jazykem C#.

Pro realizaci AI jsem zvolil techniku rozhodovacích stromů a později k tomu goal-driven architekturu. Nemohu obecně říct, která technika je pro Expanzi vhodnější, či dokonce obecněji pro jakoukoli strategickou hru. Nevhodnou implementací jedné z nich se mohl vytratit její potenciál. Přesto bych za vítěze považoval GD-AI, i když v některých situacích je druhou porážena.

Problém rozhodovacích stromů je poměrně jasný postup, jak na nich hráč založený bude ve hře postupovat. Věřím, že by se to brzy naučili i ostatní hráči. Navíc u GD-AI se mi líbí možnost její chování radikálně ovlivňovat pomocí koeficientů vhodnosti jednotlivých akcí a díky tomu snadno do jedné hry vložit několik různě hrajících hráčů. Důkazem, že AI se při jiných koeficientech chová zcela odlišně je mé využití genetického algoritmu.

Současné verze obou AI jsou schopné mě porazit, což splňuje můj osobní cíl a mojí hlavní motivaci při jejich tvorbě a vylepšování.

13.1 Diskuze nad pokračováním

Na hře jsou odvedeny dva celé semestry práce, z toho v jednom semestru práce tří osob. Přes velké množství času vloženého do projektu, hru nelze považovat za kompletní.

Expanze je nyní plně hratelná na jednom počítači v jednom až šesti hráčích na náhodně generovaných mapách. V dnešní době pro úspěch hry chybí možnost hrát po lokální síti, či lépe přes internet. Expanze neobsahuje prakticky žádné ozvučení, navíc chybí nějaký tutoriál zabudovaný přímo ve hře. Opět v dnešní době zásadní problém. Málokdo si přeje před začátkem hry otevřít manuál a číst si pravidla z něj.

Hru by určitě obohatila kampaň, předem připravené scénáře na různorodých mapách s odlišně chovajícími se soupeři. Mezi další nedostatky patří nemožnost uložit hru a později se k ní vrátit.

Z hlediska umělé inteligence je stále co vylepšovat. Ani jedna z umělých inteligencí nedokáže včas poznat, že už nepostaví další město a musí kvůli výhře postavit pevnost. Na jejich obhajobu, to samé se nepovede vždy ani mně. Dále v sobě nemají zabudovanou

¹⁰ Mario v Y36PJV, had v Y35PES, vlastní plošinovka v Y36PDA, univerzální 2D editor pro hry X36TJC

zákeřnost. Postavení cesty směrem, který není pro hráče výhodný, ale pro protihráče je velice nevýhodný.

Neumí příliš rozpoznat situaci, kdy je soupeř chce a může jednoduše uvěznit. Tedy postaví dříve město, ke kterému se soupeř nikdy nedostane před městem, které by mohl zabrat dříve než soupeř.

Kromě medailí spojených se stavbou speciálních budov, zisk ostatních medailí je vždy pouze vedlejším produktem nějaké akce. Nic nepostaví s přímým účelem obdržení medaile.

Na hře jako takové je stále a vždy bude, co vylepšovat stejně jako u jakéhokoli softwarového projektu. U všech projektů musí vždy někdo říct „dost“, tímhle to utneme a já to utnul v této fázi.

14 Citovaná literatura

1. **Lackore, Jason.** Survey on the use of artificial intelligence in video games. *Jason's Website*. [Online] [Citace: 20. leden 2011.]
<http://jlackore.com/Documents/Survey%20on%20the%20Use%20of%20Artificial%20Intelligence%20in%20Video%20Games.pdf>.
2. **Birch, Chad.** Understanding Pac-Man Ghost Behavior. *Game Internals*. [Online] 2. prosinec 2010. [Citace: 20. leden 2010.]
<http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior>.
3. **Tozour, Paul.** The Evolution of Game AI. [autor knihy] Steve Rabin. *AI Game Wisdom*.
4. **Scott, Bob.** The Illusion of Intelligence. [autor knihy] Steve Rabin. *AI Game Wisdom*.
5. **Kirby, Neil.** Solving the right problem. [autor knihy] Steve Rabin. *AI Game Wisdom*.
6. **Buckland, Mat.** *Programing AI by Example*. místo neznámé : Wordware Publishing, 2005.
7. **Bourg, David M. a Seeman, Glenn.** *AI for Game Developers*.
8. **Buckland, Mat.** *AI Techniques for Game Programming*. Ohio : Premier Press, 2002.
9. **Millington, Ian.** *Artificial Intelligence For Games*. místo neznámé : Morgen Kaufmann, 2006.
10. **Schwab, Brian.** *AI Game Engine Programming*. Hingham : Charles Rever Media Inc., 2004.
11. **Champandard, Alex J., Dawe, Michael a Herdandez-Cerpa, David.** Behavior Trees: Three Way of Cultivating Game AI. *GDC Vault*. [Online] 2010. [Citace: 11. únor 2010.]
<http://www.gdcvault.com/play/1012744/Behavior-Trees-Three-Ways-of>.
12. Duplicated code. *Source making*. [Online] [Citace: 7. únor 2011.]
<http://sourcemaking.com/refactoring/duplicated-code>.
13. Design patterns. *Source making*. [Online] [Citace: 7. únor 2011.]
http://sourcemaking.com/design_patterns.
14. **O'Brian, John.** A Flexible Goal-Based Planning Architecture. [autor knihy] Steve Rabin. *AI Game Programming Wisdom*.
15. Glider. *Conway Life*. [Online] 23. červenec 2010. [Citace: 20. leden 2011.]
<http://www.conwaylife.com/wiki/index.php?title=Glider>.
16. Life Simulation Game. *Wikipedia*. [Online] 17. leden 2011. [Citace: 20. leden 2011.]
http://en.wikipedia.org/wiki/Life_simulation_game.
17. Machine learning software. *DMOZ.org*. [Online] [Citace: 1. únor 2011.]
http://www.dmoz.org/Computers/Artificial_Intelligence/Machine_Learning/Software/.
18. jCOLIBRI CBR Framework. *Group for Artificial Intelligence Applications*. [Online] [Citace: 1. únor 2011.] <http://gaia.fdi.ucm.es/grupo/projects/jcolibri/jcolibri2/index.html>.
19. SOAR. *SOAR*. [Online] [Citace: 1. únor 2011.] <http://sitemaker.umich.edu/soar/home>.
20. Cipher Engine. *Cipher Engine*. [Online] Synaptic Soup. [Citace: 1. únor 2011.]
<http://www.cipherengine.com/>.
21. *Visual 3D Game Engine*. [Online] [Citace: 1. únor 2011.] <http://www.visual3d.net/>.
22. Quake 3 1.32 Source Code. *File Shack*. [Online] [Citace: 1. únor 2011.]
<http://www.fileshack.com/file.x?fid=7547>.
23. OpenRTS. *Libre Game Wiki*. [Online] [Citace: 1. únor 2011.]
<http://libregamewiki.org/OpenRTS>.
24. **Cochran, Matthew.** Introduction to Building a Plug-In Architecture Using C# . C# Corner. [Online] 10. září 2007. [Citace: 26. březen 2011.] http://www.c-sharpcorner.com/uploadfile/rmcochran/plug_in_architecture09092007111353am/plug_in_architecture.aspx.
25. **Brackeen, David.** *Vývoj her v jazyku Java*. Praha : Grada, 2004.

26. **Belis, Mary.** Computer and Video Game History. *About.com Inventors*. [Online] [Citace: 20. leden 2011.] http://inventors.about.com/library/inventors/blcomputer_videogames.htm.

A Obsah CD

results	
cell	Data files with results measured on Cell
x86	Data files with results measured on x86
src	
cell rt	IBM Cell Ray Tracer source
x86 client	x86 to Cell Client source
x86 rt	x86 Ray Tracer source
thesis	PDF and Microsoft Word versions of this text