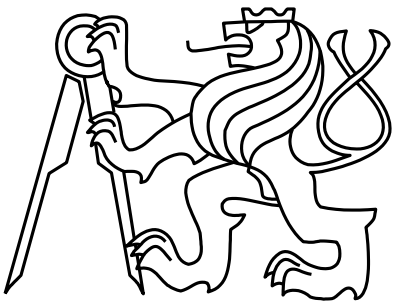
České vysoké učení technické
Fakulta elektrotechnická
Katedra počítačů



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

Declaration – Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém 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 17. ledna 2011 ……..

Lukáš Beran

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Abstrakt

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque porta vulputate dui eget convallis. Aliquam erat volutpat. Nulla facilisi. Vestibulum ante libero, mollis ac fringilla id, malesuada vel lectus. Cras id tellus dolor, vel consectetur ipsum. Aenean dignissim, sapien et viverra fermentum, lectus enim suscipit quam, id mollis neque dolor non magna. Integer volutpat est quis erat dapibus mattis. In a purus eget ligula egestas fringilla. Aenean semper risus a dolor pretium dapibus. Donec ac justo lacus. Mauris lobortis nulla lorem, vitae porttitor sapien. Sed malesuada tempus odio vitae mollis. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

„I have never let my schooling interfere with my education.“

Mark Twain

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

1 Úvod

V první kapitole, v druhé kapitole…

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 nezbýval 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 střílečce 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 testeři 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 vývojaře 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 simík 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.

1.3 Nejen soupeř je inteligentní

2 AI algoritmy a techniky

Algoritmy a techniky používané pro tvorbu herní umělou inteligenci 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.

3 Rozhodování

Zaútočit, zlepšovat obranu, přihrát spoluhráči, střelit 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.

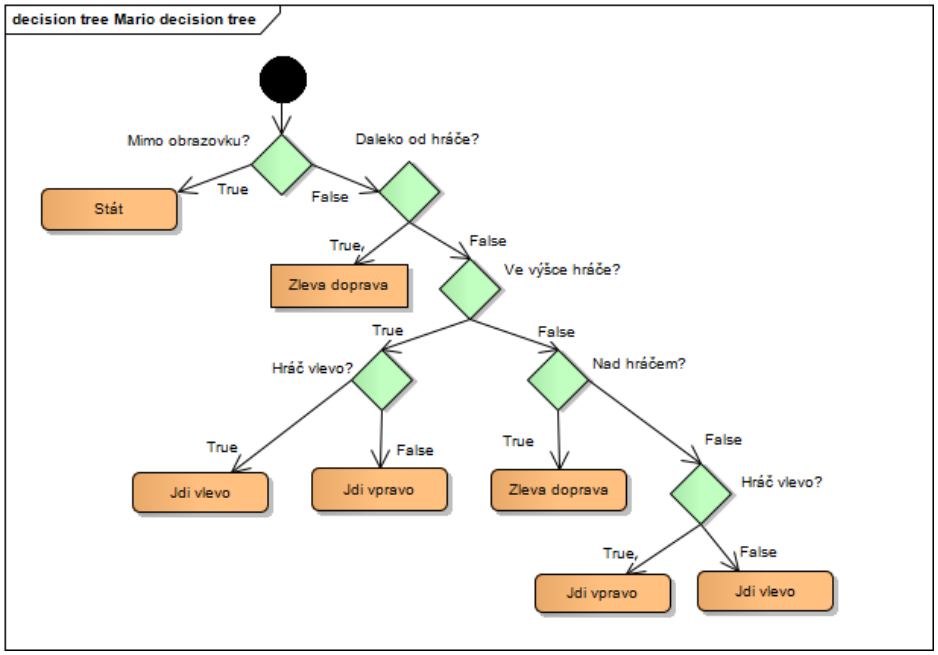
Existuje několik metod, jak ve hře rozhodování provádět. Mezi některé patří níže popsané Rozhodovací stromy, Stavové automaty, Goal-driven architektura a Fuzzy logika.

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 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 měli 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 plošince, 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

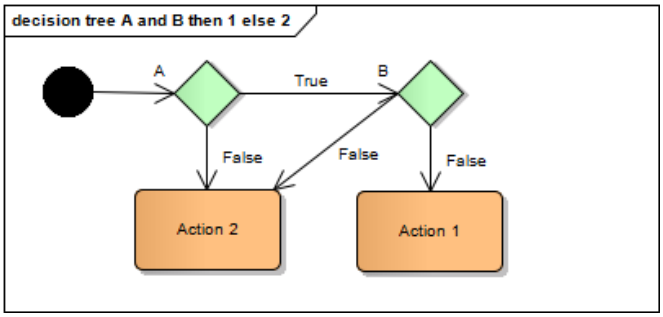
Každý vnitřní uzel by měl kontrolovat jednoduchou podmínku, zpravidla závislou na typu proměnné. Některé druhy podmínek shrnuje následující tabulka Tabulka 1.

Tabulka 1 Příklady datových typů dle [6]

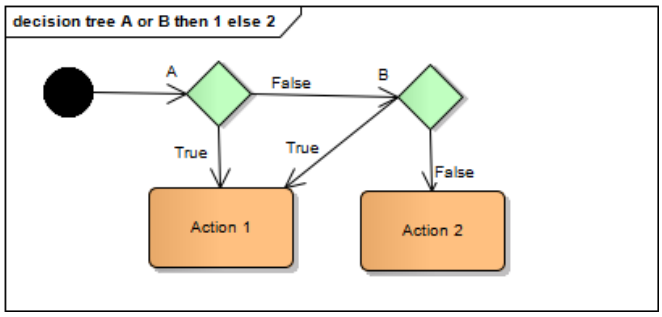
DATOVÝ TYP	ROZHODNUTÍ
boolean	hodnota je true
výčet hodnot (právě jedna z nich je možná v jednom okamžiku)	shoda s jednou z množiny hodnot
číselné	hodnota je v daném intervalu
vektor	vektor má délku v daném rozsahu (např. vzdálenost hráče a nepřítele)

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.



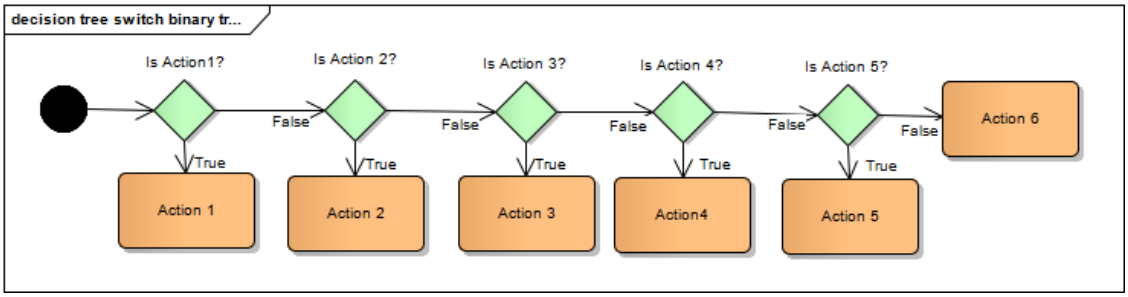
Obr. 2 Dekompozice AND pomocí dvou rozhodovacích uzlů



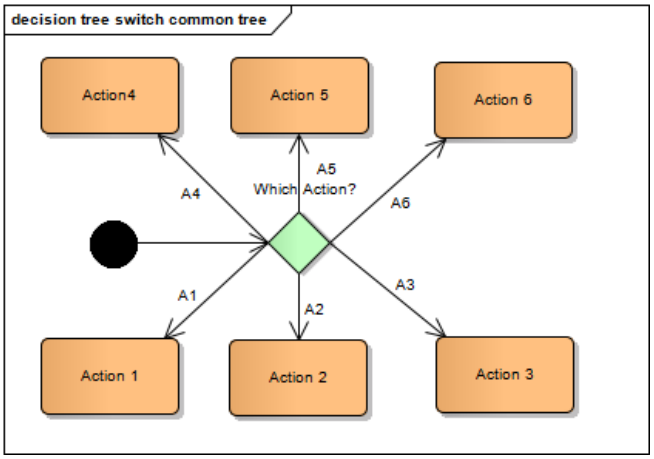
Obr. 3 Dekompozice OR pomocí dvou rozhodovacíh uzlů

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.

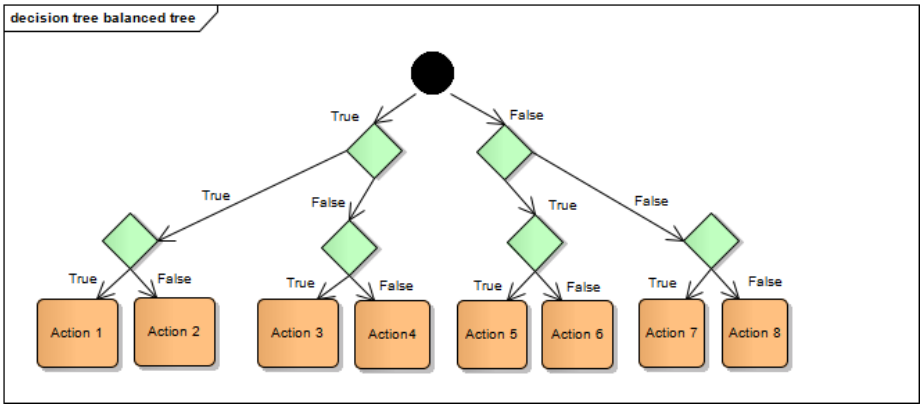
Někdy můžeme do rozhodovacího stromu zanést náhodu . Opakování stejných akcí dokola může časem působit nudně. Ozvláštněme rozhodovací strom přidáním vnitřních uzlů, které náhodně vybírají ze svých potomků. Když se houba z našeho příkladu dostane na obrazovku, vydá se vlevo. Lepší by bylo, kdyby náhoda 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 stram 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 stejné pravděpodobná. To se děje velmi zřídka, často jsou některé akce pravděpodobnější než jiné. Pokud by se akce 1 a 2 projevovaly 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 zeslábne 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

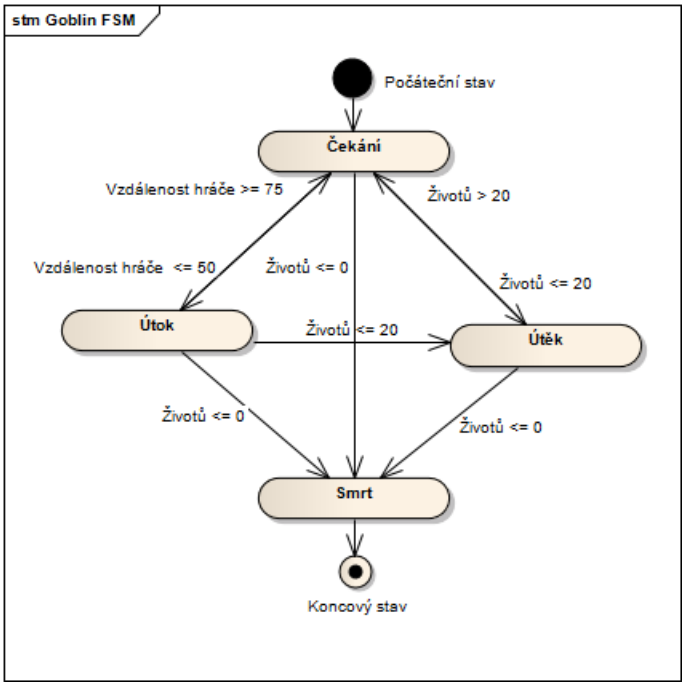
Pro zachycení tohoto chování je vhodné využít konečné stavové automaty. Konečný stavový automat (*Finite State Machine, FSM*) lze definovat jako uspořádanou pětici $(Q, \Sigma, \delta, q_0, F)$, kde Q je konečná neprázdná množina stavů. Σ je konečná množina vstupních symbolů. δ je přechodová funkce, 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. Rozšiřme 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 pojené 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 bereserkera, 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.

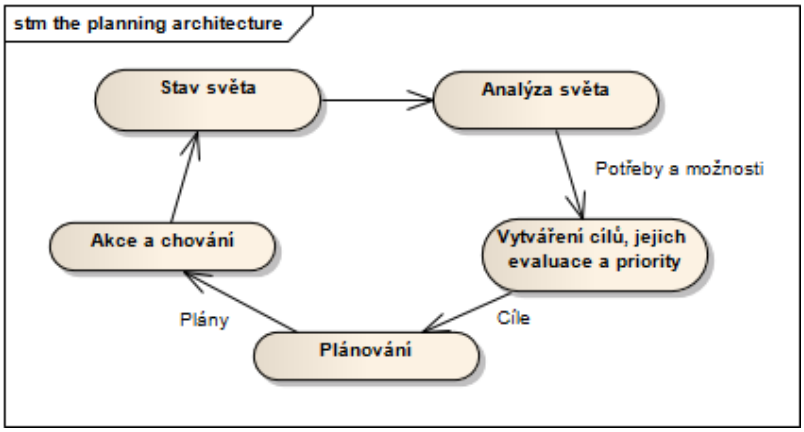
Úkoly jsou umístěny v určitě hierarchii a buď jsou složené z jiných úkolů, nebo jsou již atomické. Atomickým úkolem by mohl být přesun hrdiny z jednoho místa do druhého, výměna jedné suroviny za druhou, přesun jednotek mezi hrdiny. Jsou to úkoly, které po důkladném naplánování děláme ve hře jako hráč. 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 stupnici 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 hrady 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 sítem 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 ohlídá 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 podúkolu 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 realtimové 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, dosáhnutí 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 přemisťujeme 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. Použít špetku soli, hrnek mléka, chvilku

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 vojácích 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 míň, 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 vojácích 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 vojácích 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.

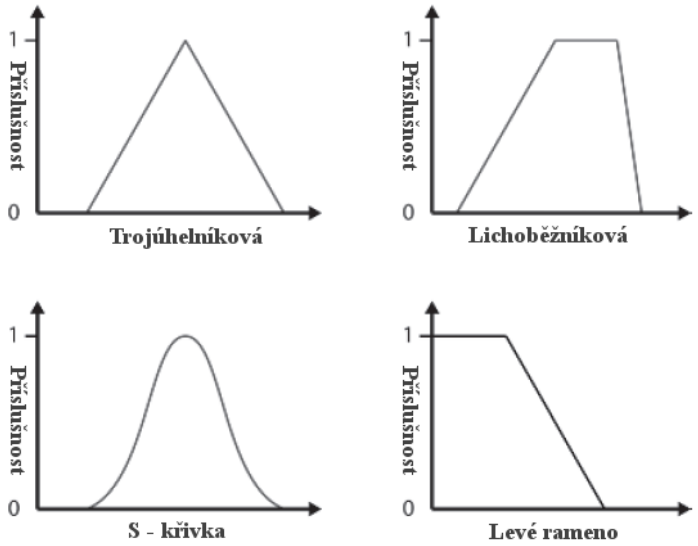
3.4.1 Fuzifikace

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).

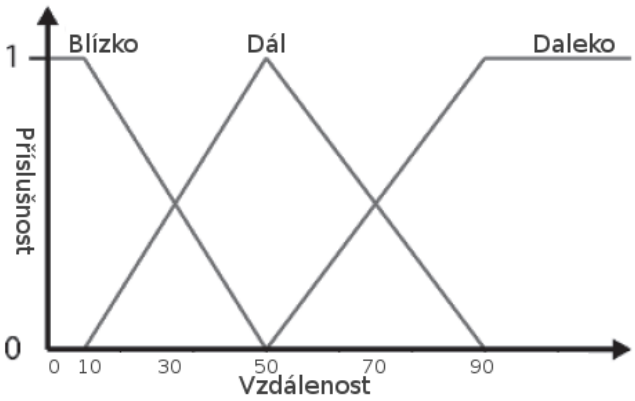
$$velikostArmady = \{MALA, STREDNI, VELKA\}$$

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

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 vzdalenostArmady by mohl vypadat obdobně jako na Obr. 10.



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



Obr. 10 Manifold funkcí příslušnosti FLV vzdalenostArmady. 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 vývojaře. 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 komutitivity, asociativity, je neklesající a 1 je její jednotkový element. Jedním z příkladů t-normu je obyčejné minimum.

$$A\ AND\ B = MIN(A, B)$$

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

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

$$A\ OR\ B = MAX(A, B)$$

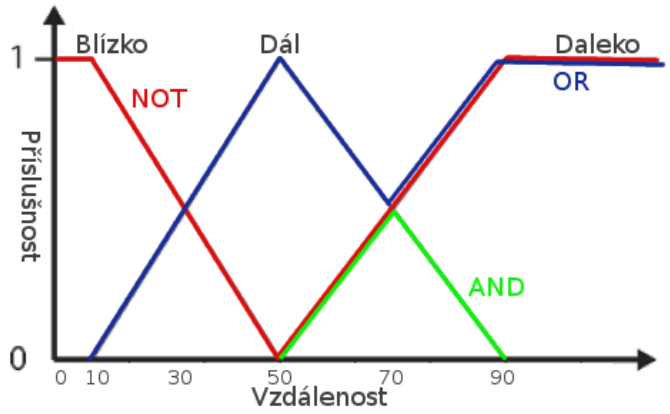
$$0{,}5\ OR\ 0{,}2 = 0{,}5$$

Unární operace NOT, negace je doplňkem do jedné.

$$NOT\ A = 1 - A$$

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

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.

$$VERY\ A = A^2$$

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

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

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.

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

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 žáber 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, nalezli 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 Selekce 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 zahozeno. 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 :

RDDLRULRU a ***LUDDLRLUR***

X

RDDL***LRLUR*** a ***LUDD***RULRU

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

RDDLRULRU a ***LUDDLRLUR***

X

R***UDDL***ULRU a ***L***DDLR***RLUR***

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

RDDLRULRU a ***LUDDLRLUR***

X

RD***DDL***UL***U***U a ***LU***DLR***RL***R***R***

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. RDDLRULRU > ***UL***DLRULRU.

Jinou mutací může být prohození jednoho příkazu s jiným, skupiny příkazů s jinou. RDDLRULRU > R***U***DLR***D***LRU. Můžeme i otočit pořadí několika prvků RDDLRULRU > RDD***URL***LRU.

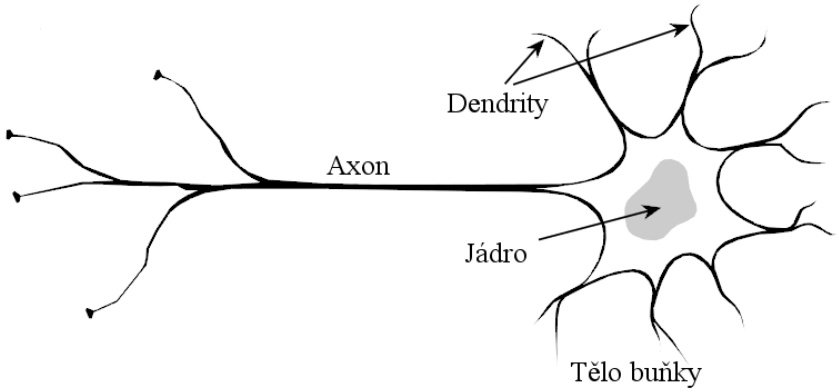
Způsob mutace, či křížení je ponechán pouze naší představivosti. Můžeme si vymyslet jiné způsoby, jež jsme neuvedli.

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.



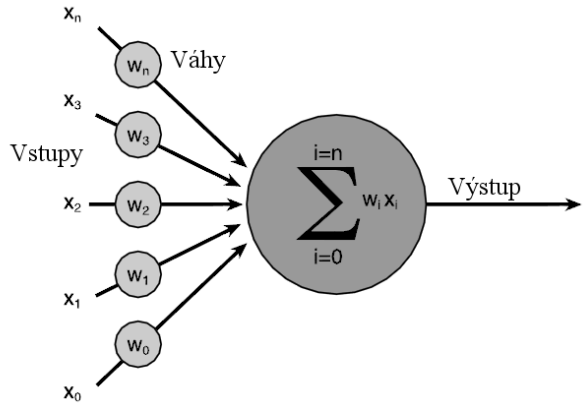
Obr. 12 Schéma neuronu. Zdroj [14].

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.

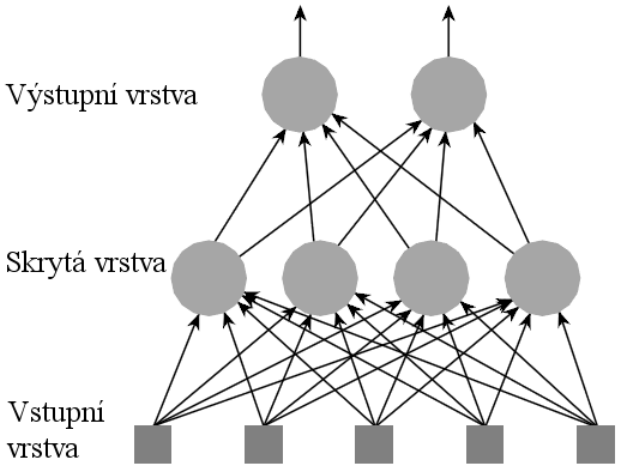
Akční potenciál je vstupem nelineární přenosové funkce. Pokud chceme na výstupu pouze binární hodnotu 0, či 1 a nic mezitím, použijeme skokovou Heavisidovu funkci, jež pro kladné vstupy vrací jedničku, pro záporné nulu.

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]

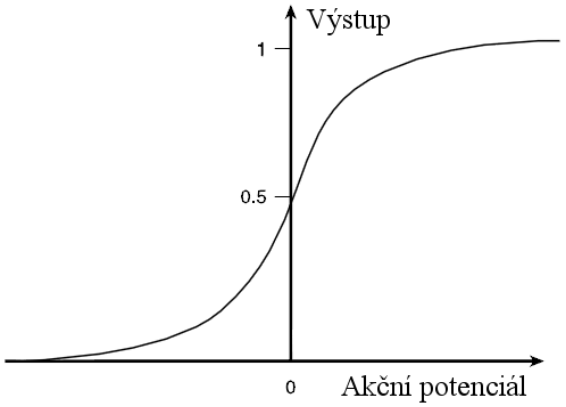
Vstupy a výstupy neuronové sítě jsou specifické dle jejich použití.

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 zatáčí 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 (*supervized training*) a učení se bez učitele (*unsupervized 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 dvou 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 spočíst, 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 Conveyově 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, čí 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 zpevňování stěn, těžbu drahokamů, či zabírání nových území. My je jako představitelé „Boha“ můžeme trestat, obětovávat, dávat jim nepřímo 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.

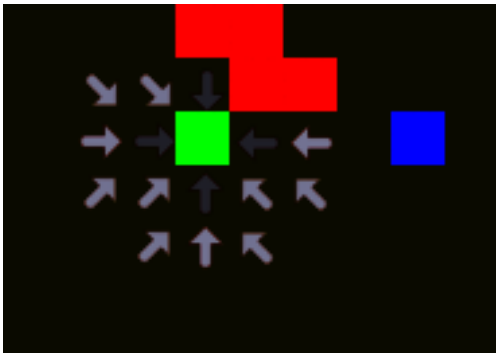
5.1.1 Prohledávání do šířky

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.

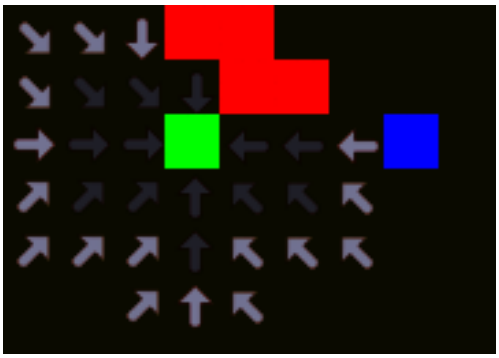
Základní princip algoritmu vysvětlíme na příkladě čtvercové 2D mapy, kde je označen jeden čtverec jako začátek (např. by to mohla být aktuální pozice jednotky), jeden čtverec jako cíl (např. místo, kam hráč klikl myší a kam chce poslat jednotku) a některé ze čtverců jsou

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 pokračuji 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 cíle 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.

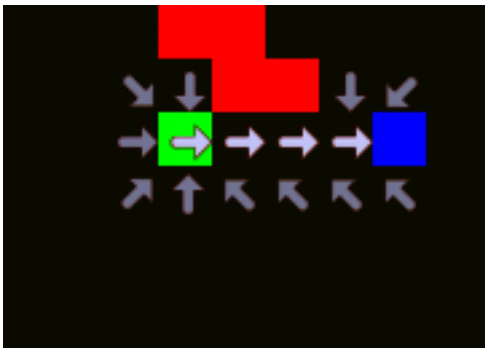
Pod předpokládanou vzdáleností si lze představit odhad vzdálenosti se zanedbáním terénů, zdí mezi kontrolovaným čtvercem a cílem. Např. u čtvercové mapy dobře funguje jako odhad tzv. Manhattonská metoda. Odhad touto metodou se spočítá jako součet absolutních hodnot rozdílu souřadnic zkoumaného čtverce 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. (Manhattonká 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 zkompilovat a slinkovat a znovu spustit. Objevíme v textech překlep, gramatickou chybu, opravíme jí, 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 poznani
DOJIT_NA(100,42);  // Strom vsevedeni
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 languge). 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 modařům. 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 Strategické hry

6.1 Real-time

6.2 Turn-based

7 Frameworky a enginy

Herních frameworků a enginů 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 ptačí perspektivy, tanky, které po sobě střílí. Pokud si spustíme ukázku 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 inteligence. 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.

8 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.

8.1 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.

8.2 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.

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.

8.3 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.

9 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.

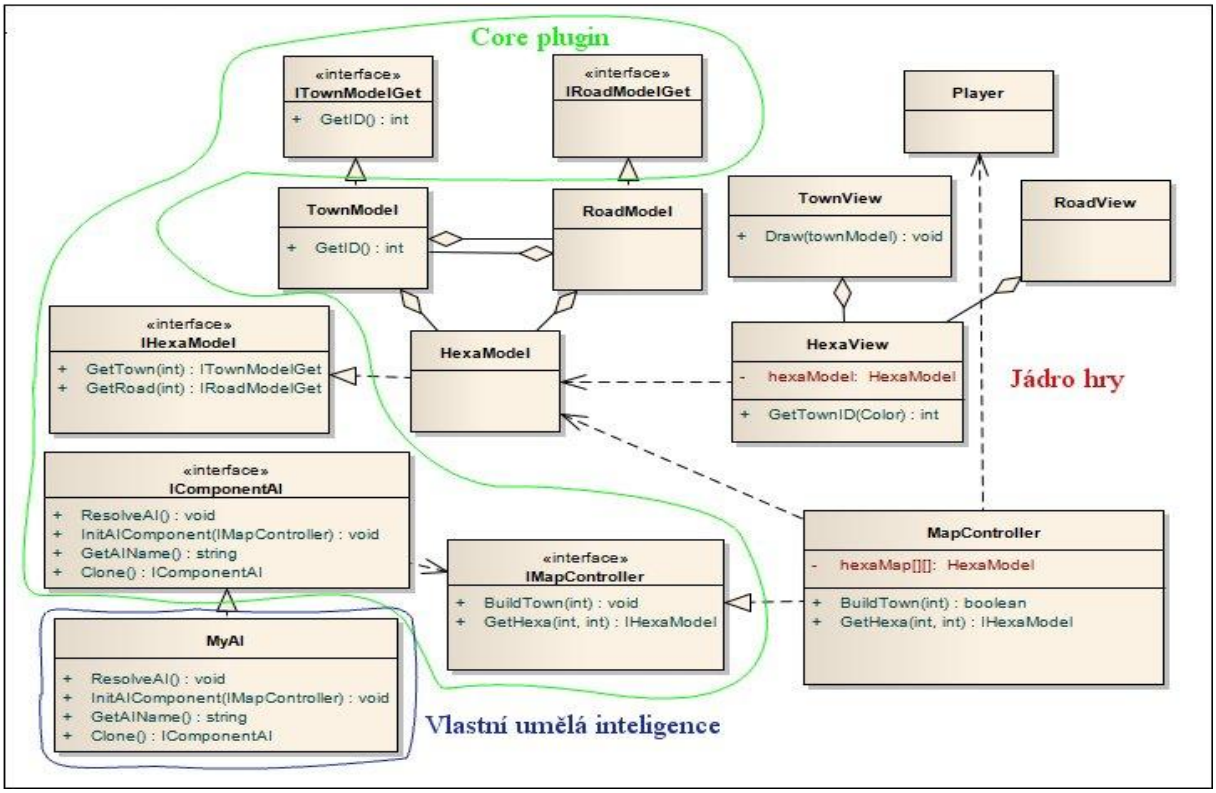
Tento přístup urychluji znatelně vývoj, při každé změně se překládá pouze dynamická knihovna a spustí se s nezměněným jádrem hry.

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.

9.1 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. 19 Zjednodušené schéma návrhu hry

Při vytváření umělé inteligence musí vývojař 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].

9.2 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 uložené 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.

9.3 Skriptovací jazyk

Způsob komunikace AI se hrou lze považovat za skriptovací jazyk založený na C#. Tento jazyk se před použitím ve hře musí zkompilovat, 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.

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 neprovedla (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 kláštera s volným slotem pro pokrok. V druhém případě může AI zavolat *InventUpgrade* v konkrétním klášteře stejně jako by to dělal lidský hráč ve hře.

9.4 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 normálním 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 zacyklit.

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 odstřihne 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.

Nemohu přímo zachytit zacyklení v umělé inteligenci. 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.

10 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.

```
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 7 Ukázka kódu AI lehká

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

Kód 7 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 stavěný 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.

11 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?

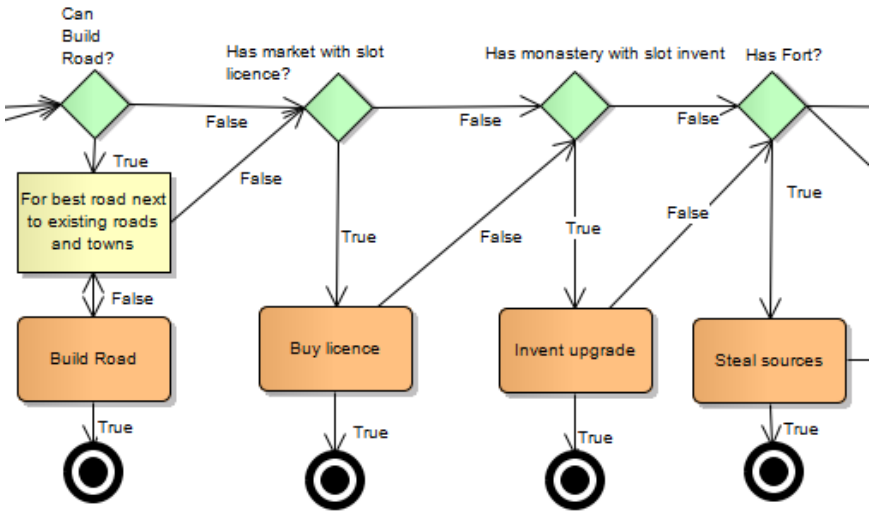
III. Je lepší vyměnit suroviny a provést jednu z akcí, nebo počkat jedno a více kol, kdy se potřebné suroviny vytěží?

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.

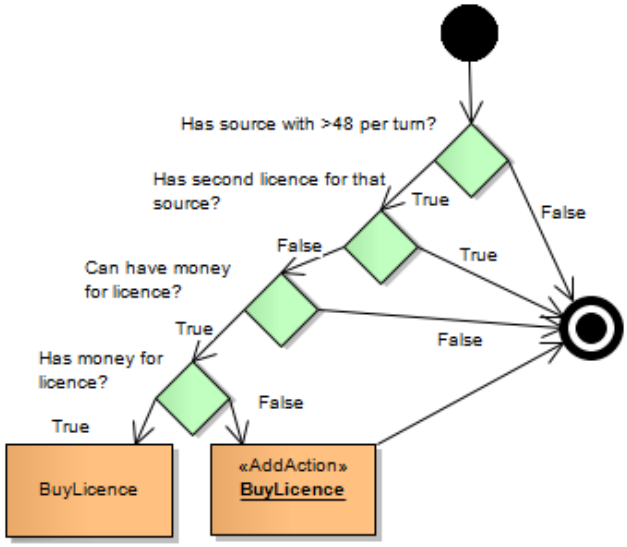
11.1 Základní struktura „stromu“

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. 21 a Obr. 22 znázorňují základní strukturu stromu. Za každým obdélníkem s oblými hranami se skrývá ještě malý podstrom. Rozdělil jsem to pro větší přehlednost. Každá z akcí[1] 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.



Obr. 21 Výřez rozhodovacího stromu.



Obr. 22 Detail na Buy licence z Obr. 21

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 akci. Jinak řečeno je

[1] 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.

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. Testy[2] 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.

11.2 Druhy uzlů

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

11.2.1 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.

[2] Viz kapitola Uživatelské testy 13.2

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 větvích stromu, které ještě nebyly dokončeny. Nyní se nachází za všemi kontrolami vhodností jednotlivých akcí a značí, že se má přejít k výměně surovin a provedení akce z *actionSource*. Také ho využívají některé rozhodovací uzli.

11.2.2 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 8 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. 21 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 prvém 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.

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 DecisionTree AI. Jak vyřešit velký stavový prostor? Inspiroval jsem fitness funkcí z genetických algoritmů. ForBestNode existuje v pěti variantách : 1) ForBestFortHexaNeigbour,
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 trueNodu nedosáhne akce, pokračuje se falseNodem dál

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 pokročuje 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.

11.3 Shrnutí

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

Oproti základní AI zde funguje plánování, při výběru stavby cest je prohlížen stavový prostor do vzdálenosti dvou měst od infrastruktury hráče.

12 Goal Driven architektura

13 Testování

13.1 Testování pravidel

13.2 Uživatelské testy

13.3 Debugování AI

Testování softwaru má za úkol odhalit chyby v softwaru, ale testeři už většinou neřeší, jak daný problém odstranit.

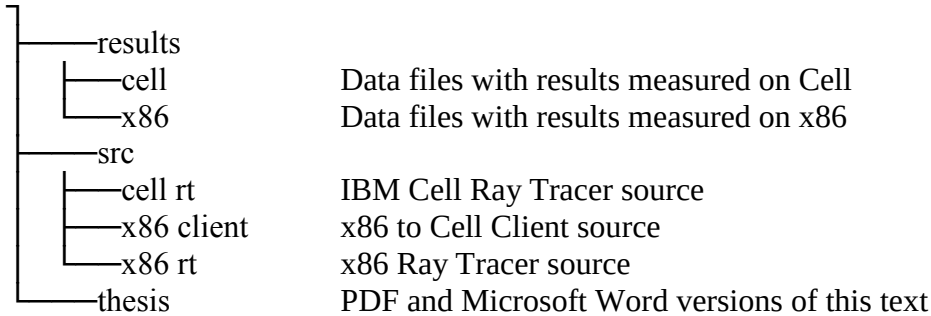
14 Závěr

15 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 Inteligence 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. *Lible Game Wiki.* [Online] [Citace: 1. únor 2011.] http://libregamewiki.org/OpenRTS.
24. **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 Ukázky kódu

B 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