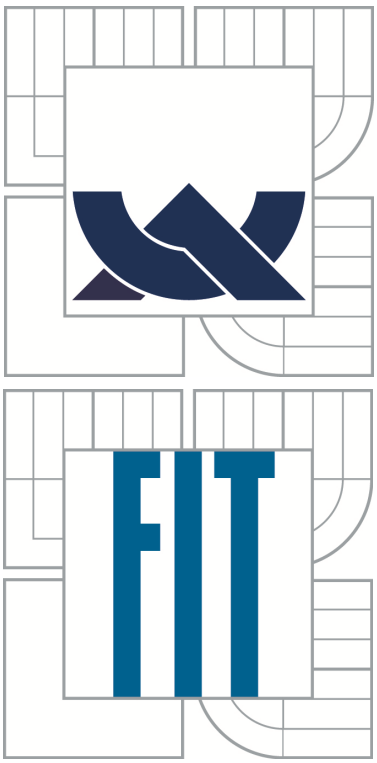


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER GRAPHIC AND MULTIMEDIA

UMĚLÁ INTELIGENCE VE STRATEGICKÝCH POČÍTAČOVÝCH HRÁCH

ARTIFICIAL INTELLIGENCE IN COMPUTER STRATEGY GAMES

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE **Lukáš Votroubek**

AUTHOR

VEDOUCÍ PRÁCE **Ing. Jiří Zuzaňák**

SUPERVISOR

BRNO 2011

Abstrakt

Tato práce se zabývá metodami používanými v umělé inteligenci strategických počítačových her, mnohé z těchto metod jsou však použitelné i v jiných oblastech. Jsou to různé metody pro rozhodování (např. stavové automaty, fuzzy logika, markovy řetězce), plánování (cílem orientované plánování, plánování monte-carlo, plánování založené na případech) a strojového učení (posilové učení, rozhodovací stromy a neuronové sítě). Cílem této práce je tyto metody z různých zdrojů nastudovat a vysvětlit základní princip několika z nich. Poté několik vybraných metod rozebrat více do dopodrobna a implementovat je (cílem orientované plánování a stavový automat). Při implementaci a následném testování byl využit herní engine ORTS, kterým se tato práce také zabývá.

Abstract

This work covers with artificial intelligence of strategy computer games, however many of these methods are usable in other areas. These are different methods used in deciding (finite state machina, fuzzy logic, Markov Process), planning (Goal-oriented action planning, Monte-carlo planning, Case-based planning) and machine learning (Reinforcement leasing, Decision Learning and Neural Networks). Objective of this thesis is to study this methods from different sources and explain their base principle. Then few of this methods resolve in more details and implement them (goal oriented planning and state machine). This thesis focuses on game engine ORTS, which is used in implementing and testing methods.

Klíčová slova

Umělá inteligence, RTS, strategické hry, ORTS, Open Real Time Strategy

Keywords

Artificial inteligence, RTS, strategy games, ORTS, Open Real Time Strategy

Citace

Lukáš Votroubek, Umělá inteligence ve strategických počítačových hrách, bakalářská práce, Brno, FIT VUT v Brně, 2011

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Zuzaňáka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

16.5.2011 Lukáš Votroubek

Poděkování

Rád bych poděkoval Ing. Jiřímu Zuzaňákovi za jeho čas strávený při konzultacích a za rady a připomínky při tvorbě této práce.

Obsah

1. Úvod

Strategické hry jsou v současné době velice populárním typem her. Vysoká konkurence vyžaduje, aby hry byly čím dál propracovanější, a nestačí již vylepšovat pouze jejich grafickou stránku. Pozornost je nutné směřovat i na chování počítačových protihráčů – na umělou inteligenci. Abychom pochopili co to umělá inteligence (UI) je, musíme nahlédnout do její historie. Ta je starší než by se mohlo zdát, první zmínky jsou dokonce mnohem starší než počítače.

První myšlenka inteligentních umělých bytostí (robotů) se vztahuje do období antiky, kde se psalo o bohovi jménem Talos - obrovském muži z bronzu chránícím Evropu před piráty a nájezdníky. Základy moderní umělé inteligence položili filozofové Aristoteles, Porfyry z Tyrosu a další, kteří se zabývali lidským myšlením a tím jak ho vyjádřit a lépe pochopit. Později se umělá inteligence stala také problémem matematickým. Myšlenkové pochody se filozofové a matematikové snažili vyjádřit různými matematickými metodami a logikami. Jako významné osobnosti této éry uvedu jména Ramon Llull, Gottfried Leibniz, Russel a Whitehead.

Jedním z nejvýznamnějších milníků byla Church-Turingova teze[a, b] která tvrdila, že mechanické zařízení pracující jednoduše jen se symboly 0 a 1 může napodobovat jakýkoli myslitelný proces matematického uvažování. Byl to základ pro vývoj mocného nástroje pro výpočty a různé simulace využitelného nejen v umělé inteligenci – počítače.

Vývoj umělé inteligence ubíhá mnoha různými směry jako robotika, získávání informací z lidské řeči a také směrem, který je pro tuto práci nejdůležitější -- UI v počítačových hrách. Jejími prvními úspěchy byly např. program hrající dámu, který napsal Christohper Strachey a šachový program, který napsal Dietrich Prinz, oba v roce 1951.

Za zrození umělé inteligence ve smyslu jak ji chápeme dnes je považován rok 1956, kdy se konala Dartmouthská konference[c]. Zde byl mj. název „Artificial Intelligence“ (česky Umělá Inteligence), přijat jako název tohoto odvětví. Umělá inteligence tu dostala svoje jméno, cíl, první úspěchy a hlavní osobnosti.

Významný milník nastal v devadesátých letech, kdy byl všeobecně přijat vzor používaný dodnes, Inteligentní agenti[d]. I když dřívější výzkumy navrhovaly způsob „rozděluj a panuj“, moderní způsoby zavedli až Judea Perl, Alan Newell a další, kteří přenesli koncepty z teorie rozhodování [e] a ekonomie do umělé inteligence.

Počítače jsou dnes již dostatečně výkonné na to, abychom se mohli soustředit nejen na grafickou stránku, jako to bylo dříve, když nestačil výkon, ale i na jiné součásti jako např. právě na UI.

Smyslem této práce je přiblížit metody používané v tomto zajímavém a stále se rozvíjejícím oboru, některé modernější metody (např. učící se umělou inteligenci) z důvodu složitosti pouze okrajově, jiné trochu detailněji, a poté podrobněji rozebrat. Dále se pokusím implementovat jednu z metod plánování a připravit základ pro její další možné rozšíření. Abych se mohl zabývat více problematikou umělé inteligence a vyhnul se řešení některých věcí jako např. grafické stránky hry nebo hledání cesty, které jsou velice složitými a rozsáhlými tématy, využil jsem již existující herní

engine (ORTS), který tyto základní problémy již umí řešit. Seznámení s ním je dalším předmětem této práce.

Text této práce se skládá ze dvou základních částí. V kapitole 2 je popis metod používaných v různých oblastech umělé inteligence. V podkapitole 2.1 jsou to metody pro rozhodování. Tyto metody jsou poměrně jednoduché, ne přiliž flexibilní, ale stále v mnohých oblastech používané a užitečné. V podkapitole 2.2 je popsáno několik metod plánování. V poslední podkapitole jsou, z důvodu složitosti spíše okrajově, popsány moderní metody strojového učení.

Další velkou částí, kterou pokrývá kapitola 3, je základní popis a princip funkce herního frameworku ORTS, který jsem využil jako základ a implementoval jsem s jeho pomocí něj jednu z metod popsaných v kapitole 2.2. Vlastní implementaci zvolené metody popisuje detailněji kapitola 4. Jakých výsledků jsem dosáhl, a celkové shrnutí mé práce uvádím v závěru.

Velká část informací čerpána z [1], Wikipedia: History of artificial intelligence.

2. Odvětví UI

Umělou inteligenci ve strategických počítačových hrách (RTS – Real Time Strategy) hrách lze podle zaměření rozdělit na tři základní části a to na pohyb, rozhodování a taktiku. V této práci se budu zabývat především taktikou a rozhodováním, pohyb jednotek je poměrné rozsáhlé téma a není proto předmětem této práce.

1) Pohyb (movement)

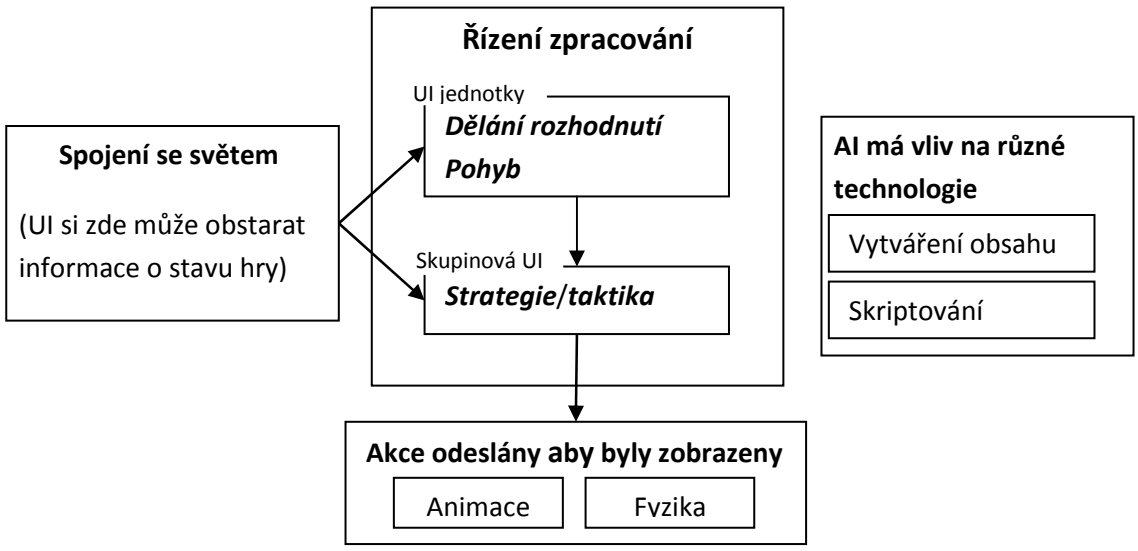
   Tímto se rozumí algoritmy pro hledání nejkratší a/nebo nejlepší cesty z bodu A do bodu B. Ve starších strategických hrách byl pohyb hlavní stránkou umělé inteligence. Jako příklad lze uvést hru Warcraft: Orcs and Humans (Blizzard Entertainment, 1994). [2]

2) Rozhodování (Decision Making)

   Rozhodování je ve hrách (nejen strategických) poměrně důležitou součástí. Lze jej rozdělit na několik úrovní, v té nižší se rozhodují např. jednotlivé jednotky (odkud lučištník trefí svůj cíl, zda se armáda kryje před palbou apod.), ve vyšší úrovni jsou to rozhodnutí v oblasti výzkumu, konstrukce budov, práce se zdroji apod. Pro každou tuto oblast je výhodné vytvořit jednu komponentu UI – agenta. Složitost kombinování těchto agentů se velice liší hra od hry. [2]

3) Taktika/strategie (Strategy)

   Je koordinace všech částí UI.



Obr. 1: Schéma součástí UI [2]

2.1. Rozhodování (Decision Making)

2.1.1. Rozhodovací stromy (Decision Trees, DT)

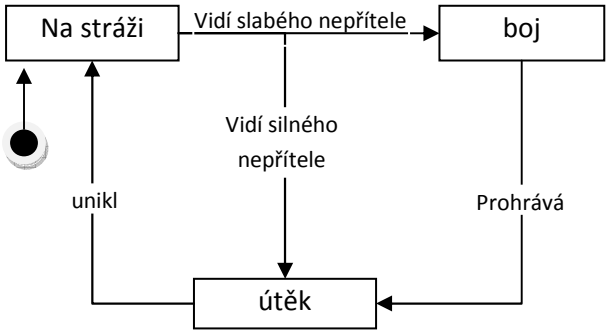
Rozhodovací stromy patří do proaktivních systémů. Jsou poměrně rychlé, velice snadno implementovatelné a jednoduché na pochopení. Základní princip je velice jednoduchý. Máme strom rozhodnutí typu ANO/NE, kde kořenem je nějaká jednoduchá otázka. Může to být například: „je nepřítel blíže než dosah zbraní?“ Pokud na otázku odpovíme ANO, strom přidá kořen „zaútoč“, pokud NE tak bude další podstrom.

Tyto stromy jsou velice často používány pro kontrolu postav a další herní rozhodnutí, např. kontrola animací. Ve strategické a taktické UI jsou dnes v této formě zřídka používány, mnohem častěji se používá jeho učící se forma. [2]

2.1.2. Konečné automaty (Finite State Machines, FSM)

Jsou jednoduché systémy, které se skládají ze dvou částí. Mají určitý počet stavů (pravidel), a tyto stavy jsou propojeny přechody. Aktivní je v daný moment pouze jeden stav. [3]

Na Obr. 2 jsou stavy znázorněny obdélníky, přechody pomocí šipek. Na obrázku je znázorněn obecný stavový stroj, nikoli konečný automat.



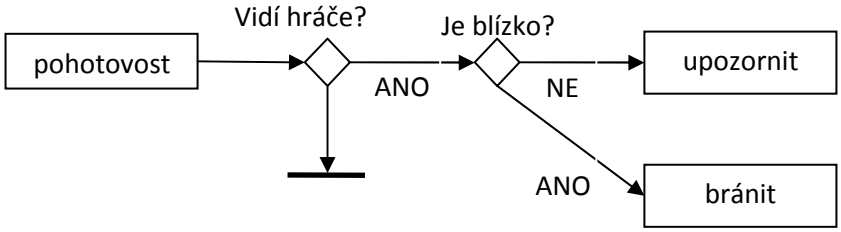
Obr. 2: Jednoduchý stavový stroj [2]

Výhodou konečných automatů je jejich rychlost a jednoduchost implementace. Jsou snadno pochopitelné, a proto se také snadno hledají chyby. Při složitějších chováních složitost návrhu ale prudce narůstá a schéma stavů se stává velice složitým. Další nevýhodou je předvídatelnost chování (počítač se chová vždy stejně). [4]

Tento způsob byl využit například ve známé hře Pac-Man, ve sportovní hře FIFA2002, nebo ve strategické hře Warcraft.

2.1.3. Kombinace DT a FSM

Implementace přechodů stavových strojů se podobá rozhodovacím stromům a této skutečnosti můžeme využít, protože tyto stromy jsou účinným způsobem porovnávání sérií podmínek. Jsou proto výhodné při implementaci přechodů. Způsob implementace je znázorněn na Obr. 3.



Obr. 3: Ukázka kombinace DT a FSM

Když jsme ve stavu „pohotovost“, máme pouze jednu možnost přechodu: přes rozhodovací strom. Ten velice rychle zjistí, zda vidíme nepřítele. Pokud nevidíme, přechod končí a nedosáhneme nového stavu. Pokud vidíme nepřítele, DT provede výběr na základě vzdálenosti. V závislosti na rozhodnutí přejde do jednoho ze stavů upozornit / bránit.

Implementace stroje na Obr. 3 bez použití rozhodovacího stromu by musela dvakrát vyhodnotit stejnou situaci (1. hráč je blízko *a nepřítel v dohledu*, a za 2. Hráč je daleko a *nepřítel je v dohledu*). Z toho vyplývá výhoda tohoto způsobu: rychlost.

Informace byly čerpány z [2], *AI for games*.

2.1.4. Skriptování (Scripting)

Skriptovací jazyky[f] rychle získávají na popularitě herních vývojářů. Několik velkých vývojářských firem již skriptování ve svých titulech využívá. Jsou to např. herní série Unreal Tournament (Epic Games), Neverwinter Nights (BioWare) a FarCry (Crytek). [4]

Výhodou těchto jazyků je, že pokud máme dlouhý kód a provedeme změny, nemusíme vše znovu kompilovat jako u klasických jazyků. Nelze je sice využít všude, např. u částí kódu kde potřebujeme velkou rychlost, ale může být výhodné je použít jako rozhodovací logiku herních agentů. Např. při použití FSM je výhodné vytvořit rozhranní mezi třídou agenta a skriptovacím jazykem, a napsat skript pro každý stav, což umožní jejich snadnou případnou úpravu, a nemusíme čekat na rekompilaci. [4]

2.1.5. Fuzzy Logika

Fuzzy logika vychází z velice vyspělé komunikační schopnosti lidí jednoduše a nepřesně se vyjádřit. Například v televizi kuchař řekne: ukrojíme dva středně „*silné* plátky“, ohřejeme pánev na „*vysokou“* teplotu, dáme „*malé množství“* černého pepře apod.

Jak ale tohoto využít v umělé inteligenci? Představme si, že hrajeme golf. Můžeme si určit několik pravidel, které by člověk jednoduše pochopil:

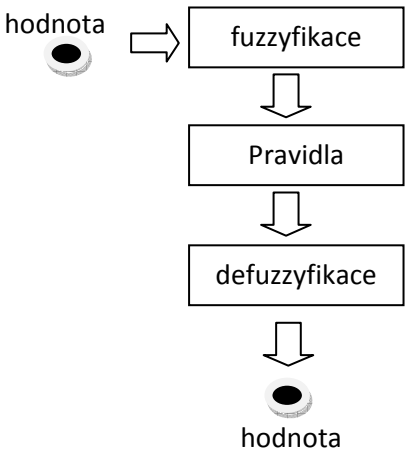
1) POKUD je míček daleko od jamky A green se klopí jemně zleva doprava, PAK uhoď míček silně A úhel lehce nalevo od vlajky.
2) POKUD je míček vel mi blízko jamky A green mezi míčkem a dírou je ve stejné úrovni, PAK uhoď míček jemně a přesně na jamku.
3) POKUD je vítr silný A vane zprava doleva A díra je daleko PAK uhoď na míček hodně a úhel daleko vpravo od vlajky.

Je však poměrně složité je přeložit do jazyka, kterému počítač rozumí. Slova jako „*daleko*“, „*velmi blízko*“ nebo „*jemně*“ nemají pro něj žádný význam. Jediné čím mu můžeme pomoci, aby těmto slovům porozuměl je definovat např. termín „v*zdálenost*“ jako množinu a přiřadit několik intervalů:

Blízko = míček je mezi nulou a dvěma metry od jamky.
Středně = míček je mezi dvěma a pěti metry.
Daleko = míček je více než pět metrů od jamky.

Co když je ale míček například 4.99 metru od jamky? Použitím těchto pravidel by nám vyšla „střední“ vzdálenost i když nás od „daleko“ dělí jen několik málo centimetrů. Pokud bychom si vzali inspiraci z lidského myšlení, část lidí by tuto vzdálenost brala jako „*střední*“, většina však jako „*daleko*“. Tuto skutečnost můžeme nejlépe vystihnout termínem „stupeň odpovědnosti“ („matter of degree“), hezky česky řečeno: „jak moc hodnota odpovídá různým termínům“.

Fuzzy logiku vymyslel Lotfi Zadeh v polovině šedesátých let. Umožňuje počítačům rozumět řečnickým termínům podobným těm lidským. Slova jako „*daleko*“ nebo „*trochu*“ nejsou reprezentovány přesnými intervaly, ale fuzzy množinami (fuzzy sets), které umožňují, aby hodnota odpovídala množině nějakou mírou. Tento proces se nazývá fuzzyfikace. Použitím fuzzyfikovaných hodnot je počítač schopen interpretovat pravidla a vyprodukovat výstup, který buď zůstává fuzzy nebo častěji, zvláště v počítačových hrách, je defuzzyfikována aby poskytla nějakou přesnou hodnotu.



Obr. 4: Proces fuzzy logiky [4]

Informace byly čerpány z [4], *Programming gameAI by example.*

2.1.6. Markovy procesy, markovy řetězce (Markov Process)

Několik numerických stavů můžeme reprezentovat jako vektor čísel. Každá pozice ve vektoru odpovídá jednomu stavu (např. jedné prioritě, nebo jednomu umístění jednotky). Takový vektor se nazývá stavový vektor.

Hodnoty tohoto vektoru nejsou nijak omezeny. Každá aplikace může mít vlastní hodnoty reprezentující např. nějaké rozložení (pak budou mít sum roven jedné). Taková rozložení jsou v matematice spojována s rozložením náhodných proměnných.

Přechodové matice jsou vždy čtvercové. Element *(i,j) v matici* reprezentuje podíl elementu *i* na starém stavu vektoru, který je přidán k elementu *j* v novém vektoru. Jedna iterace Markova procesu se skládá z násobení stavového vektoru přechodovou maticí. Používá klasická pravidla násobení matic, výsledkem je stavový vektor stejně velký jako původní. Na každém elementu nového vektoru má svůj podíl každý element starého vektoru.

Konzervativní Markovy procesy se liší tím, že sum hodnot stavového vektoru se nemění v čase. To je užitečné v aplikacích, kde by se stavový vektor neměl měnit (kde distribuuje nějaký podíl, např. pokud hodnoty reprezentují číslo nějakého objektu ve hře). Proces bude konzervativní, pokud všechny řádky v matici budou mít sum k jedné.

Iterující procesy – při těchto procesech se očekává, že se stejná přechodová matice aplikuje znovu a znovu na stavový vektor. Na určení finální, stabilní, hodnoty existují různé techniky. Stabilní hodnota je charakteristický vektor matice stejně dlouhý jako vektor. Tyto iterativní procesy formují *Markovův řetěz*.

Při aplikaci ve hrách je běžné, že máme přechodové matice různé pro různé události ve hře.

Příklad:

Sniper vybírá vhodnou pozici. Konkrétně má 4 pozice, a každá je nějak bezpečná.

$$V = \begin{bmatrix} 1.0 \\ 0.5 \\ 1.0 \\ 1.5 \end{bmatrix}$$ → sum roven 4.0.

Když sniper vystřelí z první pozice, upozorní nepřítele na svou existenci. Bezpečnost pozic se tedy změní. Zatímco se však nepřítel zaměřuje na směr, odkud sniper vystřelil, jiná pozice bude odpovídajícím způsobem bezpečnější. Mohli bychom použít např. tuto přechodovou matici:

$$M = \begin{bmatrix} 0.1 & 0.3 & 0.3 & 0.3 \\ 0.0 & 0.8 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.8 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.8 \end{bmatrix}$$

Když aplikujeme tuto matici na stavový vektor, dostaneme:

$$V = \begin{bmatrix} 0.1 \\ 0.7 \\ 1.1 \\ 1.5 \end{bmatrix} \rightarrow \text{má sum 3.4}$$

Celková bezpečnost se tedy snížila (ze 4 na 3.4). Bezpečnost bodu odkud střílel se snížila z 1 na 0.1, ale bezpečnost ostatních tří pozic se podstatně zvýšila.

Pokud bychom aplikovali pouze jeden druh matice, za nějaký čas by nebylo nikde bezpečno. To by bylo realistické, pokud by sniper pálil na nepřítele neustále. My však chceme, aby se jeho bezpečí zvýšilo, pokud nestřílí. To může zařídit matice jako např. tato:

$$M = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 1.0 & 0.1 & 0.1 \\ 0.1 & 0.1 & 1.0 & 0.1 \\ 0.1 & 0.1 & 0.1 & 1.0 \end{bmatrix}$$

Dokud pracujeme se známými pravděpodobnostmi, hodnoty přechodové matice mohou být tvořeny ručně. Úprava hodnot k požadovanému efektu může být ale poměrně složitá, závisí totiž na hodnotách použitých ve stavovém vektoru.

Použitím Markových procesů můžeme vytvořit rozhodovací nástroj, který používá numerické hodnoty ve svých stavech. Tento stavový stroj reaguje na události ve hře vykonáním přechodové matice nad stavovým vektorem. Pokud žádná podmínka ani událost nenastane, pak může být vykonána defaultní přechodová matice.

Informace pro tuto kapitolu byly čerpány z [2], Artificial Intelligence for games.

2.1.7. Systém založený na pravidlech (Rule-Based Systems)

Systém založený na pravidlech vznikal v letech 1970 – 1980. Pomocí této metody a jejího ztělesnění ve formě „expertních systémů“ bylo vytvořeno mnoho známých UI programů. Stále je to navzdory její reputaci, že je neúčinná a obtížná na implementaci, často používaná metoda. [1]

Skládá se ze dvou částí, pracovní paměti a paměti pravidel. Mnoho systémů přidává ještě třetí komponentu, soudce. Pracovní paměť ukládá známá fakta a tvrzení vytvořená pomocí pravidel. Paměť pravidel obsahuje pravidla typu „pokud nastane situace, udělej něco“ („if-then rules“), která operují nad fakty uloženými v pracovní paměti. Při průchodu pravidel může být vyvolána nějaká akce nebo změna stavu, jako ve FSM, nebo mohou modifikovat obsah pracovní paměti přidáním nových informací (tzv. *tvrzení* – anglicky *assertion*).

Jako příklad z oblasti RTS her uvedu technologický strom. V pracovní paměti budeme mít elementy reprezentující prvky technologického stromu, které mohou nabývat několika hodnot, např. Ano, Ne, Možná nebo Neznámo. Budeme uvažovat, že tento strom je našeho nepřítele, a chceme zde uchovávat jeho stav. Hodnota „Ano“ nám tak říká, že hráč vlastní danou technologii, „Ne“ že ji ještě nevlastní. Hodnotu „Možná“ použijeme pokud víme, že hráč splňuje kritéria, ale nemáme to potvrzené (např. průzkumník nám ještě u nepřítele nepotvrdil). Pokud nevíme z této oblasti nic, použijeme hodnotu „Neznámo“.

Umělý hráč může zjišťovat fakta o nepříteli a jeho technologickém stromu posíláním průzkumníků. Např. pokud pošle průzkumníka a ten objeví chrám, nastaví hodnotu „*Chrám*“ v jeho pracovní paměti na „*Ano*“. Použitím pravidel může hráč odhadnout další části technologického stromu nepřítele dříve, než je průzkumník potvrdí jako známý fakt.

Př.: **Nalezen kameník**

Pracovní paměť:

Dřevorubec - *ANO*
Kameník - *NE* --> *ANO* (průzkumník ho právě objevil)
Chrám - *Neznámo*

Nyní když byla fakta změněna, projdou se pravidla. Zde dojdeme např. k pravidlu:

POKUD (Dřevorubec je „*ANO*“ A Kameník je „*ANO*“
A Chrám je „*Neznámo*“)
PAK nastav Chrám na „*Možná*“

Dedukce může fungovat i opačným směrem. Například tedy pokud průzkumník objeví kněze, je dle technologického stromu jasné že má i kasárna, dřevorubce, kameníka a chrám.

Př.: **Objeven kněz**

Pravidlo:

POKUD (Kněz je „*ANO*“)
PAK nastav Chrám, Kasárna, Dřevorubce a Kameník
na „*ANO*“

Pro získávání odhadů pomocí této metody existují dva základní algoritmy. Jsou to běžnější dopředné řetězení (Forward chaining) a zpětné řetězení (Backward chaining).

Dopředné řetězení má tři základní fáze. Ta první zjistí, která pravidla souvisejí s fakty uloženými v pracovní paměti. Toho docílíme kontrolou části „POKUD“ („if“) každého pravidla abychom zjistili, zda odpovídá dané sadě faktů v pracovní paměti. Když jsme byli úspěšní, je vykonána část „PAK“ („then“). Potenciálně se může stát, že je vybráno více než jedno pravidlo. V tomto případě musíme spočítat, které pravidlo vybrat. Fáze, která tento problém řeší, se nazývá fáze řešení konfliktů. Běžný způsob řešení této fáze je jednoduše vybrat první pravidlo. V některých případech je výhodné použít náhodný výběr. Také lze ohodnotit každé pravidlo a následně vybrat to s nejvyšší váhou. Když je tato fáze dokončena a pravidlo vybráno, vykonáme ho. Vykonání pravidla (provedení části „PAK“) ovlivní fakta v pracovní paměti nebo případně vyvolá nějakou událost, případně další funkci provádějící další zpracování. Celý tento proces je vykonáván, dokud lze najít nějaká pravidla. Pokud žádné vyhovující pravidlo nelze najít, pracovní paměť obsahuje všechny informace které lze odvodit z daných faktů.

Zpětné řetězení je opak dopředného. Stále máme pracovní paměť a paměť pravidel, ale místo snahy o nalezení shody v části „POKUD“ zkoušíme najít shodu v části „PAK“. Jinými slovy, začínáme s nějakým výstupem nebo cílem a snažíme se zjistit, která pravidla musí být vykonána, abychom dostali hledaný výstup nebo cíl. Například tedy pokud bychom zjistili, že hráč má kavalerii (v paměti bylo nastaveno Kavalerie na „Ano“) můžeme zjistit, která pravidla musíme použít, abychom nastavili tuto hodnotu.

Například najdeme toto pravidlo:

POKUD (Kovárna je „*ANO*“) PAK Kavalerie je „*ANO*“

Dle toho usoudíme, že musí hráč mít kovárnu. Když má kovárnu, musí mít i kasárna a když má kasárna nejspíš má …. A takto můžeme dál a dál pokračovat.

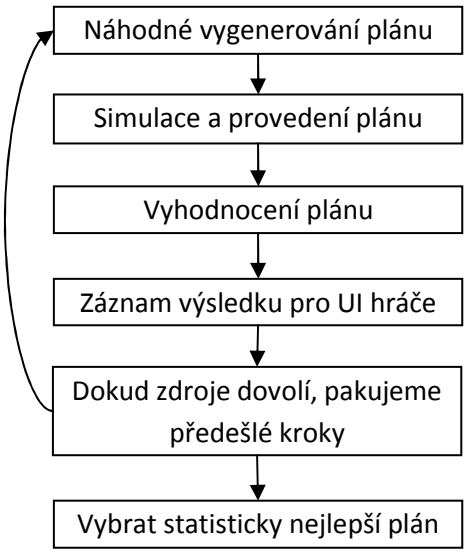
Zpětné řetězení je rekurzivní a obtížnější na implementaci. Struktura pravidel může být také poměrně složitá, proto kontrola zda odpovídá část „PAK“ faktům v pracovní paměti je velice obtížná.

Informace a příklady k tomuto tématu byly čerpány z [5], AI for game developers.

2.2. Strategie/plánování (Strategy/planning)

2.2.1. Plánování Monte Carlo (Monte-carlo planning)

Simulace metodou Monte-Carlo byly kvůli výhodě jednoduchosti a hlavně schopnosti dosáhnout vysokého výkonu s menším množstvím expertních znalostí použity např. na hry backgammon, bridge, poker nebo scrable.



Obr. 5: Princip Monte-Carlo plánování

Vypořádání se s neúplnými informacemi, stochastičností a velkým počtem skrytých stavových atributů je velice náročné. Dalším velkým problémem je, že ve strategických hrách často hrají více než dva hráči. To má za následek, že tradiční algoritmy pro prohledávání stromů určené pro úplné informace jsou nevhodné. Jedním způsobem jak se s tímto prohledáváním vyrovnat je provést abstrakci stavového prostoru. S neúplnými informacemi se můžeme vypořádat pomocí vzorkování (sampling).

Principem této metody je, že vytvoří náhodné vzorky možných plánů pro konkrétního hráče a následně vybere plán, který je pro něj nejvýhodnější (je nejlépe statisticky ohodnocen). Výhodou této metody je, že nepotřebuje žádné speciální znalosti. Např. ve Full Spectrum Command[g] totiž každý plán musí být přesně specifikován. Scénář musí být označen informací v jakých případech je aplikovatelný a musí obsahovat všechny možné akce včetně toho, jak může oponent reagovat. Takovéto plány je velice obtížné definovat v takových detailech a identifikovat jejich slabiny, opomenutí apod.

V metodě Monte Carlo existuje několik základních plánů (např. pro průzkum, útok, přesun k cíli apod.). Tyto plány jsou na sobě nezávislé a k jejich ohodnocení je použito vzorkování. Prohledávání může vyhodnocovat plány jejich vykonáváním s různými parametry

(např. kam útočit, kde prozkoumávat) a také prováděním sekvencí plánů – a to pro obě strany (já a nepřítel).

Základní princip této metody je znázorněn na Obr. 5. Nejdříve náhodně vygenerujeme plán pro umělého hráče. Poté tento plán simulujeme pro oba hráče a vykonáme jej. Vyhodnotíme konečný stav hry (spočteme, jak se plán zdá výhodný) a výsledky odešleme umělému hráči. Postup opakujeme tak často, jak to naše zdroje dovolí. Nakonec umělý hráč vybere plán, který měl nejlepší výsledky.

Abychom s přidělenými zdroji zvládli co nejvíce, je potřeba abstrakce. Tato metoda se spoléhá na spolehlivý statistický výběr vzorků, na což je však potřeba mnoho dat. Pro nejlepší výsledky je třeba úroveň abstrakce vybrat tak, aby bylo prohledávání rychlé ale užitečné. Abstrakce je volitelná. Pokud je případ použití dostatečně jednoduchý, nemusíme ji vůbec použít. Implementace funkce pro zhodnocení plánů buď vyžaduje odborné znalosti, nebo jsme nuceni použít automatické vyhodnocení, např. použitím predikční metody „temporal diference learning“ [h].

Informace v této kapitole byly čerpány z [8], Monte Carlo Planning in RTS Games.

2.2.2. Cílem orientované plánování (Goal-oriented action planning)

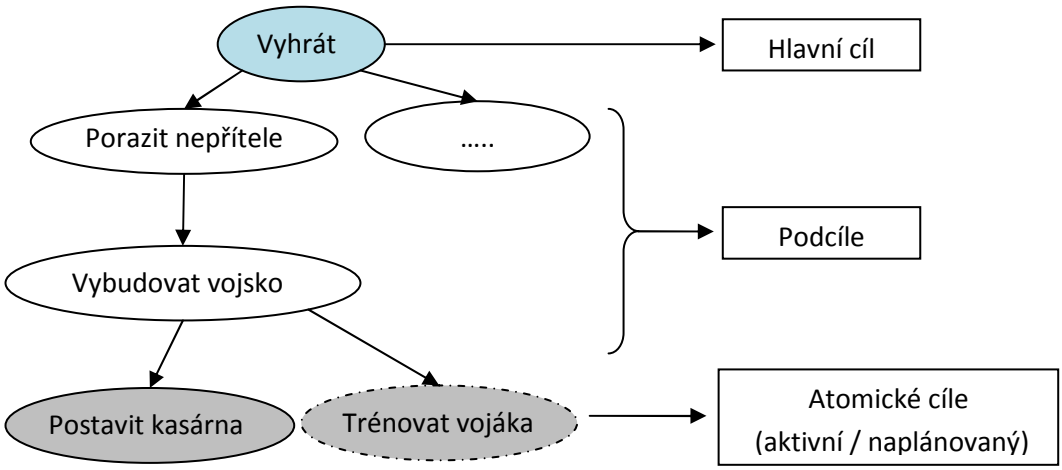
Tato metoda je v současné době poměrně hodně používaná, byla použita např. ve hře Empire: Total war (Creative Assembly) nebo v RTS taktické strategii s četnými prvky RPG - Demigod (Gas Powered games), oba tituly jsou z roku 2009.

Základní princip metody je takový, že každá postava má svoje cíle, neboli motivy, a akce, ze kterých můžeme vybírat. Smyslem je splnit daný cíl (nebo se přiblížit jeho splnění) s pomocí platných akcí.

Motivů mohou být i stovky a aktivní nemusí v daný čas být pouze jeden. Každý cíl má ve většině implementací svoji úroveň důležitosti, tzv. „insistence“ reprezentovanou číslem. Cíl s vyšší důležitostí zde má větší vliv např. na chování postavy. Pro některé účely je možno implementovat bez insistence, avšak může být obtížné vybrat cíl, na který se zaměřit.

Akce, ze kterých můžeme vybírat, mohou být generovány centrálně nebo je mohou generovat objekty ve hře. Tyto akce jsou závislé na aktuálním stavu hry.

Problém nastává, pokud je cíl příliš komplikovaný. Lze jej však vyřešit velice jednoduše, dekompozicí na jednodušší a v ideálním případě lépe proveditelné podcíle. Cíle, které již nelze dále dekomponovat se nazývají atomické a často jsou to vlastní akce, které se mají vykonat (např. stavba budovy nebo místo kam zaútočit). Takováto dekompozice nám umožňuje vložit do hry jakousi nepředvídatelnost (dekomponujeme na podcíle které NÁM mohou pomoci).



Obr. 6: Ukázka jednoduchého stromu cílů

Ve většině her, nejen strategických, máme na začátku cíl „*Vyhrát*“. Toho ale lze docílit mnoha různými způsoby. Pro každý způsob tedy máme v seznamu akcí možnost. Ze seznamu vybereme následně ty, které se nám hodí. Například tedy že „*porazíme nepřítele*“. My však nevíme co provést, abychom splnili cíl porazit nepřítele. Projdeme tedy znovu seznam akcí a narazíme například na „*vybudovat vojsko*“. Přidáme ji tedy do seznamu cílů. Stejný postup opakujeme, dokud nenarazíme na akci, která je atomická a u které víme přesně co vykonat, aby byl cíl splněn. V našem případě jsou to akce „*postavit kasárna*“ a „*trénovat vojáka*“.

Cíl může dospět do několika různých stavů. Jsou to:

1) **Dokončen** – cíl byl splněn. U atomického cíle je to jednoduché, např. budova byla postavena. U cíle neatomického, například vybudovat vojsko, může být problém složitější. Ve většině případů je neatomický cíl splněn, pokud byly splněny všechny jeho podcíle.
2) **Selhal** – cíl již nemůže z nějakého důvodu být splněn (například bychom chtěli trénovat jednotku, budova však byla zničena)

Z hlediska plánování jsou pro nás také významné tyto dva stavy:

1) **Neaktivní** – cíl je potřeba vykonat ke splnění nadřazeného cíle, zatím však není možné jej provést (například trénovat vojáka v ukázce na obr. 5 ještě nemůžeme, protože nemáme postavena kasárna, máme to však v plánu). Tyto cíle jsou při průchodu stromem vynechány.
2) **Aktivní** – cíl je aktuálně zpracováván při průchodu stromem

Informace byly čerpány z [2], Artificial Inteligence for Games, a [3], AI game programming wisdom.

2.2.3. Plánování založené na případech (Case-based planning, CBP)

Tato metoda se inspiruje tendencí lidí přirovnávat nové situace k situacím již prožitým v minulosti. Spočívá v analýze situace (vstupů), které se snaží porovnávat s databází znalostí a vybírat ty nejvhodnější výstupy vzhledem k dané situaci.

Plánování založené na případech se snaží zlepšit plánování v oblastech předcházení chybám úpravou a znovupoužitím dřívějších plánů. Od jiných technik se tedy liší v několika oblastech. Jinak se tvoří inicializační plán, jinak se reaguje na chyby v plánech a jinak se plány ukládají. Inicializační plán je vybírán s účelem předcházet problémům.

Plánovací algoritmus by měl být schopen rozpoznat plány, které selhaly a také je tak označit. Plán selhal pokud nesplnil některý z cílů, pro které určen. To může nastat ze dvou důvodů:

1) Chyba v plánování („Planning failure“)
2) Chyba po neočekávané události („Exception failure“)

Dle druhu chyby se liší reakce plánovače. Pokud dojde k chybě v plánu, je pouze pozměněn konkrétní plán. Pokud však dojde k neočekávané události, je nutné změnit chápání světa.

Při chybě v plánování si musí plánovač položit otázku „proč chyba nastala“ a využít odpovědi k vyhledání nové strategie. Při neočekávané události musí nejdříve odpovědět na otázku „co bylo špatně s plánem“ a až poté na „co bylo špatně s panováním“. Důsledkem je opravení nejen chybného plánu, ale také znalostí, které způsobily sestavení tohoto chybného plánu.

Když to celé shrnu, tak plánovač musí znát tyto věci:

1) Inicializační plán
2) Aktuální stavy hry
3) Aktuální cíle, které je třeba uspokojit (v tomto může být problém v případě, že cíl plně neuspokojí žádný plán, nebo naopak uspokojí více plánů jeden cíl)

2.3. Učení (Learning)

2.3.1. Rozhodovací učení (Decision learning)

Základem realizace učící se umělé inteligence je naučit ji dělat správná rozhodnutí. Dříve zmíněné techniky jako například stavové stroje vymezují schopnost postavy dělat rozhodnutí, zda jsou nebo nejsou aplikovatelná v dané situaci (např. pokud zbraň nemá náboje, není žádná možnost vybrat střelbu). Naproti tomu učení pracuje s pravděpodobnostmi. Vždy je nějaká pravděpodobnost (i když malá) že se provede každá možná akce. Naučit striktní omezení je velice obtížné zkombinovat s učením se základních vzorů chování tak, aby přechytračily lidské oponenty.

Učící se umělá postava má nějakou množinu možností jak se chovat. To můžou být řídící chování, animace, nebo na vyšší úrovni i strategie pro válečné hry. Tyto znalosti mohou obsahovat informace jako např. vzdálenost nejbližšího nepřítele, množství zbylé munice, relativní velikost hráčovy armády, apod. Účelem je naučit asociovat rozhodnutí s pozorováním. Časem se může umělá postava učit, která rozhodnutí pasují ke kterým pozorováním a zvyšují tak svůj výkon.

Za účelem zvýšení výkonu potřebujeme zlepšit odezvu učícího se algoritmu. Tato odezva se nazývá dohled – „supervision“. Dohled má dva různé druhy, které se liší učícími se algoritmy nebo odlišnou charakteristikou stejného algoritmu.

Silný dohled má podobu souboru správných odpovědí. Skupiny pozorování jsou asociovány s chováním, které by mělo být vybráno. Algoritmus se učí při daných vstupech vybrat správné chování. Správné odpovědi jsou nejčastěji poskytnuty lidským hráčem. Vývojář může hru nějakou dobu hrát a pozorovat při tom počítačového hráče. Počítačový hráč si uchovává postupy získané z pozorování a rozhodnutí, které lidský hráč udělá. Může tak později jednat stejným způsobem.

Slabý dohled nevyžaduje soubor správných odpovědí, místo toho je odezva dána tím jak dobré jsou výběry akcí. Tato odezva může být dána vývojářem, ale častěji je poskytnuta algoritmem, který monitoruje výkon umělého hráče.

Silný dohled je jednodušší na implementaci, je ale méně flexibilní. Vyžaduje někoho, kdo by ho učil co je dobré a co špatné. Slabý dohled se může učit co je správné a špatné sám, ale je obtížnější aby se učil správně.

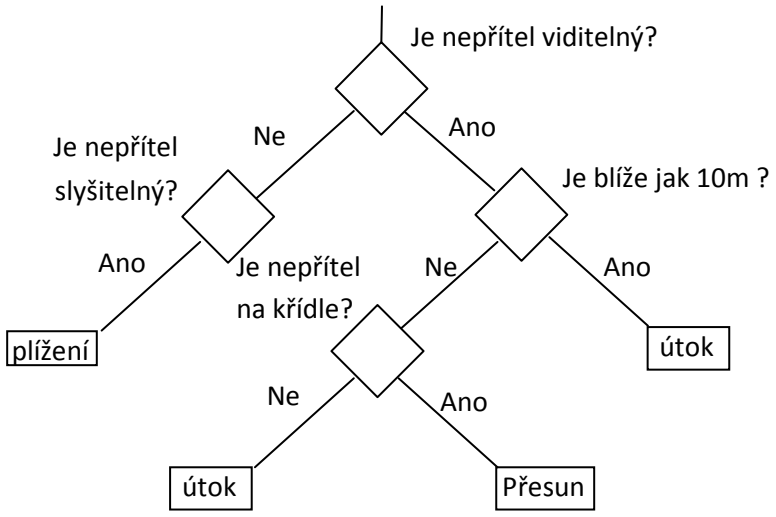
Informace byly čerpány z [2], Artificial Inteligence for Games.

2.3.2. Rozhodovací stromy (Decision Tree Learning)

V části 2.1 jsem se zmínil o rozhodovacích stromech – souboru rozhodnutí které generují akce založené na pozorování. V každé části stromu jsme zvažovali některý aspekt herního světa a vybírali kam dále směřovat. Soubory takovýchto rozhodnutí vedly k akci.

Rozhodovací stromy se mohou efektivně učit, být dynamicky konstruovány ze skupiny pozorování a akcí poskytnutých pomocí silného dohledu. Konstruované stromy mohou být k rozhodování během hry běžně použity.

Existuje množství různých algoritmů rozhodovacích stromů, které slouží ke klasifikaci, predikci a statistické analýze. Algoritmy použité v herní umělé inteligenci jsou typicky založeny na Quinlanově ID3 algoritmu.



Obr. 7: Ukázka jednoduchého rozhodovacího stromu

Algoritmus ID3

Zkratka ID3 pochází z anglických výrazů „Inductive Decision tree algorithm 3“ a „Iterative Dichotomizer 3“, oba jsou používané. První výraz by se dal přeložit jako induktivní algoritmus pro rozhodovací stromy 3, druhý jako Iterativní Dichotomizer 3. Jako všechny algoritmy i tento má mnoho optimalizací použitelných v různých situacích.

Základní ID3 algoritmus používá soubor vzorů „pozorování – akce“. Pozorování jsou v ID3 obvykle nazývány „atributy“. Začíná se s jedním koncovým uzlem, k němu je přiřazen soubor vzorů. Každý uzel rozdělí vzory do dvou skupin. Dělení je prováděno na základě atributu a je vybráno to, které vyprodukuje nejefektivnější strom. Když je rozdělení provedeno, novým dvěma uzlům je předána podmnožina vzorů, která je na ně aplikována. Algoritmus se pak opakuje pro každý z nich, je tedy rekurzivní.

Proces dělení sleduje v každém tahu všechny atributy a počítá informační zisk z každého možného rozdělení. Rozdělení s nejvyšším informačním ziskem je vybráno jako konečné rozhodnutí pro daný uzel. Informační zisk je matematický problém, který je přiblížen v následující části.

Entropie[i] a informační zisk

Abychom určili, který atribut bude v každém kroku zvažován, používá algoritmus ID3 entropii akcí v množině. Entropie je míra informace v souboru vzorků. V našem případě určuje, jak hodně akce v souboru vzorků odpovídá každé ostatní. Pokud všechny vzorky mají stejnou akci, entropie bude 0. Pokud budou akce distribuovány rovnoměrně, entropie bude 1. Informační zisk je redukce celkové entropie.

O informaci v nějaké množině můžeme přemýšlet jako o míře, kterou se tato informace podílí na celkovém výstupu. Pokud máme množinu vzorků, kde jsou všechny akce různé, pak nám to že je v dané množině mnoho o tom, kterou akci vybrat, neřekne. Ideálně chceme dosáhnout situace, kde nám přítomnost v množině řekne přesně kterou akci vzít.

Například máme dvě možné akce: útočit a bránit. Dále máme 3 atributy: zdraví, krytí a náboje. Pro jednoduchost si rozdělíme atributy pouze na pravdu nebo nepravdu: zdravý nebo raněný, krytý nebo nekrytý, s nabitou nebo s prázdnou zbraní.

Naše množina vzorků může tedy vypadat například takto:

Zdraví	**Krytí**	**Zbraň**	**Akce**
Zdravý	*Kryje se*	*Nabitá*	*Útok*
Raněný	*Kryje se*	*Nabitá*	*Útok*
Zdravý	*Kryje se*	*Prázdná*	*Bránit se*
Raněný	*Kryje se*	*Prázdná*	*Bránit se*
Raněný	*Nekrytý*	*Nabitá*	*Bránit se*

Pro dva možné výstupy, útok a obrana, je entropie množiny akcí dána vztahem:

$$E = -p_A \log_2 p_A - p_D \log_2 p_D$$

Kde p_A je podíl akce útok na množině vzorků a p_D je podíl akce bránit se. V našem případě to znamená, že entropie celé množiny je 0.971.

V prvním uzlu algoritmus se algoritmus podívá na každý možný atribut, podělí množstvím vzorků a vypočítá entropii spojenou s každým podílem.

Děleno pomocí:

Zdraví $E_{zdravý} = 1.000$ $E_{raněan} = 1.000$

Krytí $E_{krytý} = 1.000$ $E_{nekrytý} = 1.000$

Náboje $E_{nabito} = 1.000$ $E_{prázdno} = 1.000$

Informační zisk pro každé rozdělení je redukce entropie ze současné množiny vzorků (0.971). To je dáno vzorcem: $G = E_S - P_T \cdot E_T - p_\perp \cdot E_\perp$, kde P_T je poměr vzorků pro které atribut je pravdivý a E_T je entropie pro tyto vzorky. Obdobně, $p_\perp$ a $E_\perp$ se odkazuje na příklady, pro které je atribut nepravdivý. Rovnice nám ukazuje, že entropie jsou násobeny podílem vzorků v každé kategorii.

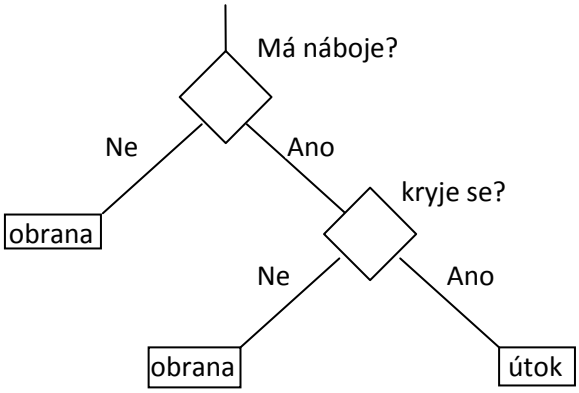
V našem příkladu můžeme spočítat pro každý atribut tuto informaci:

$$G_{zdravý} = 0.020$$

$$G_{krytý} = 0.171$$

$$G_{nabito} = 0.420$$

Takže z našich tří atributů nám nejlépe vyšel atribut nabito, což dává smysl. Bez nábojů nemá moc smysl útočit. Podle zásady učení se od nejobecnějšího, použijeme nabito jako naši první větev rozhodovacího stromu. Pokud bychom tímto způsobem pokračovali dále, získáme strom znázorněný na obrázku Obr. 8.



Obr. 8: Výsledný strom rozhodovacího algoritmu

Jak je vidět, nezáleží vůbec na atributu zdravý. Pro naše rozhodnutí je za uvedených podmínek nepodstatný. Pokud bychom uvedli více vzorků, mohla by však nastat situace, kdy bude hrát roli a rozhodovací strom ho použije.

Zatím jsme uvažovali pouze dvě akce. Stejný proces však funguje s více akcemi, musíme však generalizovat vzorec pro výpočet entropie:

$$E = -\sum_{i=1\ldots n} p_i \log_2 p_i ,$$

kde n je počet akcí a p_i je podíl každé akce na množině vzorků.

Dále jsme uvažovali logaritmus se základem 2, většina systémů však má základ jiný. Typicky je *e* nejrychlejší, bývá však i základ 10 pokud je pro něj optimalizována implementace.

Pokud máme více než dvě kategorie a tím pádem více než dva dceřiné uzly, musíme upravit i vzorec pro informační zisk:

$$G = E_S - \sum_{i=1...n} |S_i| / |S| \cdot E_S ,$$

Kde S_i je množina vzorků pro každých *n* hodnot atributu.

Informace, obrázky a vzorce v kapitole byly čerpány z [2], Artificial Inteligence for Games.

2.3.3. Posilové učení (Reinforcement Learning)

Pod pojmem posilové učení se skrývá několik technik založených na zkušenostech. V nejobecnějších technikách má 3 komponenty:

1) Část zkoumající různé akce ve hře
2) Posilová funkce dávající odezvu jak dobrá každá akce je
3) Učící pravidla, které spojují předchozí dvě komponenty dohromady

Každá komponenta má několik různých implementací závisejících na konkrétní situaci. Účelem těchto algoritmů je dát umělému hráči schopnost se sám dle okolností rozhodnout, která z akcí je nejlepší. To však není většinou v daném okamžiku jasné. Proto je důležité, abychom byli schopni dávat velice různorodé informace a dostávat z nich odezvu jen když se stane něco podstatného. Umělá postava by se měla učit, že všechny akce vedoucí k nějaké události je dobré provést. Dokonce i ty, které v okamžiku vykonání nedávají žádnou odezvu.

Q-learning

Algoritmus Q-learning se spoléhá na to, že je problém reprezentován specifickým způsobem. S touto reprezentací může ukládat a obnovovat relevantní informace tak, jak prozkoumává vhodné akce, které jsou dostupné.

Herní svět je reprezentován jako stavový stroj. V určitém čase je algoritmus v nějakém stavu. Stav by měl mít v sobě zakódovány všechny relevantní detaily o prostředí postavy a další interní data.

Pokud je tedy například zdraví podstatné pro nějakou postavu a umělé hráč zjistí, že ve dvou identických situacích má dvě různé úrovně zdraví, pak to bude považovat za dva různé stavy. Cokoli není ve stavu začleněno, nemůže být učeno. Pokud bychom tedy nezačlenili hodnotu zdraví, pak bychom ho nemohli zvažovat při rozhodování.

Stavy se skládají z faktorů jako například pozice, blízkost nepřítele nebo síla. Ve hře potřebujeme být schopni přeložit současný stav hry do jednoho stavového čísla, použitelného pro učící algoritmus. Naštěstí algoritmus nikdy nevyžaduje opak, nepotřebujeme překládat stavové číslo zpět do herních pojmů.

Q-learning je bezmodelový (model-free) algoritmus, protože se nepokouší sestavit model toho, jak svět funguje. Vše řeší pomocí stavů. Algoritmy, které nejsou bezmodelové se snaží ze stavů, které navštíví, rekonstruovat co se ve hře děje. Proto jsou algoritmy jako Q-learning podstatně jednodušší na implementaci.

Algoritmus se snaží pro každý stav najít vhodné akce. V mnohých hrách jsou však vždy dostupné všechny akce, často jsou naopak dostupné akce pouze v určitých situacích. Když umělý hráč v současném stavu provede nějakou akci, posilová funkce by měla dát pozitivní nebo negativní odezvu. Pokud není výsledek zřejmý, je odezva nulová. Hodnoty, které tato funkce vrací, nejsou nijak omezené, běžně však bývají v rozsahu [-1, 1].

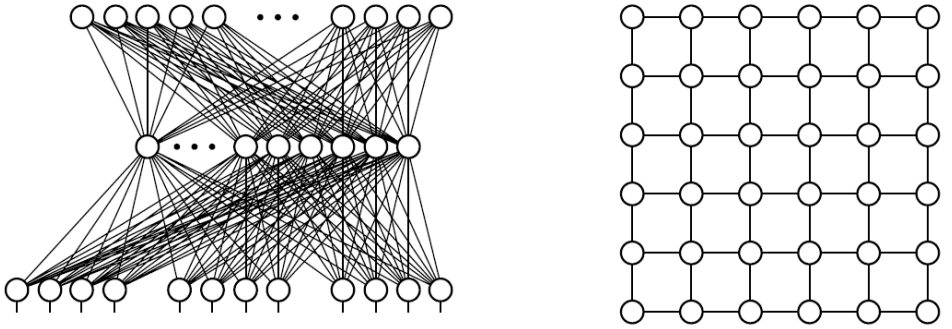
Po vykonání akce se umělý hráč dostane do nového stavu. Vykonání stejné akce ve stejném stavu nemusí vždy vést do stejného stavu hry. Ostatní hráči mají na stav hry také vliv. Velkou výhodou Q-learning a většiny algoritmů posilového učení je, že se může vypořádat s tímto druhem nepředvídatelnosti. Čtyři základní elementy – počáteční stav, vybraná akce, posilová hodnota a výsledný stav – jsou nazývány zkušenostní kolekce („experience tuple“), často zapisována jako $\langle s, a, r, s' \rangle$.

2.3.4. Neuronové sítě (Neural Networks)

Umělé neuronové sítě, zkráceně pouze neuronové sítě, jsou předvoj moderních biologií inspirovaných technik, které mají počátky v sedmdesátých letech. Jsou vhodné pro široké spektrum aplikací. Jejich primárním účelem je rozpoznávání.

Neuronové sítě se skládají z velkého množství relativně jednoduchých uzlů, v každém běží stejný algoritmus. Tyto uzly jsou umělými neurony. Původně byly určeny k simulaci operací jedné mozkové buňky. Každý neuron komunikuje s podmnožinou dalších neuronů, všechny jsou propojeny do vzorů charakteristických pro neuronové sítě. Tyto vzory jsou nazývány architekturou, nebo topologií, neuronových sítí.

Běžnou formou neuronových sítí jsou vícevrstvé perceptrony. Perceptrony jsou uspořádány ve vrstvách, kde je každý perceptron připojen do všech ostatních před a za ním (jak je znázorněno na obrázku Obr. 9). V mnohých typech neuronových sítí jsou některé spoje vstupy a některé výstupy.



Obr. 9: Vícevrstvé perceptrony

Vícevrstvé perceptrony berou vstupy ze všech uzlů v předchozí vrstvě a dávají jednu výstupní hodnotu do všech v té následující. Toto chování je známé jako, volně přeloženo, dopředná síť („Feedforward network“). Vrstva vlevo je nazývána vstupní vrstvou, je obvykle vstupem programátora, vrstva vpravo je výstupní vrstvou. Ta ve finále vykonává něco užitečného. Pokud má dopředná síť smyčky, spoje vedoucí z následující vrstvy zpět do dřívější, nazývá se tato síť rekurzivní. Rekurzivní sítě mohou mít velice složité a nestabilní chování a je mnohem obtížnější je kontrolovat. Zvláštním typem neurálních sítí jsou sítě bez specifikovaného vstupu a výstupu, kde každý spoj je jak vstupním tak výstupním.

Každý neuron je v nějakém čase v určitém stavu, což můžeme považovat za výstupní hodnotu neuronu (klasicky reprezentováno jako desetinné číslo). Algoritmus definuje, jak neuron generuje svůj stav na základě vstupů, nebo, v případě že vstupy a výstupy nejsou definovány, algoritmus generuje stavy na základě stavů připojených neuronů. Takováto architektura nám umožňuje paralelismus.

Doposud popsaný princip nedovoluje učení. Síť měla svoje vstupy a vstupní vrstvu, každý neutron dělal svoji a následně byl čten výstup z výstupní vrstvy. Je to velice rychlý proces. Aby se takováto síť mohla učit, musíme perceptrony přepnout do specifického učícího se módu. Při tomto módu začne fungovat algoritmus aplikující učící pravidla. Nejběžněji používaným algoritmem je zpětná propagace, kde síť, která je normálně dopředná (každá vrstva generuje výstup na základě předchozí vrstvy) začne pracovat v opačném směru (pracuje zpětně z výstupu).

3. Herní engine

Jako herní engine pro implementaci některých metod umělé inteligence jsem použil ORTS framework. ORTS je Open Source aplikace licencovaná pod GNU GPL. Byla vytvořena na univerzitě Alberta, v Edmontonu, Kanada. Vedoucím projektu je profesor Michael Buro, na projektu spolupracovalo i několik absolventů a dalších přispěvovatelů. První verze se objevila již v roce 2002, v roce 2003 a 4 bylo přidáno několik funkčností, jako kompilace pod MAC-OSX a Windows, 3D modely, různá zdokonalení modulů apod. Další vývoj a optimalizace probíhají doposud, avšak již ne v takové míře jako zpočátku.[6]

V rámci ORTS proběhlo i několik soutěží. Těchto soutěží se mohl účastnit kdokoli, a to v několika kategoriích (Souboj vojenských jednotek, reálná RTS hra, efektivní těžba surovin a několik dalších).

Aplikace byla vytvořena primárně k účelu testování umělé inteligence, kvůli dostupným zdrojovým kódům (a licenci GPL) však můžeme dle libosti testovat i další námi vytvořené součásti. Napsána je v jazyce C++, využívá několika skriptovaných tříd (blueprintů) a několik externích knihoven. Jsou to SDL, SDL_NET, Qt, OpenGL, GLUT a GLEW. Tyto knihovny jsou multiplatformní, stejně jako ostatní části, lze jej tedy využívat jak v systémech Windows, tak v UNIXových systémech.

ORTS je Client-server aplikace, jež využívá systému „hraní na straně serveru“ (Server-Side Game Simulation). Hra jako taková tedy běží na straně serveru, klient pro něj pouze generuje příkazy. Toto řešení nám přináší některé výhody a nevýhody. Nevýhodou je, že server může být poměrně dost vytížen, pracuje totiž s velkým množstvím dat (požadavky od klientů, grafické rozhraní – GUI – všech klientů). Avšak pro naše účely testování UI nám to vůbec nevadí, naopak nám to přináší několik výhod. Hlavní výhodou je jistota, že náš protivník nepodvádí. Pouze server zná celý stav hry, a nedovolí klientovi získat nepatřičné informace a provést nepovolené akce. Další výhodou je, že se můžeme starat jen o klientskou část hry, ne o hru jako takovou.

Mnoho informací a odkazů lze najít na domovských stránkách projektu ORTS http://skatgame.net/mburo/orts/.

Klient a server

Server je, jak jsem se již v úvodu k ORTS zmínil, odpovědný za běh hry jako takové. Určuje co který hráč (klient) může provádět a co může znát o současném stavu hry. Akce, které chce klient provést, se odesílají v cyklech. Server v každém cyklu provede požadované akce, přidá nově jednotky a odstraní ty mrtvé. Jsou změněny pozice posunujících se objektů, kolidující jsou zastaveny. Nakonec je vypočítána viditelná oblast. Akce pro pohyb a výpočet viditelnosti konzumují většinu času serveru.

Na serveru lze pomocí zjednodušeného grafického zobrazení zobrazit aktuální stav hry. Jedná se o 2D zobrazení, kde mobilní jednotky jsou zobrazeny jako kruhy, budovy jako čtverce, a hranice jsou znázorněny čarami.

Úkolem klienta je vypočítat akce které mají být provedeny a tyto akce následně odeslat serveru. Akci lze poslat ke každému klientem vlastněnému objektu, který je ve hře (není mrtvý), a to pro všechny objekty v každém tiku hry. Jeden tik trvá obvykle 1/8 sekundy nebo méně (lze nastavit na straně serveru).

U klienta lze nastavit, stejně jako u serveru, zjednodušené grafické rozhranní. Navíc lze zobrazit stav hry ve 3D.

Skriptování

Skriptovací engine vykonává některou logiku a dovoluje definovat vlastní jednotky a akce (např. definice typů jednotek a akcí které vykonávají). Tyto definice jsou uloženy jako kontejnery pro proměnné, akce a podobjekty. Z těchto kontejnerů je definována sada schémat („blueprints“), která popisuje jména, inicializační hodnoty atributů, možné akce a strukturu objektu.[6]

Když klient příjme informace o stavu hry, jsou zde obsažena i schémata. Tyto schémata může klient rozšířit a využít je např. k přidání jednoduché UI na pozadí. Po přidání 3D modelu jako podobjektu může klient specifikovat, jak má být objekt reprezentován ve hře. Dovoluje i animace závisející na aktuálním kontextu. Klient může také registrovat speciální funkce umožňující přístup do OpenGL, např. pro kreslení bitmap.[6]

Události myši a klávesnice jsou snímány klientem, a jsou převedeny do skriptu voláním akcí speciálního grafického objektu, kterému jsou předány informace o události jako parametry.[6]

Grafické rozhranní

K interakci s lidskými hráči je implementován modul grafického rozhranní. Serverová i klientská část podporuje jednoduché 2D zobrazení. Klient navíc podporuje zobrazení pomocí OpenGL, které dokáže vyrenderovat pohled na aktuální stav ve 3D.

Při 3D pohledu jsou zobrazeny další součásti jako minimax a informační panel. Tyto části a v nich zobrazované informace lze upravit pomocí skriptování. Pro komunikaci grafického modulu se serverem nám slouží GameStateModule.

Nízkoúrovňová UI

Základní herní úkony jako hledání cesty, těžba surovin nebo automatická obrana jsou implementovány na klientské straně pomocí volitelných C++ modulů.

Pokud uživatel odešle jednotku na nějaké umístění, je generována událost path finding (hledání cesty). Modul pro hledání cesty poté naplánuje cestu do cíle a navádí jednotku (posílá příkazy pro přesun). Jak je svět prozkoumáván, tento modul posílá zprávy informující o nových jednotkách a objektech.

Pokud je inicializován modul pro těžbu zdrojů, pracuje s modulem pro hledání cesty. Nejdříve odešle jednotku k danému zdroji. Když příjme potvrzení o dosažení cíle, začne těžbu. Po dokončení těžby je opět jednotka navedena na základnu.

4. Implementace klienta

Pro implementaci klienta jsem využil připravenou aplikaci sampleai, kde je již implementováno čtení parametrů hry, inicializace komunikace mezi klientem a serverem a je připravena grafika hry. Dále je zde připraveno čtení informací ze serveru, a ukázáno jak lze jednoduše odeslat požadované akce serveru.

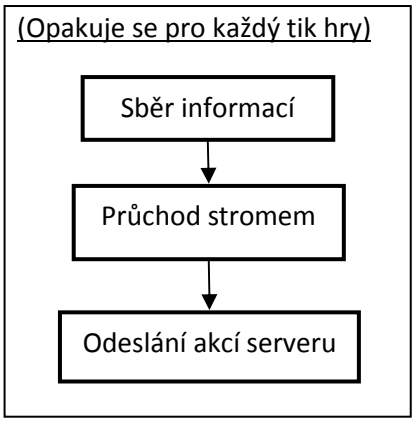
Pro čtení informací ze serveru je využita sdílená třída, kde jsou obsaženy užitečné metody pro získání změn oproti poslednímu tiku (např. seznam nových a mrtvých objektů), seznam všech objektů (např. i viditelných objektů ostatních hráčů nebo objekty nepatřící žádnému hráči), informace o mapě (rozměry mapy, informace o jednotlivých polích), počet hráčů apod. Hlavním smyslem klienta je tedy vypočítat akce, které je potřeba provést (resp. odeslat serveru).

4.1. Implementace cílem orientovaného plánování

První věcí, která je při každém plánování potřeba je uložení informací, které by šlo později využít při rozhodování. Jsou to například informace o jednotkách a budovách hráčů (mých i nepřátelských), stav mých a nepřátelských jednotek (jak pracovní síly, tak armádu), minerály v okolí apod.

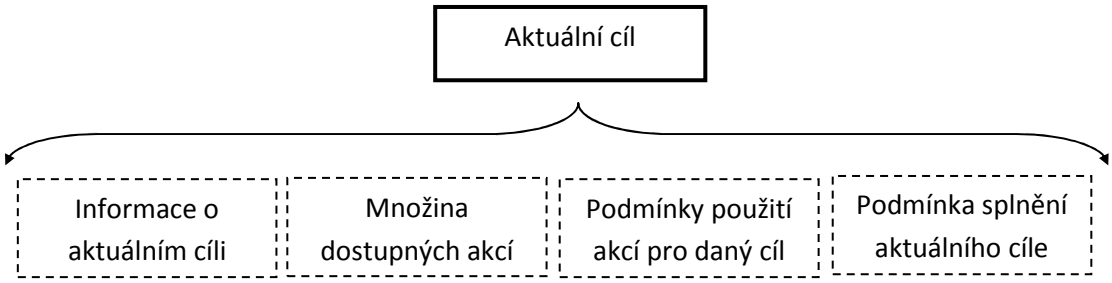
Základní princip byl detailněji popsán dříve, teď ho pouze připomenu. Máme dvě množiny: množinu akcí a množinu cílů. Množina cílů nám tvoří strom. Tento strom se generuje na základě akcí, které vybíráme podle aktuálního cíle a stavu hry.

Celý princip mé implementace vychází z tohoto principu. Hlavní části implementace jsou znázorněny na Obr. 10. Nejdříve posbíráme informace, které se změnily oproti poslednímu tiku hry. Mohou to být nové/mrtvé objekty, změněné informace o existujících objektech, zmeškané akce (ty které jsme úspěšně odeslali, ale server nepřijal), číslo zobrazovaného rámce (tiku hry) apod. Následně procházíme stromem cílů. Zde se bere v úvahu aktuální stav hry a množina akcí vhodných pro daný cíl a je postupně generován strom. Při průchodu tímto stromem jsou v případě potřeby „odesílány na server“ akce, které mají být vykonány.



Obr. 10: Základní části implementace

V mé implementaci se každý cíl skládá z několika komponent. Ty jsou znázorněny na Obr. 11. Jsou to informace o stavu cíle, rozhodování výběru následníků a podmínky splnění cíle.



Obr. 11: Struktura cíle

Informace o stavu cíle jsou uloženy stejným způsobem pro každý cíl. Jsou zde základní informace jako jméno cíle, atomičnost, následníci, hodnota cíle apod. Jméno cíle v případě že je cíl atomický určuje přímo akci, která se má vykonat. Například u jména cíle „build_barracks“ zjistíme pomocí funkce, že platí pro pracovníka a výstupem je budova. Seznam následníků je v případě atomického cíle zbytečný (žádní nejsou). V případě neatomického cíle však ukazují na potomky konkrétního uzlu. Pro případ lepší orientace obsahuje každý cíl také ukazatel na svého otce (pomáhá nám to například při rozhodování). Využití dalších proměnných se již liší cíl od cíle. Jsou zde uloženy informace jako identifikační číslo obsluhujícího pracovníka, číslo tiku hry kdy začala obsluha cíle, počítadlo kolikrát jsme se o cíl již pokusili, cílová poloha budovy apod.

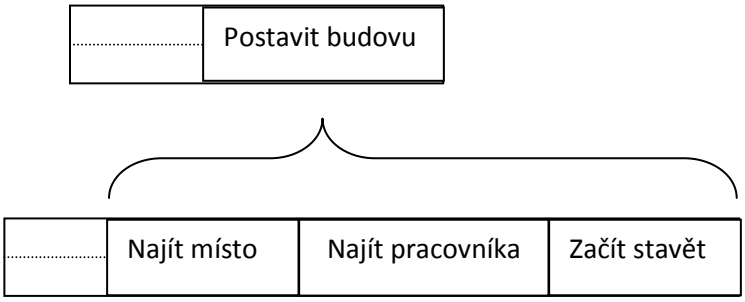
Informace o jednotlivých akcích jsou uloženy ve struktuře, která obsahuje několik informací. Jsou to název cíle, pro který akce platí, název výsledného cíle (který posléze přiřadíme jako potomka aktuálního). Název výsledného cíle je uložen ve dvou různých proměnných – jméno regenerovaného cíle a atomická akce. Pouze jedna z těchto proměnných však obsahuje jméno. Je to z toho důvodu, že snadno poznáme, zda je akce atomická či nikoli (resp. zda cíl pomocí této akce vygenerovaný bude atomický). Všechny akce jsou uloženy v proměnné typu vektor, kde jsou uloženy odkazy na struktury s jednotlivými akcemi. Tento vektor je plněn pomocí jedné inicializační funkce, kde je možné dle potřeby jednoduše přidávat nebo odebírat dostupné akce. Seznam akcí je pouze jeden, nejsou rozděleny na jednotlivé cíle. Funkci rozhodující o použití vhodných akcí jsou předány pouze ty odpovídající aktuálnímu cíli.

Další komponenta pracuje s akcemi použitelnými pro daný cíl a s informacemi o celkovém stavu hry. Toto, dalo by se říct rozhodovací, rozhranní je unikátní pro každý cíl. Specifikuje přesné podmínky kdy vybrat a aplikovat danou akci a kdy cíl aktivovat (změnit stav cíle z „inactive“ na „active“). Je to dalo by se říct nejdůležitější a nejrozsáhlejší část, která by se dala velice dlouhou dobu vylepšovat.

Další a poslední důležitou částí je část rozhodující o splnění cíle. Pokud je cíl atomický, je rozhodnutí poměrně jednoduché – například budova je postavena, jednotka vytrénována. U

neatomických to však v některých případech tak jednoduché není, to ale vysvětlím později na příkladech.

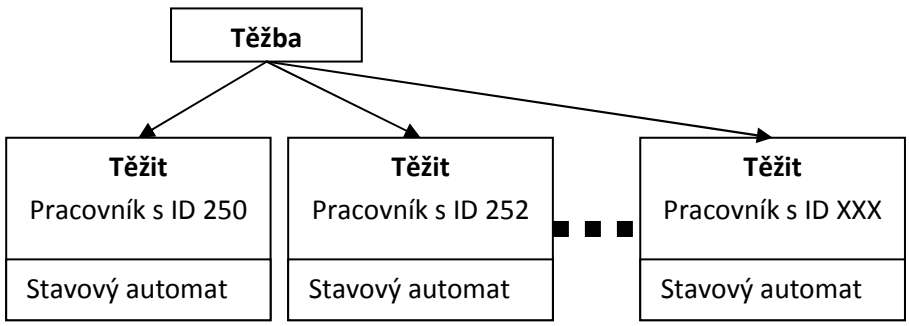
To celé propojuje dohromady funkce pro průchod stromem. Zde jsem měl na výběr ze dvou metod, rekurzivní nebo pomocí zásobníku. Z důvodu složitosti a předvídatelnosti chování jsem zvolil průchod pomocí zásobníku. Princip je znázorněn na Obr. 12. Máme cíl, který je neatomický. To znamená, že ho potřebujeme dle daných pravidel (zatím neuvažujeme) rozgenerovat. To spočívá ve vyjmutí cíle ze zásobníku a vložení cílů z něj vygenerovaných.



Obr. 12: Princip průchodu stromem pomocí zásobníku

Při průchodu stromem bereme v úvahu, zda je cíl atomický či nikoli. Pokud je neatomický musíme zkontrolovat, zda nebudeme nějaký cíl přidávat. Pokud atomický je, budeme vykonávat akci (případně pouze kontrolovat zda nebyl dokončen).

Seznam akcí, ze kterého vychází funkčnost mé implementace je na Tab. 14. Celý výsledný strom, který z těchto akcí následně vznikne (tedy by vznikl, pokud by se provedla každá akce z obrázku právě jednou) je přiložen v příloze 1. Princip funkce je myslím již dostatečně zřejmý, za zmínku však stojí cíl neatomický „*Těžba*“, a jeho atomický následník „*Těžit.*“. Atomický cíl „*těžit*“ reprezentuje jednotlivé pracovníky, kteří jsou určeni pro těžbu. Tento cíl je implementován jako stavový automat, který pracovníka při těžbě obsluhuje.



Obr. 13: Část stromu cílů obsluhující těžaře

Tab. 14: Následuje tabulka akcí pro plánování

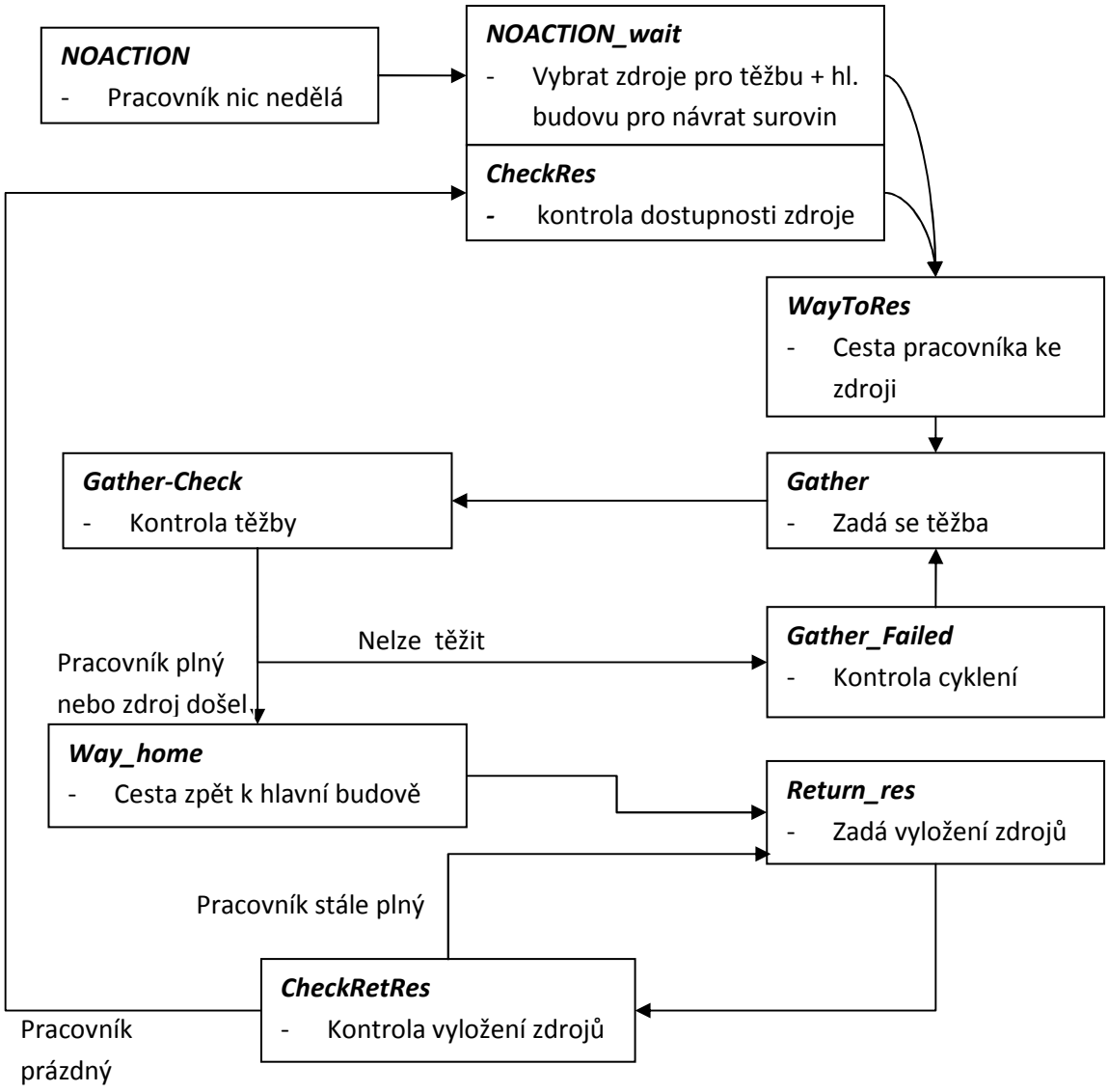
Na následujícím obrázku je seznam akcí, které když „složíme“ dohromady, vytvoří strom ukázaný v příloze 1. Každý cíl může mít nějakého následníka, a následník případně dalšího, dokud není následník atomický. Ten je koncem stromu a následníky nemá.

Název cíle	**Jméno následníka**	**atomická**
Vyhrát	Zničit_nepřítele Zajistit_suroviny	
Zničit_nepřítele	Postavit_armádu Zaútočit Rozšířit_základnu Průzkum	
Postavit_armádu	Trénuj_vojáka Trénuj_tank	
Trénuj_vojáka	Postav_budovu Trénuj_jednotku	
Trénuj_tank	Postav_budovu Trénuj_jednotku	
Trénuj_workera	Trénuj_jednotku Postav_budovu	
Postav_budovu	Najdi_misto Najit_workera Presun_workera Stavej	Ano Ano Ano Ano
Trénuj_jednotku	Najdi_budovu Najdi_shromžďovací bod Trenuj	Ano Ano Ano
Zajistit_suroviny	Težba Trenuj_workera Rozšířit_základnu	
Těžba	Těžit	Ano
Zaútočit	zničit_objekt	
Průzkum	Průzkum_mapy Průzkum_místa	

Princip je automatu, který je součástí atomického cíle „těžit“, je znázorněn na obrázku ……. Nejdříve je pracovník naveden ke zdroji (předem vybranému), poté je zadána těžba a kontroluje se, zda je pracovník plný. Když je, je odeslán k hlavní budově, kde je zadána akce pro vyložení surovin. Když jsou suroviny vyloženy, opakuje se celý stavový automat znovu. V průběhu je třeba počítat s nepředvídatelnými problémy jako například že se pracovník nedostane

k požadovanému zdroji, nepodaří se vyložit suroviny, nebo že zdroj dojde, aniž by se pracovník stihl naplnit. V případě neošetření by pracovník zamrzl a zůstal neaktivní do konce hry.

Ve hře funguje ještě jeden stavový automat. Dá se říct nadstavba tohoto pro těžbu. Každá jednotka vykonává v určitém čase nějakou činnost. Ať už to je stavba budovy, některý ze stavů těžby nebo že je neaktivní. Každý stav musí mít smysl a musí mít pokračování. Jinak by se mohlo stát, podobně jako u pracovníka, že jednotka zamrzne v nějakém stavu, který nic neprovádí.



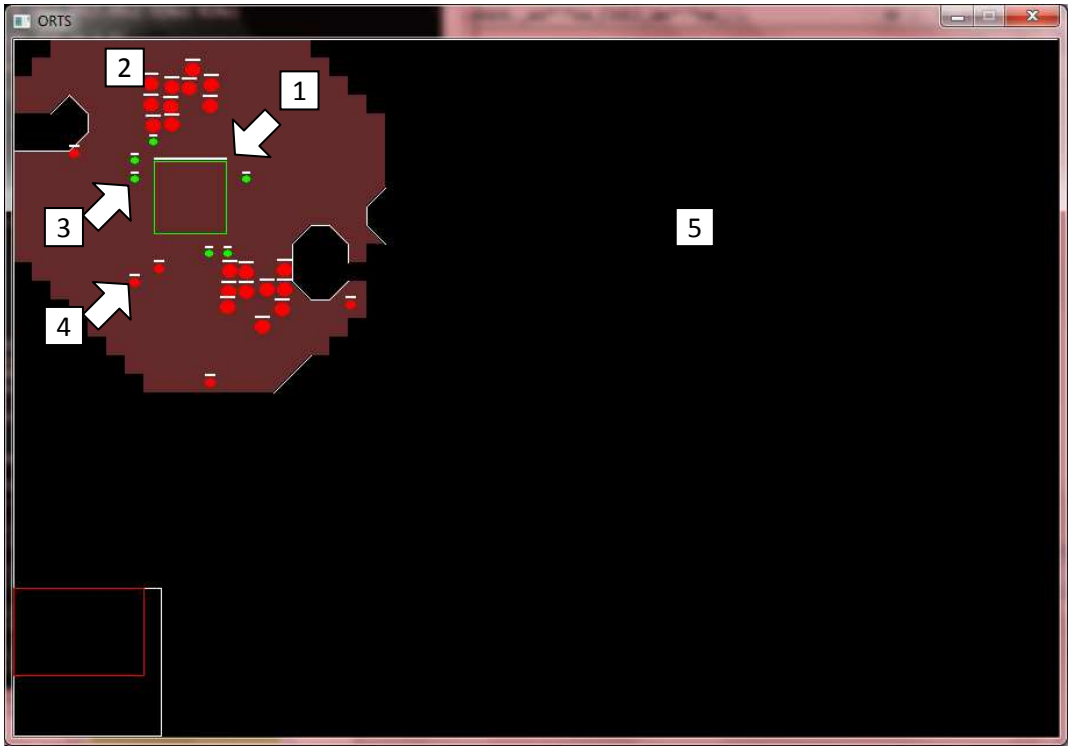
Obr. 15: Stavový automat těžby surovin

4.2. Ukázka výsledné aplikace

V následující ukázce je spuštěna moje implementace klienta ve 2D režimu.

A) Začátek hry

1) Hlavní budova
2) Pole krystalů (větší červená kolečka)
3) Pracovník (malé zelené kolečko)
4) Náhodně se pohybující neutrální jednotka (malá červená kolečka)
5) Neprozkoumaná mapa

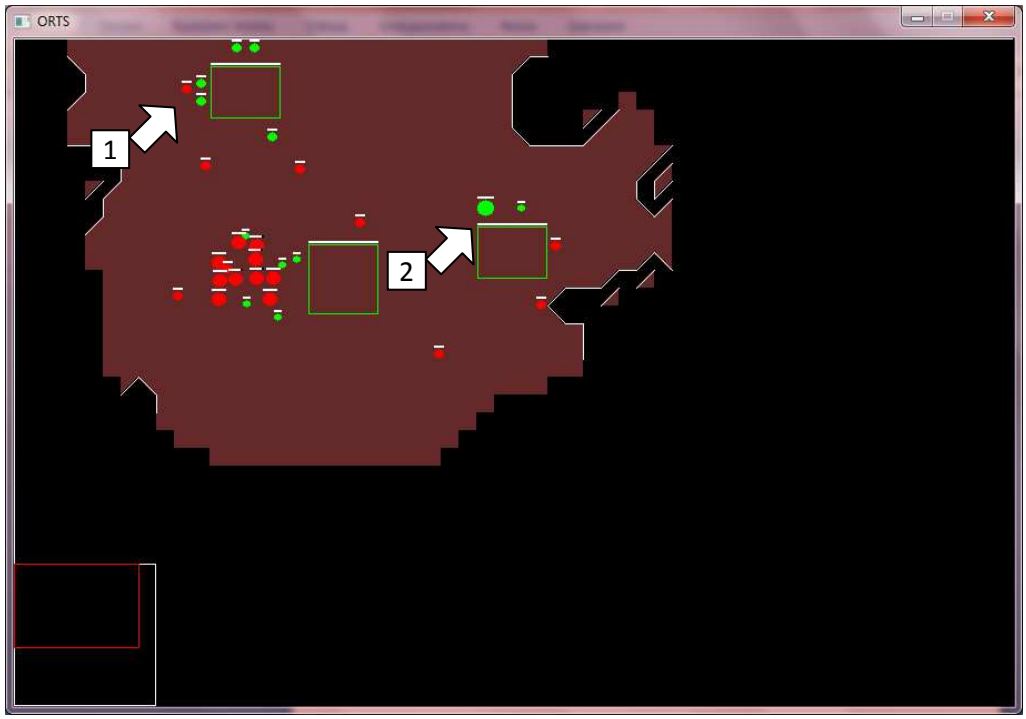


B) Začíná těžba surovin + staví se nová budova

C) Postaveny budovy, vytrénováno několik jednotek

1) Vojáci (o něco větší zelené kolečko než pracovník)

2) Tank (největší zelené kolečko)



5. Závěr

V této práci jsem prostudoval několik publikací jako Artificial inteligence for games (od pana Millingtona, 2006), AI game programming Wisdom (Steve Rabin, 2002) a *AI for game developers* (David M. BOURG, 2004) týkajících se umělé inteligence. Pochopil jsem tak a v této práci vysvětlil mnohé z těchto metod, některé podrobněji, některé spíše obecněji (například poměrně složité metody strojového učení). Dále jsem nastudoval z různých materiálů a zdrojových kódů princip herního engine ORTS, který jsem využil k testování mé implementace. Implementoval jsem metodu umělé inteligence z oblasti plánování – cílem orientované plánování. V rámci této implementace jsem také navrhl a implementoval stavový automat.

Práce na tomto tématu byla pro mne velice zajímavá a dozvěděl jsem se mnoho věcí nejen z oblasti umělé inteligence, ale také jsem pochopil základní algoritmy použité ve skutečně funkční hře. Další cenné zkušenosti mi přinesla už samotná práce s takto rozsáhlým projektem.

Za vývojem herní umělé inteligence často stojí tým více lidí a pracují na ní i několik let, proto, je třeba tuto implementaci brát spíše jako základ, na který lze postupně aplikovat různá vylepšení. I z tohoto hlediska bylo použití metody cílem orientovaného plánování výhodné, jeho základní princip spočívá v práci se stromem a upravit (resp. vyměnit či doplnit) část stromu je poměrně jednoduché.

Jak jsem již zmínil, téma mě velice zajímá a určitě se v budoucnu pokusím implementovat alespoň některá rozšíření. Například je třeba doplnit průzkum mapy, a to jak z hlediska hledání zdrojů a překážek využitelných v boji, tak z hlediska získávání informací o nepříteli. Dále je třeba doplnit expanzi základny, nejlépe s využitím numerických pro výpočet nejvýhodnějšího místa. Další věcí je útočení na nepřítele a začlenění alespoň nějakých bojových taktik. Pro lepší rozhodování a práci s informacemi by bylo výhodné také implementovat Fuzzy logiku. Práce se surovinami je také část, kterou lze stále vylepšovat.

Seznam obrázků:

Sezam příloh:

Zdroje:

[1] History of artificial intelligence. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 13 October 2005 , last modified on 31 January 2011 [cit. 2011-03-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/History_of_artificial_intelligence>.

[2] MILLINGTON, Ian. *Artificial Intelligence for games*. San Francisco : Morgan Kaufmann Publishers, 2006. 895 s. ISBN-10: 978-0-12-497782-2, ISBN-13: 978-0-12-497782-2.

[3] *AI game programming wisdom*. Steve Rabin. Hingham : CHARLES RIVER MEDIA, 2002. 672 s. ISBN 1-58450-077-8.

[4] BUCKLAND, Mat. *Programming gameAI by example*. Plano, Texas : Wordware Publishing, 2005. 521 s. ISBN 1-55622-078-2

[5] BOURG, David M.; SEEMAN, Glenn. *AI for game developers*. [s.l.] : O'Reilly, 2004. 400 s. ISBN 0-596-00555-5.

[6] BURO, Michael; FURTAK, Timothy. *ON THE DEVELOPMENT OF A FREE RTS GAME ENGINE* [online]. Montreal : [s.n.], 2005 [cit. 2011-03-21]. Dostupné z WWW: <http://skatgame.net/mburo/ps/orts05.pdf>.

[7] BURO, Michael; FURTAK, Timothy. *RTS Games and Real–Time AI Research* [online]. Arlington : [s.n.], 2004 [cit. 2011-03-21]. Dostupné z WWW: <http://skatgame.net/mburo/ps/BRIMS-04.pdf>.

[8] CHUNG, Michael; BURO, Michael; SCHAEFFER, Jonathan. Monte Carlo Planning in RTS Games [online]. Edmonton (Alberta) : Department of Computing Science, University of Alberta, 2005 [cit. 2011-05-13]. Dostupné z WWW: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.7452&rep=rep1&type=pdf>.

Další informace:

[a] Churchova-Turingova teze. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 13. 1. 2006, last modified on 28. 2. 2011 [cit. 2011-03-06]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Churchova-Turingova_teze>.

[b] *The Church-Turing Thesis (Stanford Encyclopedia of Philosophy)* [online]. c2002 [cit. 2011-03-06]. The Church-Turing Thesis. Dostupné z WWW: <http://plato.stanford.edu/entries/church-turing/>.

[c] *Dartmouth Artificial Intelligence (AI) Conference* [online]. c1996-2010 [cit. 2011-03-06]. Dartmouth Artificial Intelligence (AI) Conference. Dostupné z WWW: <http://www.livinginternet.com/i/ii_ai.htm>.

[d] Intelligent agent. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 20 September 2005, last modified on 23 February 2011 [cit. 2011-03-06]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Intelligent_agent>.

[e] Decision theory. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 29 January 2004, last modified on 4 March 2011 [cit. 2011-03-06]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Decision_theory>.

[f] *Skriptovací jazyk. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2. 6. 2007, last modified on 16. 12. 2010 [cit. 2011-03-20]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Skriptovací_jazyk>.*

[g] Game Production Services: Full Spectrum Command [online]. c2003-2008 [cit. 2011-05-13]. Full Spectrum Command. Dostupné z WWW: <http://www.gameprodsvcs.com/project_fsc>.

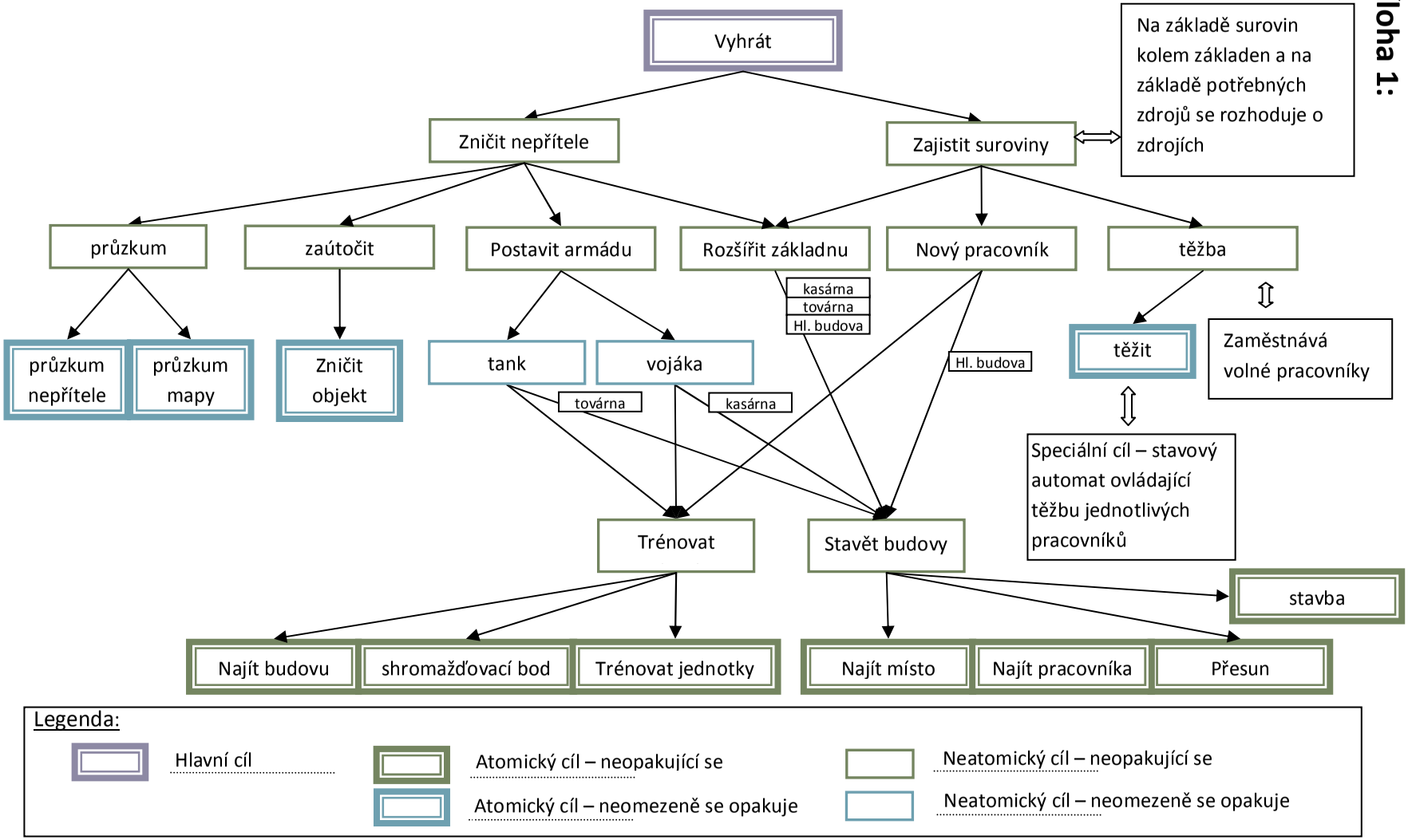
[h] Temporal difference learning. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 25 November 2004 , last modified on 5 April 2011 [cit. 2011-05-13]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Temporal_difference_learning>.

[i] Entropie. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 8. 9. 2005, last modified on 14. 3. 2011 [cit. 2011-05-13]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Entropie>.

Citace generovány pomocí portálu:

www.citace.com – „Portál Citace.com je věnován citacím a citování. Zaměřuje se na citování dokumentů dle normy ČSN ISO 690 a ČSN ISO 690-2."

Příloha 1:


Vyhrát
Zničit nepřítele
Zajistit suroviny
Na základě surovin kolem základen a na základě potřebných zdrojů se rozhoduje o zdrojích
průzkum
zaútočit
Postavit armádu
Rozšířit základnu
Nový pracovník
těžba
kasárna
továrna
Hl. budova
průzkum nepřítele
průzkum mapy
Zničit objekt
tank
vojáka
továrna
kasárna
Hl. budova
těžit
Zaměstnává volné pracovníky
Speciální cíl – stavový automat ovládající těžbu jednotlivých pracovníků
Trénovat
Stavět budovy
stavba
Najít budovu
shromažďovací bod
Trénovat jednotky
Najít místo
Najít pracovníka
Přesun
Legenda:
Hlavní cíl
Atomický cíl – neopakující se
Atomický cíl – neomezeně se opakuje
Neatomický cíl – neopakující se
Neatomický cíl – neomezeně se opakuje