



Univerzitet
Singidunum

УНИВЕРЗИТЕТ СИНГИДУНУМ
БЕОГРАД
ДЕПАРТМАН ПОСЛЕДИПЛОМСКИХ СТУДИЈА

МАСТЕР РАД

**МОДЕЛ ВЕШТАЧКЕ ИНТЕЛИГЕНЦИЈЕ
КРОЗ ТЕОРИЈУ ИГАРА**

МЕНТОР
Проф. др Младен Веиновић

КАНДИДАТ
Јелена Каљевић, 112/2007

БЕОГРАД, фебруар 2010. године

Сажетак

У овом раду биће представљена једна од најранијих примена вештачке интелигенције у компјутерским играма - симулиране верзије игара на табли. AI (Artificial Intelligence- вештачка интелигенција) технике за игре на табли се доста разликују од оних које се користе у играма у реалном времену, па тамо имају ограничену примену. Понекад се користе за одлуке на стратегијском нивоу, правећи дугорочне одлуке у ратним играма.

Основни алгоритми се могу користити за било какве игре на табли. Посматраћемо minimax фамилију алгоритама, најпопуларнију AI технику за игре на табли. Од скора се појавила и нова фамилија алгоритама са бољим перформансама: memory-enhanced test driver (MTD). Обе фамилије се заснивају на претраживању стабла и захтевају посебну репрезентацију стања у игри у виду стабла. На крају ћемо размотрити употребну вредност ових техника у сложенијим потезним стратегијама (turn-based strategy).

Биће приказан процес доношења одлука који је само један део структуре AI-а за игру. Многе игре користе једноставне механизме за одлучивање: коначне аутомате (state machines) и стабла одлучивања (decision trees). Системи засновани на правилима су ређи (rule-based systems). У последње време и сложенији системи за доношење одлука су коришћени, као што је fuzzy логика. Све технике које ће бити приказане се могу користити и на вишем, стратешком и тактичком нивоу.

Кључне речи: Вештачка интелигенција, Теорија игара, Принцип минимаксовања, Minimax алгоритам, Алфа – Бета одсецање, Negamax, Negascout, Табеле транспозиције и меморија, MTD алгоритам, Потезне стратегије, Доношење одлука, Стабла одлучивања, Коначни аутомати, Fuzzy логика, Понашање на основу циља, Продукциони системи, Blackboard архитектура, Скриптовање

Abstract

In this paper one of the earliest applications of artificial intelligence in computer games – simulated versions of board games will be represented. AI (Artificial Intelligence) techniques for board games significantly differentiate from the ones used in real-time games, and therefore have limited application. Sometimes they are used for decisions on strategic level, making long-term decisions in war games.

Basic algorithms can be used for any kind of board games. Minimax family of algorithms, the most popular AI technique for board games, will be researched. Recently, a new family of algorithms with better performances has appeared: memory-enhanced test driver (MTD). Both families are based on stem research and require special game situation stem representation. Finally, applicable value of these techniques in more complex turn-based strategies will be researched.

The process of decision-making, which is only a part of AI game structure, will be represented. Numerous games use simple decision-making mechanisms: state machines and decision trees. Systems based on rules (rule-based systems) are less common. Recently, more complex decision-making systems have been used, such as fuzzy logic. All techniques which will be presented can also be used on a higher, strategic and tactics levels.

Key words: Artificial Intelligence, Game theory, Minimax principle, Alpha-Beta severance, Negamax, Negascout, Transposition and memory tables, Memory-enhanced test driver (MTD) algorithm, Turn-based strategy, Decision-making, Decision trees, State machines, Fuzzy logics, Goal-oriented behaviour, Rule-based systems, Blackboard architecture, Scripting

САДРЖАЈ

1. Увод	4
1.2. Модел вештачке интелигенције	4
1.2.1. Кретање	5
1.2.2. Доношење одлука.....	5
1.2.3. Стратегија.....	5
2. Игре на табли	9
2.1 Теорија игара.....	10
2.1.1. Типови игара	10
2.1.2. Стабло игре.....	11
2.2. Принцип минимаксовања	14
2.2.1. Статичка функција процене.....	14
2.2.2. MINIMAXING	16
2.3. Minimax алгоритам	18
2.4. Алфа – Бета одсецање.....	20
2.5. Minimax алгоритам са алфа – бета одсецањем	23
2.6. Negamax.....	25
2.7. Negamax са алфа – бета одсецањем.....	26
2.8. Прозор претраге код алфа-бета одсецања.....	28
2.9. Negascout.....	29
2.10. Табеле транспозиције и меморија	31
2.11. Memory-enhanced test driver (MTD) алгоритам.....	34
2.12. Књиге отварања и друге секвенце потеза	37
2.13. Даља унапређења	38
2.14. Потезне (turn – based) стратегије	38
3. Доношење одлука.....	41
3.1. Стабла одлучивања (decision trees)	42
3.2. Коначни аутомати (state machines).....	45
3.3. Fuzzy логика.....	48
3.4. Понашање на основу циља (goal-oriented)	54
3.5. Продукциони системи (rule-based).....	58
3.6. Blackboard архитектура	61
3.7. Скриптовање	63
4. Закључак.....	65
Литература	67

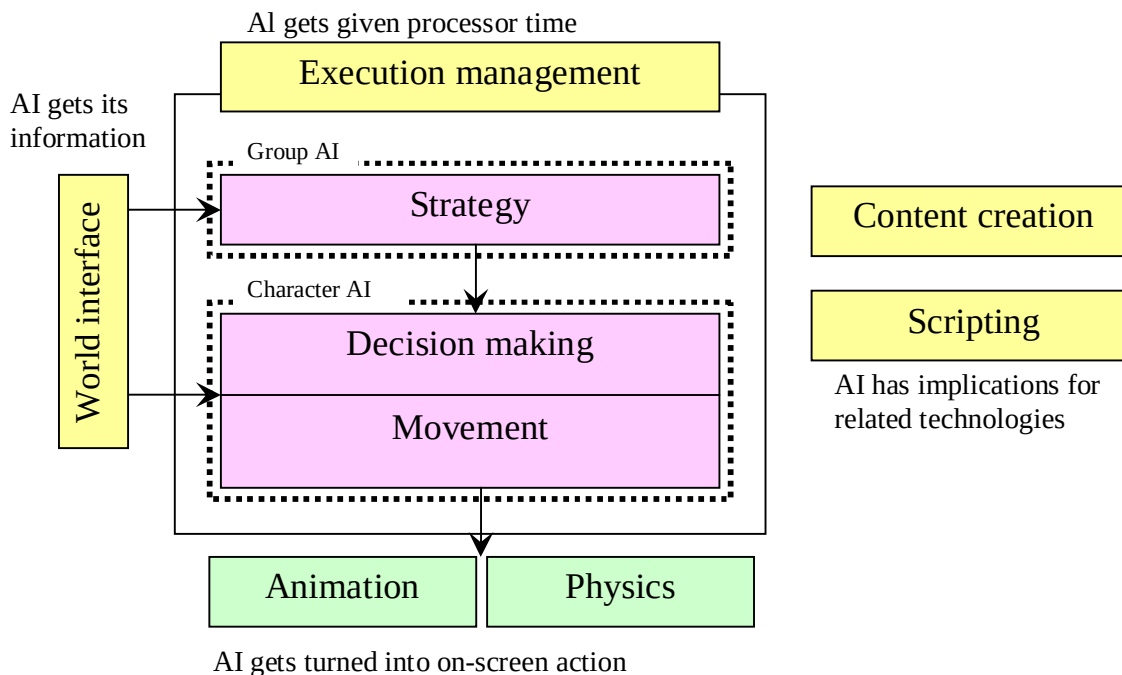
1. Увод

Од својих скромних почетака 1960.-их и 1970.-их година компјутерске игре су значајно напредовале у погледу обима и комплексности. Први шаховски програми су писани још раније 1950-их година. Модерне компјутерске игре су достигле висок ниво реализма, посебно у областима као што су графика и симулација физике. Следећи корак ка још реалнијим играма је у даљем унапређењу вештачке интелигенције. Напредак на пољу процесора и графичких картица омогућава да се више ресурса додели софистициранијој вештачкој интелигенцији.

Циљ овог рада је да направи преглед неких популарнијих техника које се користе у пројектовању вештачке интелигенције за игре.

1.2. Модел вештачке интелигенције

Један од модела који се могу користити за поделу техника дат је на слици 1.1.



Слика 1.1 AI модел 1

У њему је AI (Artificial Intelligence- вештачка интелигенција) подељен на три секције: покретање, доношење одлука и стратегију. Секције за покретање и доношење одлука ће садржати алгоритме који се баве појединачним карактерима у игри, док ће се секција која садржи стратегију бавити алгоритмима за један тим или целу страну.

Нису све три секције потребне за све игре. По овом моделу, у играма на табли као што су ризико и шах потребан је само стратешки ниво, док су ниво за кретање и доношење одлука непотребни, јер карактери у игри не доносе сопствене одлуке. Са друге стране, у неким играма није потребан стратешки ниво јер су карактери у игри у потпуности реактивни, и само одговарају на акције играча.

1.2.1. Кретање

Кретање се односи на алгоритме који претварају одлуке у неки вид покретања. Алгоритми могу бити једноставни као праћење и пресретање играча, да би могао да се нападне када се приђе довољно близу. Могу бити и комплекснији ако је потребно избегавати препреке или вршити навигацију кроз серију соба. Чувар би ако примети играча требао да активира аларм, што захтева да притисне дугме за узбуну, које може бити далеко и може захтевати комплексну навигацију око препрека и кроз собе. Када АИ одлучи да је неки карактер гладан, потребно га је довести до најближег извора хране.

1.2.2. Доношење одлука

Односи се на доношење одлука на нивоу једног карактера. Обично сваки карактер има одређени скуп акција које може да изврши : нападај, стоји мирно, сакриј се, истражуј, патролирај, спавај и слично. Систем за доношење одлука треба да одлучи која је акција од могућих најпогоднија за тај тренутак у игри. На основу одлуке ће се карактери померати и извршити одређене анимације. Правила за одабир понашања могу бити врло једноставна, као понашање животиња које ће да стоје на месту осим ако се играч приближи када ће да побегну од њега. На другом крају, процес одлучивања може бити веома комплексан, где ће противнички тим пробати више различитих стратегија да нападне играча, коришћењем граната, заклона, међусобног покривања и слично. Неким одлукама је потребан и систем за кретање, противнички карактер мора да приђе играчу да би га напао. Другима је потребна само анимација карактера, пошто је одлучено да карактер треба да једе и он је код извора хране само ће се покренути анимација њега како једе. Када АИ одлучи да се истражује нова технологија то се врши само мењањем стања игре, без кретања и анимација.

1.2.3. Стратегија

Доста се може урадити само са кретањем и доношењем одлука, али да би се координисао цео тим неопходан је и стратешки ниво АИ – а. По овом моделу, стратегија се односи на приступ који узима група карактера. У тој категорији су алгоритми који не конторлишу само један карактер, већ утичу на понашање читавог скупа карактера. Сваки карактер у групи ће обично имати сопствене алгоритме за кретање и доношење одлука, али ће то бити под утицајем стратегије целе групе. Ако играча напада група непријатеља, покушаваће да га опколе, један од њих може да протрчи поред играча да би могао да га нападне са бока уз подршку осталих из групе. У новим играма које покушавају да симулирају војне операције овакав тимски АИ постаје све бољи.

АИ алгоритми су само део приче. Да би се направио АИ који ради у игри потребна је додатна инфраструктура. Та инфраструктура се грубо може поделити на општи механизам за менаџмент АИ понашања (који одређује када се које понашање извршава и слично) и део који садржи интерфејс ка свету који служи за доношење информација до АИ-а.

Захтеви за кретање се претварају у акције помоћу анимација, или симулације физике. Вештачкој интелигенцији су потребне информације од игре да би могла да донесе смислене одлуке. То се понекад назива перцепцијом и представља шта карактер зна о свету око себе. Обично подразумева интерфејсе између света у којем се одвија игра и вештачке интелигенције.

Потребан је и одређени менаџмент да би AI добио неопходне ресурсе, односно, процесор и меморију.

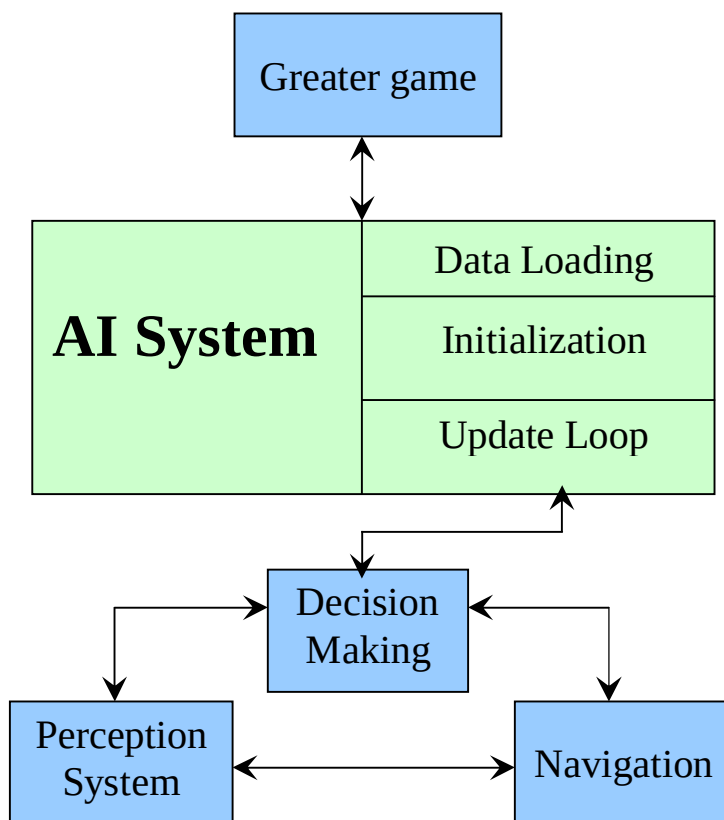
Модел са слике 1.1 је модел базиран на агентима. У овом контексту, циљ вештачке интелигенције базиране на агентима је у прављењу аутономних карактера који скупљају информације од света у којем је игра, одлучују на основу тих информација шта треба да раде и врше те акције.

Овај приступ се може посматрати као пројектовање од дна ка врху. Почиње се са појединачним карактерима и имплементира се преко AI-а њихово понашање. Свеукупно понашање света зависи од понашања појединачних карактера и њихове интеракције. По овом моделу, прва два нивоа, кретање и доношење одлука се односе на AI за појединачног агента.

Другачији приступ, односно, приступ без агената се може сматрати пројектовањем од горе ка доле. AI покушава да одреди како све треба да се понаша од вишег нивоа ка нижем и покушава да направи један систем који ће да симулира све. У игри која симулира један град на нивоу саобраћаја и пешака може да на основу доба дана, дела града и дана у недељи одреди какав саобраћај и колико пешака генерише у играчевом видеокругу на основу саобраћајног протока који се одређује на вишем нивоу.

Углавном се због флексибилности користе мешане технике. Карактери у игри су обично агенти, док се користе и технике које су део приступа од горе ка доле.

Модел AI-а са слике 1.1 је само један од могућих модела, па је на слици 1.2 представљен још један модел.



Слика 1.2 AI модел 2

У овом моделу AI је подељен на секцију за доношење одлука, навигацију и систем за перцепцију уз везе са остатком игре.

Систем за доношење одлука, као и у претходном моделу, има за циљ да на основу добијених информација и знања о свету доноси потребне одлуке. У самој игри то значи да противник кога контролише AI скупља информације о свету и прави разумне одлуке као одговор. Дакле, AI је одређен типом информација које може да добије од света и скупом акција које може да обави. У овом моделу, систем за доношење одлука је задужен и за појединачне карактере и за групне одлуке. Одлуке се могу поделити на стратешке и на тактичке. Стратешка решења су уобичајено дугорочна, решења на вишем нивоу која захтевају више акција да би се спровела. Тактичка решења су обично краткорочнија која захтевају неку физичку акцију или вештину да би се спровела. Према овој подели, јурење играча по мапи би била стратешка одлука, која садржи проналажење играча, прилазак играчу и када се довољно приђе нападање играча. У борби би избегавање напада играча била тактичка одлука. Многе игре захтевају и стратешке и тактичке одлуке, па се користи ова подела да би се проблем поделио и да би се могле користити различите технике за њихово решавање.

Систем за навигацију се обично своди на налажење пута између тачке А и тачке Б уз неке критеријуме и избегавање препрека. Ради реалности модерне игре подразумевају комплексна окружења са различитим теренима, препрекама, објектима који се могу померати и слично. Захваљујући роботизици где је навигација у реалним окружењима један од основних проблема постоје развијени алгоритми за навигацију. Навигација обично садржи део за налажење пута и део за избегавање препрека. У играма се мапа по којој се крећемо може представити помоћу мреже хехова (тада мапа личи на пчелиње саће) или помоћу квадрата. У првом случају се из тренутног хеха можемо кретати у 6 смерова, а у другом случају се из текућег квадрата можемо кретати у 8 смерова. Свако поље може имати неке особине, као што су тип терена, постојећа инфраструктура, проходност и слични који утичу на одабир пута. У великом броју случајева се користи A* алгоритам или нека његова варијанта. У комплекснијим тродимензионалним окружењима се могу користити графови који повезују тачке које се налазе на местима које је поставио дизајнер нивоа, и које су повезане са суседним тачкама. Могуће је и аутоматско генерисање графа повезаности, ако се подлога по којој је могуће кретање састоји од полигона. Чворови графа могу бити центри тих полигона, а повезаност чворова се одређује суседношћу полигона. Онда се на таквом графу искористи A* алгоритам. Користе се и комбиновани системи.

Систем за перцепцију се може описати као скуп стимуланса у игри на које желимо да имамо реакцију елемената који су у игри. Те информације могу бити једноставне као локација играча, до пажљивог записа јединица, њиховог типа, времена опажаја, последње локације које је противничка страна детектовала у некој стратегији.

Различити типови опажаја се не морају истом фреквенцијом освежавати. Освежавање неких опажаја би можда било сувише временски скупо да се стално обавља, док је неке друге опажаје потребно освежавати само са времена на време јер нису критични за доношење одлука.

Време реакције је пауза док AI не прихвати промену у окружењу. Са временом паузе која се задаје на насумичан начин на основу нивоа тежине и вештине тог карактера глобално понашање ће деловати хуманије и фер.

Могу постојати минималне и максималне границе интензитета опажаја на које ће AI реаговати. Тако се могу симулирати карактери који слабије чују, смањењем звучног

опсега на који ће да реагују, или повећањем визуелног опсега на који ће да реагује неки карактер са надпросечним видом. Такође, опсези се могу мењати у складу са акцијама у игри. Ако је ноћ или ако је магла, видљивост ће бити смањена као пример утицаја спољашњих фактора, или ако неки карактер баци заслепљујућу гранату сви карактери у близини ће имати привремено смањен и звучни и визуелни опсег као пример утицаја акција на опажаје.

У наставку овог рада ће се користити AI модел са слике 1.1 и помоћу њега ће се илустровати где одређени скупови техника припадају и које су технике алтернатива једне другима. Многе технике могу испуњавати улоге у различитим категоријама. Pathfinding (налажење путева) може бити и техника за кретање и за доношење одлука. Слично се и технике које анализирају претње и могућности у окружењу могу користити да се донесе одлука на нивоу једног карактера или да се направи стратегија на нивоу целокупног тима.

Да би се сматрале реалним, одлуке и акције које спроводи компјутерски контролисан противик би требале да буду хумане. То подразумева вештачку интелигенцију која прави грешке, као и људи. Наравно, ово не важи у свим случајевима (Деер Blue суперрачунар који је играо шах и успео да победи шаховског светског шампиона Гарија Каспарова није био дизајниран да прави грешке). Противници који блокирају све нападе, никад не промашују или користе велике речнике у играма где је потребно спајати речи на основу добијених слова не би били забавни и фрустирали би играча. Циљ је у дизајнирању AI-а направити баланс између изазова и забаве. Како се играчи разликују по вештини у одређеној игри једно решење је увођење нивоа тежине. Могу бити статички нивои тежине, које поставља играч пре почетка игре или динамички, који се мењају у току игре на основу вештине играча. Углавном се користи статичко намештање нивоа тежине. Ниво тежине се може огледати у играма које траже добре рефлексе као смањено време одзива противника на већим нивоима тежине, или у питању шаха као дубље претраживање или коришћење комплекснијих алгоритама.

Приликом дизајнирања игре и пројектовања вештачке интелигенције, потребно је пазити да играч не може да користи неке акције или тактике за које противник контролисан вештачком интелигенцијом нема решења иначе ће игра бити сувише лака. На другом крају је ситуација да компјутер користи неке акције на које играч нема одговора, па игра може бити фрустрирајућа.

2. Игре на табли

Једне од најранијих примена вештачке интелигенције у компјутерским играма су биле у симулираним верзијама игара на табли. На западу то је углавном био шах, и у последњих 40 година се види драстичан напредак у квалитету рачунара који играју шах. У исто време, друге игре као што су икс-окс, Connect Four, Reversi(Othello) су истраживане и AI различитих квалитета су биле створене.

Квалитет AI алгоритама се може грубо сврстати у 5 категорија :

- (1) **Решене игре.** Игра је решена ако програм увек може да постигне најбољу позицију (у оквиру игре) у односу на противника, без обзира на његове потезе. У ову категорију спадају игре : Connect Four (Allis,1988), Go-Moku (Allis,1994) и друге.
- (2) **Боље од шампиона.** У ову категорију спадају игре где су програми јачи него светски шампион у тој игри. Овде спадају мице (Checkers). Рачунари су достигли ниво већи од људског још од победе Chinook-а над Tinsley-јем 1994. Такође, то је постигнуто и у Backgammon-у(Tesauro,1994), Othello(Buro,1997) и Scrabble(Sheppard,2002).
- (3) **На нивоу шампиона.** Најпознатији пример је шах. Deep Blue је победио тадашњег шампиона Kasparov-а, међутим нерешени мечеви садашњих најбољих програма Deep Fritz -а и Deep Junior –а против Kramnik-а и Kasparov-а показују да су још увек на нивоу светских шампиона.
- (4) **На нивоу велемајстора.** Ту спадају програми који могу играти на нивоу велемајстора али ипак бивају побеђени од шампиона. Покер спада у ову категорију.
- (5) **На нивоу аматера.** Програми играју на нивоу аматера. Go спада у ову категорију.

Даље истраживање игара из (2) категорије може довести до њиховог решавања. Постоје и такмичења као што су Computer Olympiad, где се такмиче програми у одређеним играма. Такође, јаки програми могу бити одлични тутори људским играчима.

AI технике за игре на табли се доста разликују од оних које се користе у играма у реалном времену, па тамо имају ограничену примену. Понекад се користе за одлуке на стратегијском нивоу, правећи дугорочне одлуке у ратним играма.

Основни алгоритми се могу користити за било какве игра на табли. Посматраћемо minimax фамилију алгоритама, најпопуларнију AI технику за игре на табли. Од скора се појавила и нова фамилија алгоритама са бољим перформансама : memory-enhanced test driver (MTD). Обе фамилије се заснивају на претраживању стабла и захтевају посебну репрезентацију стања у игри у виду стабла.

На крају ћемо размотрити употребну вредност ових техника у сложенијим потезним стратегијама (turn-based strategy).

Време је јако битан фактор у играма јер уобичајено имају временско ограничење. Због временског ограничења игре пенализују неефикасност. Примера ради, имплементација A* алгоритма у претраживању која је 10% спорија ће захтевати нешто више времена да се комплетира, док 10% спорији шаховски програм, у партији где је ограничено време размишљања има већу шансу да ће да изгуби. Тако да је истраживање на пољу игара изродило интресантне идеје како ефикасно користити расположиво време, када није могуће доћи до оптималних одлука. То питање је битно због комплексности проблема које покушавамо да решимо вештачком интелигенцијом.

2.1 Теорија игара

Теорија игара је математичка дисциплина која се бави истраживањем абстрактних, идеализованих игара. Посматра свако окружење са више агената као игру, под условом да је међусобни утицај агената значајан, без обзира да ли су агенти кооперативни или противнички настројени.

Има малу употребну вредност у играма у реалном времену. Терминологија у потезним (turn-based) играма је настала од ње. Увешћемо довољно теорије да би се могли схватити концепти, без много комплекснијих упуштања у математику.

Пре даљих разматрања потребно је рећи нешто о терминологији која се користи. Потез једног играча се обично зове **ply**, док је једна рунда потеза свих осталих играча **move**. Ова терминологија потиче из шаха, где се један **move** састоји од потеза сваког играча. Пошто су многи алгоритми за потезне игре засновани на алгоритмима за шах терминологија је остала. Међутим, у другим играма терминологија је другачија и тамо се **move** третира као потез једног играча.

2.1.1. Типови игара

Према теорији игара игре се класификују на основу броја играча, типа циљева које ти играчи имају и информација које имају о игри.

Број играча

Игре на табли које су инспирисале потезне (turn-based) AI алгоритме углавном имају 2 играча. Тако да су многи ограничени на два играча у својој основној форми. Могу се прилагодити играма са више играча, али се углавном презентују у својој основној форми за два играча. Исто, многе оптимизације тих алгоритама претпостављају да су у питању само два играча. Док, су основни алгоритми адаптивни, то се не може рећи за многе оптимизације.

Циљ игре

У многим стратешким играма циљ је победити. Играч побеђује ако сви противници изгубе. То је zero-sum игра. Она описује ситуацију у којој је добитак или губитак сваког учесника избалансиран губицима или добицима осталих учесника. Другим речима, ако неко победи остали су изгубили. Ако играч добије 1 поен за победу, то је исто као да је добио -1 поен за губитак. Ово није случај у казино играма, ако се казино не посматра као играч, где би сви могли да добију или изгубе.

У zero-sum играма није битно да ли покушаваш да победиш или покушаваш да натераш противника да изгуби, јер је исход исти. За игру која није zero-sum, где сви могу победити или изгубити, је пожељније фокусирати се на сопствену победу, него на пораз противника.

У играма са више од два играча ситуација се компликује. Чак и у zero-sum игри, није увек најбоља стратегија натерати сваког играча на пораз. Може бити боље искористити слабије играче да би се заједничким снагама елиминисао најјачи играч. Потом би се могли они елиминисати.

Информације

У играма као што је шах или го оба играча знају све шта се може знати о тренутном стању игре. Знају шта ће бити резултат сваког противничог потеза и које ће бити опције за наредне потезе. Све је ово познато од почетка партије. Овај тип игара се назива играма са савршеним информацијама. Иако се не зна који ће потез начинити противник, комплетно знање о могућим потезима које он може начинити и њиховим ефектима је доступно.

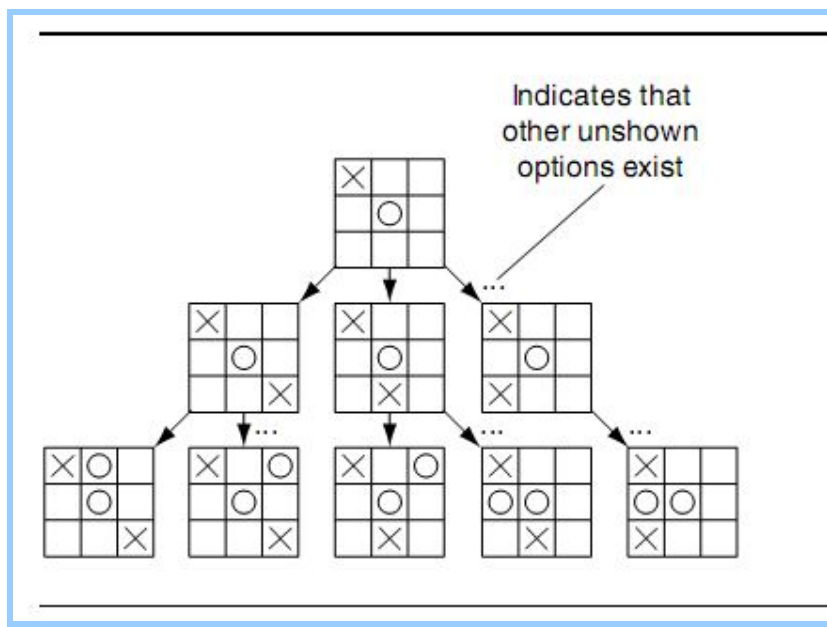
У играма као што је Backgammon постоји и случајни елемент. Не могу се унапред знати дозвољени потези, јер бацање коцкица одређује могуће потезе. Слично, не могу се унапред знати потези противника, јер се не може предвидети противничко бацање коцкица са потпуном сигурношћу. Овај тип игара се зове игре са несавршеним информацијама.

Многе потезне стратегије су игре са несавршеним информацијама, јер постоје насумични фактори (насумичност у борби и слично). Игре са савршеним информацијама су лакше за анализу. Доста алгоритама и техника за потезну вештачку интелигенцију подразумева доступност савршених информација. Обично се могу прилагодити али уз слабије резултате.

Најпознатији и најразвијенији алгоритми за потезне игре најбоље раде за zero-sum игре, за два играча са савршеним информацијама. Такви алгоритми ће бити представљени у наставку, међутим, могу бити прилагођени и другим типовима игара.

2.1.2. Стабло игре

Било која потезна игра се може представити стаблом игре. На слици 2.1 је део стабла игре за игру икс-окс. Сваки чвор у стаблу представља позицију у игри, а свако гранање представља један могући потез. Потези пресликавају једну позицију на табли у другу.

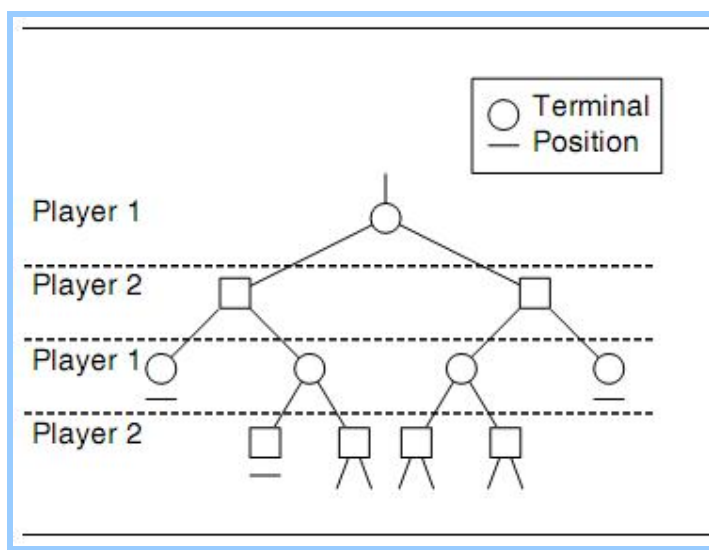


Слика 2.1. Део стабла игре за игру икс-окс

Пошто играчи играју један после другог, сваком играчу одговара неки ниво у стаблу и то ако је првом играчу одговарао N -ти ниво другом играчу ће одговарати $N+1$ -ви ниво, уколико је игра са само два играча. Пошто је игра потезна до промене на табли долази само када се одигра потез.

Број гранања од сваке позиције на табли, односно од сваког чвора, је једнак могућем броју потеза које тај играч може да одигра. У икс-окс-у тај број у првом потезу је 9, па је у другом 8, у трећем 7 и тако даље. У многим играма може постојати и стотина, чак и више, могућих потеза из текућег чвора стабла.

Неке позиције на табли немају више могућих потеза, Зову се терминалне позиције, и означавају крај партије. За сваку терминалну позицију даје се резултат сваком играчу. Тај резултат може бити само једноставно +1 за победника и -1 за губитника, или тај резултат може рефлектовати квалитет победничке позиције. Нерешене позиције би доносиле 0. У zero-sum играма би коначни збир резултата свих играча био 0. У играма које нису zero-sum резултати би рефлектовали квалитете завршних позиција свих играча. Обично се користе апстрактна стабла, без чворова као приказа стања на табли, али са финалним резултатима.



Слика 2.2. Абстрактно стабло игре

Фактор гранања и дубина стабла

Број гранања од сваког чвора у стаблу игре се зове фактором гранања, и добар је индикатор колико ће сама игра бити компликована компјутеру за играње.

Другачије игре такође имају другачију дубину стабла. Односно, другачији максимални број потеза до терминалне позиције. У икс-окс-у сваки играч додаје сопствени симбол и пошто је на почетку било 9 места на табли максимални број потеза је 9. Исто се догађа у Othello-у, који се игра на табли димензија 8 са 8. Пошто су на почетку присутне 4 фигуре, могуће је максимално одиграти 60 потеза. Игре као шах могу имати практично бесконачан број потеза (правило у шаху на турнирима лимитира ово). Стабло игре за такву игру би било изузетно дубоко, чак и за релативно мали фактор гранања.

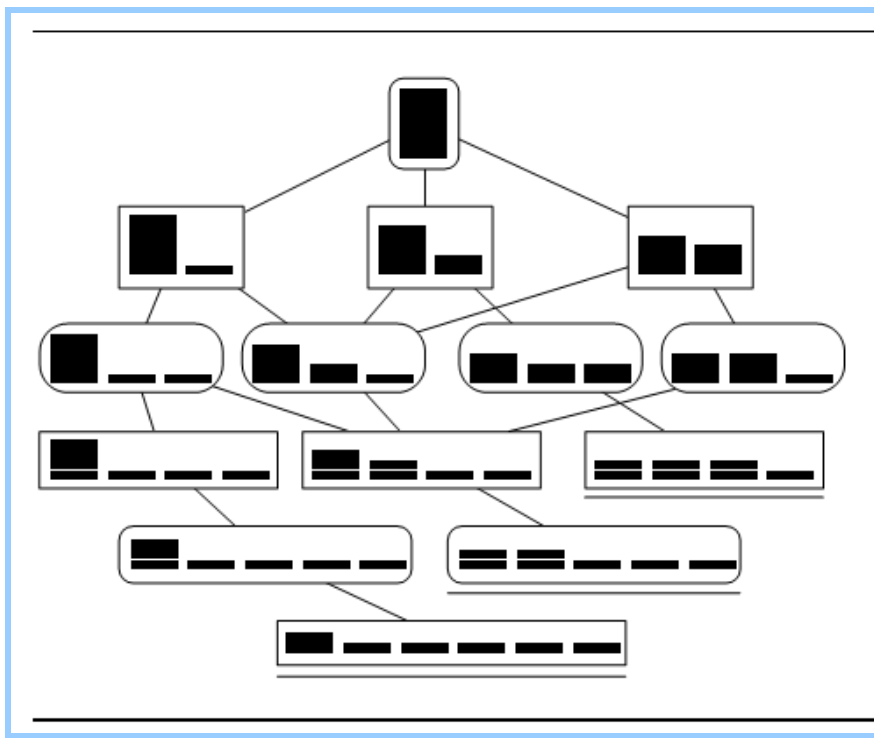
Компјутерима је лакше да играју игре са малим фактором гранања а са дубоким стаблима, него игре са плитким стаблима а са великим фактором гранања.

Транспозиција

У многим играма је могуће доћи до истог стања на табли неколико пута у току партије. У другим играма је могуће доћи до истог стања на табли различитим комбинацијама потеза.

Исто стање на табли до којег се дошло различитим комбинацијама потеза се назива транспозицијом. То значи да стабло игре за многе игре и није стабло, јер се гранања стапају.

Split-Nim, варијација старе Кинеске игре Nim, почиње са једном гомилом новчића. Сваког потеза је потребно да играч подели једну гомилу новчића у две са неједнаким бројем новчића у њима. Последњи играч који може да направи потез побеђује. На слици 2.3 је дато комплетно стабло игре за варијанту Split-Nim игре са 7 новчића у почетној гомили. На слици се види велики број стапајућих гранања.



Слика 2.3. Стабло игре за варијанту Split-Nim игре са 7 новчића у почетној гомили

Алгоритми базирани на Minimax-у су дизајнирани да раде са нормалним стаблима игре. Могу бити прилагођени да раде са стаблима са стапајућим гранањима, али ће дуплицирати посао за свако стапајуће гранање. Могу бити проширени са табелама транзиција да би избегли дупликацију посла када се стопе гранања. Друга група алгоритама memory-enhanced test driver (MTD) су дизајнирани за игре са транспозицијом.

Дакле, формално се игре могу представити као проблеми претраге са следећим компонентама :

- (1) Иницијално стање, које укључује почетно стање на табли и има информацију о играчу који је следећи на потезу.
- (2) Функцију која враћа легалне потезе у виду листе која садржи парове (потез, стање), који означавају легални потез и резултујуће стање.
- (3) Тест терминалности, који означава када је дошло до краја партије.
- (4) Функцију која враћа нумеричку вредност за терминална стања. У шаху би исход био победа, пораз или нерешена партија, којима одговара +1, -

1 и 0 респективно. Неке игре имају шире опсеге могућих исхода. Један такав пример је Backgammon, где би опсег био од +192 до -192.

Прве две компоненте (иницијално стање и легални потези са резултујућим стањем) чине стабло игре.

2.2. Принцип минимаксовања

Компјутер игра потезне игре разматрајући могуће акције у сопственом потезу. Да би одабрао један потез, мора да зна који су потези бољи од других. Ово се постиже хеуристиком која се зове статичка функција процене. Због брзине се користи хеуристика, а статичка је јер разматра само текуће стање без разматрања будућих стања.

2.2.1. Статичка функција процене

У потезној игри статичка функција процене ће да оцени позицију на основу тренутног стања на табли из перспективе једног играча. Ако је позиција терминална, онда ће та процена бити за финално стање у партији. У шаху ако је бели матирао црног, за победу ће добити +1, док ће црни добити -1. Дакле, у крајњој позицији ће један имати позитиван скор, а други негативан, ако је zero-sum игра.

Теже је дати процену у половини партије. Та процењена вредност би требала бити индикатор колико је вероватно да је из тог стања могуће победити. Тако да ако у тој позицији један играч има огромну предност, нумеричка процена треба да буде блиска нумеричкој вредности позиције у којој је играч са предношћу победио.

У Othello-у онај играч са највише фигура на табли у терминалној позицији побеђује. У току партије је стратешки боље због природе игре имати мање фигура на табли, јер то даје већу иницијативу. Да би се направила добра статичка функција процене потребно је узети у обзир начин на који се игра та игра. Алгоритми који ће се у наставку разматрати неће узимати у обзир стратешке аспекте. Све потребне стратешке информације ће морати да буду садржане у статичкој функцији процене, на основу које ћемо знати које су боље позиције.

Распон вредности статичке функције процене

У принципу, статичка функција процене може да врати било који број. У многим имплементацијама враћа целобројни број. Неки од алгоритама се ослањају на чињеницу да је у питању целобројни број. Иначе, обично је целобројна аритметика бржа од аритметике у покретном зарезу.

Распон могућих вредности није исувише битан. Неки алгоритми раде боље са мањим распонима (од -100 до +100), док неки раде боље са већим распонима. Велики део рада на AI алгоритмима потиче од рада на шаховским програмима. Скала од +1000 до -1000 је уобичајена за шах. Она омогућава финије стратешко одлучивање.

Такође, распон вредности које се добијају треба да буде мањи од вредности за победу или пораз. Ако би процена давала +1000 за позицију близу победи а само +100 за победу, програм би покушавао да буде близу победе али да не победи, јер ће му то деловати привлачније.

Комбиновање функција процене

Може постојати више различитих механизма за процењивање који раде у исто време. Сваки разматра одређени стратешки аспект игре. Један алгоритам може

разматрати број јединица које свака страна поседује, други распоред снага, трећи зоне опасности и слично. Пожељно би било искомбиновати све појединачне процене да би добили реалнију глобалну процену. То се може постићи једноставним множењем одговарајуће процене са тежинском вредношћу и сабирањем свих добијених вредности.

Тако да је принцип за конструисање статичке функције процене преко суме усредњених вредности (weighted sum) фактора које узимамо у разматрање за процену текуће позиције. У шаху би статичка функција процене могла бити

$$c_1 * \text{material} + c_2 * \text{mobility} + c_3 * \text{king safety} + c_4 * \text{center control} + \dots$$

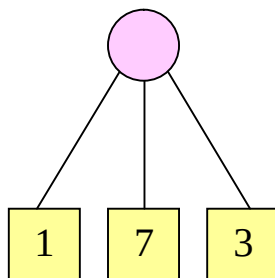
где би за почетнички ниво AI-а само материјал био битан фактор. Свакој фигури би била додељена одговарајућа нумеричка вредност на основу типа фигуре (за противника би се користила иста вредност само супротног знака) и стање на табли би се добило као сума вредности свих фигура на њој. Неке функције могу бити окренуте више ка материјалном, па као резултат алгоритми који их користе би више ценили тактику него стратегију.

Могуће је у току игре применити једноставан алгоритам за учење, који би на основу искуства мењао тежинске вредности. У различитим фазама игре је могуће користити другачије комбинације процена, на тај начин потенцирамо одређене аспекте игре у одређено време.

Комбиновањем стварамо комплекснију анализу ситуације на основу појединачних једноставнијих анализа.

Једноставан одабир потеза

Са добром статичком функцијом процене, компјутер би одабрао потез процењујући позиције које би добио на основу свих легалних потеза и одабрао потез који води до позиције са најбољом проценом. На слици 2.4 је дата позиција са три могућа потеза и проценама резултујуће позиције. Одабран је други потез због највеће вредности функције процене.



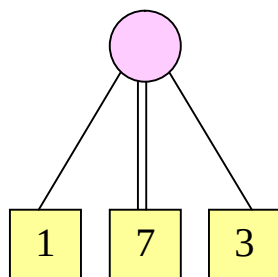
Слика 2.4. Позиција са три могућа потеза и проценама резултујуће позиције

Ако би постојала савршена функција процене, AI би требао само да на основу резултата функције процене бира најбољи потез. Међутим, савршена функција процене не постоји, чак и најбоље функције процене показују лоше резултате ако се користе искључиво на овај начин. AI ће морати да претражује на основу могућих потеза других играча, па на основу одговора на те потезе и тако даље. Исти процес користе и људи када разматрају позиције више потеза унапред, али људи имају бољу интуитивну представу ко побеђује, за разлику од хеуристика, па је компјутерима потребно процењивање на основу више потеза унапред од људи.

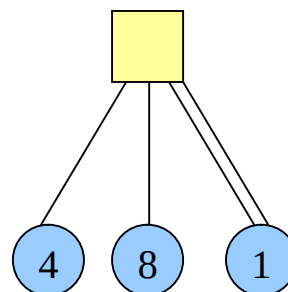
Најпопуларнији алгоритам за претраживање у играма је minimax.

2.2.2. MINIMAXING

Ако је мој потез, вероватно ћу одабрати потез који води до добре позиције. Може се предпоставити да ћу одабрати потез који води до најбоље позиције на располагању. Другим речима, покушаћу да максимизујем сопствени резултат. Илустровано на слици 2.5.



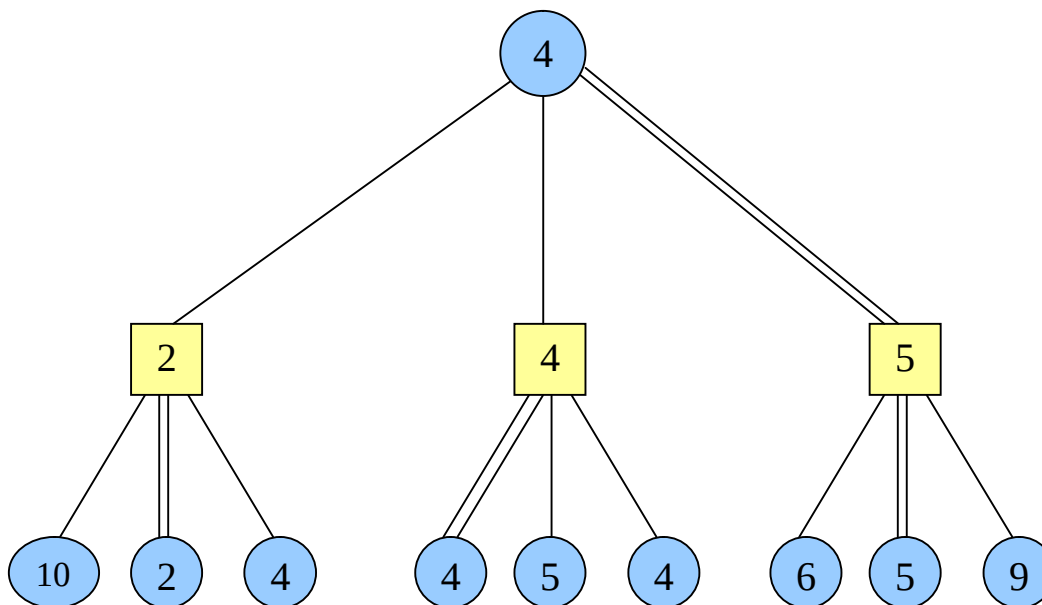
Слика 2.5. Мој потез



Слика 2.6. Противнички потез

Ако је противнички потез, предпоставићу да ће начинити потез који ће ме оставити у најгорој позицији од потеза на располагању. Противник покушава да минимизује мој резултат. Илустровано на слици 2.6.

Када се врши претраживање у циљу налажења противничких одговора на моје потезе, треба имати у виду да је потребно максимизовати сопствени резултат, док ће противник покушавати да га минимизује. Ова транзиција између максимизовања и минимизовања док се претражује стабло игре се назива *minimaxing*. Стабла игре на сликама 2.5 и 2.6 су дубока само један потез, међутим, потребно је разматрати и противничке потезе.



Слика 2.7. Стабло игре дубоко два потеза

Ако направим први потез могу да дођем до позиције која се вреднује са 10 поена, али се предпоставља да ће противник начинити потез који ће довести до позиције вредне 2 поена. Слично, ако начиним потез два се предпоставља да ће противнички потез довести до позиције вредне 4 поена. Трећим потезом противник

бира потез који води до коначне позиције вредне 5 поена, што је и најбољи потез који могу да направим у овом примеру.

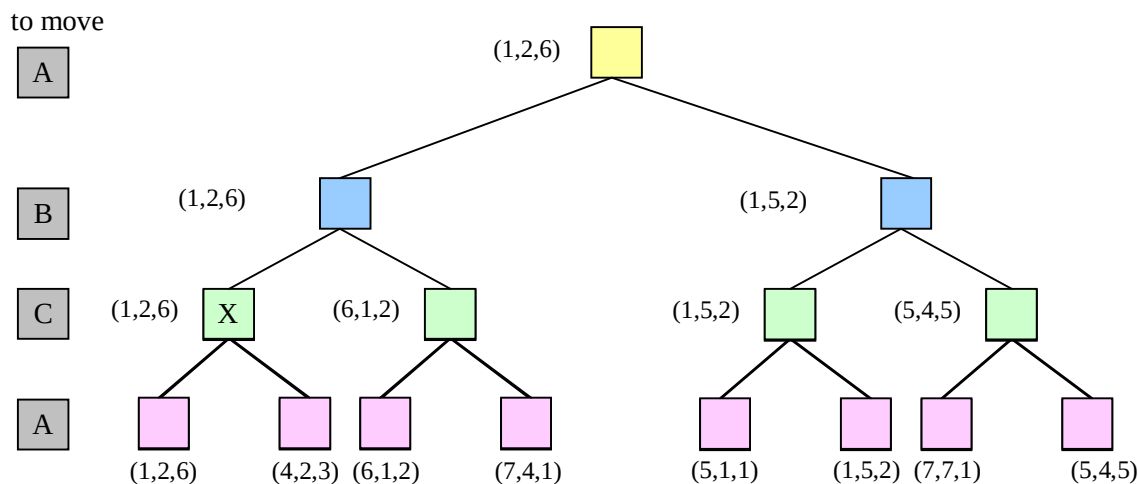
Стартујући од дна стабла игре бирамо процене стања на основу $\min\max$ правила. У сопственом потезу бира се највећа процена, у противничком потезу најмања. После пропагације добијамо процене потеза за текућу позицију, потом једноставно бирамо потез на основу највеће вредности. Овај процес је у суштини оно шта ради $\min\max$ алгоритам.

Да би одредио колико је добар неки потез, претражује стабло игре за одговорима на тај потез, па одговорима на те одговоре све док не дође до терминалних стања или до тражене дубине претраге. Тада се користи статичка функција процене да се добије нумеричка вредност процене те позиције. Потом се на основу тога чији је потез бирају одговарајуће вредности и пропагирају ка врху, односно ка текућем чвору у стаблу игре, да би се добиле процене за сваки од могућих потеза. Онда је могуће одабрати потез, као потез са највећом вредношћу процене. Чак и за претраге које гледају само пар потеза унапред, $\min\max$ даје значајно боље резултате него приступ само са хеуристиком.

Игре са више играча

За случај са три играча А, Б, Ц потребно је за сваког памтити процену. То се може урадити у виду вектора који би био придружен сваком чвору у стаблу игре. У терминалном стању тај вектор ће бити резултат статичке функције процене. У zero-sum играма са два играча довољна је једна вредност која је придружена чвору, јер је вредност за другога играча само супротног знака.

Формирање процена у случају нетерминалних стања ће бити илустрован на примеру на слици 2.8. Вектори одговарају (А, В, С) проценама.



Слика 2.8. Игра са три играча

Разматраће се чвор означен са X на слици. Играч С одлучује шта да ради. На располагању су му два потеза који воде ка терминалним стањима са проценама (1,2,6) и (4,2,3). Пошто је потез играча С, и пошто је 6 веће од 3 одабраће први потез са проценом (1,2,6), па се тај вектор пропагира ка горе. У општем случају, бира се онај вектор чији је одговарајући елемент највећи за играча који је на потезу и чији се резултат максимизује.

У многим играма са више играча постоје и савези, било формални или неформални. Ако су играчи А и В у слабијој позицији од играча С могу направити савез да би заједно напали играча С, јер би их он могао појединачно уништити. Када С ослаби услед заједничког напада савез прва два играча губи на значају. Они играчи који

раскидају договоре се могу сматрати неповерљивим. Потребно је одредити да ли је краткорочни циљ раскидања договора исплативији од поверљивости као дугорочног циља.

У играма које нису zero-sum сарадња може постојати и ако су само два играча. Ако би постојало терминално стање (100, 100) које је најбоље за оба играча, оптимална стратегија за оба је да достигну то стање.

2.3. Minimax алгоритам

Дати алгоритам је рекурзиван. Сваком рекурзијом покушава да израчуна исправну вредност текуће позиције на табли.

То ради тако што разматра сваки могући потез из текуће позиције. За сваки потез рекурзивно израчунава вредност резултујућег стања на табли после тог потеза. Да би се онемогућило бесконачно претраживање (ако је стабло игре врло дубоко) у алгоритму постоји и максимална дубина до које ће се вршити претраживање. Ако је тренутна дубина једнака максималној дозвољеној дубини или смо досли до терминалног стања завршавамо са рекурзијом, позивамо статичку функцију процене и враћамо резултат.

Ако алгоритам разматра позицију у којој је текући играч на потезу онда враћа максималну вредност, иначе, враћа минималну. То је промена између сопственог и потеза противника, односно, између максимизујуће и минимизујуће фазе. Ако је тренутна дубина претраге 0, сачуваће се најбољи потез. То ће уједно бити потез који треба начинити. На слици Алгоритам 2.1 је дат псеудокод Minimax алгоритма.

```

1  def minimax(board, player, maxDepth, currentDepth):
2
3  # Check if we're done recursing
4  if board.isGameOver() or currentDepth == maxDepth:
5      return board.evaluate(player), None
6
7  # Otherwise bubble up values from below
8
9  bestMove = None
10 if board.currentPlayer() == player: bestScore = -INFINITY
11 else: bestScore = INFINITY
12
13 # Go through each move
14 for move in board.getMoves():
15
16     newBoard = board.makeMove(move)
17
18     # Recurse
19     currentScore, currentMove = minimax(newBoard, player,
20                                         maxDepth, currentDepth+1)
21
22 # Update the best score
23 if board.currentPlayer() == player:
24     if currentScore > bestScore:
25         bestScore = currentScore
26         bestMove = move
27 else:
28     if currentScore < bestScore:
29         bestScore = currentScore
30         bestMove = move

```

```

31
32     # Return the score and the best move
33     return bestScore, bestMove

```

Алгоритам 2.1. Minimax алгоритам

Minimax функција ће враћати и најбољи резултат и најбољи потез. Константа INFINITY представља вредност већу од највеће вредности која се може добити на основу статичке функције процене. Слично је и за $-\text{INFINITY}$. Користи се да будемо сигурни да ће се увек наћи најбољи потез, без обзира колико је лош према вредности.

Minimax функција се може позивати из једноставније функције која ће само да врати најбољи потез. Илустровано псеудокодом на слици Алгоритам 2.2.

```

1     def getBestMove(board, player, maxDepth):
2
3     # Get the result of a minimax run and return the move
4     score, move = minimax(board, player, maxDepth, 0)
5     return move

```

Алгоритам 2.2. Помоћна функција за Minimax алгоритам

У Алгоритму 2.1 објекат табла има следеће особине које су дате на слици Алгоритам 2.3. Објекат табла је задужен за одређивање расположивих потеза и за њихову примену.

```

1     class Board:
2     def getMoves()
3     def makeMove(move)
4     def evaluate(player)
5     def currentPlayer()
6     def isGameOver()

```

Алгоритам 2.3. Објекат Табла

Метода getMoves() враћа листу могућих потеза који одговарају потезима који се могу начинити из текуће позиције на табли. Метода makeMove(move) враћа нову таблу која одговара табли у којој је потез move начињен. Метода evaluate(player) је статичка функција процене из перспективе играча player. Метода currentPlayer() враћа играча чији је ред да игра. Тај играч не мора да буде онај чији резултат желимо да максимизујемо. Метода isGameOver() враћа индикатор да ли је позиција у партији терминлана. Ова структура одговара играма за два играча.

Више играча

Минимизација ће се вршити у сваком потезу у којем не игра играч чији резултат желимо да максимизујемо, односно када буду потези свих осталих играча, а максимација ће се вршити када буде потез тог играча. Ово је илустровано помоћу `board.currentPlayer() == player`, где се проверава да ли је питању играч чији резултат максимизујемо. Ако би постојала три играча, корак максимизације ће следити два корака минимизације.

Перформансе

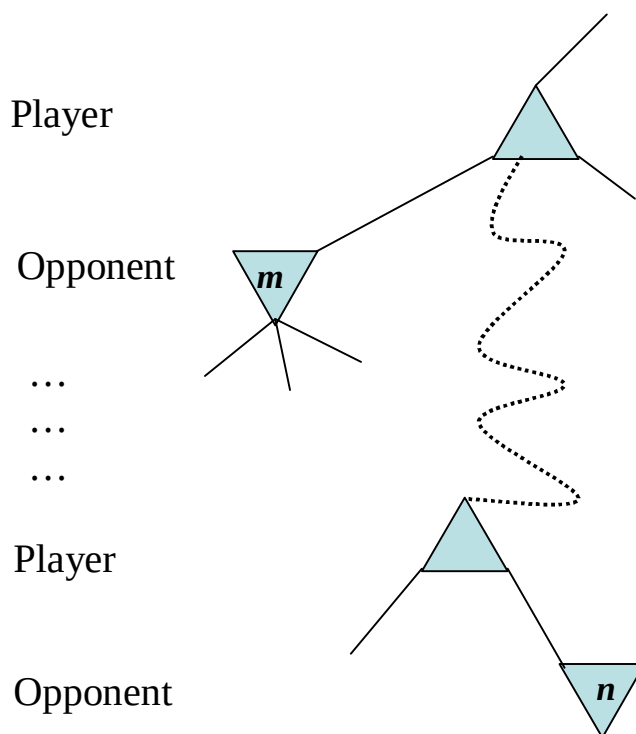
Minimax алгоритам изврши комплетно дубинско истраживање стабла игре. Ако би дубина стабла била m , а број могућих потеза у сваком чвору b , онда је његова

временска комплексност $O(b^m)$. Комплексност је $O(bm)$ у меморији. Цена је врло велика и непрактична за комплексније игре, па ће се размотрити нека унапређења.

2.4. Алфа – Бета одсецање

Проблем са minimax алгоритмом је да је број чворова који треба да се обиђе експоненцијалан по дубини стабла. Нажалост, не може се уклонити експоненцијална зависност, али се може преполовити зависност од дубине стабла. Да би се то урадило потребно је донети исправну minimax одлуку без гледања свих чворова. Делови стабла се одсецањем избацују из даљег разматрања. Техника се зове алфа – бета одсецање и када се примени на стандардно minimax стабло, враћа исти резултат као и minimax, али одсеца гране које сигурно не могу утицати на исход одлуке. Састоји се из алфа и бета одсецања.

Алфа – Бета одсецање се може применити на стабла игре било које дубине, и често се одсецају читава подстабла из разматрања. Основни принцип је приказан на слици 2.9.

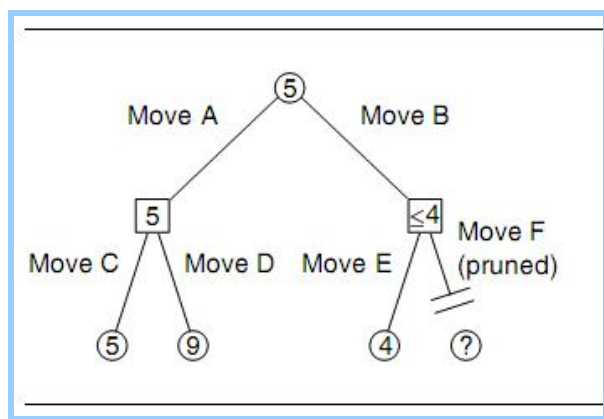


Слика 2.9. Основни принцип Алфа-Бета одсецања

Разматрамо такав чвор n негде у стаблу да играч може да га одабере за разматрање. Ако играч већ има бољи избор m негде изнад у стаблу чвор n никад неће бити достигнут у игри. Када сазнамо довољно о n , истраживањем његових потомака, можемо донети одлуку да неће бити достигнут и може се одсећи.

Алфа одсецање

Да би одсецали стабло на овај начин потребно је да водимо евиденцију о најбољем потезу који можемо да одиграмо. Алфа граница је доња граница резултата који се може постићи. Претрагом се може наћи боља секвенца потеза, али никад нећемо прихватити секвенцу потеза са лошијим резултатом. Вођењем евиденције о алфа вредности, избегавамо разматрање свих потеза у којима би нас противник водио до лошије позиције. На слици 2.10 је дат пример алфа одсецања.



Слика 2.10. Алфа одсецање

Ако играч начини потез А, противник одговара са С дајући резултат 5. Сада алгоритам разматра потез В. Види да је први одговор потез Е који даје резултат 4. Без обзира на вредност процене стања после потеза F противник може одабрати потез Е, тако да је коначни резултат потеза В једнак 4. Чак и без разматрања потеза F играч зна да је то погрешан потез јер може да добије резултат 5 потезом А. У овом случају потез F се одсеца.

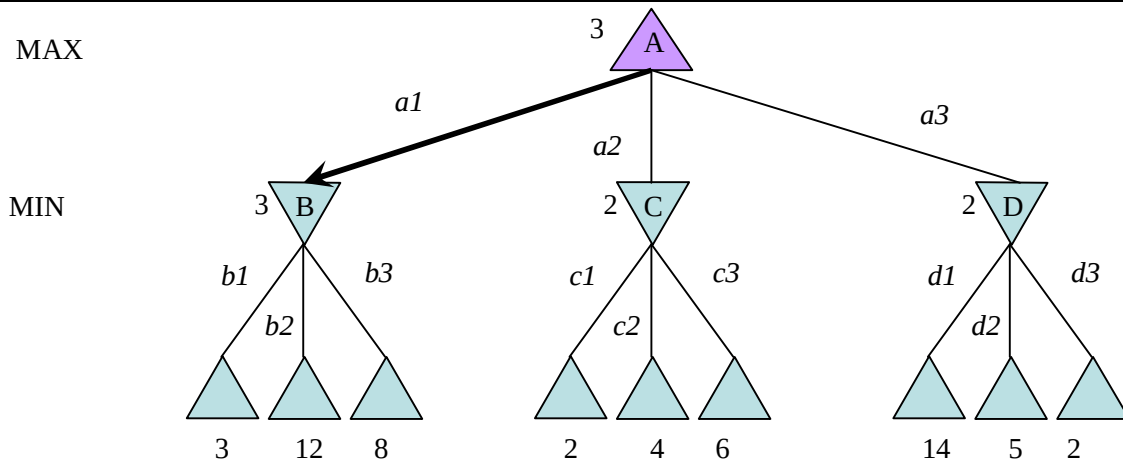
Бета одсецање

Бета одсецање ради на сличан начин. Бета вредност води евиденцију о горњој граници резултата које можемо да постигнемо. Ова вредност се ажурира када се нађе секвенца потеза на коју нас натера противник. Тада знамо да је то горња граница која се може постићи и противник може само још више да нас ограничи. Ако нађемо секвенцу потеза са проценом већом од бета вредности знамо да противник неће дати прилику да се одигра и тај део може да се одсече из разматрања.

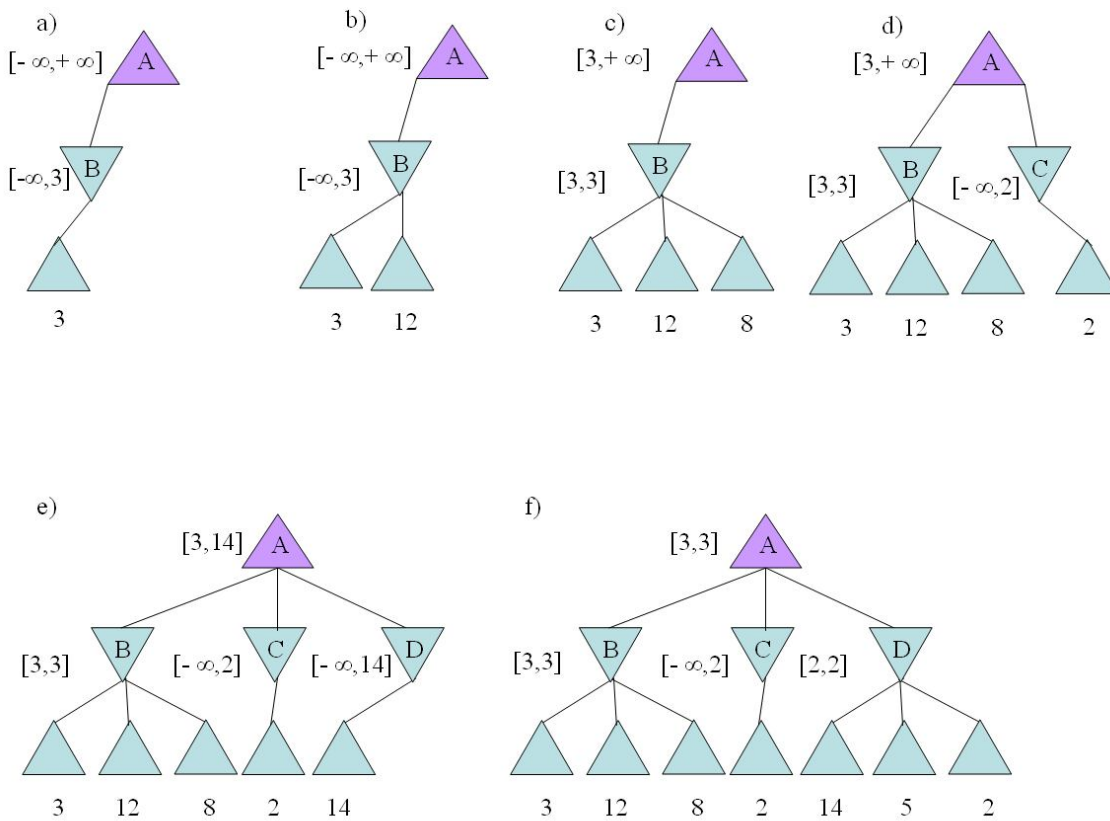
Заједно Алфа и Бета вредности чине интервал могућих вредности резултата који се разматрају. Никад нећемо играти потезе са лошијом проценом од алфа и противник никад неће дозволити да се одигра потез са проценом већом од бета. Коначни потез ће бити ограничен тим вредностима. Како се претражује стабло игре алфа и бета вредности се ажурирају и гране које их не задовољавају се одсецају.

Због промена између минимизовања и максимизовања за сваког играча, само се једна вредности треба проверавати у свакој позицији на табли. Када је противнички ред да игра, резултати се минимизују па се само минимални резултат може променити, па се проверава само алфа. Ако је наш потез, резултати се максимизују, па је потребна провера са бета.

На слици 2.11 је дат минимакс пример, а на слици 2.12 верзија минимакса са алфа – бета одсецањем. Min бира најмању вредност од 3,12,8 за позицију В, најмању од вредности 2, 4, 6 за позицију С и најмању од вредности 14, 5, 2 за позицију D. Max бира највећу од вредности 3, 2, 2 а то је 3 која одговара потезу a1.



Слика 2.11. Minimax алгоритам



Слика 2.12. Алфа – Бета одсецање

На слици 2.12 су дати кораци у одређивању оптималне одлуке. У сваком кораку су дате алфа и бета вредности у заградама. У стању А тах одређује потез, док у стањима В, С и D min одређује потез.

Корак:

- Први лист испод В има вредност 3, па је максимална вредност чвора В који је min чвор једнака 3. То је бета вредност.
- Други лист испод В има вредност 12, која је већа од бета вредности и тај потез ће min да избегне, тако да је вредност чвора В још увек највише 3.

- (c) Трећи лист испод В има вредност 8, чиме смо видели све његове наследнике и вредност В је тачно 3. Може се рећи да је вредност В најмање 3. То је алфа вредност и пошто су једнаке и алфа и бета вредности то је уједно и једина вредност.
- (d) Први лист испод С има вредност 2. С који је min чвор има вредност од највише 2. То је бета вредност. Међутим, знамо да чвор В има вредност 3, па max никад не би бирао тај потез. Дакле, нема потребе даље гледати потомке чвора С. То је пример алфа – бета одсецања.
- (e) Први лист испод D има вредност 14, тако да чвор D има вредност од највише 14. То је бета вредност. И даље је већа од најбоље алтернативе max –а, па је потребно наставити обилазак потомака чвора D.
- (f) Други лист испод D има вредност 5, па се наставља обилазак. Трећи потомак има вредност 2, па је вредност чвора D тачно 2.

Коначна одлука max –а је да одигра потез који води ка стању В, јер има вредност 3.

Перформансе алфа – бета одсецања

Перформансе јако зависе од редоследа којим се посећују потомци. У претходном примеру да смо у чвору D прво посетили потомка са вредношћу 2, могли би да одсечемо остале потомке. Било би добро када би могли да прво посетимо такве потомке који би нам омогућили да вршимо одсецање, односно најбоље потомке прво.

Када би то било могуће, алфа – бета одсецање би морало да обиђе само $O(b^{m/2})$ чворова да одабрало најбољи потез, што је у поређењу са $O(b^m)$ за minimax значајно боље. Наравно, то се не може савршено урадити. Ако се потомци посећују насумичним редоследом потребно је обићи приближно $O(b^{3m/4})$ чворова. Додавањем динамичких шема за одабир, као што је прво обићи чворове који су раније нађени као најбољи нас приближава теоретским перформансама. Разлика између теоријских перформанси алфа – бета одсецања и нормалног minimax-а је у томе што би на истим хардверским ресурсима могли да претражујемо дупло дубље, што у игри као шах чини разлику између почетника и експерта.

2.5. Minimax алгоритам са алфа – бета одсецањем

На слици Алгоритам 2.4 је дат псеудокод Minimax алгоритма са алфа – бета одсецањем.

```

1  def abMinimax(board, player, maxDepth, currentDepth, alpha, beta):
2
3  # Check if we're done recursing
4  if board.isGameOver() or currentDepth == maxDepth:
5      return board.evaluate(player), None
6
7  # Otherwise bubble up values from below
8
9  bestMove = None
10 if board.currentPlayer() == player: bestScore = -INFINITY
11 else: bestScore = INFINITY
12
13 # Go through each move
14 for move in board.getMoves():
15
16     newBoard = board.makeMove(move)
17

```

```

18  # Recurse
19  currentScore, currentMove = minimax(newBoard, player,
20                                     maxDepth, currentDepth+1, alpha, beta)
21
22  # Update the best score
23  if board.currentPlayer() == player:
24      if currentScore > bestScore:
25          bestScore = currentScore
26          bestMove = move
27
28      # If we're outside the bounds, then prune: exit immediately
29      if bestScore >= beta:
30          return bestScore, bestMove
31      alpha = max(alpha, bestScore)
32
33  else:
34      if currentScore < bestScore:
35          bestScore = currentScore
36          bestMove = move
37
38      # If we're outside the bounds, then prune: exit immediately
39      if bestScore <= alpha:
40          return bestScore, bestMove
41      beta = min(beta, bestScore)
42
43  # Return the score and the best move
44  return bestScore, bestMove

```

Алгоритам 2.4. Minimax алгоритам са алфа – бета одсецањем

Разлика у односу на стандардни minimax алгоритам је у позивању, јер имамо још алфа и бета параметре. Приликом претраживања проверавамо да ли текући потез упада у интервал одређен алфа и бета вредностима, ако не само вратимо најбољи потез и његов резултат. Алфа и бета вредности се ажурирају и тако се сужава интервал и ограничава претрага. Овај алгоритам се може позвати из функције дате на слици Алгоритам 2.5.

```

1  def getBestMove(board, player, maxDepth):
2
3  # Get the result of a minimax run and return the move
4  score, move = abMinimax(board, player, maxDepth, 0, -INFINITY, INFINITY)
5  return move

```

Алгоритам 2.5. Помоћна функција за minimax алгоритам са алфа – бета одсецањем

Почетне вредности за алфа и бета параметре у функцији су `-INFINITY` и `INFINITY`, које представљају најмању и највећу вредност који статичка функција порцене може да врати.

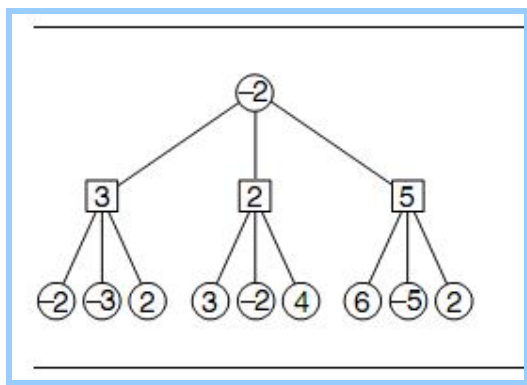
2.6. Negamax

Minimax рутина оцењује потезе на основу перспективе једног играча. Садржи код којим се прати чији је потез, и на основу тога да ли треба да максимизује или минимизује резултате. За неке игре је оваква флексибилност добродошла, али у одређеним случајевима се може побољшати.

За игре које су zero-sum са два играча, знамо да је добитак једног играча губитак другог. Та чињеница се може искористити да би се поједноставио minimax алгоритам.

У сваком кораку, уместо бирања или најмање или највеће вредности, сви резултати из претходног корака промене знак. Тада су то исправни резултати за текућег играча, али не морају ад буду исправни за играча који је покренуо претраживање. Пошто сваки играч покушава да максимизује сопствени резултат, бира се највећа вредност.

Због инвертовања знака и бирања максимума у сваком кораку, алгоритам се зове negamax. Даје исте резултате као minimax, али сваки корак му је идентичан, без обзира на играча. На слици 2.13 је дат пример рада negamax алгоритма.



Слика 2.13. Negamax алгоритам

Negamax и статичка функција процене

Статичка функција процене оцењује позицију на табли из перспективе једног играча.

У случају minimax-а на сваком кораку се процењује на основу истог играча. Сама функција процене треба да има параметар који говори на основу којег играча да врши процењивање.

Пошто negamax алгоритам у сваком потезу мења перспективу из које се посматра (на основу играча), статичка функција процене увек треба да врши процену на основу играча чији је потез на табли. Да би се то имплементирало, статичкој функцији процене више није потребан играч као параметар, већ може само да погледа који играч је на потезу. Та информација је доступна из класе табле.

На слици Алгоритам 2.6 је дат псеудо код negamax алгоритма

```

1  def negamax(board, maxDepth, currentDepth):
2
3  # Check if we're done recursing
4  if board.isGameOver() or currentDepth == maxDepth:
5      return board.evaluate(), None
6
7  # Otherwise bubble up values from below
8

```

```

9     bestMove = None
10    bestScore = -INFINITY
11
12    # Go through each move
13    for move in board.getMoves():
14
15        newBoard = board.makeMove(move)
16
17        # Recurse
18        recursedScore, currentMove = negamax(newBoard,
19                                             maxDepth, currentDepth+1)
20        currentScore = -recursedScore
21
22        # Update the best score
23        if currentScore > bestScore:
24            bestScore = currentScore
25            bestMove = move
26
27    # Return the score and the best move
28    return bestScore, bestMove

```

Алгоритам 2.6. Негатах алгоритам

Пошто није више потребан статичкој функцији процене, параметар играч се не појављује у псеудокоду. Због тога се објекат табла разликује, што је приказано на слици Алгоритам 2.7.

```

1     class Board:
2     def getMoves()
3     def makeMove(move)
4     def evaluate()
5     def currentPlayer()
6     def isGameOver()

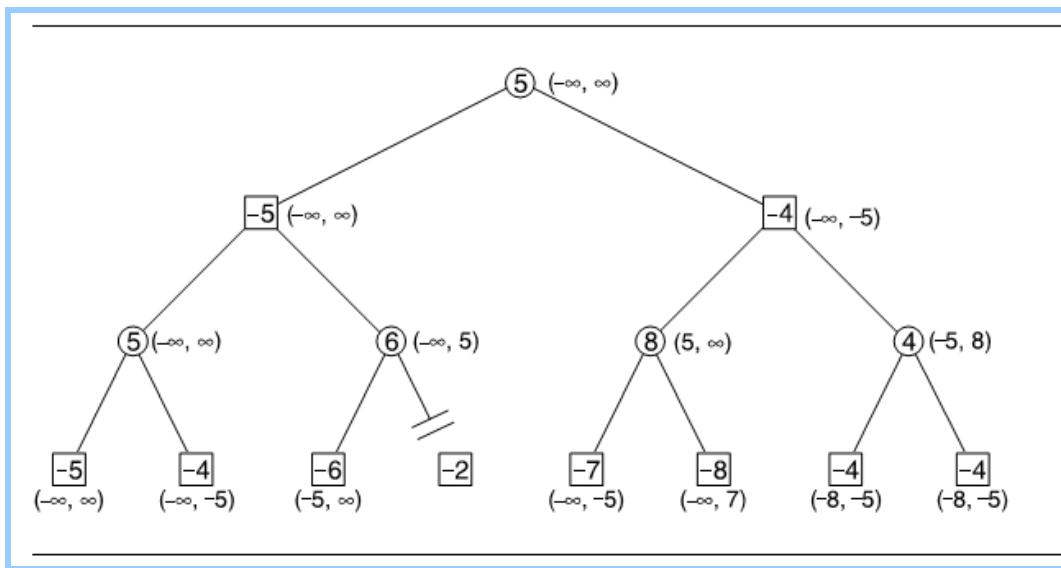
```

Алгоритам 2.7. Објекат табла

Перформансе негатах алгоритма су у истом рангу као и за *minimax* алгоритам. *Negamax* је једноставнији за имплементацију и нешто бржи, али за велика стабла има сличне перформансе.

2.7. Negamax са алфа – бета одсецањем

Уместо наизменичних провера са алфа и бета вредностима у суседним корацима, Алфа – бета негатах замени и инвертује (промени знак) алфа и бета вредности, на исти начин на који је то радио са резултатима следећег нивоа. На слици 2.14 је дат пример алфа – бета негатах-а. Поред сваког чвора у стаблу игре су дате алфа и бета вредности. Како алгоритам претражује са лево на десно, алфа и бета вредности постају све ближе, ограничавајући претрагу. Види се и како алфа и бета вредности мењају знак и мењају места у суседним корацима. На слици Алгоритам 2.8 дат је псеудокод негатах –а са алфа – бета одсецањем.



Слика 2.14. Негатах са алфа – бета одсецањем

```

1  def abNegamax(board, maxDepth, currentDepth, alpha, beta):
2
3  # Check if we're done recursing
4  if board.isGameOver() or currentDepth == maxDepth:
5      return board.evaluate(player), None
6
7  # Otherwise bubble up values from below
8
9  bestMove = None
10 bestScore = -INFINITY
11
12 # Go through each move
13 for move in board.getMoves():
14
15     newBoard = board.makeMove(move)
16
17     # Recurse
18     recursedScore, currentMove = abNegamax(newBoard,
19                                           maxDepth,
20                                           currentDepth+1
21                                           -beta,
22                                           -max(alpha, bestScore))
23     currentScore = -recursedScore
24
25     # Update the best score
26     if currentScore > bestScore:
27         bestScore = currentScore
28         bestMove = move
29
30     # If we're outside the bounds, then prune: exit immediately
31     if bestScore >= beta:
32         return bestScore, bestMove
33
34     return bestScore, bestMove
    
```

Алгоритам 2.8. Негатах са алфа – бета одсецањем.

Овај код се може позивати из функције која је дата на слици Алгоритам 2.9.

```

1  def getBestMove(board, maxDepth):
2
3  # Get the result of a minimax run and return the move
4  score, move = abNegamax(board, maxDepth, 0, -INFINITY, INFINITY)
5  return move

```

Алгоритам 2.9. Помоћна функција

Перформансе су опет истог реда као перформансе одговарајућег minimax алгоритма. Међутим, иако су истог реда, пегатах са алфа – бета одсецањем ће имати боље перформансе. Опет се јавља проблем са редоследом обилазака чворова да би се могло ефикасно вршити одсецање непотребних делова стабла. Дакле, овај алгоритам је бољи него основни пегатах, а нешто бржи од minimax-а са алфа-бета одсецањем.

2.8. Прозор претраге код алфа-бета одсецања

Интервал који формирају алфа и бета вредности у алгоритму се назива прозор претраге. Разматраће се само потези чије су процене у том опсегу, остали се игноришу, односно, одсецају.

Што је мањи прозор претраге то је већа вероватноћа да ће се грана одсећи. На почетку претраге прозор је бесконачно широк (-INFINITY, INFINITY), где су +-INFINITY вредности највећа могућа вредност и најмања могућа вредност које се могу добити од статичке функције процене. У току рада алгоритма тај прозор се сужава. Било шта што би смањило прозор, што брже могуће, ће повећати број одсецања и убрзати претраживање.

Редослед разматрања потеза

Ако би се највероватнији потези први разматрали, прозор претраге би се брже сужавао. Мање вероватни потези би се могли касније разматрати и вероватно одсећи.

Само одређивање потеза који је бољи је поента вештачке интелигенције. Да знамо који су бољи потези не би ни вршили претрагу. Потребно је наћи баланс између више знања (тако би унапред знали који су потези бољи и било би потребно мање претраживања) и више претраживања (имамо мање знања на располагању па морамо више да претражујемо).

У најједноставнијем случају је могуће користити статичку функцију процене да би одредити добар редослед потеза за разматрање. Пошто статичка функција процене даје приближну процену колико је добра нека позиција, може бити ефективна у смањењу обима претраге кроз одсецање. Мада, често позивање статичке функције процене у неким случајевима може чак и да успори извршавање.

Још ефектнији начин је да се користе резултати претходних minimax претрага. То могу бити резултати претрага на другим дубинама или претходних потеза.

Фамилија memory-enhanced test driver (MTD) алгоритама користи овај приступ да би поређала потезе пре њиховог разматрања. И на minimax фамилију алгоритама се може применити принцип распоређивања потеза.

И без одређивања редоследа разматрања потеза перформансе minimax-а са алфа бета одсецањем су значајно боље од основног minimax-а, док са распоређивањем постају изузетно боље. То значи да је могуће вршити дубље претраживање, односно, још пар потеза унапред.

Смањивање прозора пре почетка претраге

Мали интервал прозора претраге чини огормно убрзање претраживања, тако да има смисла вештачки га смањити. Уместо позивања алгорита са прозором димензија $(-\text{INFINITY}, \text{INFINITY})$, алгоритама се може позвати са прозором процењених димензија. Те димензије се начивају аспирацијом, а алгоритама са алфа-бета одсецањем који га користи алгоритмом са претрагом на основу аспирације.

Што је мањи прозор то је бржи алгоритам, али може се догодити да не постоје потези који задовољавају тај прозор. Претрага ће бити неуспешна и неће бити нађени погодни потези. Потребно је поновити претрагу са ширим прозором.

Сама аспирација може бити базирана на резултатима претходне претраге. Ако би се у претходној претрази позиција на табли оценила са 5, када се играч нађе поново у истој позицији може се покренути претрага са аспирацијом $(5 - \text{величина прозора}, 5 + \text{величина прозора})$. Величина прозора зависи од распона вредности које могу бити добијене од статичке функције процене.

```

1  def aspiration(board, maxDepth, previous):
2  alpha = previous - WINDOW_SIZE
3  beta = previous + WINDOW_SIZE
4
5  while True:
6
7      result, move = abNegamax(board, maxDepth, 0, alpha, beta);
8      if (result <= alpha) alpha = -NEAR_INFINITY;
9      else if (result >= beta) beta = NEAR_INFINITY;
10     else return move;
```

Алгоритам 2.10. Код који би вршио претрагу на основу аспирације

2.9. Negascout

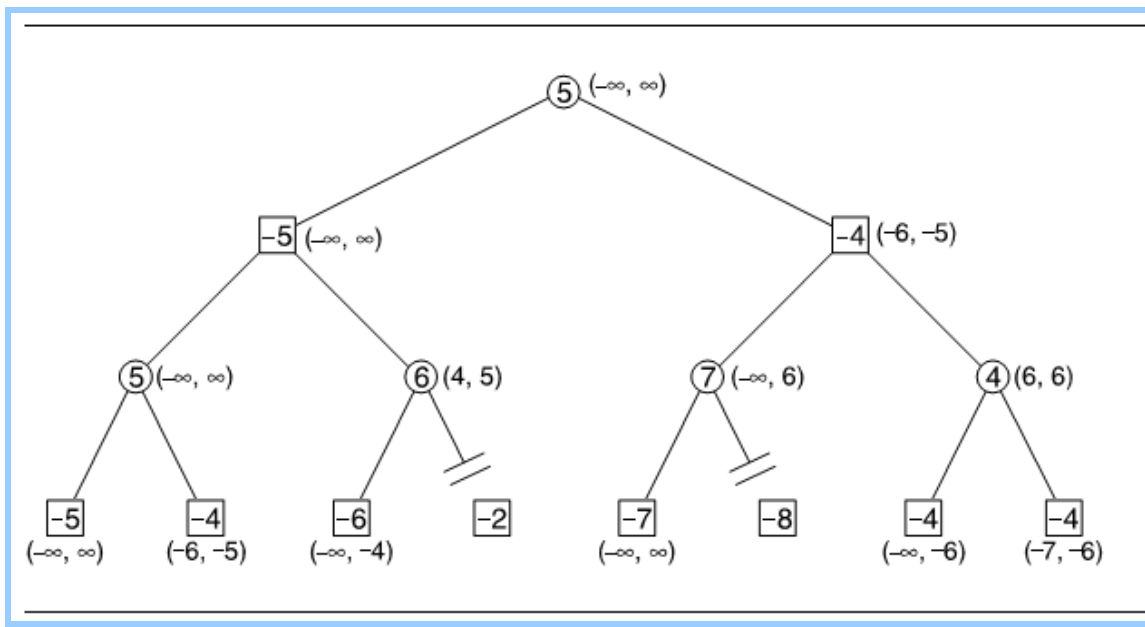
Сужавање прозора претраге се може довести до екстемности, када је његова величина једнака нули. На тај начин претрага ће бити веома брза, али ће одсећи скоро све корисне гране заједно са бескорисним. Тако да ако се покрене алгоритам са погрешним прозором, претрага ће бити неуспешна. Прозор величине нула може се сматрати тестом који тестира стварни резултат са претпоставком.

Основна верзија алфа-бета пегатах-а ће вратити или алфа или бета вредност у случају неуспеха (и то у зависноти на основу које границе је неуспешно претраживање). Та информација омогућава да се прилагоди претпоставка и да се понови претрага са разумнијим прозором.

Negascout алгоритам користи алфа-бета пегатах. Прво се врши потпуна претрага само првог потеза за сваку позицију на табли. То се ради са широким прозором да би сама претрага била успешна. Следећи потези се разматрају на основу резултата првог потеза. Ако је претрага на основу информација добијених из првог потеза неуспешна, претрага се понавља са прозором пуне ширине, исто као код алфа-бета пегатах-а.

Иницијална претрага првог потеза са широким прозором даје добру апроксимацију за тест. Тако се избегава превише неуспеха и дозвољава одсецање великог броја грана.

На слици 2.15 је дат пример negascout алгоритма. Алфа и бета вредности више варирају него код пегатах-а. У општем случају negascout доминира над пегатах алгоритмом јер потребно да обиђе исти број или мање чворова.



Слика 2.15. Пример Negascout алгоритма

На слици Алгоритам 2.11 дат је псеудокод Negascout алгоритма.

```

1  def abNegascout(board, maxDepth, currentDepth, alpha, beta):
2
3  # Check if we're done recursing
4  if board.isGameOver() or currentDepth == maxDepth:
5      return board.evaluate(), None
6
7  # Otherwise bubble up values from below
8
9  bestMove = None
10 bestScore = -INFINITY
11
12 # Keep track of the Test window value.
13 adaptiveBeta = beta
14
15 # Go through each move
16 for move in board.getMoves():
17
18     newBoard = board.makeMove(move)
19
20     # Recurse
21     recursedScore,
22     currentMove = abNegamax(newBoard,
23                             maxDepth,
24                             currentDepth+1,
25                             -adaptiveBeta,
26                             -max(alpha, bestScore))
27     currentScore = -recursedScore
28
29     # Update the best score
30     if currentScore > bestScore:
```

```

31
32     # If we are in 'narrow-mode' then widen and
33     # do a regular AB negamax search
34     if adaptiveBeta == beta || currentDepth >= maxDepth-2:
35         bestScore = currentScore
36         bestMove = move
37
38     # Otherwise we can do a Test
39     else:
40         negativeBestScore,
41         bestMove = abNegascout(newBoard,
42                                maxDepth,
43                                currentDepth+1,
44                                -beta,
45                                -currentScore)
46     bestScore = -negativeBestScore
47
48     # If we're outside the bounds, then prune: exit immediately
49     if bestScore >= beta:
50         return bestScore, bestMove
51
52     # Otherwise update the window location
53     adaptiveBeta = max(alpha, bestScore) + 1;
54
55     return bestScore, bestMove

```

Алгоритам 2.11. Negascout алгоритам

Комбиновањем претраживања на основу аспирација са negascout алгоритмом се добија моћан алгоритам који се користи у неким од најбољих AI-а који играју игре на табли. Такви програми су способни да играју шах на шампионском нивоу. Перформансе су опет истог реда величине али је алгоритам ефикаснији него negamax са алфа – бета одсецањем.

Negascout алгоритам се ослања на резултат првог потеза за сваку позицију на табли да водио даље претраживање. Из тог разлога још бољи резултати се постижу када се потези распореде. Ако се прво обиђе најбоља секвенца потеза, предикција иницијалног прозора ће да буде веома тачна и биће мање неуспешних претрага.

Додатно, због потребе да се поново претражују делови стабла доста се добија на перформансама ако се користи меморијски систем којим се могу користити резултати претходних претраживања.

2.10. Табеле транспозиције и меморија

Алгоритми који су разматрани до сада су предпостављали да сваки потез води до уникатне позиције на табли. Видели смо у случају игре nim да то није увек тачно. Иста позиција на табли може бити резултат различите секвенце потеза. Чак је могуће исту ситуацију на табли достићи неколико пута у току партије.

Да би се избегло поновно претраживање исте позиције више пута, може се користити у алгоритму табела транспозиције. Неки алгоритми табелу транспозиције користе као радну меморију за позиције које су размотрене. Један такав алгоритам је memory-enhanced test driver (MTD) алгоритам.

Табела транспозиције чува записе о позицији на табли и резултата пертраге од те позиције. Када алгоритам разматра позицију на табли може прво да провери да ли је па позиција у меморији и да искористи сачуван потез.

Само поређење комплетних стања је скуп процес, јер стање може садржати, у зависности од игре, мноштво информација. Поређење би дуго трајало. Да би се убрзао процес поређења може се користити хеш функција.

Хеширање стања у игри

Иако би било који хеш алгоритам радио, постоје специфичности у хеширању. Многа могућа стања се вероватно неће догодити. Дobar хеш алгоритам ће да користи функцију која ће да могуће позиције развуче кроз цео свој распон. Такође, у многим играма се позиције мало мењају у суседним потезима, па је корисно да се вредности хеш функције доста разликују за мале промене на табли. Тако се смањује вероватноћа да се те две позиције бити у колизији.

Zobrist кључеви

Створи се табела случајних вредности фиксне дужине, где свака позиција на табли има придружену вредност за свако могуће стање у којем може бити. Шах има 64 поља, на сваком пољу може бити једна фигура од шест или поље може бити празно, свака фигура може бити бела или црна. Zobrist кључ за шах ће имати $64 * 2 * 6 = 768$ улаза.

Да би се одредио хеш код, за свако непразно поље се нађе Zobrist кључ и хог-ује се са текућим укупним хешом. Када се то уради за сва поља добијемо хеш вредност.

Могу постојати и додатни Zobrist кључеви за различите специфичности везане за игру. Неке имплементације избегавају такве додатне кључеве, јер је претпоставка да ће се мало користити.

Те вредности се користе на исти начин као и претходне, односно, нађу се њихове вредности и хог-ују се са текућем вредношћу хеша. После обраде свих вредности добије се финална хеш вредност.

Дужина кључева зависи од игре. У случају шаха 32 бита је довољно, али су боље перформансе са 64 битним кључевима. Комплексније потезне игре могу тражити и 128 бита.

За једноставне игре као што је икс-окс приказан је псеудокод на слици Алгоритам 2.12. Довољно је $2 * 9 = 18$ улаза, јер имамо 2 играча и девет места.

```

1  # The Zobrist key.
2  zobristKey[9*2]
3
4  # Initialize the key.
5  def initZobristKey():
6      for i in 0..9*2:
7          zobristKey[i] = rand32()
```

Алгоритам 2.12. Zobrist кључ иницијализација

Када је табела направљена, могу се хеширати позиције на табли. То је илустровано на слици Алгоритам 2.13.

```

1  # Calculate a hash value.
2  def hash(ticTacToeBoard):
3  # Start with a clear bitstring
```

```

4     result = 0
5
6     # XOR each occupied location in turn
7     for i in 0..9:
8         # Find what piece we have
9         piece = board.getPieceAtLocation(i)
10
11        # If its unoccupied, lookup the hash value and xor it
12        if piece != UNOCCUPIED:
13            result = result xor zobristKey[i*2+piece]
14
15    return result

```

Алгоритам 2.13. Хеширање позиције на табли

Да би се подржало хеширање потребно је изменити класу табле. Хеш вредност се може чувати у самој класи табле, па када је потребно користити ту вредност може се дохватити без потребе за поновно речунање. Илустровано на слици Алгоритам 2.14.

```

1     class Board:
2         # Holds the current hash value for this board. This saves it
3         # being recalculated each time it is needed.
4         hashCode
5
6         def getMoves()
7         def makeMove(move)
8         def evaluate()
9         def currentPlayer()
10        def isGameOver()
11        def hashValue()

```

Алгоритам 2.14. Хеширање позиције на табли

Дакле, хеш табела чува вредност придружену позицији на табли, такође се може чувати и најбољи нађени потез. Додатно, и дубина претраге којом је нађен тај потез се може сачувати, јер је вреднији потез који је нађен претраживањем 10 потеза унапред него онај који је добијен претрагом 3 потеза унапред. У случају алгоритма са алфа – бета одсецањем може се чувати податак да ли је израчуната процена за позицију добијена алфа одсецањем, бета одсецањем или је у питању тачна вредност.

У случају конфликта је потребно одредити шта треба да се ради. Најједноставнија техника је да се увек пребрише улаз ако постоји конфликт. То је једноставно за имплементацију и обично је довољно. Друга хеуристика је да се замени само ако је у питању каснији потез. Тако да ако позиција на табли у потезу 6 је у конфликту са позицијом у потезу 10, позиција на табли у потезу 10 се користи. Базира се на претпоставци да је каснија позиција кориснија. Постоје и друге комплексније стратегије замене, али изгледа да су одређене стратегије оптималније за неке игре од других. Неки програми су користили и више табела транспозиције са разним стратегијама. У једном потезу свака се провери и тако се балансирају мане сваког приступа појединачно.

Међутим, постоје и проблеми са табелама транспозиције. Један од њих је зависност од пута. Неке игре имају резултате које зависе од секвенце потеза.

У шаху би повлачење истог потеза три пута довело до нерешене партије. Такве позиције ће увек бити исто оцењиване. Решење је користити Zobrist кључеве са додатним специфичностима. Тако да ће се добити различите хеш вредности и чуваће се одвојено. Уопштено, игре чије процене зависе од секвенце ће користити сложеније хеширање или ће имати посебан код који ће детектовати специјалне ситуације.

Други проблем је нестабилност. Пошто се сваки улаз у табели може пребрисати, нема гаранције да ће се иста вредност враћати сваки пут када се анализира позиција на табли. Иако се врло ретко догађа, могућа је ситуација када вредносту флукутирају између две вредности, што у неким алгоритмима који имају поновно претраживање (не у основном negascout-у), доводи до бесконачне петље.

Коришћење противничког потеза за размишљање

Табела транспозиције се може искористити да би се дозволило AI-у да побољша претраживање док људски играч размишља. У противничком потезу, компјутер може да претражује за потезом који би одиграо да је на потезу. Резултати ове претраге се чувају у табели транспозиције. Када буде ред да компјутер игра, претрага ће бити бржа зато што се велики део позиција на табли већ обишао и сачувао. Многи комерцијални програми за игре на табли користе ову технику.

2.11. Memory-enhanced test driver (MTD) алгоритам

Ослања се на постојање ефикасне табеле транспозиције која ће да буде радна меморија алгоритма.

У суштини, memory-enhanced test driver (MTD) алгоритам је negamax са алфа – бета одсецањем који ради на ширини прозора од нула. Меморија му омогућава да скакуће по стаблу игре тражећи потезе са највише потенцијала. Рекурзивна природа negamax алгоритма му не дозвољава да скакуће по стаблу игре, већ ће се он рекурзивно спуштати по стаблу.

Имплементација тест функције

Пошто је ширина прозора нула, довољно је користити само једну вредност уместо алфа и бета вредности, јер су исте. Та вредност ће се означити са гама. Додатак поједностављеном negamax алгоритму ће бити код којим се приступа табели транспозиције. На слици Алгоритам 2.15 је дата функција.

```

1  def test(board, maxDepth, currentDepth, gamma):
2
3  if currentDepth > lowestDepth: lowestDepth = currentDepth
4
5  # Lookup the entry from the transposition table
6  entry = table.getEntry(board.hashValue())
7  if entry and entry.depth > maxDepth - currentDepth:
8
9      # Early outs for stored positions
10     if entry.minScore > gamma:
11         return entry.minScore, entry.bestMove
12     if entry.maxScore < gamma:
13         return entry.maxScore, entry.bestMove
14
15  else:
16
```

```

17     # We need to create the entry
18     entry.hashValue = board.hashValue()
19     entry.depth = maxDepth - currentDepth
20     entry.minScore = -INFINITY
21     entry.maxScore = INFINITY
22
23     # Now we have the entry, we can get on with the test
24
25
26     # Check if we're done recursing
27     if board.isGameOver() or currentDepth == maxDepth:
28         entry.minScore = entry.maxScore = board.evaluate()
29         table.storeEntry(entry)
30         return entry.minScore, None
31
32     # Now go into bubbling up mode
33     bestMove = None
34     bestScore = -INFINITY
35
36     for move in board.getMoves():
37
38         newBoard = board.makeMove(move)
39
40         # Recurse
41         recursedScore, currentMove = test(newBoard,
42                                         maxDepth,
43                                         currentDepth+1,
44                                         -gamma)
45         currentScore = -recursedScore
46
47         # Update the best score
48         if currentScore > bestScore:
49
50             # Track the current best move
51             entry.bestMove = move
52
53             bestScore = currentScore
54             bestMove = move
55
56     # If we pruned, then we have a min score, otherwise
57     # we have a max score.
58     if bestScore < gamma: entry.maxScore = bestScore
59     else: entry.minScore = bestScore
60
61     # Store the entry and return the best score and move.
62     table.storeEntry(entry)
63     return bestScore, bestMove

```

Алгоритам 2.15. Помоћна функција тест за MTD алгоритам

Зато што све претраге имају нулте ширине прозора мала је вероватноћа да ће се наћи тачан резултат, али се може после неколико покушаја доћи до могућег опсега резултата. Табела транспозиције чува и најмање и највеће резултате. Јер се само те две вредности чувају, садржај табеле транспозиције је дат на слици Алгоритам 2.16.

```

1 struct TableEntry:
2     hashValue
3     minScore
4     maxScore
5     bestMove
6     depth

```

Алгоритам 2.16. Садржај табеле транспозиције

MTD алгоритам

Тест функција се стално зове из драјвер рутине. Одговорност драјвер рутине је да користи тест функцију (обично се назива МТ) и да се приближава коректној минимакс вредности, као и да одреди следећи потез. Алгоритми овог типа се називају тест драјвери потпомогнути меморијом.

Уобичајени MTD алгоритам ради следеће :

- (1) Води евиденцију о горњој граници вредности процене. Зваћемо је гама (да би се разликовала од алфа и бета).
- (2) Нека гама буде прва претпоставка за резултат. Може бити било која фиксна вредност или се може добити на основу претходног рада алгоритма.
- (3) Израчунати другу претпоставку позивањем функције тест за текућу позицију на табли, максималне дубине, нулте ширине прозора и са гама вредношћу.
- (4) Ако процена није иста као гама вредност, вратити се на корак (3). Овим потврђујемо да је процена тачна. Ако се не може доћи до поклапања користи се лимит на број итерација.
- (5) Вратити процену као резултат, ако је тачна.

Потребно је да MTD алгоритми користе параметар који служи као нагађање. Што је тачније то нагађање то ће се алгоритам брже извршавати.

Такође, MTD алгоритми се ослањају на велику меморију. Перформансе му се дегрдирају када постоје колизије у табели транспозиције и када се различите позиције на табли пресликају у исти улаз у табели. У најгорем случају алгоритам не може да врати резултат ако му се стално меморија пребрисује. Величина табеле зависи од фактора гранања, дубине претраге и квалитета алгоритма хеширања. За шах су потребне табеле од неколико мегабајта, са милионима улаза.

На слици Алгоритам 2.17. је дат псеудокод MTD алгоритма

```

1 def mtd(board, maxDepth, guess):
2     for i in 0..MAX_ITERATIONS:
3
4         gamma = guess
5         guess, move = test(board, maxDepth, 0, gamma)
6
7         # If there's no more improvement, stop looking
8         if gamma == guess: break
9
10    return move

```

Алгоритам 2.17. MTD алгоритам

Перформансе су опет истог реда за време, док је за простор $O(s)$, где је s број улаза у табели транспозиције. MTD алгоритам је ривал negascout алгоритму по питању перформанси као најбржи алгоритам за претраживање стабала игара. Тестови показују да је често MTD бржи, али остаје да се види који ће моћи још да се оптимизује. Многи бољи алгоритми користе negascout као алгоритам за игре на табли, међутим, доста модерног AI-а се ослања на MTD језгро. Једини сигуран начин је да се оба пробају за тражену апликацију и да се тако одреди бољи.

2.12. Књиге отварања и друге секвенце потеза

У многим играма експерти, помоћу вишегодишњег искуства, су способни да одреде који су потези бољи од других на почетку партије. То је врло видљиво на примеру шаха, где се проучавају велике базе фиксних почетних потеза, учећи најбоље одговоре на неке потезе.

Књига отварања је листа потеза, са индикацијама колико ће бити добра крајња позиција помоћу те секвенце. Коришћењем тих правила компјутер не мора да користи алгоритам за претраживање да би одредио најбољи потез. Потребно је само да бира следећи потез из секвенце, све док је крај секвенце користан. За популарне игре на табли се могу наћи бесплатне књиге отварања, као и комерцијалне.

Сама имплементација може бити хеш табела, која је слична табели транспозиције. За разлику од обичне табеле транспозиције, може постојати више препоручених потеза из текуће позиције на табли. Позиције до којих воде ти потези могу припадати у више улаза, па ће сама отварања имати структуру стабла. Дакле, AI ће само погледати тренутну позицију у књизи отварања и добија скуп потеза које може да направи.

Књиге отварања се могу користити и у функцији процене као један њен елемент. Ако се тренутна позиција налази у књизи отварања, онда ће тежински фактор бити повећан. Када је игра унапредовала од фазе отварања, други елементи функције процене ће се користити и више вредновати, док ће се савет књиге отварања игнорисати.

Неки програми долазе са иницијалном књигом отварања и са слојем за учење. На тај начин се процене ажурирају да би се боља отварања могле бирати. Један начин је да се чува статистички успех програма са сваким отварањем. Ако је отварање добро оцењено, али компјутер стално губи са њим, мења му се оцена да се не би често користило у будућности. Велика пажња се посвећује оцењивању сваког отварања и базирано је на дугој историји интернационалних партија. Али, сваки алгоритам за игру има различите карактеристике. Неко отварање може касније довести до позиције коју алгоритам неће добро одиграти, тако да је корисно додати статистички слој за учење да би се отварања прилагодила алгоритму.

Неке игре ће саме учити секвенце. После хиљада партија нека отварања ће се често појављивати и тако учити. Иницијално, компјутер може преко свог алгоритма претраге да их оцењује. Током времена се те вредности могу узети средње вредности са статистичким информацијама о броју победа. Многе књиге отварања за мање популарне игре се на овај начин формирају, тако што компјутер игра сам са собом гомилу партија.

Иако се такве секвенце потеза користе на почетку, могу се и касније у току партије користити. Многе игре имају фиксне комбинације потеза које се догађају у току партије, посебно на крају.

Због самог броја могућих стања на табли, није вероватно да ће позиција бити иста као она из базе. Потребан је механизам који посматра обрасце и он ће да тражи одређене обрасце који се налазе у структури позиције. Најчешћи случај овог типа су базе са само деловима табле који се траже у текућој позицији. Ово је посебно значајно у играма као што је Go и шах.

Завршна база

Врло касно у партији позиција на табли се значајно поједностави у неким играма. У њима је могуће користити приступ сличан књигама отварања (у играма као што су шах, мице или backgammon).

Постоје завршне базе за шах, које покривају најбоље начине да се матира противник са различитим фигурама на располагању. Обично нису потребне у играма са експертима, где ће играч предати меч када је у лошој ситуацији.

2.13. Даља унапређења

Велика мана компјутерских играча у потезним играма је ефекат хоризонта. Ефекат хоризонта настаје када се фиксна секвенца потеза заврши са потезом који изгледа као одлична позиција, међутим, један потез унапред ће се показати да је та позиција ужасна.

У шаху компјутер може наћи серију потеза којом ће се заузети непријатељска краљица. Нажалост, одмах после заузимања непријатељске краљице противнички играч може да шах-матира компјутер. Да је компјутер мало дубље претраживао, видео би овај резултат и не би одабрао фаталан потез.

Без одзира на дубину претраживања, овај ефекат и даље може бити присутан. Ако је претрага изузетно дубока, компјутер може имати времена да изабере бољи потез када види опасност. Ако се претрага не може вршити на већој дубини због великог фактора гранања, а ефекат хоризонта је још увек присутан, онда минимакс алгоритам може користити технику звану екстензије.

Екскреније су техника променљиве дубине, где се потези са највећим потенцијалом претражују до веће дубине. Пажљивим бирањем потеза дубина претраге се значајно повећава, чак и за 10 -ак потеза. Значајно зависи од извора потеза који ће се дубље претраживати.

2.14. Потезне (turn – based) стратегије

AI за игре на табли има доста сличности са AI-ем за потезне стратегије. Додуше, комерцијалне стратегије обично не користе алгоритме за претраживање стабла игре који су до сада изложени као главни алгоритам. Сама комплексност тих игара значи да се алгоритми претраживања загушити пре него што дођу до разумних одлука.

Такође, многе технике за претраживање су дизајниране за игре са два играча, које су zero-sum са савршеним информацијама и многе оптимизације не могу се прилагодити општим стратешким играма.

Просте потезне стратегије могу користити алгоритме претраживања. На примеру потезне стратешке игре у којој се гради држава се види да то није могуће због броја одлука у сваком потезу и самим тим, великим фактором гранања. Потребно је за

десетине, понекад чак и стотине градова одредити какве објекте ће да праве, како ће грађани унутар њих бити распоређени, одредити какве технологије ће се истраживати и каква ће бити глобална и локална економија.

Потребно је одредити и политичко уређење. Поред тога, потребно је одредити дипломатске и војне акције. Мноштво трупа је потребно померати на битне локације. Обично је у војним акцијама укључен и фактор случајности поред особина те војне јединице, и терена на којем је. Број могућих потеза је огроман. У теорији представљени алгоритми се могу користити и за овакве игре, али нису практични. Ако би се у таквој игри са десетинама хиљада могућих потеза користио обичан минимакс ништа се не би постигло.

Дакле, главни проблем за комплексне потезне игре је огромна величина стабла игре. Постоји више приступа за поједностављење.

Подели и владај

Један приступ је подели и владај приступ, којим се групишу могући потези да би се смањио број опција у сваком потезу.

Могу се разматрати и глобалне стратегије уместо индивидуалних потеза. Компјутер би могао да нападне суседну нацију. У том случају AI се понаша као горњи ниво у AI-у са више нивоа. Тада је AI на нижем нивоу задужен да преузме више атомских акција да то уради, на основу стратегије на вишем нивоу.

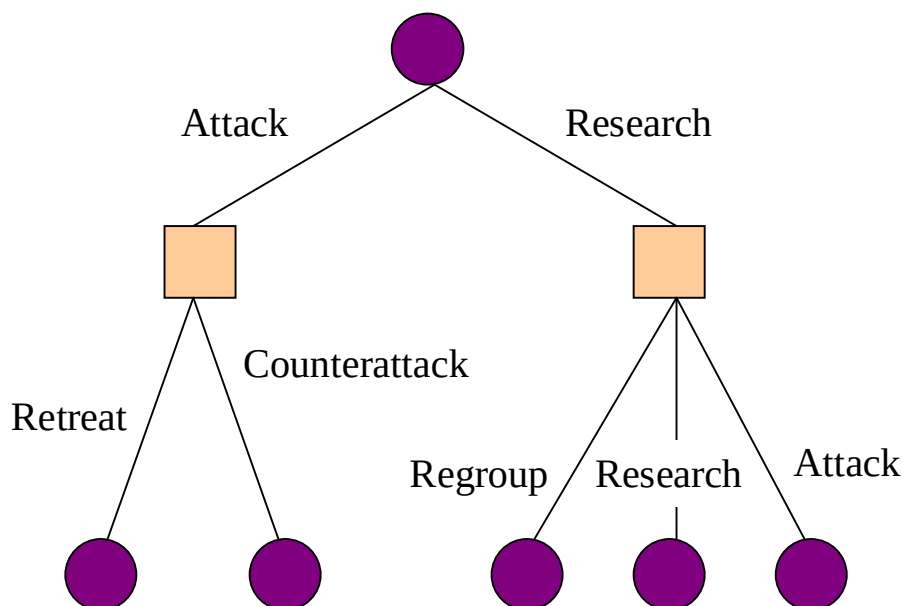
На слици 2.16 је дат пример стратешког одлучивања на основу минимакс алгоритма.

Овај приступ је могућ и у играма у реалном времену.

Хеуристике

Чак и са агресивним приступом подели и владај, проблем остаје велик. AI за стратешке игре ће бити толико заснован на хеуристикама, да ће се често напустити коришћење минимакса и користиће се само хеуристика.

Саме хеуристике могу узимати у обзир параметре као што су величина територије која се контролише, близина непријатељских снага, технолошка супериорност, задовољност популације и сличне.

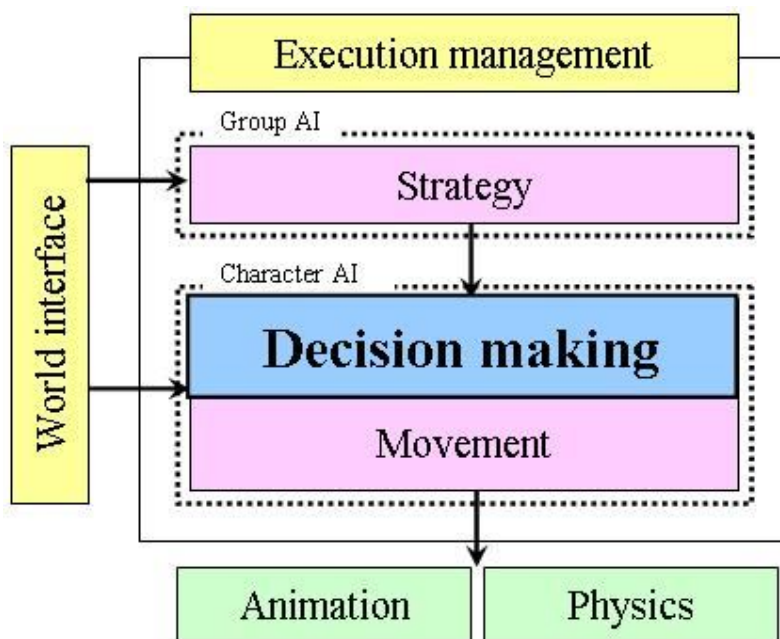


Слика 2.16. Стабло стратешког одлучивања

Постоје сличности у AI-у у потезним стратегијама са AI-ем у стратегијама у реалном времену (RTS). Многи алгоритми који су развијани за RTS-ове су применљиви и на потезне игре. Посебно технике као мапирање утицаја, анализа терена, стратешке скрипте и планирање на високом нивоу. Мапирање утицаја се првобитно користило у потезним играма.

3. Доношење одлука

Доношење одлука је обично само један део структуре AI-а за игру, што се види са модела којег смо усвојили. Дат је на слици 3.1.

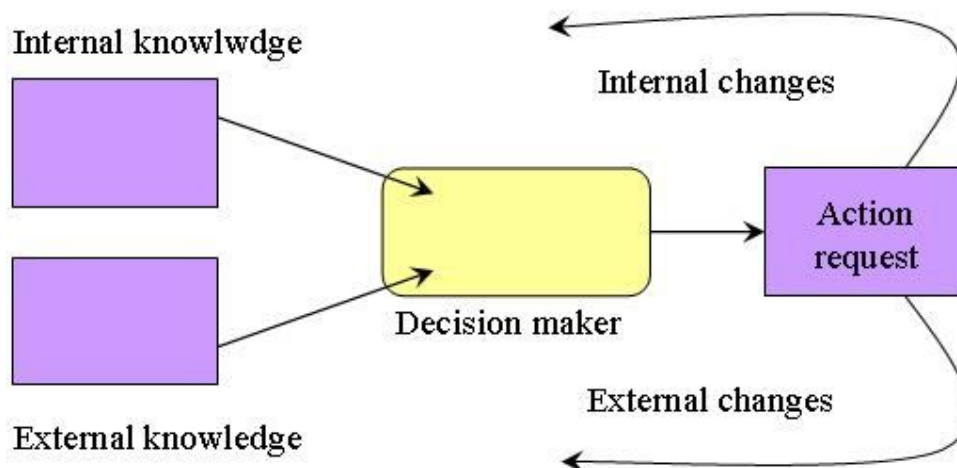


Слика 3.1. Структура AI-а за игру

Многе игре користе једноставне механизме за одлучивање : коначне аутомате (state machines) и стабла одлучивања (decision trees). Системи засновани на правилима су ређи (rule-based systems). У последње време и сложенији системи за доношење одлука су коришћени, као што је fuzzy логика.

Све технике које ће бити приказане се могу користити и на вишем, стратешком и тактичком нивоу.

Карактер поседује одређене информације помоћу којих је потребно генерисати акцију. Улаз је знање које карактер поседује, а излаз је захтев за акцијом. Знање може бити екстерно и интерно. Екстерно би било знање карактера о свету око себе, а интерно је знање о самом стању тог карактера (његовим циљевима нпр). Акције могу мењати екстерно стање карактера, тако што се утиче на окружење, или могу мењати интерно стање карактера. Те интерне промене су мање изражене, али су битне за одлучивање. Дакле, репрезентација интерног стања ће варирати од технике до технике, док би се екстерне акције представљале на исти начин. Формат и количина знања зависи од захтева игре и само знање је веома повезано са већином алгоритама за доношење одлука. Сам систем одлучивања се може представити са сликом 3.2.



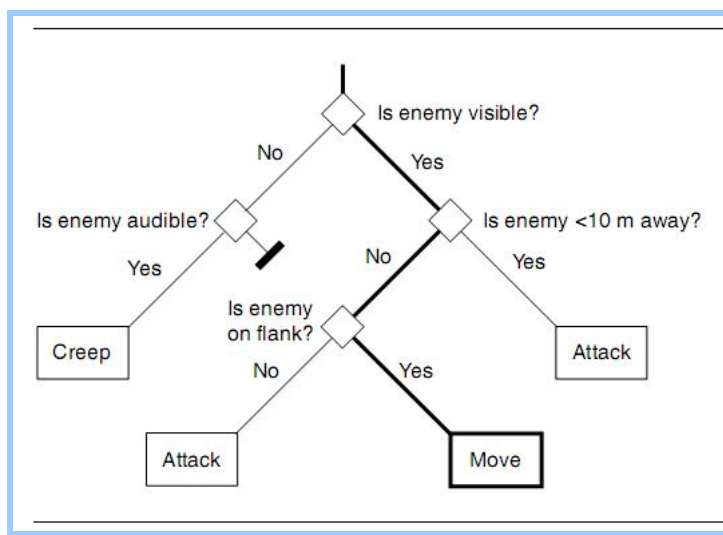
Слика 3.2. Систем одлучивања

3.1. Стабла одлучивања (decision trees)

Стабла одлучивања су брза, лака за имплементацију и једноставна за разумевање. Она су најједноставнија техника за доношење одлука, мада могу помоћу одређених унапређена постати веома софистицирана. Предност је што су модларна и лака за креирање. Користила су се за све од једноставног одлучивања до комплексног стратешког и тактичког AI-а. Такође, могу се учити у току игре и учење је релативно брзо у поређењу са осталим техникама.

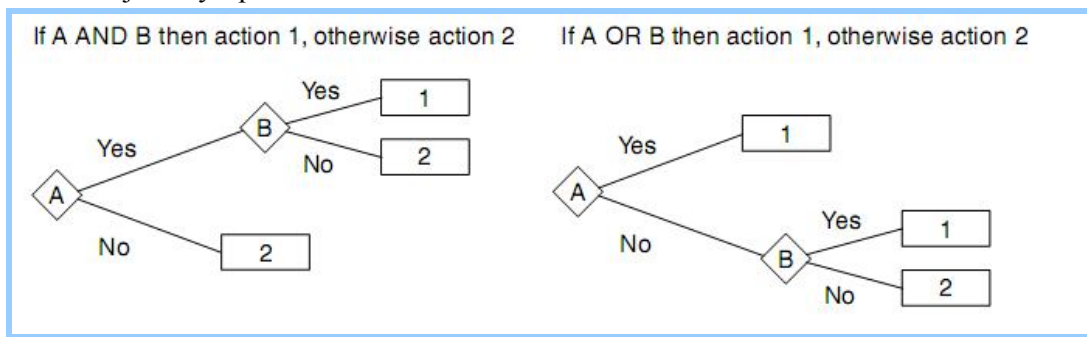
Потребно је на основу пасположивог знања генерисати одговарајућу акцију из скупа могућих акција. Мапирање између улаза и излаза може бити комплексно. Потребан је метод који ће лако груписати улазе за одговарајућу акцију, дозвољавајући битнијим деловима улаза да више утичу на добијену акцију.

Стабла одлучивања се састоје од повезаних чворова одлучивања. Стабло има почетну одлуку, која је корен. У сваком чвору се бира једна опција на основу улаза. Свака одлука се базира на знању карактера. Алгоритам наставља по стаблу, бирајући одлуку у сваком чвору док не стигне до ситуације када нема више одлука за разматрање. Сваком листу у стаблу је придружена акција. Када алгоритам стигне до акције, та акција се одмах изврши.



Слика 3.3. Стабло одлучивања са донетом одлуком

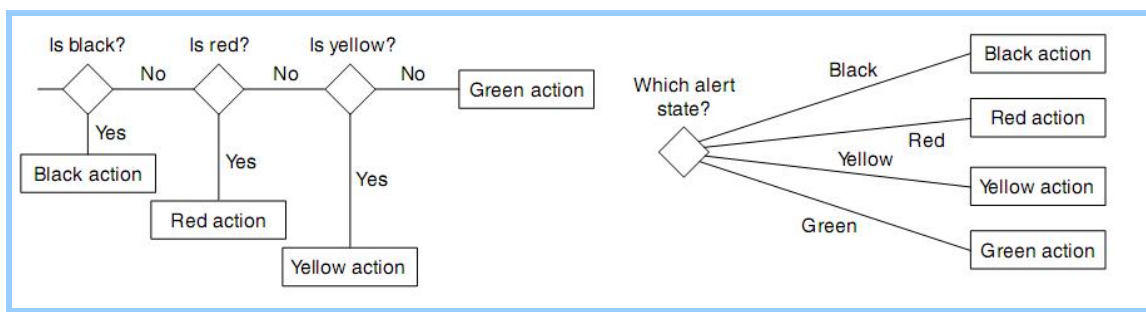
Саме одлуке су једноставне. Обично проверавају једну вредност и не садрже boolean логику (тј. нема одлука спојених са AND и OR). Сама вредност која се проверава може бити било којег типа. Помоћу стабла одлучивања је могуће репрезентовати и boolean комбинације. Илустровано на слици 3.4.



Слика 3.4. Boolean комбинације у стаблу одлучивања

Број одлука које је потребно размотрити је обично значајно мањи од укупног броја одлука у стаблу. Стабла одлучивања се лако праве и могу се конструисати у етапама. Једноставно стабло се прво конструира, и пошто је AI који га користи тестиран, додају се додатне одлуке да би се третирали специјални случајеви и нова понашања.

До сада смо приказали стабла која бирају између две опције и таква стабла се називају бинарним, међутим, могу се конструисати стабла са већим бројем опција. На слици 3.5 је лево бинарно а десно стабло са већим бројем опција. Структура стабла са десне стране је ефикаснија.



Слика 3.5. Бинарно и стабло са већим бројем опција

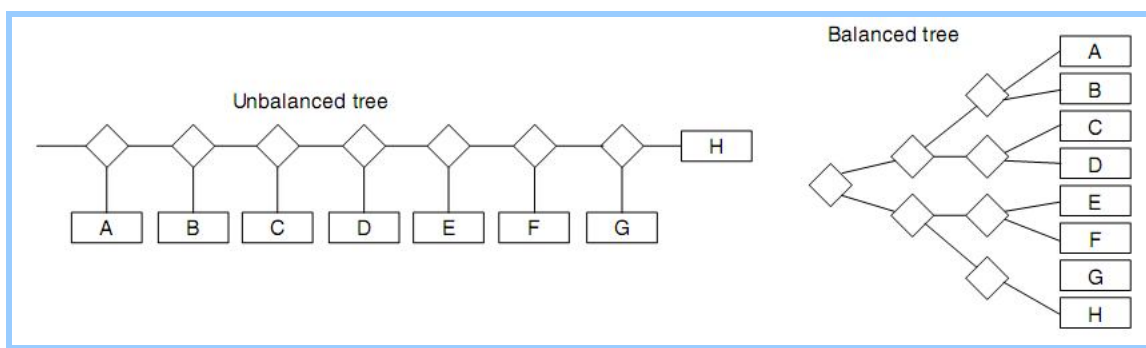
Стабла одлучивања су обично бинарна, јер се лакше оптимизују. Додатно, неки алгоритми за учење који раде са стаблима одлучивања захтевају да буду бинарна. Све шта се може урадити са сложенијим стаблом се може урадити и са бинарним. Убрзање које се добија са сложенијим стаблима је обично маргинално.

Одлуке се могу доносити на основу било каквих типова података који су специфични за игру, што значи да није потребно преводити знања у другачији формат од остатка игре. Такође, стабла одлучивања се обично имплементирају да могу директно да приступају стању игре. Ако је потребно знати колико је далеко играч од противника, вероватно ће директно приступити играчу и противнику да би им одредио локацију. Ово може довести до грешака које би настале ако се структура стања игре мења у току развоја.

Перформансе су линерне са бројем посећених чворова одлучивања. Ако претпоставимо да је за сваку одлуку потребно исто време и да је стабло балансирано, перформансе ће бити $O(\log_2 n)$, где је n број чворова у стаблу.

Обично је време потребно за одлуке константно. Ако је потребно проверити видљивост биће потребне сложеније калкулације на основу геометрије нивоа и брзина одлучивања у стаблу ће зависити од ове сложене провере.

За добре перформансе стабла одлучивања је потребно балансирати стабло. Балансирано стабло има отприлике исти број листова у свакој грани.



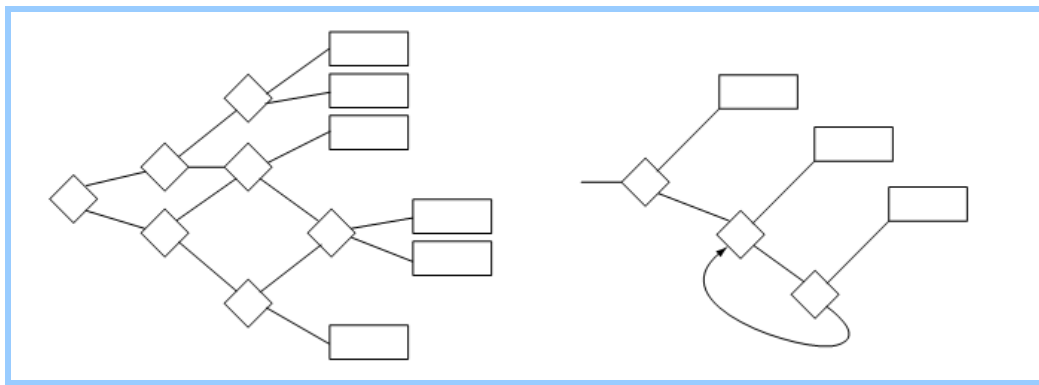
Слика 3.6. Неизбалансирано и балансирано стабло

На слици 3.6 је дат пример стабла које је веома неизбалансирано и истог стабла када се избалансира. Оба имају 8 понашања и 7 одлука. Да би се дошло до понашања H, у неизбалансираном стаблу је потребно 8 одлука, док је у балансираном потрено само 3 одлуке. Да су све одлуке поједнако вероватне, у неизбалансираном стаблу је потребно просечно 4.375 одлука, док су у балансираном увек довољне 3 одлуке. Дакле, увек желимо да избалансирамо стабло што је више могуће због перформанси. У најгорем случају, у изузетно неизбалансираном стаблу перформансе стабла одлучивања су реда $O(n)$, што је значајно лошије од теоретског $O(\log_2 n)$.

Међутим, нису све одлуке подједнако вероватне. Ако би у примеру стабла са слике 3.6 понашање A било највероватније, прво стабло је боље јер у једном кораку долази до A, док су у другом стаблу потребна 3 корака. Одлука која је веома скупа за рачунање би требала бити што је ниже у стаблу, чак иако ће због тога стабло бити неизбалансирано.

Потребно је балансирати стабло, уз стављање честих понашања ближе корену и стављања скупих одлука ниже у стаблу.

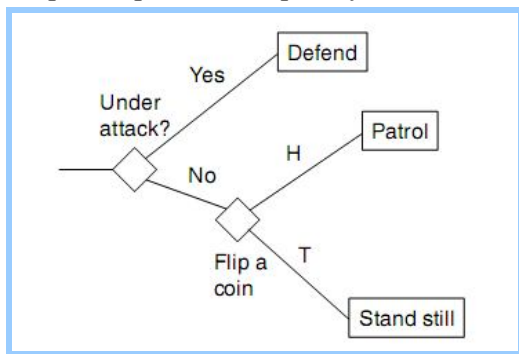
Стабло одлучивања се може проширити да дозволи вишеструка гранања која се стапају у једну одлуку. Треба пазити да се не створе петље у стаблу, јер би тако процес одлучивања могао бесконачно да траје. Стриктно, валидна структура би била усмерени ациклични граф (DAG).



Слика 3.7. Исправно стабло и неисправно које садржи петљу

Често, не желима да избор понашања буде комплетно предвидљив. Неки насумични елементи су потребни. Додавање одлуке са случајним елементима је

једноставно, само је потребно генерисати случајни број и на основу њега вршити одабир гране. Пошто се стабла одлучивања често користе, потребно је пажљиво користити случајне елементе да би понашање било реално. Ако би се честим коришћењем стабла са слике 3.8. карактер стално у току патроле често заустављао понашање би било неприродно. Решење је памтити претходну одлуку, и њено коришћење када се поново разматра исти чвор одлучивања.



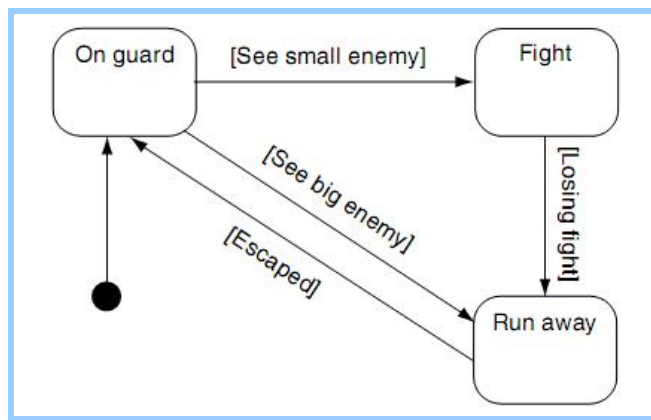
Слика 3.8. Стабло у коме се карактер зауставља

3.2. Коначни аутомати (state machines)

Често ће се карактери понашати на један од могућих начина. Радиће исту ствар док их неки догађај из окружења не натера да промене понашање. Чувар ће бити на свом месту док не види играча, када ће да га нападне.

Овакво понашање се може подржати помоћу стабала одлучивања, али лакше је користити технику створену за овакве ситуације: коначне аутомате. Ова техника заједно са скриптовањем чини већину система за доношење одлука у тренутним играма. Коначни аутомати воде бригу о свету око њих као и о сопственом унутрашњем стању.

Карактер је у једном стању у коначном аутомату. Акције или понашања су придружена сваком стању. Све док је карактер у том стању, вршиће исту акцију. Стања су повезана транзицијама. Свака транзиција води од једног стања ка другом и свака има скуп услова потребних за прелазак. Када се испуне услови, стање карактера постаје циљно стање одређено транзицијом. Каже се да су се услови окинули, и да је стање испаљено. На слици 3.9 је дат пример.



Слика 3.9. Структура FSM

У играма се оваква структура обично назива finite state machine (FSM).

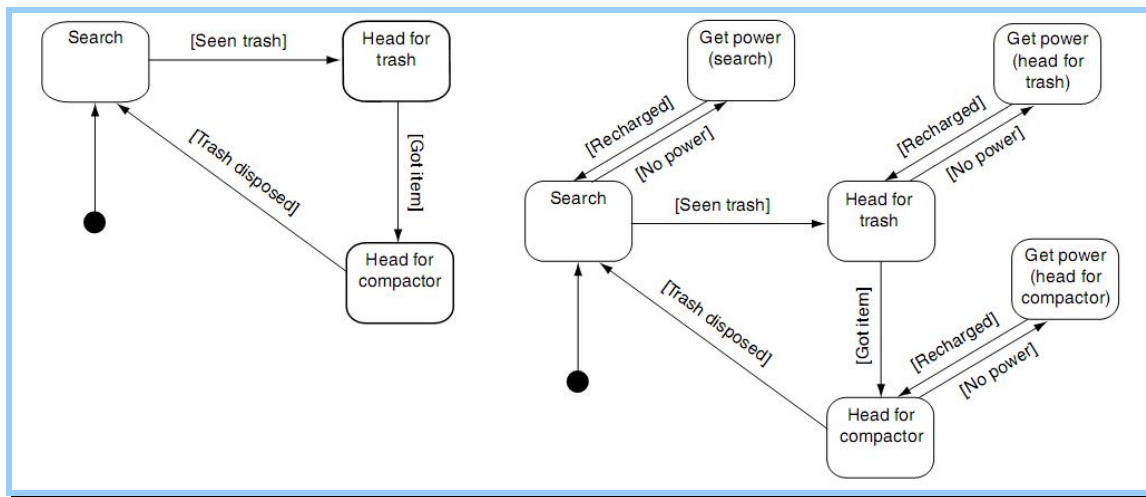
Потребно је да коначни аутомат води евиденцију о могућим стањима и о стању у којем је тренутно. За свако стање постоји и скуп транзиција из тог стања са одређеним условима. На сваку промену ажурира се коначни аутомат и проверава се да ли су испуњени услови неке транзиције. Транзиција чији су услови окинути се распореди за извршавање. Потребне акције у текућем стању се изврше и ако је потребно, прелази се у ново стање помоћу транзиције. Раздвајање окидања и извршавања транзиције омогућава да се и транзицијама придруже акције које ће се извршити у текућем стању.

Коначни аутомати траже меморију у којој чувају окинуту транзицију и тренутно стање. Комплексност је $O(1)$ у меморији, $O(m)$ у времену, ако је m број транзиција по стању.

Раније су скоро сви коначни аутомати били hard-coded. Правила за транзиције и извршавање акција су били део кода игре. Тај приступ постаје ређи, јер дизајнери нивоа добијају све више контроле на процесом прављења логике коначних аутомата (потребно је да подесе понашања карактера на нивоу који праве), али је и даље битан. Иако су лаке за писање, hard-coded коначни аутомати су јако тешки за одржавање. Такође, мана је ако се промени понашање једног карактера потребно је рекомпајлирати игру, што је проблем у великим пројектима где време копајлирања дуго траје. Комплексније структуре, као што су хијерархијски коначни аутомати, се тешко координишу користећи hard-coded коначне аутомате.

Нека понашања су незгодна за имплементацију. Размотримо ситуацију у којој имамо робота чији је посао да чисти подове. То се обавља на основу коначног аутомата. Може да тражи објекте који су бачени, покупи их и да их однесе до корпе. Нажалост, робот може остати без струје, па је потребно пре него што се то деси да се напуни на одређеној тачки. Без обзира шта тренутно ради, прекида акцију и када је напуњен треба да настави тамо где је стао. Период пуњења може омогућити играчу да се прошуња или да искључи целокупно напајање у згради и тако онеспособи робота.

У питању је механизам аларма: неки догађај прекине нормално понашање да би се одговорило на нешто битно. Ако се репрезентује у коначном аутомату води до дуплирања стања, илустровано на слици 3.10.

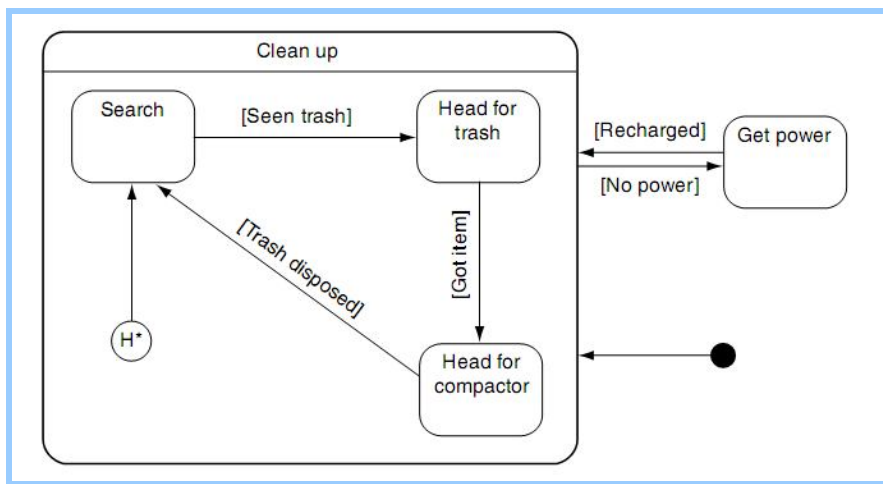


Слика 3.10. Дуплирање стања у механизму аларма

Са једним нивоом аларма ово није проблем, али ако желимо да робот има још битније понашање бежи када почне борба, ситуација се значајно компликује. После краја борбе би наставио пуњење, а после пуњења би наставио акцију коју је прекинуо.

Уместо комбиновања све логике у један коначни аутомат, логика се може поделити на неколико аутомата. Сваки алармни механизам има сопствени коначни аутомат и они су поређени у хијерархију, тако да се коначни аутомат ниже у

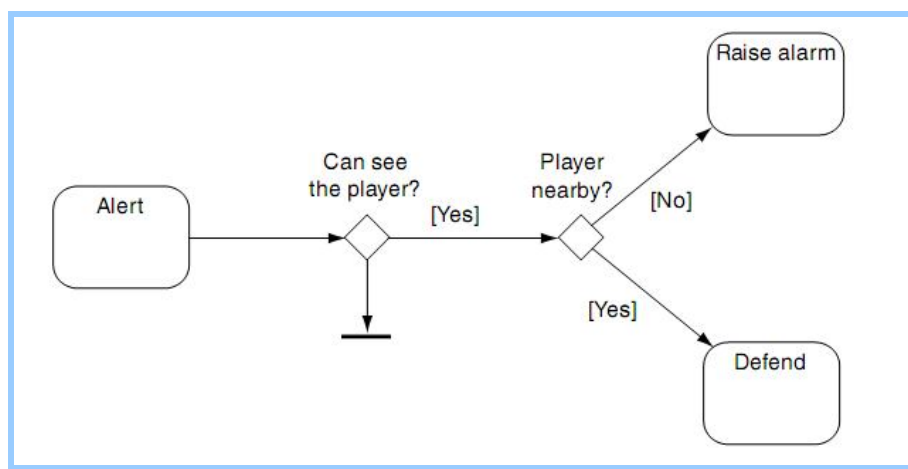
хијерархији разматра само када онај више и хијерархији не одговара на аларм. На слици 3.11 је дат хијерархијски аутомат који одговара слици 3.10.



Слика 3.11. Хијерархијски аутомат

Помоћу угњежавања смо направили хијерархијски коначни аутомат. Када се композитно стање први пут обилази H^* одређује у које подстање треба да се уђе. Ако се композитно стање већ раније обишло H^* одређује из ког стања настављамо. Из тог разлога се назива стањем историје. На тај начин водимо евиденцију о стању чишћења када робот пуни батерије. Када се напуни може да настави где је стао. Може се сматрати да се налазимо у више стања одједанпут, али због стриктне хијерархије никад нема дилеме које стање се користи.

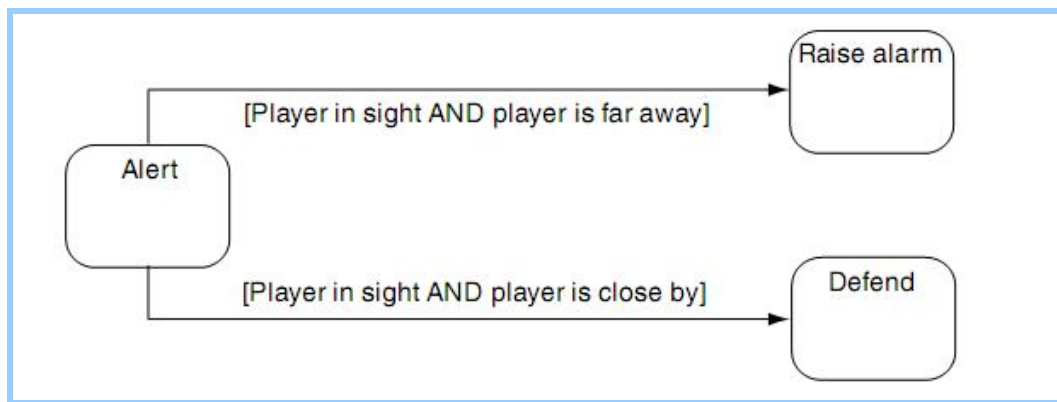
Могуће је комбиновати стабла одлучивања са коначним аутоматима. Стабла одлучивања представљају ефикасан начин да се обради скуп услова и то је корисно за одређивање да ли су услови транзиција у коначним аутоматима окинати. Приступ се могу комбиновати заменом транзиција из стања у стање са стаблима одлучивања. Листови стабла одлучивања уместо акција, представљају прелазе у нова стања. Пример је дат на слици 3.12.



Слика 3.12. Комбиновање стабла одлучивања са коначним аутоматима

Када је у стању узбуне, чувар има само једну могућу транзицију преко стабла одлучивања. Прво се одреди да ли види играча, уколико не види нема прелазе у ново стање. Ако види играча, на основу чињенице да ли је играч близу прелази се у једно од два могућа стања.

Да би се имплементирао исти случај без стабла одлучивања потребно је израчунати два комплексна услова и оба ће морати исте информације да рачунају (видљивост играча и његова близина), што може бити значајно спорије него имплементација са стаблом одлучивања. Илустровано на слици 3.13.



Слика 3.13. Имплементирање случаја без стабла одлучивања

3.3. Fuzzy логика

Услови и одлуке су биле тачно, нетачно, односно, веома црно – беле. Фази логику чини сет математичких техника које се баве сивим областима.

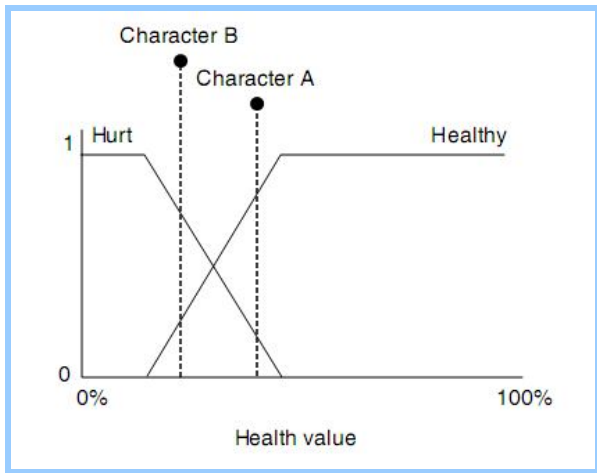
Ако се прави AI за карактера који се креће кроз опасну област, могу му се придружити два стања у приступу са коначним аутоматима : опрезан и сигуран. Када је опрезан шуњаће се, истовремено пажљиво осматрати. Када је сигуран, нормално се креће. Како се креће кроз ниво може често мењати понашање кроз ова два стања. То може деловати чудно. Помоћу фази логике можемо замути границу између ова два стања, дајући му читаву палету нивоа сигурности. Са фази логиком и даље можемо правити одлуке, као ходај полако када си опрезан, али и полако и опрезан могу имати опсег вредности.

Фази скупови. У традиционалној логици постоји појам предиката. Предикатом се означава неки квалитет или опис нечега. Карактер може бити повређен или не, немамо информацију колико је повређен. Фази логика предикату придружује вредност. Карактер може бити повређен са вредношћу 0.7 што је више од карактера који је повређен са вредношћу 0.3. Уместо да припада неком скупу или не, може се делимично припадати скупу и неке ствари ће више припадати од других. Ти скупови се називају фази скуповима, а нумеричка вредност се назива степеном припадности. Степен припадности са вредношћу 1 се даје нечему што потпуно припада фази скупу, а 0 нечему што је у потпуности ван скупа. За вредности припадности које су 1 и 0 правила традиционалне логике важе. Могу се користити и друге вредности за имплементацију (од 0 до 255 рецимо).

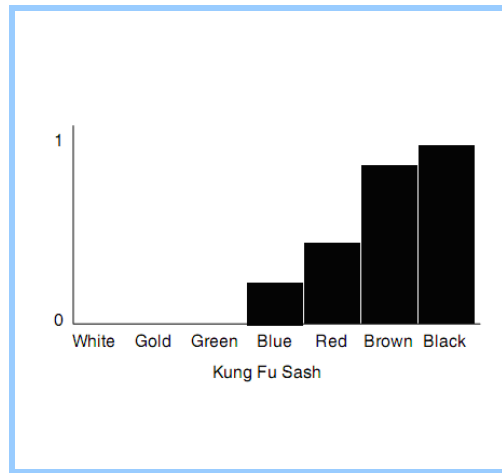
Припадност у више скупова. Било шта може бити члан више скупова истовремено. Карактер може бити и гладан и повређен. За разлику од традиционалне логике, карактер може бити и повређен и неповређен у исто време (са степенима припадности 0.5 и 0.5). Ограничење је да збир степена припадности за узајамно искључиве скупове буде једнак 1.

Фазификација. Фази логика ради искључиво са степенима припадности фази скуповима. Пошто се подаци у игри обично не чувају на тај начин, потребна је конверзија. Претварање података у степене припадности је фазификација, обрнут процес је дефазификација. Један приступ је нумеричка фазификација, помоћу које се нумеричка вредност претвара у степен припадности фази скуповима. Карактери могу имају број који означава колико су здрави који се може претворити у степен

припадности скуповима повређен и неповређен. Ово се постиже са функцијом припадности, која мапира улазну вредност у степен припадности. Илустровано на слици 3.14.



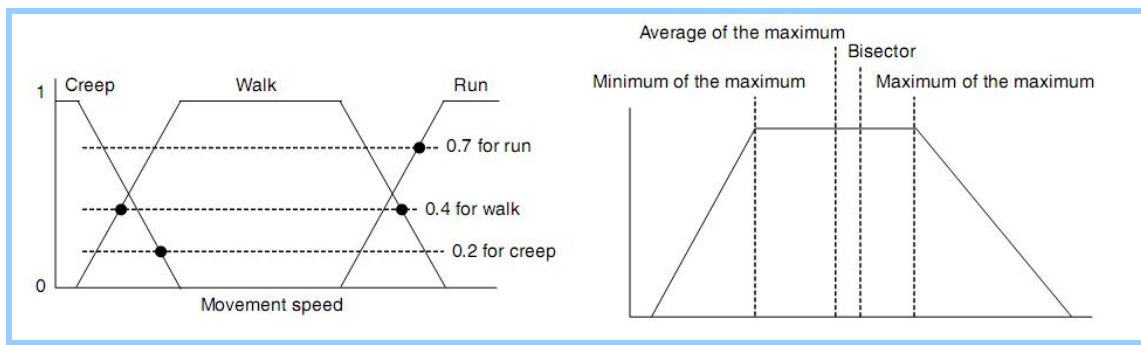
Слика 3.14. Функција припадности



Слика 3.15. Степени припадности

Често је потребно фазификовати boolean и enumeration вредности. Приступ је да се сачувају преодређене вредности степена припадности за сваки релевантан скуп. Ако би се фази скуп односио директно на boolean вредност, онда би степени припадности били 0 и 1. Слично је за enumeration вредности, где може постојати више могућности и свака има сопствену вредност степена припадности. Пример је на слици 3.15.

Дефазификација. После примене потребне фази логике, добили смо скуп степена припадности за фази скупове. Да би их претворили у корисне податке користи се техника дефазификације. Потребно је обрнути процес фазификације и претворити вредности степена припадности у једну излазну вредност. То се ради на основу функција припадности, када је потребно наћи вредност којој одговарају дате вредности степена припадности. Ово је ретко директно могуће. Ако имамо вредности степена припадности од 0.2, 0.4 и 0.7 за леву ситуацију са слике 3.16. не постоји вредност за кретање која ће их задовољити.



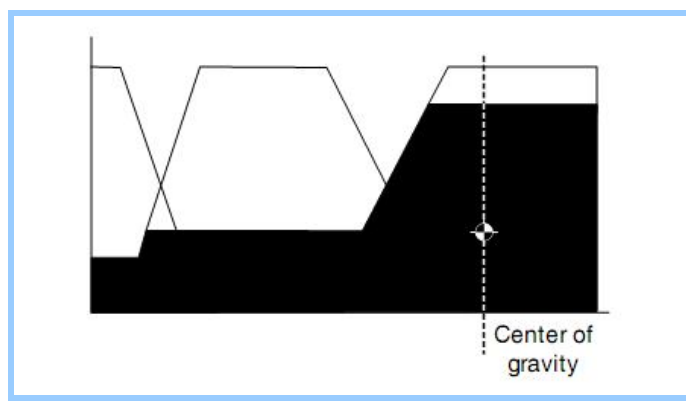
Слика 3.16. Дефазификација

Један приступ је дефазификација базирана на највећем степену припадности, којим се бира излаз на основу највећег степена припадности. У примеру лево са слике 3.16. би се бирала брзина која одговара трчању. Постоје 4 значајне тачке : минимална и максимална вредност када функција враћа 1, њихов просек и бисектор тачка површину коју заузима функција дели на једнаке делове, илустровано десно на слици 3.16. Ово је брза техника, али даје грубу дефазификацију. Карактер са стапенима припадности 0 creep, 0 walk, 1 run ће имати исту излазну брзину као и карактер са 0.33 creep, 0.33 walk, 0.34 run.

Решење је користити техинку мешања базрану на припадности, којом се користи и сама вредност поред степена припадности. Тако да карактер са степенима припадности 0 creep, 0 walk, 1 run ће користити брзину за трчање, док ће карактер са 0.33 creep, 0.33 walk, 0.34 run користити брзину одређену са $0.33 * \text{брзина шуњања} + 0.33 * \text{брзина ходања} + 0.34 * \text{брзина трчања}$. Битно је да су фактори нормализовани да се не би добила неисправна брзина.

Техника центра гравитације узима у обзир све вредности степена припадности, уместо само највеће. Прво се свака функција припадности одсеца на вредности степена припадања за одговарајући скуп. За пример лево са слике 3.16 би се за степен припадности од 0.4 функција припадности одсекла изнад 0.4. Илустовано на слици 3.17. За излазну вредност се узима центар масе добијене фигуре. За интеграцију којом се одређује центар масе је потребно време и не може се урадити унапред као за одређивање вредности бисектора.

Иако се техника центра гравитације често користи, релативно је комплексна за имплементацију а резултати добијени техником мешања су обично довољно добри и значајно бржи за израчунавање.



Слика 3.17. Техника центра гравитације

Да би се добила boolean вредност, користи се један фази скуп и гранична вредност. Ако је степен припадности мањи од граничне вредности, излаз се сматра нетачним, иначе је тачан. Ако постоји више фази скупова који доприносе одлуци, онда се они комбинују користећи фази правила у један скуп, који се онда дефазификује за излазну boolean вредност.

Комбиновање чињеница. Фази логика је слична традиционалној логици. Помоћу логичких оператора (као AND, OR и NOT) се врши комбиновање једноставних чињеница да би се добиле комплексније. За разлику од традиционалне логики, свака чињеница није само тачна и нетачна, већ има и нумеричку вредност којом се одређује степен припадности одговарајућем фази скупу. У традиционалној логици се користи табела истинитости. У фази логици сваки оператор има нумеричко правило помоћу којег се може израчунати степен истинитости на основу степена истинитости сваког улаза. Правила су дата на слици 3.18.

Expression	Equivalent	Fuzzy Equation
NOT A		$1 - m_A$
A AND B		$\min(m_A, m_B)$
A OR B		$\max(m_A, m_B)$
A XOR B	NOT(B) AND A	$\min(m_A, 1 - m_B)$
	NOT(A) AND B	$\min(1 - m_A, m_B)$
A NOR B	NOT(A OR B)	$1 - \max(m_A, m_B)$
A NAND B	NOT(A AND B)	$1 - \min(m_A, m_B)$

Фази правила комбинују степене припадности да би генерисали нове степене припадности другим фази скуповима. Имамо правило које каже : “Ако сам близу кривине и брзо се крећем онда треба да закинем.” Правило се односи на два улазна скупа : “Близу кривине” и “Брзо се крећем”. Потребно је израчунати степен припадности трећем скупу “Треба да закинем”. Са слике 3.18 се види

$$m_{(Should\ Brake)} = \min(m_{(Close\ to\ the\ Corner)}, m_{(Travelling\ Quickly)})$$

Ако би степен припадности скупу “Близу кривине” био 0.6 и степен припадности скупу “Брзо се крећем” 0.9, онда би степен припадности скупу “Треба да закинем” био 0.6.

Коришћење фази логики за доношење одлука

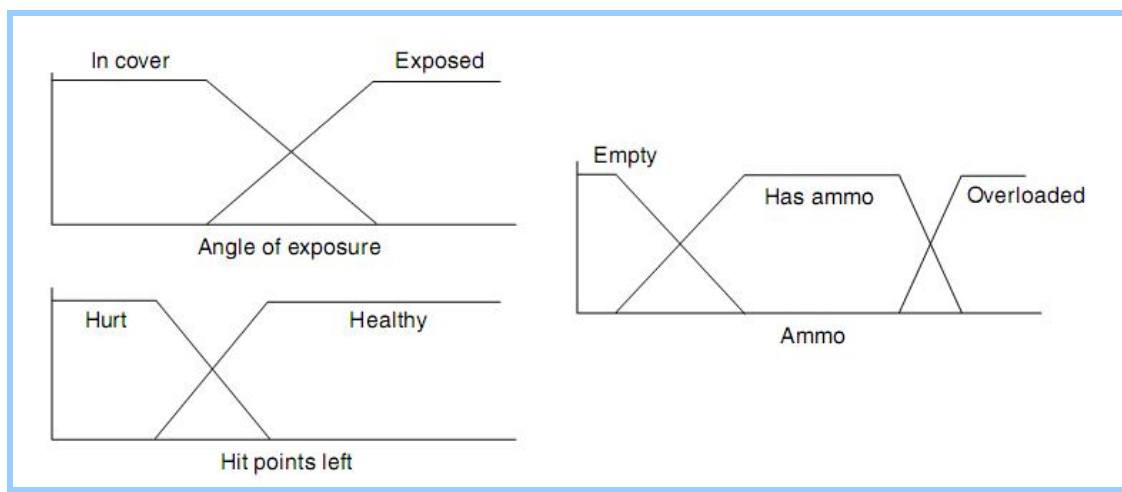
Може се користити у систему који садржи традиционалну логику, засновну на AND, OR и NOT операторима. Може се користити да одреди да ли су транзицијама окинати услови. Такође, може се користити у системима заснованим на праилима (rule-based).

Разматраће се структура за доношење одлука која користи фази логику и само AND оператор. Припада подобласти фази логики која се зове фази контрола. Обично се користи у индустријским контролерима који врше акције на основу скупа улаза.

У многим ситуацијама постоји скуп акција које се могу предузети и није лако одредити која је најбоља. Посебно је тешко доћи до одлуке ако акције нису искључиво on / off типа, већ се могу примењивати у неком опсегу. Ако се вози аутомобил, могуће је скретати и убрзавати са одређеним интензитетом. Понекад се екстремни могу одмах елиминисати, али интензитет одлуке у осталим ситуацијама је проблем. То ће разликовати одличне возаче од просечних.

Помоћу система за доношење одлука базираним на фази логици можемо обрадити такве ситуације. Циљ је добити разумне, мада не увек оптималне, одлуке.

Систем за доношење одлука има било који број улаза, који могу бити нумеричке, boolean и enumeration вредности. Сваки улаз се мапира у фази скуп помоћу функције припадности. Илустровано на слици 3.19.



Слика 3.19. Мапирање улаза у фази скуп помоћу функције припадности

Поред улазних стања, имамо и скуп излазних стања. Она су нормална фази стања, која представљају различите акције које карактер може да предузме.

Повезивање улазних и излазних стања са врши преко фази правила. Правила су обично типа :

input 1 state AND ... AND input n state THEN output state

Чест захтев је постојање комплетног скупа правила, односно, једног правила за сваку комбинацију стања улаза. Да би се генерисао излаз, потребно је проћи кроз свако правило и израчунати степен припадности сваком излазном стању. Пошто су сва правила формирана помоћу AND оператора, то значи да је потребно наћи минималну вредност степена припадности улаза у правилу. Коначни степен припадности неком излазном стању ће бити максимална вредност добијених степена припадности из применљивих правила (OR). На слици 3.20 је дат скуп правила, степени припадности и резултати тих правила.

corner-entry AND going-fast THEN brake	Corner-entry: 0.1
corner-exit AND going-fast THEN accelerate	Corner-exit: 0.9
corner-entry AND going-slow THEN accelerate	Going-fast: 0.4
corner-exit AND going-slow THEN accelerate	Going-slow: 0.6
$\text{Brake} = \min(0.1, 0.4) = 0.1$ $\text{Accelerate} = \min(0.9, 0.4) = 0.4$ $\text{Accelerate} = \min(0.1, 0.6) = 0.1$ $\text{Accelerate} = \min(0.9, 0.6) = 0.6$	

Слика 3.20. Скуп правила, степени припадности и резултати правила

Финална вредност за кочење је 0.1, а за убрзавање је максимална вредност од добијених 0.4, 0.1 и 0.6, дакле, 0.6. Не морају се рачунати све вредности за сва правила. После рачунања другог правила знамо да је убрзање најмање 0.4, па када рачунајући треће правило видимо вредност 0.1 знамо да ово правило неће имати утицаја на коначну одлуку, следи да се може престати разматрање овог правила.

После одређивања вредности степена припадности за излазна стања, може се извршити дефазификација да би одредили шта да радимо (може се вратити нумеричка вредност која нам говори колико да убрзавамо или кочимо).

Као рестрикција, обично се у скуповима користе правила чији улаз садржи сва стања и да су у скупу правила присутне све комбинације стања улаза. Ако се користе таква правила, у овај скуп се могу пресликати било какав скуп правила (који није контрадикторан). Међутим, овај приступ није скалабилан. Ради добро са малим бројем улаза и са малим бројем стања по улазу. За веће системе, број правила које је потребно направити је огроман, па је потребно користити мање правила или Combs метод за креирање правила.

Combs метод. Своди се на трансформацију већег правила у мања :

IF a_1 AND ... AND a_n THEN c
IF a_1 THEN c
⋮
IF a_n THEN c

Од правила која су имала све могуће комбинације стања смо добили једноставна правила са једним стањем у IF делу и једним стањем у THEN делу.

Уместо великог броја правила, постојаће исти број као и број улазних стања. Скуп правила се може превести у једноставнија али могуће су неконзистентности у самом скупу. На примеру на слици 3.21 после трансформације не знамо да ли треба да кочимо или убрзавамо ако идемо брзо. Дакле, док се једно правило може увек декомпоновати то се не може тврдити за сваки генерални скуп правила. Да би се користио Combs метод потребно је правила дизајнирати специфично са њим у виду. Тако се смањује флексибилност у прављењу правила. Пример са слике 3.20 се може представити у формату као са слике 3.22 која за исте улазе даје резултат који ће резултовати у сличном убрзању.

IF corner-entry AND going-fast THEN brake
IF corner-exit AND going-fast THEN accelerate
IF corner-entry THEN brake
IF going-fast THEN brake
IF corner-exit THEN accelerate
IF going-fast THEN accelerate

Слика 3.21. Примеру понашања у ситуацији после трансформације

corner-entry THEN brake	Corner-entry: 0.1
corner-exit THEN accelerate	Corner-exit: 0.9
going-fast THEN brake	Going-fast: 0.4
going-slow THEN accelerate	Going-slow: 0.6
Brake = 0.4	
Accelerate = 0.9	

Слика 3.22. За исте улазе добија се резултат који ће резултовати у сличном убрзању

Фази коначни аутомати. Фази коначни аутомат може користити фази стања уместо нормалних стања или користити фази логику за окидање услова транзиција. Може користити оба приступа.

Обични коначни аутомати су корисни ако је карактер или у једном или у другом стању. Фази приступ се користи ако желимо да омогућимо да карактер буде у више стања у исто време.

У конвенционалном аутомату стања водимо евиденцију о текућем стању преко једне вредности. У фази коначном аутомату можемо бити у више стања у исто време са одређеним степенима припадности. Свако стање има сопствену вредност степена припадности. Да би одредили која стања су тренутно активна (имају степен припадности већи од нуле) потребно је погледати сва стања.

У свакој итерацији коначног аутомата, транзиције које припадају свим активним стањима добијају прилику да се изврше. Прва транзиција у сваком активном стању се изврши. Ово значи да је могуће да се догоди више транзиција у једној итерацији.

Преко транзиције се може доћи до више нових стања. Транзицији је придружен степен транзиције. Потребан је, јер се транзиција може представити као : у почетном смо стању AND услови транзиције су окинати ПРОУЗРОКУЈЕ активирање крајњег стања. Степен припадности циљног стања се добија помоћу AND операције над степеном припадности текућег стања и степена транзиције. Такође, може постојати

више транзиција које воде до истог крајњег стања, што значи да ће се коначни степен припадности добити помоћу OR операције. Слично се гледа и за почетно стање, када се транзиција изврши почетно стање се искључи.

Акције у фази логици су обично повезане са дефазификованим вредностима и дефазикација обично користи више стања. Могуће је одвојити захтеве за акцијама од коначног аутомата и користити екстерни дефазификациони процес који ће да одреди потребну акцију.

3.4. Понашање на основу циља (goal-oriented)

До сада смо се фокусирали на приступе који су реактивни : скуп улаза је доступан карактеру и он на основу њих бира одговарајућу акцију. Нема разматрања жеља или циљева. Карактер само реагује на улаз.

Ако је могуће, желимо да изгледа да карактер има циљеве или жеље, чак иако користимо једноставне технике за доношење одлука. Ловац ће желети да улови дивљач, тражиће је по природи и ако је нађе напаћи ће је. Исти ловац има жељу да преживи па ако наиђе на неку опасност сакриће се. Структура којом се доноси одлука ће опет реаговати на улаз, али понашање карактера не мора изгледати искључиво реактивно.

Карактери треба да покажу своје емоционално и физичко стање бирањем одговарајућих акција. Ако су гладни треба да једу, ако су уморни да спавају, да причају са пријатељима када су усамљени и слично. Може се користити стабло одлучивања да би се одабрала одговарајућа акција на основу емоционалног и физичког стања, али ако постоји мноштво могућих опција стабло ће бити огромно. Бољи приступ је направити скуп могућих акција и бирати ону акцију која најбоље задовољава тренутне потребе. Тај приступ представља понашање на основу циља.

Понашање на основу циља покрива било коју технику која узима у обзир жеље и циљеве. Могу се креирати помоћу продукционих система. Разматраће се техника која може да бира из скупа могућих акција ону која задовољава текуће потребе.

Карактер може имати било који број циљева, који се такође називају мотивима. Такође, било који број мотива може бити тренутно активан. Сваки мотив има придружен степен важности (настојање), који је репрезентован бројем. Мотиви са великим настојањем ће имати више утицаја на понашање карактера. Карактер ће покушавати да испуни циљ или да му смањи степен важности. Неке игре дозвољавају да се циљеви у потпуности испуне, док су у другима циљеви увек присутни, и само им се мења степен важности. Нулта вредност степена важности одговара испуњеном циљу.

Мотиви и циљеви су генерално врло различити али се у овом контексту користе као синоними. Циљеви се могу имплементирати и без степена важности, али се тако теже бирају циљеви на које да се фокусирамо.

Поред скупа циљева потребан је скуп акција које се могу бирати. Те акције се могу генерисати глобално (карактер увек може да се прошећа) или на основу окружења (ако постоји кауч у соби, карактер може да легне на њега). Дакле, доступне акције зависе од стања игре. Циљ који акција треба да задовољи може бити удаљен неколико корака. Ако је карактер гладан, може покупити комад сировог меса, али тек пошто га стави у рерну и испече може да га поједе и задовољи глад. У неким играма се једна акција састоји од секвенце атомских акција.

Имамо скуп акција и скуп циљева. Акције испуњавају различите циљеве.

Goal: Eat = 4 Goal: Sleep = 3 Action: Get-Raw-Food (Eat – 3) Action: Get-Snack (Eat – 2) Action: Sleep-In-Bed (Sleep – 4) Action: Sleep-On-Sofa (Sleep – 2)
--

Слика 3.23. Технике за доношење одлука ради одабира акције

Може се користити више различитих техника за доношење одлука да би се одабрала акција. Једноставан приступ је да се одабере најхитнији циљ (циљ са највећом вредношћу степена важности) и наћи акцију која га у потпуности испуњава или му највише смањује степен важности. У примеру са слике 3.23 то би била друга акција, којом се налази сирова храна (што може водити ка кувању те хране и јелу). Сама промена степена важности придружена акцијама је хеуристичка процена везана за корисност. Ако више акција има исту хеуристику, може се насумично одабрати једна од акција.

Претходни алгоритам је прво бирао најбитнији циљ, па је онда бирао најбољи начин да му смањи степен важности. Бирање најбитнијег циља може имати споредне ефекте на друге циљеве. У примеру са слике 3.24 скала је до 5.

Goal: Eat = 4 Goal: Bathroom = 3 Action: Drink-Soda (Eat – 2; Bathroom + 3) Action: Visit-Bathroom (Bathroom – 4)
--

Слика 3.24.

Карактер који је гладан и жели да иде у тоалет вероватно не жели да пије сок. Сок ће му ублажити глад али ће повећати потребу за тоалетом на максималну вредност. Нежељена интеракција може бити и фатална. Карактер у ратној игри који жели да покупи оклоп може упасти у заседу. Потребно је узимати и споредне ефекте одлука у обзир.

То се може урадити додавањем нове вредности која представља незадовољство карактера. Рачуна се на основу свих вредности степена важности, где велике вредности степена важности чине карактера незадовољнијим. Циљ карактера је да смањи укупно незадовољство. Више се не фокусира на један циљ, него на читав скуп. Могу се само сабрати вредности степена важности да би се добила вредност незадовољства, али је бољи приступ скалирати вредности пре сабирања тако да велике вредности степена важности доприносе више. Квадрирање битних вредности је обично довољно.

Пример са слике 3.24 може се представити као пример са слике 3.25.

Goal: Eat = 4 Goal: Bathroom = 3 Action: Drink-Soda (Eat – 2; Bathroom + 2) ■ afterwards: Eat = 2, Bathroom = 5: Discontentment = 29 Action: Visit-Bathroom (Bathroom – 4) ■ afterwards: Eat = 4, Bathroom = 0: Discontentment = 16
--

Слика 3.25

Да би се донела одлука, редом се разматрају могуће акције. Прави се предикција за резултујуће незадовољство пошто се изврши акција. Акција која води до најмањег незадовољства се бира. Види се са слике 3.25 да се сада бира друга акција, која води до тоалета. Незадовољство је само скор који желимо да минимизујемо, може се називати било како. У академском AI-у се назива energy metric (везано је за потребну енергију).

До сада нисмо разматрали време потребно да се нека акција изврши. Да би се донела добра одлука за потребну акцију, карактер треба да зна колико је времена потребно да се акција изврши. Ако карактер нема енергије, можда му је боље да поједе чоколадицу него да спава 8 сати. Самим акцијама је придружено и време потребно за комплетирање, што ће систем за одлучивање користити. У случају секвенца акција време једне акције представља само део времена потребног да се задовољи циљ (пример са сировим намирницама које је потребно спремити). Тајмирање је подељено на две компоненте. Акције обично захтевају да се дође до одређене локације пре него што могу да се започну, такође, акције захтевају време да се комплетирају. Само време потребно да се дође до потребне локације може бити засновано на хеуристици (ваздушни пут) или бити тачно израчунато (на основу алгоритма за налажење пута). Да би узимали у обзир време може се додати као фактор у степен незадовољства или користити принцип да се прво извршавају краће акције.

У неким играма вредности степена важности циљева се мењају са временом. Карактер може постајати све уморнији и гладнији како време тече. Када је то ситуација, акција не утиче само на неке циљеве, већ протоком времена и други ће се природно мењати. Ако знамо како се током времена мењају степени важности циљева можемо факторисати те промене у рачунање степена незадовољства. Илустровано на примеру на слици 3.26.

```
Goal: Eat = 4 changing at + 4 per hour
Goal: Bathroom = 3 changing at + 2 per hour
Action: Eat-Snack (Eat - 2) 15 minutes
■ afterwards: Eat = 2, Bathroom = 3.5: Discontentment = 21.25
Action: Eat-Main-Meal (Eat - 4) 1 hour
■ afterwards: Eat = 0, Bathroom = 5: Discontentment = 25
Action: Visit-Bathroom (Bathroom - 4) 15 minutes
■ afterwards: Eat = 5, Bathroom = 0: Discontentment = 25
```

Слика 3.26

У примеру ће карактер прво појести нешто пре него што буде ишао у тоалет. Бираће између ужине и главног јела. Пошто се главно јело дуго спрема и потреба за купатилом ће бити велика, бира се ужина. Одлазак директно у тоалет није најбоља опција због глади.

У неким играма је промена степена важности циљева фиксна и одређена од стране дизајнера. Чак иако степен промене није константан, већ варира, игра може знати који је. Дакле, може се директно дохватити исправна вредност и користити. У неким ситуацијама директан приступ није могућ. Ако се карактер у ратној игри повреди, не можемо унапред знати колико ће бити повређен, већ то зависи од дешавања у игри.

У овој ситуацији можемо апроксимирати степен промене. Једноставан начин да се то уради је да се прате промене у степенима важности циљева. Сваки пут када се покрене алгоритам проверимо колико су се промениле вредности и на основу тога вршимо процену. Илустровано на слици 3.27.

```
rateSinceLastTime = changeSinceLastTime / timeSinceLast
basicRate = 0.95 * basicRate + 0.05 * rateSinceLastTime
```

Слика 3.27

Процена се добија сабирањем помножене старе вредности са фактором 0.95 и промене са фактором 0.05. Могу се користити било какве вредности тежинских фактора, ако су

у збиру једнаки јединици. Овако је омогућено једноставно учење. Ако карактер губи, почеће да се понаша све дефанзивније, јер очекује да ће га свака акција коштати здравља, док ако побеђује понашаће се све храбрије.

Предпостављали смо да су акције на избору само када је карактер способан да их изврши. Разматрали смо метод који узима у обзир ефекте акција на многе циљеве и бира акцију која свеукупно даје најбољи резултат.

Међутим, постоји проблем, ако акције омогућују или онемогућују друге акције.

Размотримо фантазијски свет, у којем чаробњак има енергетске кристале на располагању. Моћне магије захтевају више енергетских кристала. Чаробњаку је потребно лечење али би желео да се одбрани од чудовишта. На слици 3.28.

Goal: Heal = 4

Goal: Kill-Ogre = 3

Action: Fireball (Kill-Ogre – 2) 3 energy-slots

Action: Lesser-Healing (Heal – 2) 2 energy-slots

Action: Greater-Healing (Heal – 4) 3 energy-slots

Слика 3.28

Најбоља комбинација је мало се излечити па бацити ватрену лопту, користећи 5 магичних слотова. Међутим, по тренутном алгоритму чаробњак ће одабрати веће лечење, јер га то оставља у бољој ситуацији. Ако је имао 5 енергетских кристала, више не може да се брани од чудовишта. У овом примеру, покушаваћемо да минимизујемо број искоришћених магичних слотова. Да би омогућили карактеру да предвиђа ефекте и искоришћава секвенце акција, потребно је додати неки ниво планирања.

Основни алгоритам разматра ефекте само једне акције. Акција даје индикацију како ће мењати вредности осталих циљева, и на основу тога се може израчунати степен незадовољства. Може се проширити и на секвенце акција. Ако желимо да нађемо најбољу секвенцу четири акције, можемо израчунати степен незадовољства за све комбинације акција и одабрати ону секвенцу са најмањим степеном незадовољства. Ово је основа новог алгоритма, који покушава да нађе најбољу секвенцу акција, која дугорочно задовољава циљеве. Проблем у овом алгоритму је сам број могућих комбинација акција, друго, потребно је знати какве ће акције бити на располагању када се акција изврши. Потребно је одредити будуће стање света за акцију и користити то стања да одредимо могуће акције у том стању. Такође, када се одређује исход акције потребно је одредити све ефекте. Да би се ово омогућило користи се модел света, помоћу којег се представља стање света у облику којим се може лако манипулисати без мењања стварног стања. Сама репрезентација света може бити поједностављена и обично се чува само разлика у односу на тренутно стање света. На тај начин алгоритам податак прво тражи међу разликама па тек онда гледа стварно стање.

Из модела света (који може да осликава стварно стање или веровања карактера) добијамо могуће акције. Планирање се ограничава максималном дужином. Алгоритам ради итеративно, обрађујући једно стање света по итерацији. Ако је тренутна дубина једнака максималној рачуна се степен незадовољства и ако је најбољи сачува се секвенца акција. Ако је дубина мања од максималне алгоритам узима следећу могућу необрађену акцију, генерише ново стање света на основу те акције и повећа тренутну дубину. Ако нема више могућих акција онда се смањи дубина за један. Када дубина буде једнака нули завршили смо претрагу.

Коришћењем хеуристике се може убрзати алгоритам. Потребно је израчунати степен незадовољства у свакој итерацији и сачувати га. Ако је степен незадовољства већи него на претходној дубини, онда се може игнорисати текући модел, па се смањује тренутна дубина и проба друга акција.

3.5. Продукциони системи (rule-based)

Продукциони системи су се повремено користили и у играма у последњих 15 – ак година. Слична понашања се могу добити и коришћењем једноставнијих техника, као што су стабла одлучивања и коначни аутомати. Предност им је када карактери треба да доносе одлуке на начине који се не могу лако предвидети и имплементирати у стаблу одлучивања.

Продукциони системи се састоје од: базе са чињеницама и скупа IF – THEN правила. Правила чији су услови испуњени се могу применити, каже се да су им услови окинати, а сама правила се онда испаљују. Многи продукциони системи имају и арбитра, као трећи компоненту, који одређује које се правило чији су услови окинати може извршити. Илустровано на слици 3.29.



Слика 3.29. Продукциони системи

Captain's health is 51
Johnson's health is 38
Sale's health is 42
Whisker's health is 15
Radio is held by Whisker

Слика 3.30. База података
о здрављу људи у тиму

Услови правила се проверавају са базом чињеница, успешно поклапање окида услове правила. Сам услов правила се типично састоји од чињеница које су повезане логичким операторима, као што су AND, OR и NOT. У примеру на слици 3.30 имамо базу података о здрављу људи у тиму и чињеницу код кога је радио. Једно могуће правило које врши акцију ако су му услови испуњени је :

IF Whisker's health is 0 AND Radio is held by Whisker
THEN Sale: pick up the radio

Покупити радио је акција у игри: продукциони систем је као доносилац одлука одлучио да је потребно извршити акцију. Игра ће да одлучи да ли ће акција успети (карактер који је имао радио је можда пао са литице) и ажурираће базу знања ако успе. Неке чињенице су везане за стање игре, док су неке везане само за AI и немају утицаја у остатку игре. На слици 3.31. прва три су везана за стање игре, док је последње везано за AI. Дато је и правило које ажурира циљ пилота, када у патроли види непријатеље. Није било неопходно уклонити чињеницу везану за патролирање ако желимо да настави да патролира после бробе.

1500 kg fuel remaining	IF number of sighted enemies > 0 and currently patrolling
100 km from base	THEN
enemies sighted: Enemy 42, Enemy 21	remove(currently patrolling)
currently patrolling	add(attack first sighted enemy)

Слика 3.31. Пример акције у игри

Дакле, желимо да комбинујемо правила која резултују акцијама у игри са онима који само ажурирају стање у бази.

Користи се директно и повратно уланчавање. Директно уланчавање се чешће користи у играма од повратног уланчавања. Директно уланчавање почиње са базом

чињеница и покушава да примени правила нађе нове чињенице. У контексту игре, примењује правила која резултују у промени стања базе знања (или директно или мењањем стања игре преко акција карактера).

Повратно уланчавање покушава да одреди да ли је циљ у упиту испуњен. То ради тако што покушава да нађе чињенице које испуњавају циљ. Прво гледа саме чињенице, па онда правило чији THEN део одговара циљу упита, где ће услови тог правила постати нови подциљеви и то се ради рекурзивно док се не дође до одговора.

Потребно је на погодан начин чувати чињенице у бази знања, да би се могле једнозначно идентификовати. Илустровано на слици 3.32.

Captain's-weapon = rifle
Johnson's-weapon = machine-gun
Captain's-rifle's-ammo = 36
Johnson's-machine-gun's-ammo = 229

Слика 3.32. Чување чињенице у бази знања

Да би сазнали колико муниције има капетан потребно је проверити да ли постоје чињенице везане за капетанову муницију за пушку, за митраљез, за пиштољ и слично. Ово није најпогоднији начин. Може се користити хијерархијски формат за податке, где би та структура садржала конкретну вредност или скуп таквих структура. Подаци су угњеждени на било којој дубини. Тако се могу чувати сви подаци за неког карактера или за сва оружја у игри.

Почиње се са базом знања која садржи неке чињенице. Потребне су функције којима ће се пребацити подаци из тренутног стања игре у базу. Додатни подаци се могу убацили у базу, као што је интерно стање карактера. Скуп правила је доступан. IF део садржи услове правила повезане са логичким операторима AND, OR, NOT, XOR и сличним. Предпоставићемо да имамо правила везана за акције. Алгоритам ради у итерацијама, а база се може мењати или директно преко правила или преко последица акција. Систем проверава да ли постоји правило чији су услови окинути. Ако нађе такво правило изврши придружене акције.

Ово је алгоритам који је погодан само за једноставне продукционе системе, јер проверава сваку могућност, за комплексније се користи Rete алгоритам. Rete алгоритам представља индустријски стандард за упаривање услова правила.

Неколико правила може да се окине у бази знања у исто време. Свако правило може да се примени, али само једно може да се изврши. Ако су правила поређана по приоритету можемо увек да прво правило које задовољи услове извршимо. Генерално, арбитар у продукционом систему одлучује које правило треба да се изврши ако имамо више кандидата.

Један приступ је први могући. Правила су поређана по приоритету, ранија правила имају већи приоритет. Прво правило које задовољи услове се извршава. Може настати озбиљан проблем ако правило не уноси промене у бази, тада ће се то правило стално окидати. То може бити понашање које се тражило, али може довести до проблема са бесконачним понављањем. Проблем се може ублажити ако окинуто правило суспендујемо (уколико не треба да се стално окида), и скидамо суспензију када се база знања промени.

Овај приступ и даље може доводити до нежељене ситуације, јер се стање игре обично често мења, али чињенице које доводе до проблема могу бити стабилне. Потребно је водити евиденцију о чињеницама које окидају правило, и суспендовати правило док се оне не промене (ако је то тражено понашање).

Други приступ је последње коришћено. Користи се листа правила, где се редом разматрају правила из листе и прво правило које задовољи услове се изврши. Међутим, оно се ставља на крај листе. Овај начин је праведнији, јер даје могућност свим правилима да се изврше. Овде нема проблема као са претходним приступом.

У насумичном приступу се ако се окине више правила бира једно на случајан начин и дозвољава му се да се изврши. Овај арбитар ће морати да направи листу свих окинutih правила, и тек онда ће да бира једно. Претходни приступи су налазили прво правило које задовољава и њега извршавали. Ова шема је мање ефикасна.

Може се применити приступ најспецифичнијег услова. Ако се услови правила могу лако задовољити, онда ће се правило често извршавати. Специфичнија правила се преферирају у односу на општа правила. Ако су услови представљени преко boolean вредности, њихов број је добар индикатор колико је неко правило специфично. Разлика у односу на друге технике је у томе што се правила поређају по специфичности пре коришћења. За сам избор се може користити принцип први могући.

Могуће је динамички мењати приоритете. Динамички приоритети се могу враћати преко сваког правила на основу тога колико је акција придружена правилу битна у тренутној ситуацији. Ако имамо чињеницу да карактер нема прибор за прву помоћ, а постоји правило које каже да ако немамо прибор почнемо да га тражимо. Ако је карактер тренутно здрав, то правило ће вратити низак приоритет, односно, тражиће прибор ако нема алтернативе. У другом случају, када је веома рањен, правило враћа висок приоритет, што ће га натерати да прекине тренутне акције и почеће потрагу. Овакво понашање се може имплементирати помоћу више правила (једно за доброг здравља и једно за врло повређен). Коришћењем динамичких приоритета је омогућено да правило постепено постаје све значајније. Сам арбитар ће наћи сва правила која се могу извршити и за свако ће добити приоритет. На основу највећег приоритета бира правило које ће се извршити. Ово је најфлексибилнији, али и најспорији од представљених алгоритама арбитрације.

Постоје разне оптимизације и технике које се користе у продукционим системима, међутим, само један мали део је применљив на продукционе системе који се користе у играма.

У играма је често потребно баратати са великим скупом правила. Ако би се продукциони систем користио за тимски AI у потезној стратегији, скуп правила би био огроман, и са сваком новом игром из те серије би се повећавао. Нова правила би омогућавала компјутеру да користи нове опције, које су додаване у новим играма. Такође, нова правила би била потребна да би се покривањем више могућих ситуација AI побољшао. Све је ово водило ка непрегледном скупу правила, тако да су и перформансе су биле угрожене.

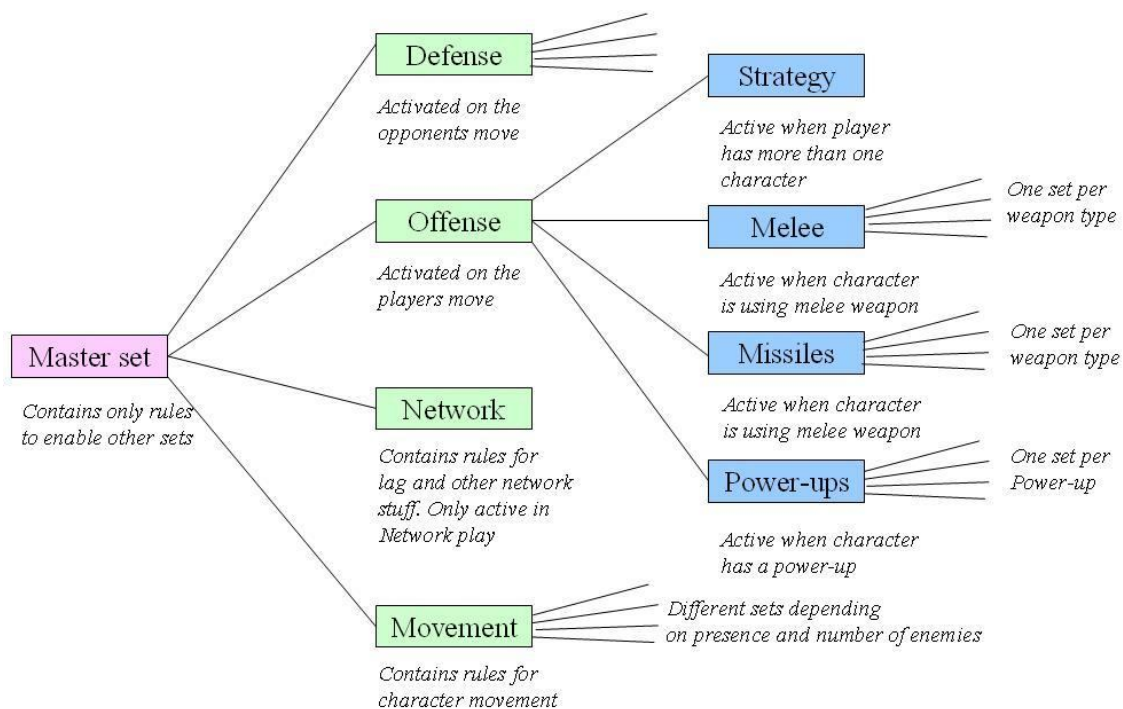
Решење је груписати правила. Правила из сваког скупа би се могла убацивати и избацивати из разматрања. Само би се правила из активних скупова могла извршавати. Правила из неактивних скупова немају прилику да се испале. Овај приступ је смањио број правила која се требају разматрати са неколико хиљада на пар стотина активних правила. Ова техника се користи у многим великим продукционим системима.

Продукциони систем има један скуп правила која су увек активна. Унутар њега је било који број правила и било који број скупова осталих правила. Правила на нивоима испод се могу укључивати и искључивати из разматрања. Активирање и деактивирање се може вршити преко саме игре или преко акција правила. Типично, сваки скуп правила на високом нивоу садржи неколико правила и скупова правила, који се могу активирати или деактивирати на основу акција правила. Цео систем је организован у хијерархију, тако да скупови правила на ниском нивоу садрже правила која обављају конкретан посао. Остали углавном служе за активирање и деактивирање скупова правила.

У свакој итерацији продукционог система, највиши ниво хијерархије ради на нормалан начин и доставља једно правило ако су му услови окинuti. Другачији метод арбитрације је могуће имати у сваком скупу у хијерархији. На тај начин имамо велики степен флексибилности на располагању.

На једној тачки хијерархије је могуће укључити скуп правила који одговара опреми који има карактер. Само активирање одговарајућег скупа је зависило од саме игре. На другој тачки хијерархије постоје скупови правила који су одговорни за

кретање карактера у близини непријатеља или када нема опасности. Пошто су информације о суседним непријатељима додате у базу, та два скупа смо укључивали и искључивали на основу правила. На слици 3.33 је дат пример.



Слика 3.33. Продукциони систем (rule-based)

Да би се велики скупови правила ефикасно дебаговали потребно је додати неки систем за објашњавање. У контексту игре, поред објашњења самог процеса закључивања потребно је додати и објашњења када сама игра директно мења стање базе знања. Наравно, ако се нека чињеница стално мења а није неопходна може се изоставити комплетан запис за ту чињеницу. На овај начин може се доћи до проблематичних почетних правила у бази знања и изменити их. У игри није потребно објашњавати играчу зашто је нешто урађено, али је ова техника користна за исправљање грешака у дизајну.

Глобално, продукциони системи са Rete алгоритмом упаривања су способни да подрже изузетно софистицирано понашање. Заједно са напредним техникама из доношења одлука на основу циља чине најнапредније технике које се користе у играма.

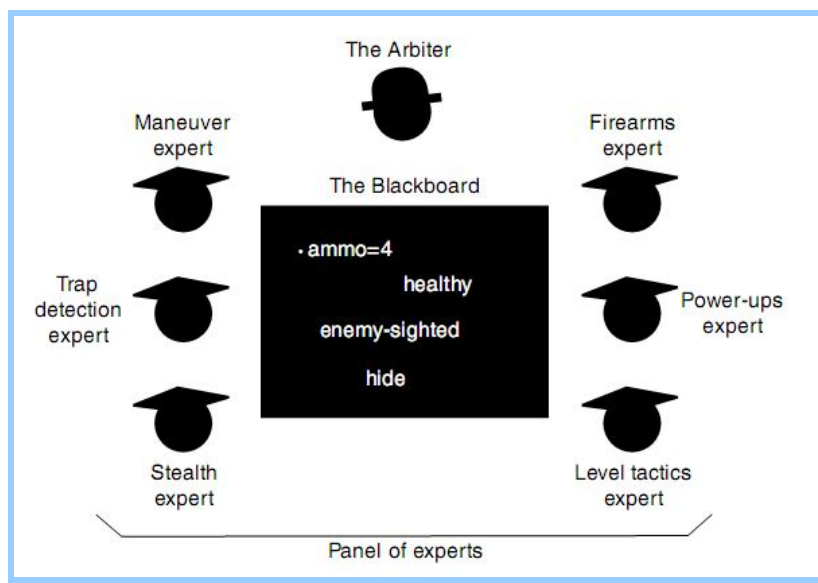
3.6. Blackboard архитектура

Blackboard архитектура сама по себи није механизам за доношење одлука, већ механизам за координисање акција више досилаца одлука. Индивидуални системи за доношење одлука се могу имплементирати на било који начин : од стабла одлучивања до продукциониг система. Ова флексибилност чини blackboard архитектуру привлачном.

Дакле, желели би да координишемо доношење одлука неколико техника. Свака техника ће моћи да даје сугестије у вечи тога шта би могли да радимо следеће, али ће финална одлука бити донесена на основу кооперације. Потребно је развити AI за тенк. Може постојати систем за доношење одлука чији је посао да прати непријатељске мете у видеокругу. Други тип AI-а ће бити коришћен да би се одабрала мета коју ће напасти. Када је мета одабрана, сам тенк можда није на доброј позицији за пуцање, па је потребно другим AI-ем довести га на добро место. Сада ће балистички AI израчунати

потребне параметре за пуцање. У овом случају је могуће ставити сваки део AI у ланац. Овај приступ је чест, али не дозвољава ток информација у супротном смеру. Ако балистички AI израчуна да не може да прецизно пуца са текуће позиције, потребно је померити се на нову позицију. Желели би механизам у којем би сваки AI могао да комуницира слободно без претходог експлицитног намештања свих могућих канала за комуникацију.

Основна структура blackboard система има три дела : скуп доносилаца одлука, црну таблу и арбитра. Илустровано на слици 3.34. Црна табла је део меморије коју сваки експерт може да користи за читање и упис. Сваки експерт би требало да чита и пише истим језиком, мада ће обично постојати неке поруке које неће моћи сви да разумеју. Сваки експерт гледа да ли на црној табли има нека порука коју могу да користе. Ако постоји таква порука, траже дозволу да добију креду и сунђер за брисање на неко време. Када добију контролу могу да размишљају, уклоне информације са црне табле и да додају нове информације на црну таблу. После неког времена експерт ће завршити и вратити контролу, што ће омогућити другим експертима да добију приступ. Арбитар одређује који експерт ће добити приступ. Експерти треба да имају неки механизам којим ће огласити да имају нешто да кажу. Арбитар ће одабрати једног и дати му контролу.



Слика 3.34. Blackboard архитектура

Алгоритам ради у итерацијама.

1. Експерти посматрају таблу и огласе своје занимање.
2. Арбитар одабере експерта коме ће дати контролу.
3. Експерт ради, успут ће вероватно модификовати таблу.
4. Експерт по завршетку рада врати контролу.

Препоручене акције се могу писати на табли на исти начин као и што пишу остале информације. На крају итерације (или више итерација) акције које су постављене на црну таблу се могу уклонити и извршити. Често се уз препоручене акције на таблу смешта и скуп флегова за договор. Тек када се релевантни експерти сложе, акција се извршава. Наредба за пуцање би имала флег којим се од балистичког експерта тражи потврда. Ово је потребно радити само за експерте који би могли да нађу разлог зашто се акција не би требала извршити.

Сам формат језика који се користи на табли зависи од апликације. Заједничке особине језика којим причају разни експерти су : вредност, идентификација типа и семантичка идентификација. Пошто сама вредност може бити у различитом опсегу на

основу типа, потребна је идентификација. Идентификацијом типа се експерту каже којег је формата податак, али му не помаже како да га схвати. Дакле, потребна је и семантичка идентификација, којом ће се експерту рећи шта значи податак.

Када смо разматрали blackboard систем рекли смо да се састоји од три компоненте : црне табле која садржи податке, скупа експерата (имплементираних на било који начин) који читају са табле и пишу на табли и арбитра који одређује који експерт има контролу. Сама структура је слична структури продукционог система. Продукциони систем има сваки од три елемента : садржи базу знања са подацима, свако правило се понаша као експерт и чита из базе знања и пише у базу знања, а постоји и арбитар који одређује које правило ће да се изврши. Сличност није случајна, већ је blackboard архитектура настала на основу генерализације продукционих система. Споредни ефекат је да ако се планира употреба и blackboard архитектуре и продукционог система, довољно је имплементирати само blackboard систем. Уз додатну подршку ће моћи да управља са правилима.

И коначни аутомати су подкуп blackboard система (они су подкуп продукционих система па самим тим и blackboard система). Црна табла се замани са једним стањем. Експерти се замењују са транзицијама, одлучујући на основу екстерних фактора и мењајући једини податак на црној табли. Прва транзиција чији су услови окинати би се извршила. То је само приступ арбитражи који извршава прву применљиву транзицију. У случају коначних аутомата то није једини начин арбитражије. Може се користити било који начин арбитражије који је представљен за продукционе системе, али се користи најједноставнији, јер је коначни аутомат дизајниран да буде једноставан.

3.7. Скриптовање

Значајан део система за доношење одлука у играма не користи ниједну од описаних техника. У раним и средњим 1990 –им, већина AI –а је била hard-coded са посебним кодом који би доносио одлуке. Ово је брз приступ, али није погодан за комплексније пројекте. И даље се користи за неке једноставне игре (добар део игара за мобилне телефоне нпр).

Како су игре постајале комплексније, јавила се потреба да се дизајн понашања одвоји од енцина. Многи тимови су прешли на коришћење других техника описаних у овом поглављу, док су неки прешли на скриптовани језик одвојен од кода, којим би програмирали понашање карактера. Ако је језик за скриптовање довољно једноставан, дизајнери нивоа су добили могућност да креирају и прошире понашање карактера.

Неочекивани споредни ефекат језика за скриптовање је могућност да сами корисници прошире игру и направе нова понашања за карактере. Тај процес се назива модовање, и помоћу њега се значајно продужава животни век игре.

Није вероватно да ће се помоћу скрипти направити врхунски AI за карактере. Међутим, корисне су за скриптовање окидача, понашања нивоа у игри (тек када играч дође на задату позицију пушта се анимација којом напредује радња) и прављења почетног прототипа за AI карактера.

Језици за скриптовање су или интерпретирани, бајт-компајлирани или потпуно компајлирани.

Интерпретирани језици се уносе као текст. Интерпретер обрађује сваку линију, одређује шта значи и извршава акцију која је специфицирана.

Бајт-компајлирани језици се конвертују из текстуалног облика у интерни формат, који се назива бајт код. Бајт код је обично знатно компактнији него текстуални формат. Пошто се може оптимизовати за извршавање, може се значајно брже извршавати. Потребан је процес компајлирања текстуалног језика у интерни формат. Тај процес је обично скуп и може се урадити приликом учитавања игре (или

нивоа) или чешће, пре него што се игра испоручи. Дакле, потребно му је више времена на почетку, али се после брже извршава.

Потпуно компајлирани језици креирају машински код. Тај код се мора повезати са главним кодом игре.

Приликом пројектовања језика за скриптовање је потребно водити бригу и о даљим могућностима коришћења, јер је могуће да ће се тај језик користити у серији игара чије захтеве не можемо одмах да проценимо. Битно је имати језик који лако може да позива функције или користи класе из кода игре.

Корисно је и да скрипте могу да буду ge-entrant. Тако да ако скрипта истроши додељено време, следећи пут када се активира може да настави где је стала. Не би све скрипте требале да буду ge-entrant. Пример је скрипта која посматра ситуацију која се рапидно мења, која би требала да се следећи пут извршава испочетка. Могу се и неки делови означити тако да се не прекида њихово извршавање.

Постоји мноштво комерцијалних језика за скриптовање који се могу користити. Предност коришћења већ готових језика је њихова робустност и оптимизација. Такође, ако се игра развија за РС, коришћењем познатог језика се омогућава корисницима да без много учења могу да модификују игру. Мана коришћења готових језика је мањак контроле. Када се пише сопствени језик он ће радити тачно оно шта је неопходно.

Примери познатијих језика за скриптовање су: Lua, Scheme и Python.

4. Закључак

Овим радом се покривене неке елементарне технике које се могу користити за пројектовање вештачке интелигенције у играма. Одабир одговарајуће технике, као и увек, зависи од потреба игре за коју дизајнирамо вештачку интелигенцију. Ако се прави компјутерска верзија неке игре на табли, користиће се нека од техника из друге области.

Основне технике које су разматране у другој области су : minimax, minimax са алфа-бета одсецањем, negamax, negamax са алфа-бета одсецањем, negascout и memory-enhanced test driver (MTD) технике. Основни minimax се не користи у реалним апликацијама, због лоших перформанси, већ служи за илустровање самог концепта минимаксовања. Слична је ситуација и са основним negamax-ом. Варијанте minimax-а са алфа-бета одсецањем и negamax-а са алфа-бета одсецањем се могу користити за једноставну вештачку интелигенцију, где је negamax-а са алфа-бета одсецањем бржи од minimax-а са алфа-бета одсецањем.

Дакле, ако правимо вештачку интелигенцију за неку једноставну игру или желимо једноставан AI, negamax-а са алфа-бета одсецањем ће бити довољан.

Битан фактор који треба узети у разматрање је могућност оптимизације која се може извршити над одабраним алгоритмом за игру, које су специфичне за ту игру. Шах као пример проблема који се у контексту вештачке интелигенције изучава више од 50 година има велики скуп могућих оптимизација које су тешко, или никако, применљиве на остале игре.

Међу најбрже алгоритме из ове категорије, које смо обрадили, спадају negascout са претраживањем на основу аспирација и memory-enhanced test driver (MTD). До недавно је negascout са претраживањем на основу аспирација био најбољи избор за алгоритме који играју игре на табли. Међутим, нови MTD алгоритми су се показали бољим у многим случајевима. Ниједан није теоретски бољи, али се у случају MTD алгоритма прави значајан напредак по питању брзине. Negascout са претраживањем на основу аспирација је коришћен као основа за AI у многим програмима који су побеђивали шампионе, укључујући шах, мице и Othello. MTD – f варијанта MTD алгоритма је ривал је negascout-у са претраживањем на основу аспирација по питању најбржег алгоритма за претраживање стабла игре. Тестови показују да је MTD – f често значајно бржи, али се и даље води дебата који од ова два алгоритма може више да се оптимизује. Најбољи начин да се за конкретну игру одабере одговарајући алгоритам је пробати све оне који су узети у разматрање и профилисати их.

На примеру једне потезне стратегије смо видели да су алгоритми из друге области теоретски применљиви, али због обима стабла игре нису применљиви у пракси. У том случају, као и у случају осталих игара је потребно користити друге технике, које су описане у трећој области.

Опет је потребно прецизно дефинисати жељено понашање. Ако је у питању неко једноставно понашање, стабла одлучивања или коначни аутомати ће бити довољни. Нема потребе имплементирати комплексније технике ако смо сигурни да ће тражено понашање бити једноставно.

Стабла одлучивања су брза, лака за имплементирање и једноставна за разумевање. Такође, модуларна су и лако се креирају. Бинарна стабла се не скалирају добро, и за комплексније пројекте могу бити тешка за одржавање.

Коначни аутомати су једноставнији за креирање од стабала одлучивања у одређеним случајевима (ако је потребно вршити исту акцију док се не промени ситуација). Коначни аутомати су такође интуитивни и лаки за разумевање. Омогућавају да се проблем разбије на мање целине и потребно је дефинисати везе између тих делова. За велике пројекте опет постоји проблем скалирања.

Језици за скриптовање омогућавају додавање новог садржаја у игру без потребе за програмерима. Користе поједностављен систем да би додали AI елементе, логику и

нова понашања. Изузетно су битни због могућности да корисници мењају игру. У комерцијалним играма се често имплементирају, јер могућност модовања повећава вредност игре.

Фази логика нам омогућава да правила дефинишемо на лингвистички природнији начин, што резултује у лакшем одржавању система. И сам број правила може бити мањи од традиционалних система. Такође, одлуке које су донесене на основу доступних информација ће бити фино подешене у односу на степен истинитости релевантних података. На тај начин се омогућују природнија понашања.

У неким случајевима је потребно дугорочније планирање, односно, разматрање секвенци могућих акција на основу текућег глобалног и унутрашњег стања (којим се моделирају мотиви и жеље). То се може постићи техникама из области : понашање на основу циља. На тај начин се ствара илузија да карактери имају жеље и мотиве.

Помоћу продукционих система је могуће створити врло реактивно и софистицирано понашање, ако се користи довољно велики скуп добро формираних правила. У случају великог скупа правила, чак и са комплексним методама за упаривање може постојати проблем са перформансама. Да би се исправиле грешке и додале нове могућности, потребно је додати нова правила, што у крајњем случају може довести до непрегледности скупа правила.

Коначни аутомати и продукциони системи су подскуп blackboard система. Разматрајући ове три технике, коначни аутомати представљају најједноставнију технику, док blackboard системи представљају најфлексибилнију технику. Blackboard систем има за циљ да координише акције више различитих експерата. Има велику флексибилност, јер су сами експерти пројектовани коришћењем одговарајуће технике за сопствени домен. Управо због своје флексибилности blackboard архитектура је веома привлачна.

Коначно, сваки жанр игара има сопствене технике које се користе за решавање одређених проблема.

Литература

Списак књига

- (1) Stuart J. Russel, Peter Norvig “Artificial Intelligence, a modern aproach – second edition”, 2003
- (2) Ian Millington “Artificial Intelligence for games “, 2006
- (3) Brian Schwab “ AI game engine programming ”, 2004
- (4) David M. Bourg, Glenn Seeman “AI for Game Developers “, 2004
- (5) Mat Buckland “Programming Game AI by Example “, 2004
- (6) Mark Watson “Practical Artificial Intelligence Programming With Java “, 2008
- (7) Деведић Владан: “Експертни системи за рад у реалном времену”, Институт “Михајло Пупин”, Београд, 1994.
- (8) Драган Бојовић, Душан Велашевић, Војислав Мишић, “Збирка Задатака из Експертних Система”, Електротехнички факултет, Београд, 1996
- (9) Татјана Николић, Матеја Опачић: “Вештачка интелигенција и неуронске мреже”, ИБН Центар, Београд, 1995.
- (10) Душан Тошић, Радивој Протић: “Пролог кроз примере”, Техничка књига, Београд,
- (11) Петар Хотомски, “Системи Вештачке интелигенције”, Технички факултет “Михајло Пупин”, Зрењанин, 1995.
- (12) Перо Субашић: “Фази логика и неуронске мреже”, Техничка Књига, Београд, 1998.
- (13) Петар Хотомски, Марија Кујачић: “Математичка логика и принципи програмирања”, Технички факултет “Михајло Пупин”, Зрењанин, 1992.
- (14) Срђан Миленковић: “Вештачке неуронске мреже”, Задужбина Андрејевић, Београд, 1997

Интернет референце

- (1) <http://www.gameai.com/>
- (2) <http://ai-junkie.com/ai-junkie.html>
- (3) <http://www.cs.cmu.edu>
- (4) <http://www.neuron.yale.edu>
- (5) <http://www.jneurosci.org/>
- (6) <http://www.mannanetwork.com>