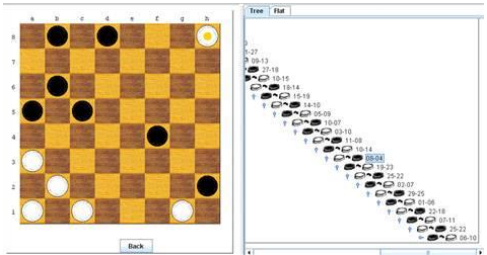
Игре и вештачка интелигенција

Категорије

- **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.
- **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).
- **На нивоу шампиона.** Најпознатији пример је шах. Deep Blue је победио тадашњег шампиона Каспарова, међутим нерешени мечеви садашњих најбољих програма Deep Fritz -а и Deep Junior –а против Крамника и Каспарова показују да су још увек на нивоу светских шампиона.
- **На нивоу велемајстора.** Ту спадају програми који могу играти на нивоу велемајстора али ипак бивају побеђени од шампиона. Покер спада у ову категорију.
- **На нивоу аматера.** Програми играју на нивоу аматера. Go спада у ову категорију.



Типови игара

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

Број играча

- Игре на табли које су инспирисале потезне (turn-based) AI алгоритме углавном имају 2 играча.
- Многи су 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.
- Многе оптимизације тих алгоритама претпостављају да су у питању само два играча

Циљ игре

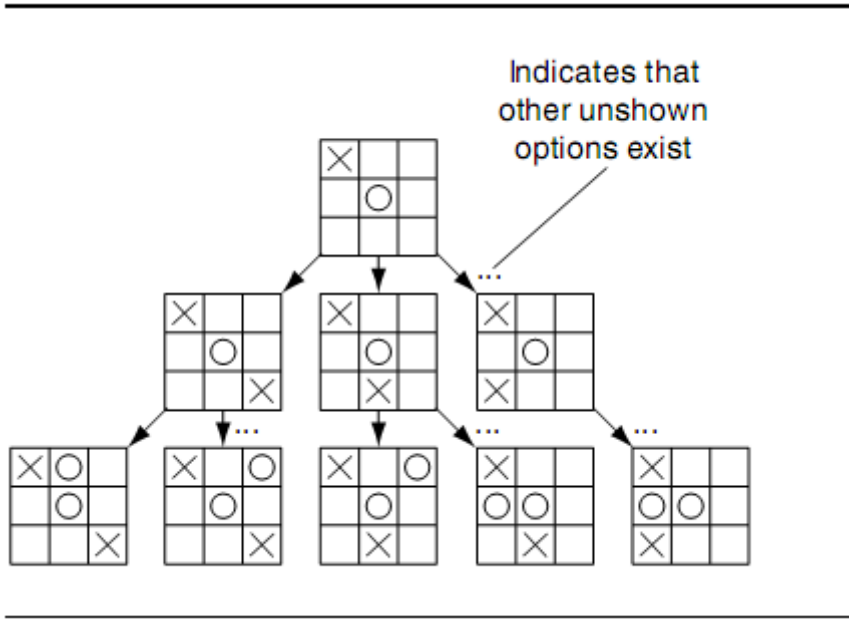
- У многим стратешким играма циљ је победити
- Играч побеђује ако сви противници изгубе (zero-sum игра)
  - Описује ситуацију у којој је добитак или губитак сваког учесника избалансиран губицима или добицима осталих учесника - ако неко победи остали су изгубили.
  - Није случај у казино играма, ако се казино не посматра као играч, где би сви могли да добију или изгубе.
- 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 је боље искористити слабије играче да би се заједничким снагама елиминисао најјачи играча, а затим се слабији елиминишу

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 - игре 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).
- 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**.

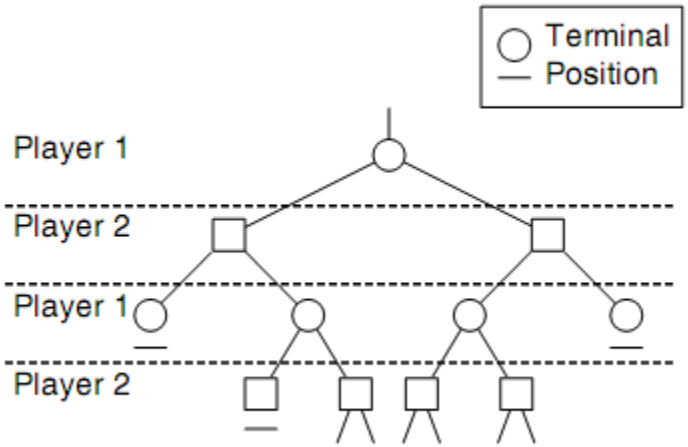
Стабло игре

- Било која потезна игра се може представити стаблом игре
- Сваки чвор у стаблу представља позицију у игри, а свако гранање представља један могући потез.
- Потези пресликавају једну позицију на табли у другу.



Стабло игре

- 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.
- Игра је потезна. па до промене на табли долази само када се одигра потез.
- Број гранања од сва заког чвора, је једнак могућем броју поте икс-окс-у тај број у првом потезу је 9, г ље. У многим играма може постојати и ст з текућег чвора стабла.
- Неке позиције на та ову се терминалне позиције, и означав
- За сваку терминалн играчу.
- Тај резултат може б ка и -1 за губитника, или тај резултат мо ке позиције. Нерешене позиције
- У zero-sum играма б ча био 0. У играма које нису zero-sum резултати би рефлектовали квалитете завршних позиција свих играча.
- Користе абстрактна стабла, без чворова као приказа стања на табли, али са финалним резултатима.

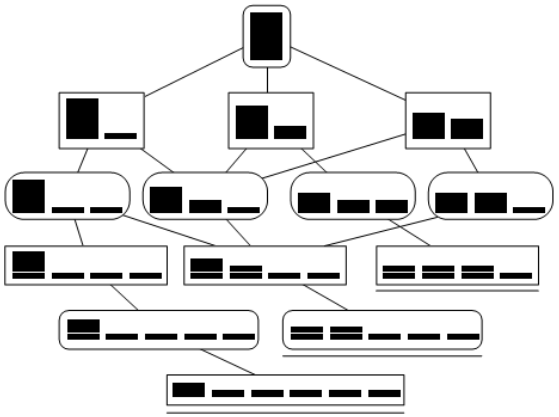


Фактор гранања

- У икс-окс-у сваки играч додаје сопствени симбол и пошто је на почетку било 9 места на табли максимални број потеза је 9 – укупно 26830 листова
- 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
- Стабло игре за такву игру би било изузетно дубоко, чак и за релативно мали фактор гранања.
- Лакше је играти 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. У сваком потезу потребно је да играч подели једну гомилу новчића у две са неједнаким бројем новчића у њима. Последњи играч који може да направи потез побеђује.



Транспозиција

- 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.
- Формално се игре могу представити као проблеми претраге са следећим компонентама :
  - Иницијално стање, које укључује почетно стање на табли и има информацију о играчу који је следећи на потезу.
  - Функцију која враћа легалне потезе у виду листе која садржи парове (потез, стање), који означавају легални потез и резултујуће стање.
  - Тест терминалности, који означава када је дошло до краја партије.
  - 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

Статичка функција процене

- Због брзине се користи хеуристика, а статичка је јер разматра само текуће стање без разматрања будућих стања.
- 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.
- Теже је дати процену у половини партије. Та процењена вредност би требала бити индикатор колико је вероватно да је из тог стања могуће победити.
- У Othello-у онај играч са највише фигура на табли у терминалној позицији побеђује.
- Али, 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.

Статичка функција процене

- Статичка функција процене може да врати било који број.
- У многим имплементацијама враћа целобројни број.
- 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.
- 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.

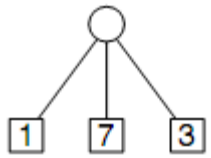
Функције процене

- Може постојати више различитих механизама 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.
- У шаху би статичка функција процене могла бити

  $\mathbf{c_1}$ *** material +** $\mathbf{c_2}$ *** mobility +** $\mathbf{c_3}$ *** king safety +** $\mathbf{c_4}$ *** center control + ...**
- 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.
- 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.

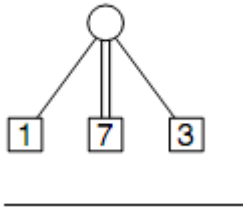
Избор потеза

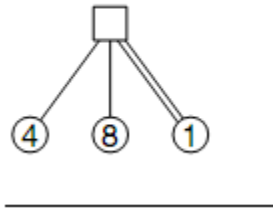
- Са добром статичком функцијом процене, рачунар би одабрао потез процењујући позиције које би добио на основу свих легалних потеза и одабрао потез који води до позиције са најбољом проценом.
- Дата позиција са три могућа потеза и проценама резултујуће позиције. Одабран је други потез због највеће вредности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



MINIMAX

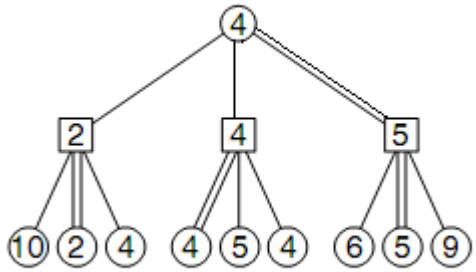
- Ако је мој потез одабраћу потез који води до најбоље позиције на располагању - покушаћу да максимизујем сопствени резултат.
- Ако је противнички потез, он ће начинити потез који ће ме оставити у најгорој позицији од потеза на располагању - противник покушава да минимизује мој резултат.
- Транзиција између максимизовања и минимизовања док се претражује стабло игре се назива minimaxing





MINIMAX

- Играч који је на потезу може да дође до позиције која се вреднује са 10 поена, али се претпоставља да ће противник начинити потез који ће довести до позиције вредне 2 поена.
- Ако се одигра потез два се претпоставља да ће противнички потез довести до позиције вредне 4 поена.
- Трећим потезом противник бира потез који води до коначне позиције вредне 5 поена.

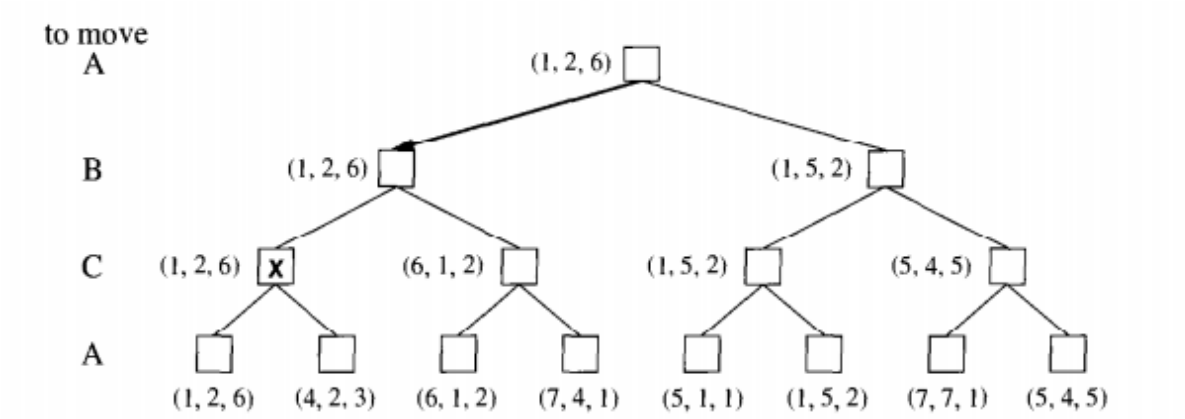


MINIMAX

- Стартујући од дна стабла игре бирамо процене стања на основу minimax правила.
- У сопственом потезу бира се највећа процена, у противничком потезу најмања.
- minimax алгоритам - после пропагације добија се процена потеза за текућу позицију, потом се бира потез на основу највеће вредности.
- Да би одредио колико је добар неки потез, претражује се 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.
- 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.
- Могуће је одабрати потез, као потез са највећом вредношћу процене.
- Чак и за претраге које гледају само пар потеза унапред, minimax даје значајно боље резултате него приступ само са хеуристиком.

Игра са више играча

- За случај са три играча А, Б, Ц потребно је за сваког памтити процену - у виду вектора који би био придружен сваком чвору у стаблу игре.
- У терминалном стању тај вектор ће бити резултат статичке функције процене.
- Чвор Х - играч С одлучује шта да ради- бира 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.



Игра са више играча

- У многим играма са више играча постоје и савези, било формални или неформални.
- Ако су играчи A и B у слабијој позицији од играча C могу направити савез да би заједно напали играча C, јер би их он могао појединачно уништити.
- Када C 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.

MINIMAX алгоритам

- Алгоритам је рекурзиван. Сваком рекурзијом покушава да израчуна исправну вредност текуће позиције на табли.
- Разматра се сваки могући потез из текуће позиције.
- За сваки потез рекурзивно се 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 дошли 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 - промена између максимизујуће и минимизујуће фазе.
- Ако је тренутна дубина претраге 0, сачуваће се најбољи потез и то је потез који треба начинити.

MINIMAX алгоритам

```
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,  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
```

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.
- Minimax функција се може позивати из једноставније функције која ће само да врати најбољи потез.

```
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
```

MINIMAX алгоритам

- **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.
  - 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 меморије је O(*bm)*.
  - Цена је врло велика и непрактична за комплексније игре, па је потребно увести унапређења.

Алфа – Бета одсецање

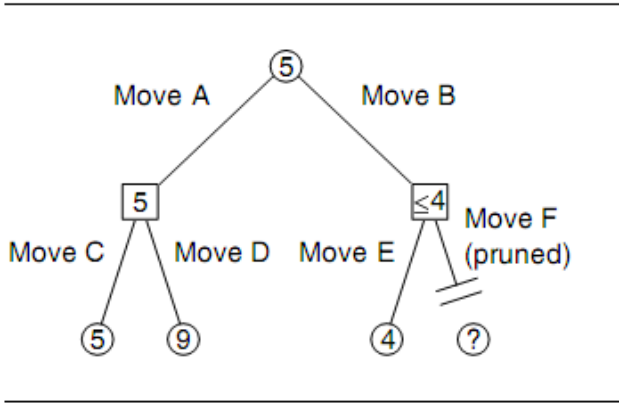
- Проблем са minimax алгоритмом је да је број чворова који треба да се обиђе експоненцијалан по дубини стабла.
- Не може се уклонити експоненцијална зависност, али се може преполовити зависност од дубине стабла.
- Да би се то урадило потребно је донети исправну minimax одлуку без гледања свих чворова - одсецањем
- 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.
- Алфа – Бета одсецање се може применити на стабла игре било које дубине, и често се одсецају читава подстабла из разматрања.
- Нека је 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

- Потребно је да водимо евиденцију о најбољем потезу који можемо да одиграмо.
- Алфа граница је доња граница резултата који се може постићи
- Може се 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.

Алфа одсецање

- Ако играч начини потез A, противник одговара са C дајући резултат 5.
- Потез B - први одговор је потез E који даје резултат 4.
- Без обзира на вредност процене стања после потеза F противник може одабрати потез E, тако да је коначни резулатат потеза B једнак 4.
- Чак и без разматрања потеза F играч зна да је то погрешан потез јер може да добије резултат 5 потезом A.
- У овом случају потез F се одсеца.



Бета одсецање

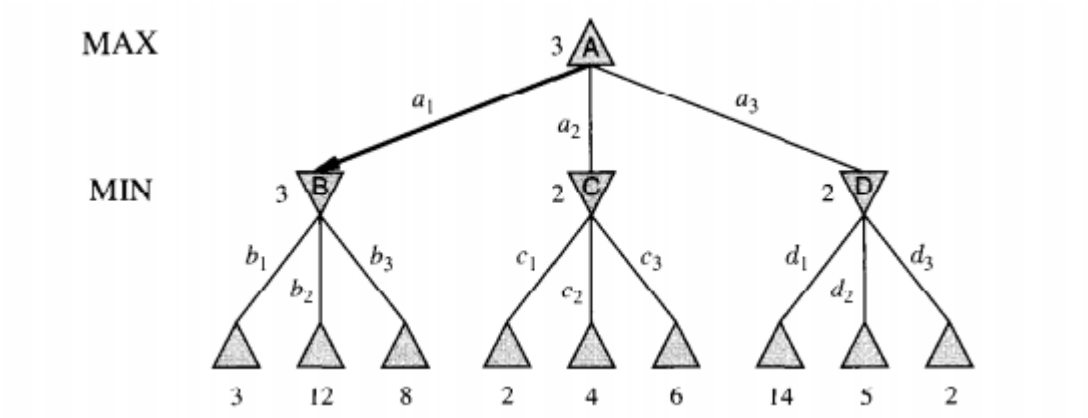
- Бета одсецање се обавља на сличан начин
- Бета вредност води евиденцију о горњој граници резултата која може да постигне
- Ова вредност се ажурира када се нађе секвенца потеза на коју нас натера противник.
- Тада се зна да је то горња граница која се може постићи и противник може само још више да нас 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.

Алфа – Бета одсецање

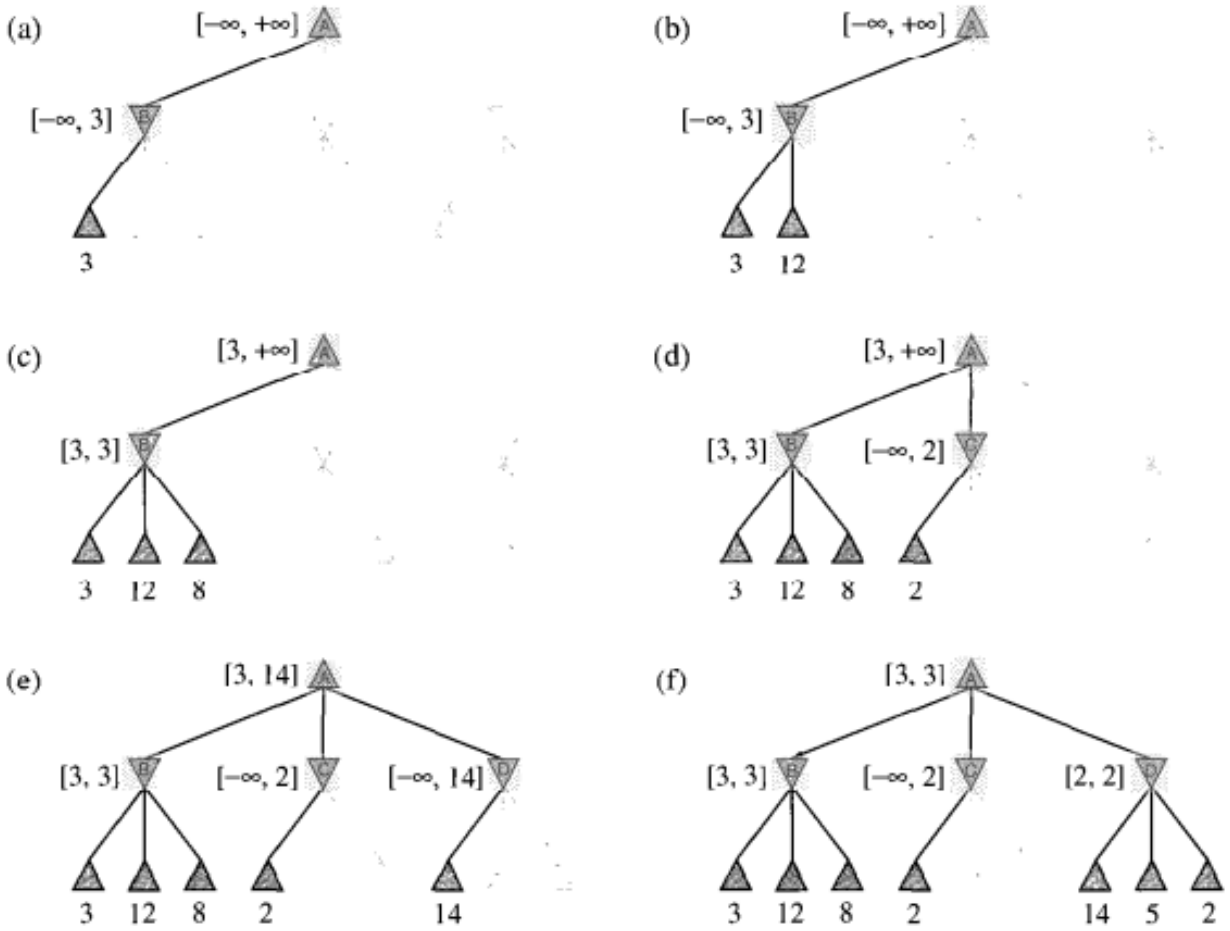
- Заједно Алфа и Бета вредности чине интервал могућих вредности резултата који се разматрају.
- Никад нећемо играти потезе са лошијом проценом од алфа и противник никад неће дозволити да се одигра потез са проценом већом од бета.
- Коначни потез ће бити ограничен тим вредностима.
- Како се претражује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

Алфа – Бета одсецање

- Min бира најмању вредност од 3,12,8 за позицију B, најмању од вредности 2, 4, 6 за позицију C 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.



Алфа – Бета одсецање



Алфа – Бета одсецање

- У стању A max одређује потез, док у стањима B, C и D min одређује потез:
  - (a) Први лист испод B има вредност 3, па је максимална вредност чвора B који је min чвор једнака 3. То је бета вредност.
  - (b) Други лист испод B има вредност 12, која је већа од бета вредности и тај потез ће min да избегне, тако да је вредност чвора B још увек највише 3.
  - (c) Трећи лист испод B има вредност 8, чиме смо видели све његове наследнике и вредност B је тачно 3. Може се рећи да је вредност B 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 C има вредност 2. C који је min чвор има вредност од највише 2. То је бета вредност. Међутим, знамо да чвор B има вредност 3, па max никад не би бирао тај потез. Дакле, нема потребе даље гледати потомке чвора C. 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 –a, 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 је да се одигра потез који води ка стању B, јер има вредност 3.

Перформансе

- Перформансе јако зависе од редоследа којим се посећују потомци.
- Било би добро када би могли да прво посетимо такве потомке који би нам омогућили да вршимо одсецање, односно најбоље потомке прво (да смо прво посетили D одсекли бисмо све његове потомке)
- Када би то било могуће, алфа – бета одсецање би морало да обиђе само $O(b^{m/2})$ чворова да одабрало најбољи потез - у односу на minimax значајно боље.
- 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.

Алгоритам

```
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

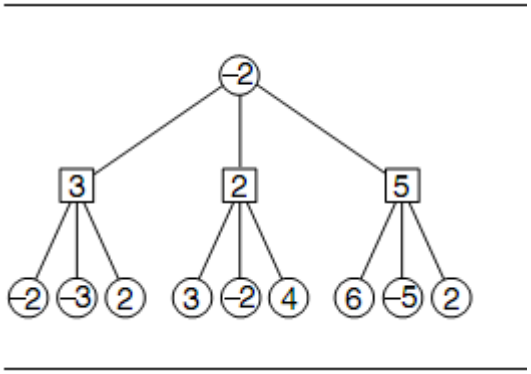
- Разлика у односу на стандардни minimax алгоритам је у позивању, јер имамо још алфа и бета параметре.
- Приликом претраживања се проверава да ли текући потез упада у интервал одређен алфа и бета вредностима,
- Ако то није случај само се врати најбољи потез и његов резултат.
- Алфа и бета вредности се ажурирају и тако се сужава интервал и ограничава претрага.

```
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
```

- 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.

Negamax

- Minimax рутина оцењује потезе на основу перспективе једног играча.
- Прати се 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.
- 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 да буду исправни за играча који је покренуо претраживање.
- Како сваки играч покушава да максимизује сопствени резултат, бира се највећа вредност.
- Због инвертовања знака и бирања максимума у сваком кораку, алгоритам се зове negamax.



Negamax

- У случају minimax-а врши се процена на основу истог играча.
- Сама функција процене треба да има параметар који говори на основу ког играча да врши процењивање.
- Како 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.
- Перформансе negamax 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.

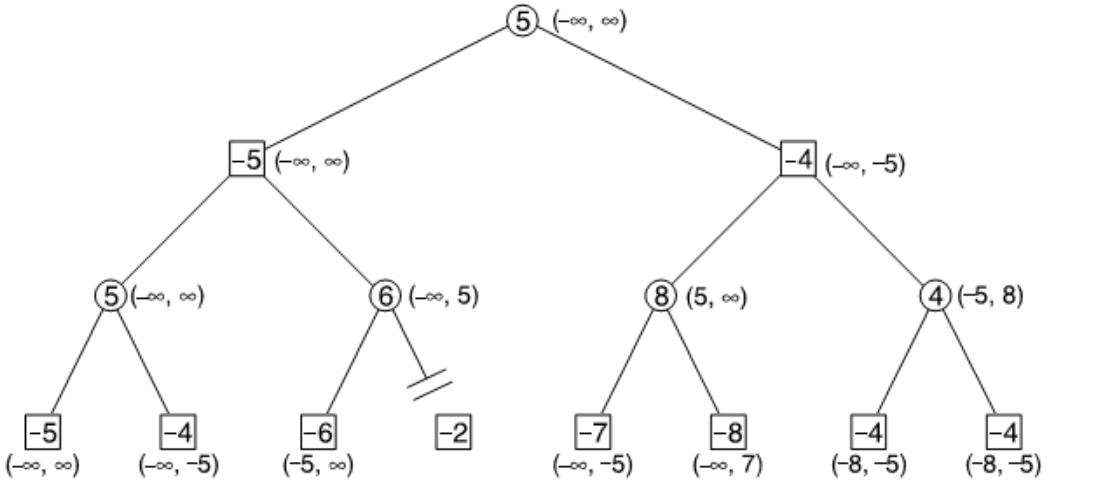
Алгоритам

```
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
```

Negamax са алфа – бета одсецањем

- Уместо наизменичних провера са алфа и бета вредностима у суседним корацима, Алфа – бета negamax замени и промени знак алфа и бета вредностима, на исти начин на који је то радио са резултатима следећег нивоа.
- Како алгоритам претражује са лево на десно, алфа и бета вредности постају све ближе, ограничавајући претрагу.



Алгоритам

```
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
```

Перформансе

- Перформансе су опет истог реда као перформансе одговарајућег minimax алгоритма.
- Међутим, иако су истог реда, negamax 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.
- Овај алгоритам је бољи него основни negamax, а нешто бржи од minimax-а са алфа-бета одсецањем.

Прозор претраге

- Интервал који формирају алфа и бета вредности у алгоритму се назива прозор претраге.
- Разматрају 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, па поступак који смањује 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 касније вероватно одсећи.
- 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.
- Како статичка функција процене даје приближну процену колико је добра нека позиција, може бити ефективна у смањењу обима претраге кроз одсецање.
- Још ефектнији начин је да се користе резултати претходних minimax претрага.
- Фамилија memory-enhanced test driver (**MTD**) алгоритама користи овај приступ да би поређала потезе пре њиховог разматрања.
- 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.

Смањивање прозора

- Мали интервал прозора претраге чини огромно убрзање претраживања
- Уместо позивања алгоритма са прозором димензија (-INFINITY, INFINITY), алгоритам се може позвати са прозором процењених димензија.
- Те димензије се начивају аспирацијом, а алгоритам са алфа-бета одсецањем који га користи алгоритмом са претрагом на основу аспирације.
- Може се догодити да не постоје потези који задовољавају одабрани 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 – величина прозора, 5 + величина прозора).
- Величина прозора зависи од распона вредности које могу бити добијене од статичке функције процене.

Алгоритам

```
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;
```

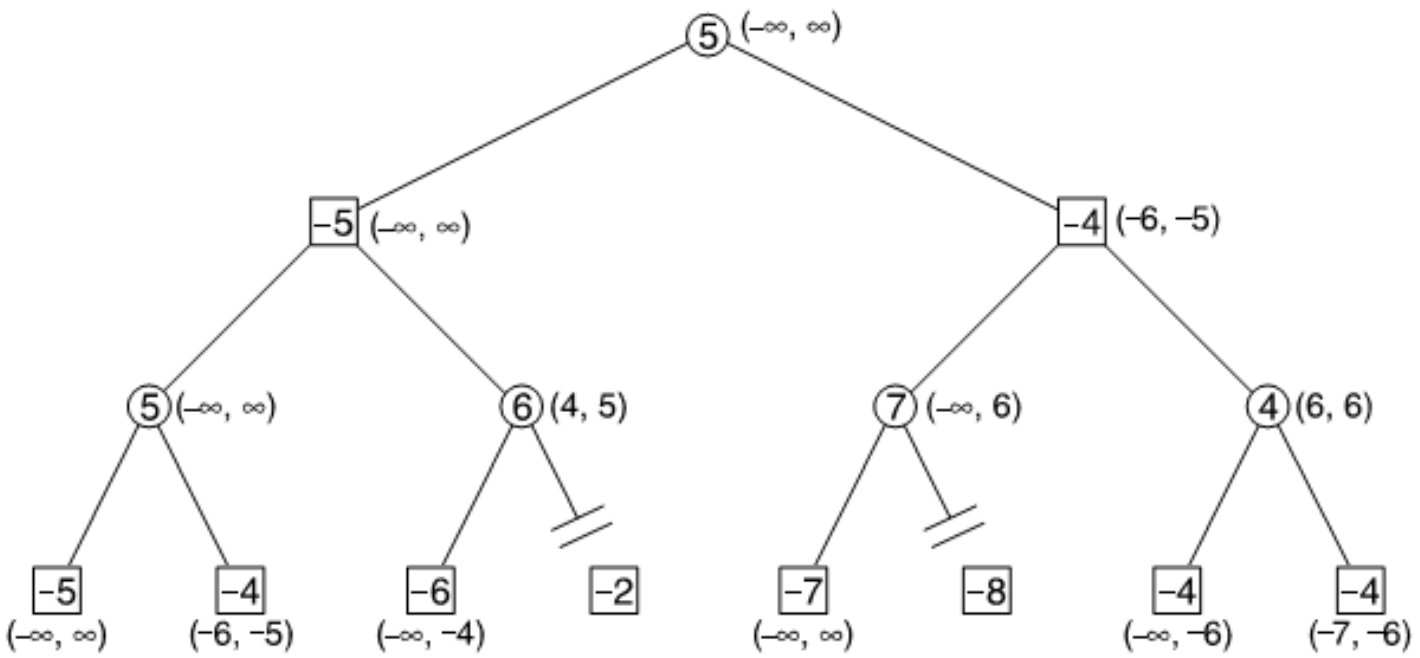
Negascout

- Сужавање прозора претраге може довести да је његова величина једнака нули.
- На тај начин претрага ће бити веома брза, али ће одсећи скоро све корисне гране заједно са бескорисним.
- Ако се покрене алгоритам са погрешним прозором, претрага ће бити неуспешна.
- Основна верзија алфа-бета negamax-а ће вратити или алфа или бета вредност у случају неуспеха (и то у зависности на основу које границе је неуспешно претраживање).
- Та информација омогућава да се прилагоди претпоставка и да се понови претрага са разумнијим прозором.

Negascout

- Negascout алгоритам користи алфа-бета negamax.
- 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 negamax-а.
- Иницијална претрага првог потеза са широким прозором даје добру апроксимацију за тест. Тако се избегава превише неуспеха и дозвољава одсецање великог броја грана.
- У општем случају negascout доминира над negamax алгоритмом јер потребно да обиђе исти број или мање чворова.

Negascout



Алгоритам

```
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
```

Алгоритам

```
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
```

Перформансе

- 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 би 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.