Zaawansowane metody przeszukiwania grafów przestrzeni stanów gier dwuosobowych

Informatyka

Laboratorium Sztucznej Inteligencji

2006

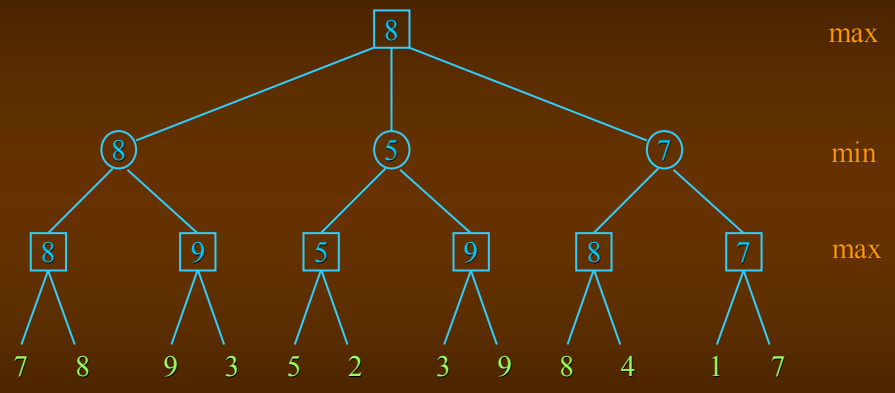
Teoria gier w dziedzinie SI

- Liczba graczy
  - jednoosobowe, dwuosobowe oraz wieloosobowe
- Suma wypłat
  - gry o sumie zerowej (zyski i straty uczestników bilansują się)
  - gry o sumie niezerowej (wygrane i przegrane nie muszą się bilansować)
- Dostępna wiedza
  - gry z pełną informacją (precyzyjna wiedza o sytuacji i celach przeciwnika)
  - gry z niepełną informacją (brak wiedzy na temat przeciwnika)

Gry dwuosobowe

- Dwóch przeciwników posiadających pełną informację stanie gry i wszystkich możliwych ruchach
- Gracz nr 1 nosi nazwę Max, bo:
  - maksymalizuje rezultat końcowy
  - wzrost wartości funkcji celu oznacza jego zysk oraz równoważną stratę dla przeciwnika
- Gracz nr 2 nosi nazwę Min, bo:
  - minimalizuje rezultat końcowy
  - spadek wartości funkcji celu oznacza jego zysk oraz równoważną stratę dla przeciwnika

Zasada min-max

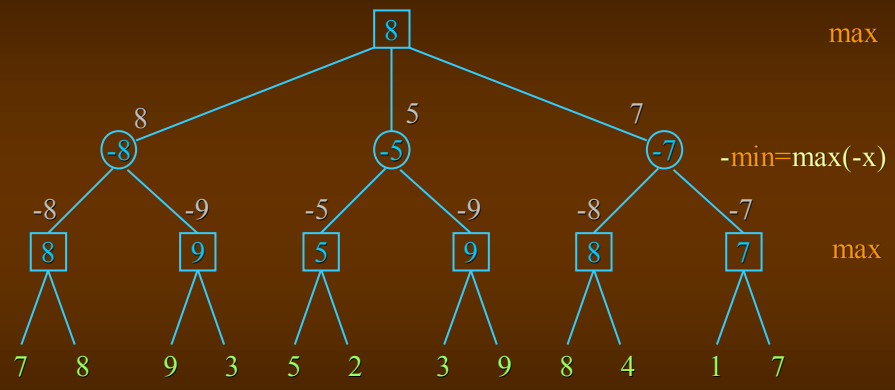


Algorytm min-max

wywołanie: `result = MinMax(s, MAXDEPTH, MAX)`

```
int MinMax(state s, int depth, int type)
{
   if( is_terminal_node(s) || depth==0 ) return(Eval(s));
   if( type == MAX){
      best = -∞;
      for(child=1; child<=NumOfSucc(s); child++) {
         val = MinMax(Succ(s,child), depth-1, MIN);
         if( val > best ) best = value;
      } //endfor
   }
   else { // type == MIN
      best = ∞;
      for(child=1; child<=NumOfSucc(s); child++) {
         val = MinMax(Succ(s,child), depth-1, MAX);
         if( val < best ) best = value;
      } //endfor
   }
   return best;
}
```


Zasada neg-max



Algorytm neg-max

wywołanie: `result = NegMax(s, MAXDEPTH)`

```
int NegMax(state s, int depth)
{
   if( is_terminal_node(s) || depth==0 ) return(Eval(s,depth));
   best = -∞;
   for(child=1; child<=NumOfSucc(s); child++) {
      val = -NegMax(Succ(s,child), depth-1);
      if( val > best ) best = value;
   } //endfor
   return best;
}
```

Funkcja heurystycznej oceny stanu musi uwzględniać, kto wykonywałby ruch w ocenianym stanie. Jeżeli gracz MAX to ocena jest w postaci prostej, jeśli gracz MIN - w postaci zanegowanej.

Koszty algorytmu min-max (lub neg-max)

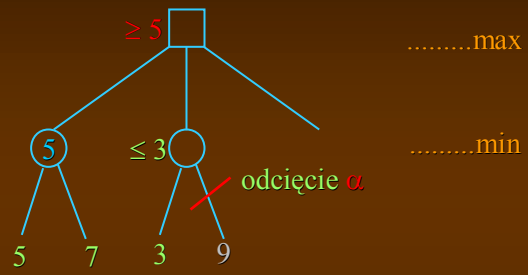
Przyjmując określony branching factor (b) oraz stałą głębokość przeszukiwania (d)

- Złożoność pamięciowa $O(bd)$
- Złożoność czasowa $O(b^d)$

Czy można ten wynik poprawić?

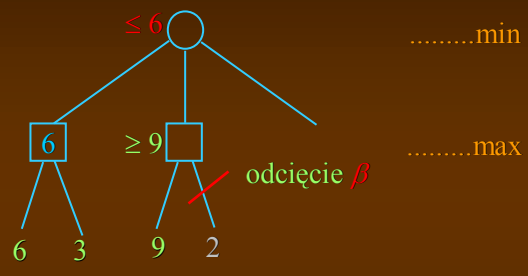
- Tak!
  Branch&bound

Ograniczenie dolne - *odcięcie α*



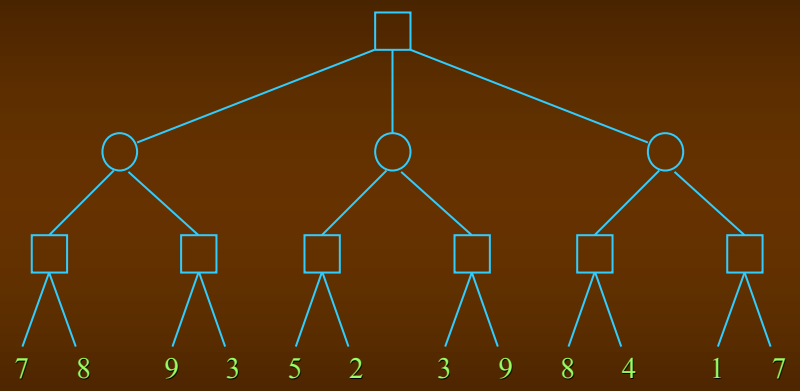
Analiza lewego poddrzewa pokazała, że MAX ma ruch o wartości 5. Po sprawdzeniu lewego liścia środkowego poddrzewa widać, że wartość ruchu będzie mniejsza lub równa 3 (ruch wykonuje MIN). Analiza pozostałych liści nie ma zatem sensu, gdyż decyzja MAXa w korzeniu grafu nie może już ulec zmianie niezależnie od ich wartości.

Ograniczenie górne - *odcięcie β*

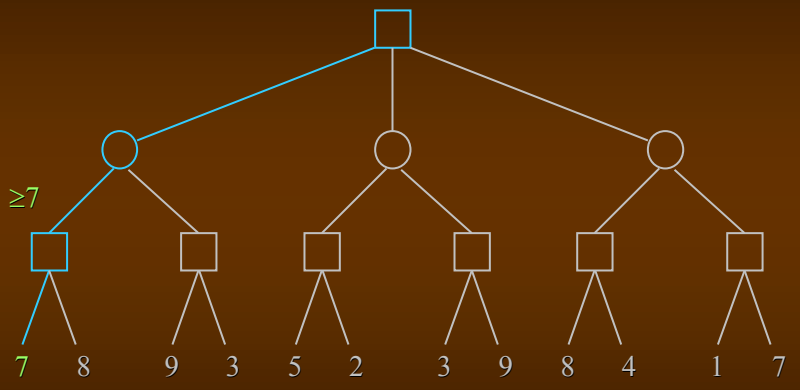


Analiza lewego poddrzewa pokazała, że MIN ma ruch o wartości 6. Po sprawdzeniu lewego liścia środkowego poddrzewa widać, że wartość ruchu będzie większa lub równa 9 (ruch wykonuje MAX). Analiza pozostałych liści nie ma zatem sensu, gdyż decyzja MINa w korzeniu grafu nie może już ulec zmianie niezależnie od ich wartości.

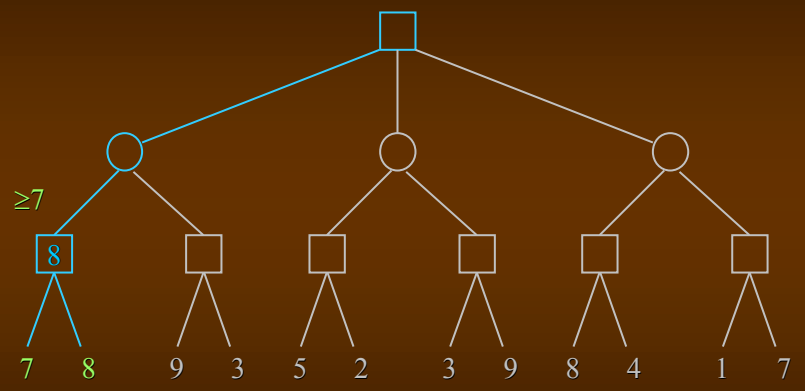
Odcięcia α-β: przykład



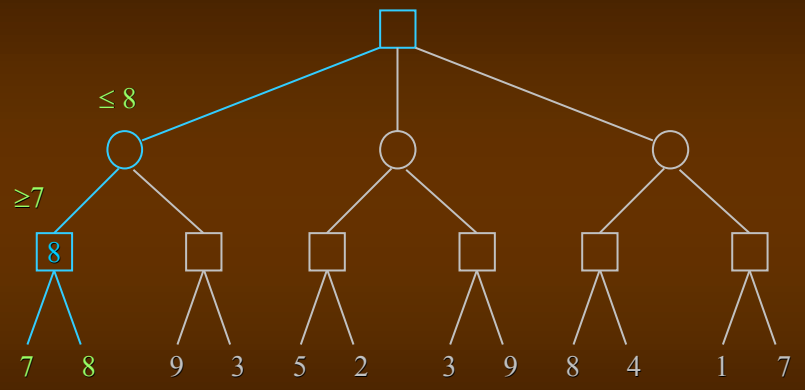
Odcięcia α-β



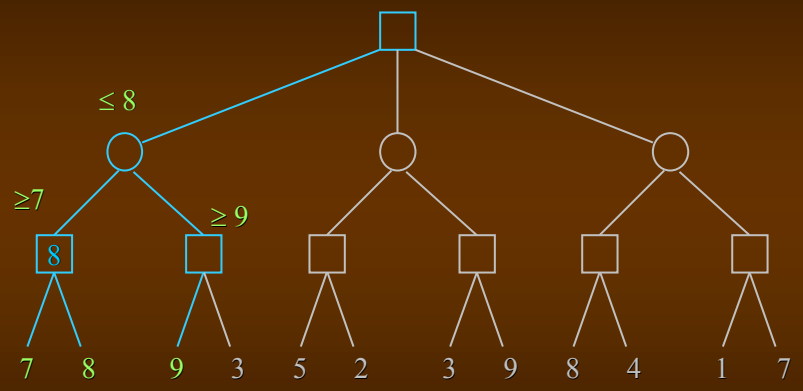
Odcięcia α-β



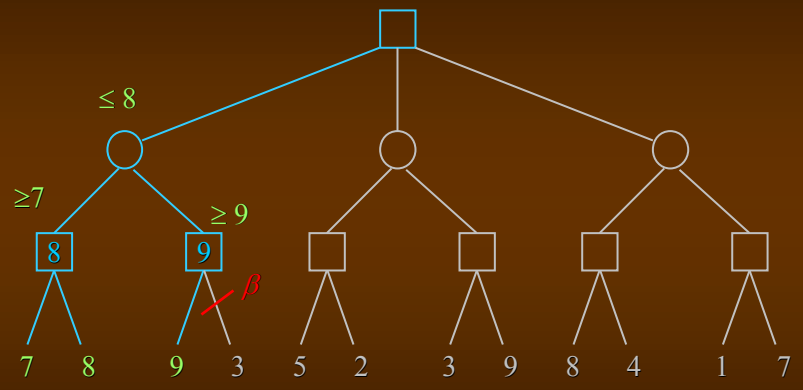
Odcięcia α-β



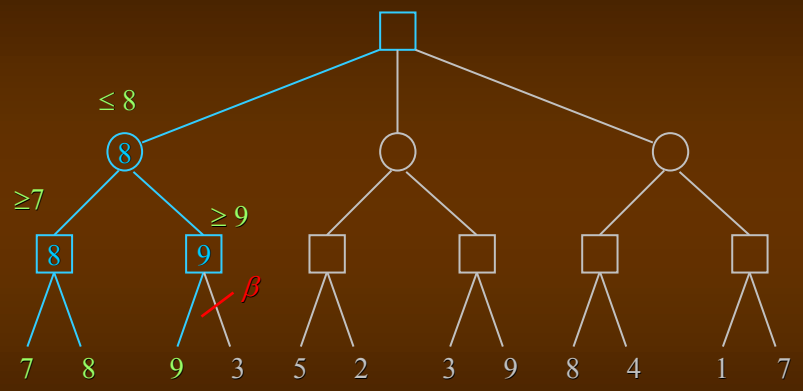
Odcięcia α-β



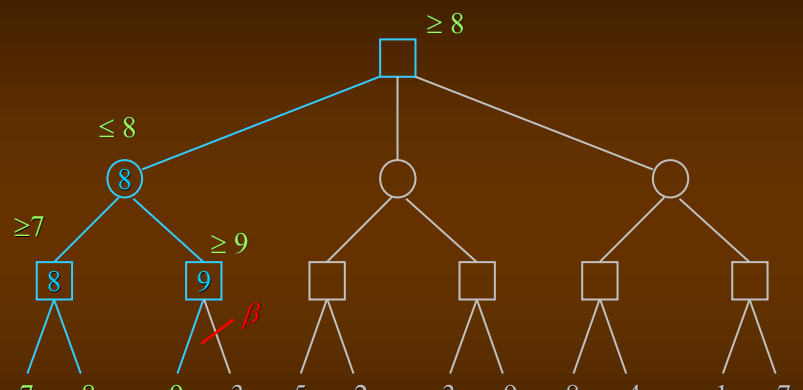
Odcięcia α-β



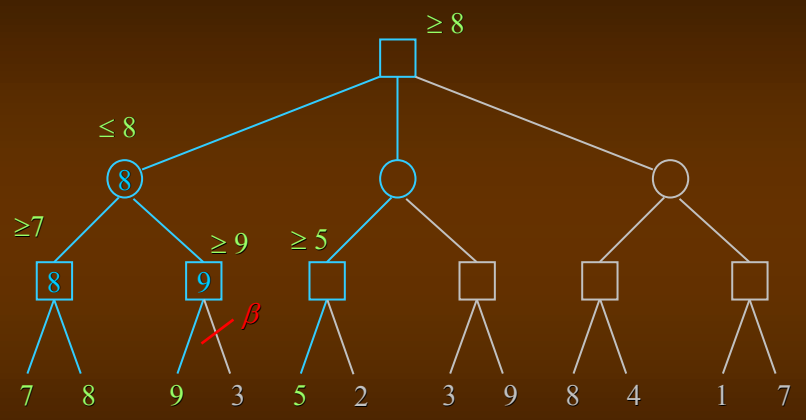
Odcięcia α-β



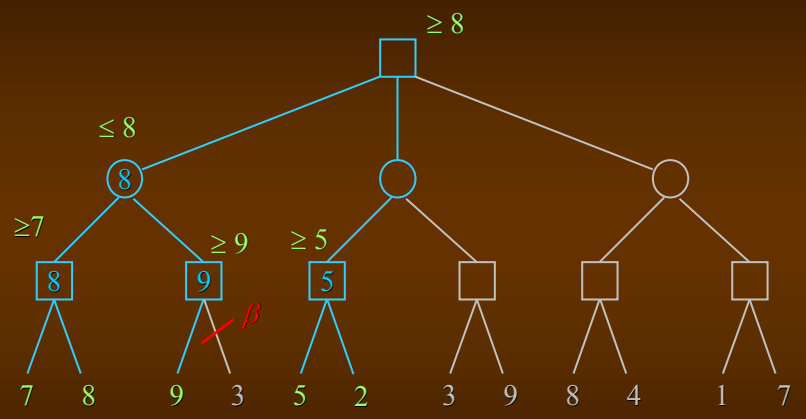
Odcięcia α-β



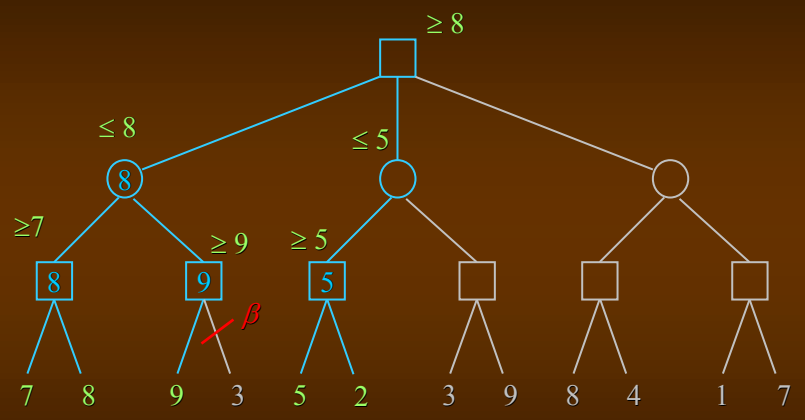
Odcięcia α-β



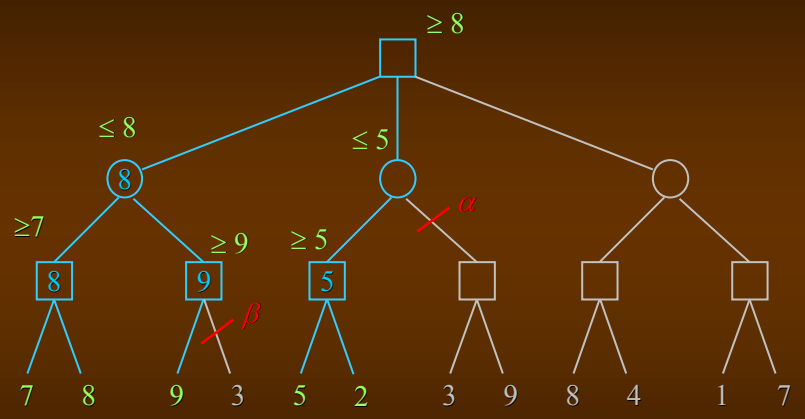
Odcięcia α-β



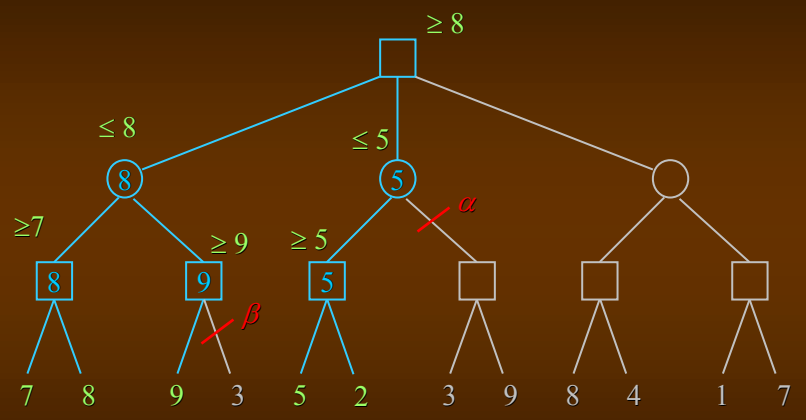
Odcięcia α-β



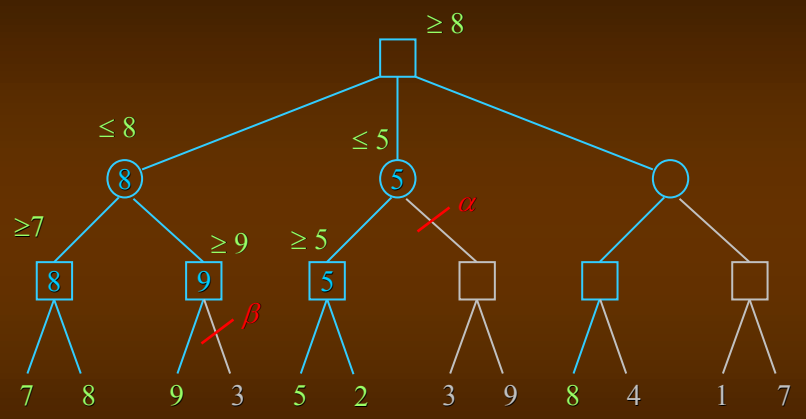
Odcięcia α-β



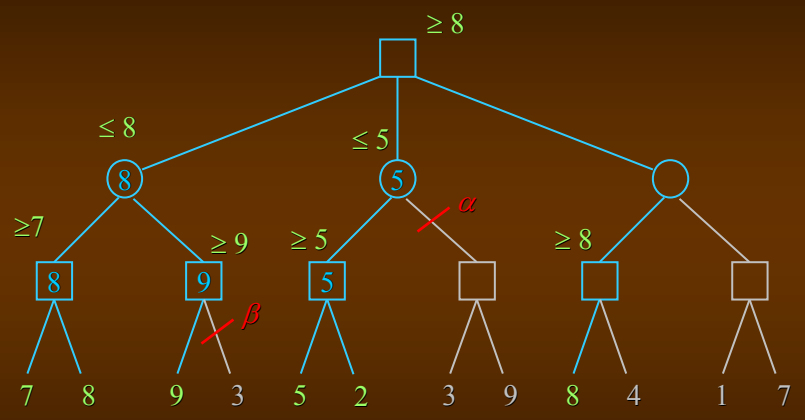
Odcięcia α-β



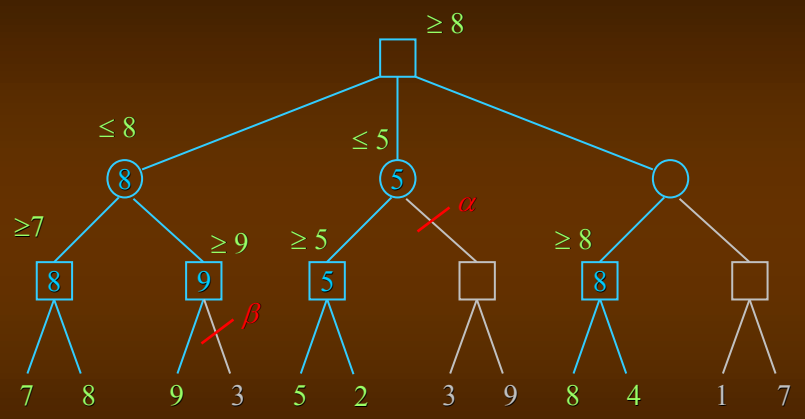
Odcięcia α-β



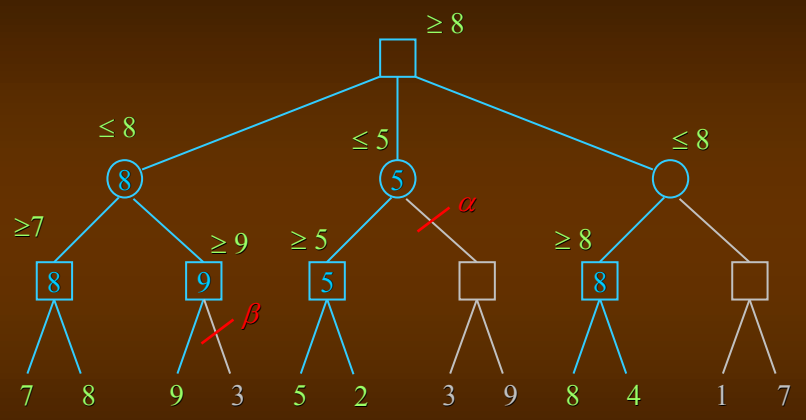
Odcięcia α-β



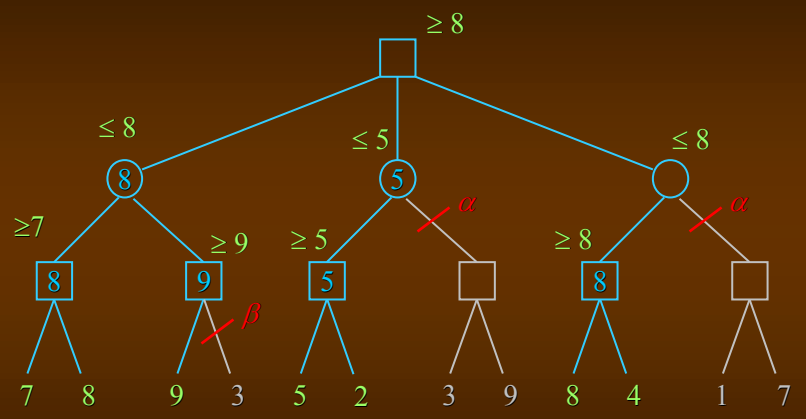
Odcięcia α-β



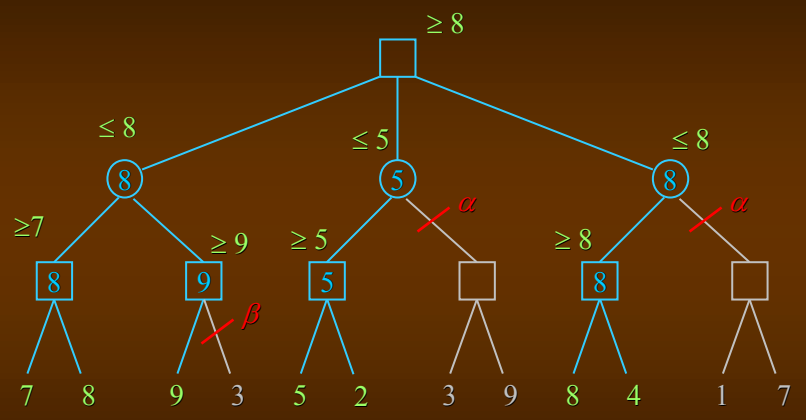
Odcięcia α-β



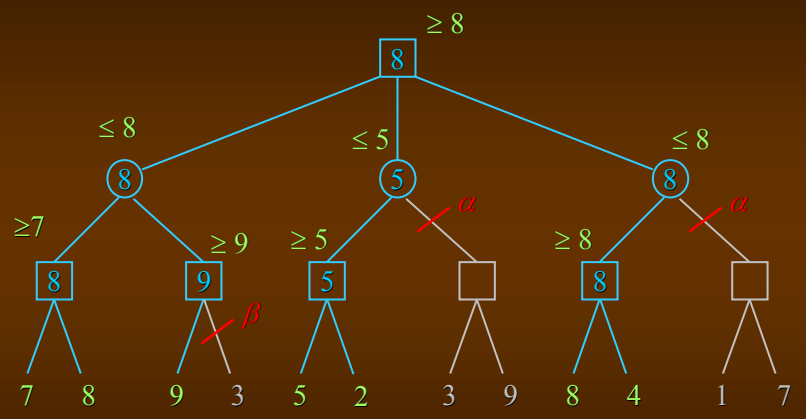
Odcięcia α-β



Odcięcia α-β



Odcięcia α-β



Mechanizm odcięć *alfa-beta*

- Dwa ograniczenia:
  - α – dolne ograniczenie dla wierzchołków Max (najwyższa wartość jaką dotychczas osiągnął gracz Max)
  - β – górne ograniczenie dla wierzchołków Min (najniższa wartość jaką dotychczas osiągnął gracz Min)
- Wartość ograniczenia α ustalana jest w wierzchołku Max, a wartość ograniczenia β - w wierzchołku Min
- Odcięcie α wykonywane jest w wierzchołku Min, a odcięcie β - w wierzchołku Max
- Kiedy tylko zachodzi warunek $\alpha \geq \beta$, nie ma potrzeby analizowania dalszych następników danego stanu

Algorytm *AlfaBeta* (zapis min-max)

wywołanie: `result = AlphaBeta(s, MAXDEPTH, -∞, ∞, MAX)`

```
int AlphaBeta(state s,int depth,int alpha,int beta,int type)
{
   if( is_terminal_node(s) || depth == 0 ) return(Eval(s));
   if( type == MAX){
      for(child=1; child<=NumOfSucc(s); child++) {
         val = AlphaBeta(Succ(s,child),depth-1,alpha,beta,MIN);
         alpha = max(val, alpha);
         if( alpha >= beta ) return beta; //cutoff
      } //endfor
      return alpha;
   }
   else { // type == MIN
      for(child=1; child<=NumOfSucc(s); child++) {
         val = AlphaBeta(Succ(s,child),depth-1,alpha,beta,MAX);
         beta = min(val, beta);
         if( alpha >= beta ) return alpha; //cutoff
      } //endfor
      return beta;
   }
}
```

Sformułowanie neg-max dla *AlfaBeta*

- Sformułowanie *min-max* wymaga przemiennych wywołań rekurencyjnych dwóch graczy (raz dla gracza MAX, dwa dla gracza MIN, itd.)
- Sformułowanie *neg-max* opiera się tylko na graczu MAX (jedna funkcja rekurencyjna)
- Przy wyjściu z rekurencji negujemy zwracaną wartość
- Przy zagnieżdżeniu rekurencyjnym w wersji *neg-max* negujemy ograniczenia i zamieniamy miejscami

Algorytm *AlfaBeta* (zapis neg-max)

wywołanie: `result = AlphaBeta(s, MAXDEPTH, -∞, ∞)`

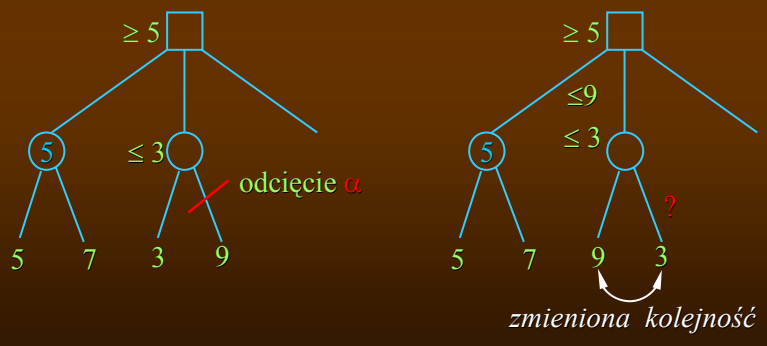
```
int AlphaBeta(state s, int depth, int alpha, int beta)
{
   if( is_terminal_node(s) || depth==0 ) return(Eval(s,depth));
   for(child=1; child<=NumOfSucc(s); child++) {
      val = -AlphaBeta(Succ(s,child),depth-1,-beta,-alpha);
      if( val > alpha ) alpha = val;    // alpha=max(val,alpha);
      if( alpha >= beta ) return beta; // cutoff
   } //endfor
   return alpha;
}
```

Cechy algorytmu *AlfaBeta*

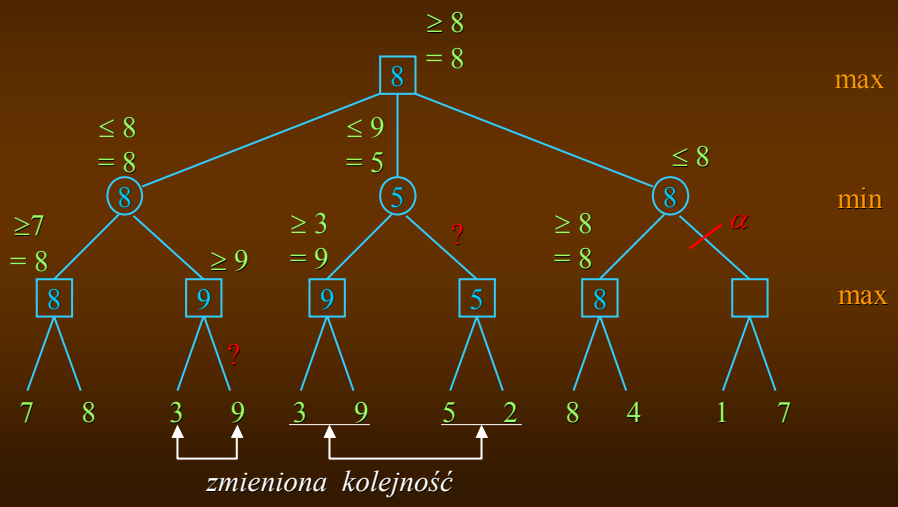
- Ścieżka krytyczna (ang. principal variation) – ścieżka w grafie przeszukiwania od korzenia do najlepszego liścia
- Wartości zwracane:
  - w wersji *minmax*: ze względu na gracza w korzeniu
  - w wersji *neg-max*: ze względu na tego czyj jest ruch w liściu
- Bardzo zawikłany kod – ewentualny błędy pozostają długo ukryte (problemy można zauważyć tylko wtedy, gdy niepoprawne wartości zostaną przepropagowane do korzenia grafu)
- Efektywność algorytmu zależy w ogromnym stopniu od kolejności następników i występowania odcięć

Analiza algorytmu *AlfaBeta* (2)

- Sytuacja idealna – jeśli odcięcie ma się pojawić, to powinno wystąpić jak najszybciej, czyli zaraz po sprawdzeniu pierwszego następnika



Odcięcia α-β (mniej odcięć!)



Złożoność algorytmu *AlfaBeta*

Dla danej głębokości (*d*) i stałego braching factor *(b)*

- Najlepszy przypadek: $O(b^{d/2})$
- Najgorszy przypadek: brak odcięć (czyli jak MinMax)
- Średni przypadek: $O((b/\log b)^d)$ [Knuth&Moore'75]

- Istnieje silna korelacja pomiędzy głębokością przeszukiwania a jakością osiąganych wyników (w szachach [Thompson'82])
- Pogłębienie przeszukiwania o jeden poziom przynosi znaczną poprawę rezultatów
- Wprowadzenie odcięć α -β pozwala zazwyczaj zwiększyć dwukrotnie głębokość przeszukiwania przy tej samej zajętości pamięci

Wady algorytmu *AlfaBeta*

- Efekt horyzontu (ang. horizont effect)
  - „Niewidoczny” spadek wartości stanu tuż za wyznaczoną głębokością przeszukiwania
  - Występuje we wszystkich odmianach algorytmu
  - Wykrywania stanów narażonych na wystąpienie efektu horyzontu i prowadzenia przeszukiwania za tym stanami - problem otwarty

Wariant *fail-soft* algorytmu *AlfaBeta*

- Klasyczna postać algorytmu – wartości zwracane zawsze z przedziału $[\alpha, \beta]$
- Wariant *fail-soft* algorytmu *AlfaBeta* [Fishburn’81] – zwraca dowolne wartości niezależnie od początkowego zakresu $[\alpha, \beta]$

Algorytm *AlfaBeta fail-soft* (zapis neg-max)

wywołanie: `result = AlphaBetaFS(s, MAXDEPTH, -∞, ∞)`

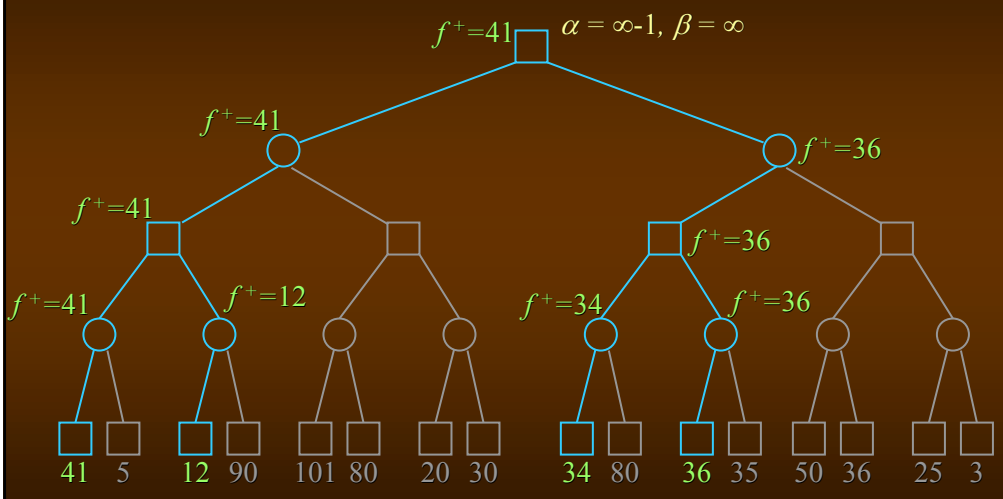
```
int AlphaBetaFS(state s, int depth, int alpha, int beta)
{
   if( is_terminal_node(s) || depth==0 ) return(Eval(s));
   best = -∞;
   for(child=1; child<=NumOfSucc(s); child++) {
      val = -AlphaBetaFS(Succ(s,child),depth-1,-beta,-alpha);
      if( val > best ) best = val;
      if( best >= beta ) break;        // cutoff
      if( best > alpha) alpha = best;
   } //endfor
   return best;
}
```


Znaczenie zakresu α-β

Załóżmy, że dla wierzchołka n o faktycznej wartości f procedura *AlfaBetaFS*(n, α, β) zwraca wartość g.
Możemy wyróżnić trzy następujące sytuacje:

- $\alpha < g < \beta$ (*sukces*) – g jest równe faktycznej wartości f
- $g \leq \alpha$ (*failing low*) – g jest górnym ograniczeniem dla f (oznaczane jako f^+), tzn. $f \leq g$
- $g \geq \beta$ (*failing high*) - g jest dolnym ograniczeniem dla f (oznaczane jako f^-), tzn. $f \geq g$

Zakresy α-β: sytuacja *failing low* ($g \leq \alpha$)



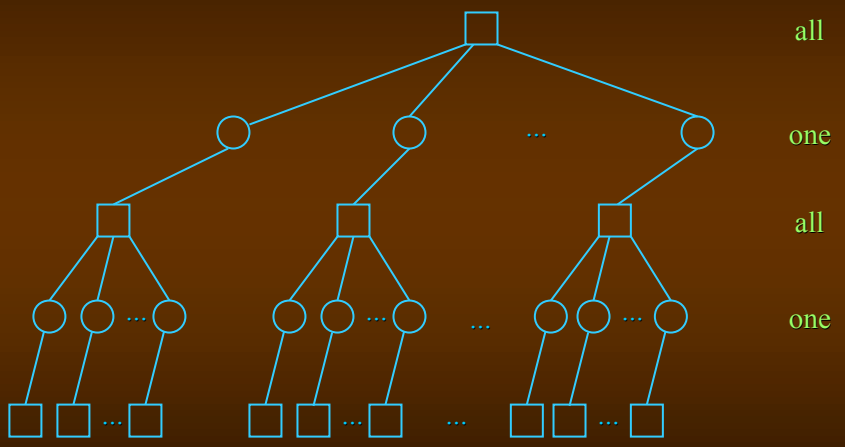
Zakresy α-β: *failing low*

Wywołanie procedury alfa-beta dla wierzchołka *n* z parametrami *AlfaBetaFS(n*, ∞-1, ∞*)* (wszystkie liście mają wartości mniejsze) spowoduje:

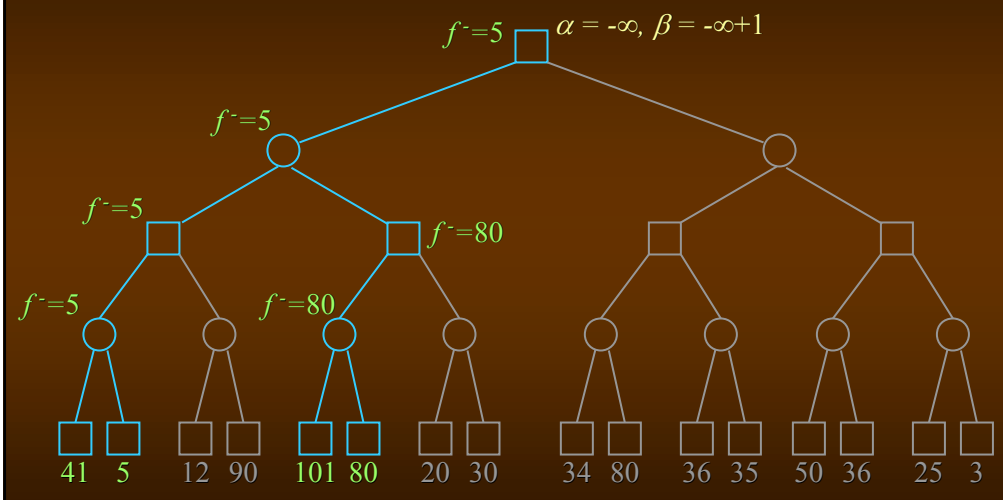
- we wszystkich wierzchołkach MIN wystąpienie odcięć α, bo wartości wszystkich następników są $g \leq \alpha = \infty\text{-}1$
- we wszystkich wierzchołkach MAX brak jakichkolwiek cięć β, bo wartości wszystkich następników są $g < \beta = \infty$

Otrzymane drzewo przeszukiwania będzie zawierać po jednym potomku dla każdego wierzchołka MIN i wszystkie potomne dla każdego wierzchołka MAX.

Failing low – dlaczego górne ograniczenie?



Zakresy α-β: sytuacja *failing high* ($g \geq \beta$)



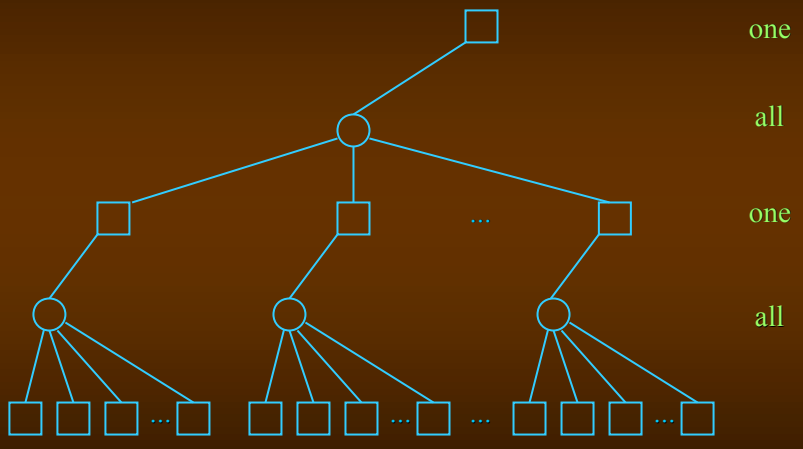
Zakresy α-β: *failing high*

Wywołanie procedury alfa-beta dla wierzchołka *n* z parametrami *AlfaBetaFS(n*, -∞, -∞+1*)* (wszystkie liście mają wartości większe) spowoduje:

- we wszystkich wierzchołkach MAX wystąpienie odcięć β, bo wartości wszystkich następników są $g \geq \beta = -\infty+1$
- we wszystkich wierzchołkach MIN brak jakichkolwiek cięć α, bo wartości wszystkich następników są $g > \alpha = -\infty$

Otrzymane drzewo przeszukiwania będzie zawierać po jednym potomku dla każdego wierzchołka MAX i wszystkie potomne dla każdego wierzchołka MIN.

Failing high – dlaczego dolne ograniczenie?



Poprawianie algorytmu *AlfaBeta*

- Modyfikacje sposobu przeszukiwania grafu
  - iteracyjne pogłębianie
  - zastosowanie pamięci (tzw. tablica przejść)
  - porządkowanie następników
  - manipulowanie zakresem α-β
  - zmienna głębokość przeszukiwania
  - przeszukiwanie eksploracyjne
- Doskonalenie funkcji oceny stanu (funkcji heurystycznej)
- Rozwiązania sprzętowe (np. obliczenia równoległe)

AlfaBeta jako przeszukiwanie w głąb

- Jak określić właściwą głębokość przeszukiwania?
- Czym można przeszukiwać do wierzchołków terminalnych?
  - Najczęściej nie! (Zbyt duża przestrzeń)
- Przeszukiwanie do ustalonej głębokości:
  - Niewłaściwa kolejność następników (ruchów) może doprowadzić do ogromnego grafu przeszukiwania
  - Co w sytuacji, gdy głębokość jest za mała?
  - Co w sytuacji, gdy głębokość jest za duża?

Iteracyjne pogłębianie

```
int iterative_deepening(state s)
{
  depth = 0;
  { depth++;
    value = AlfaBeta(s, depth, -∞, ∞);
    if( rescources_up() ) break;   // stop
  } while( depth < MAXDEPTH )
  return(value);
}
```


Iteracyjne pogłębianie

Zalety

- Osiąganie maksymalnej możliwej głębokości przeszukiwania przy aktualnie dostępnych zasobach (obliczenia w systemach czasu rzeczywistego!)
- Gwarancja znalezienia najlepszego rozwiązania do określonej głębokości przeszukiwania

Wady

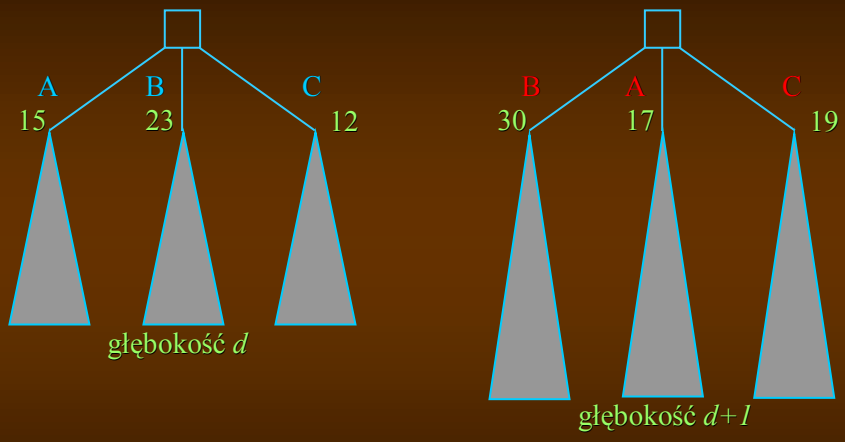
- Wielokrotne przeszukiwanie tych samych obszarów przestrzeni stanów

Iteracyjne pogłębianie

Potencjalne korzyści z poprzednich iteracji

- Przed przejściem do przeszukiwania na głębokości *d+1* można uporządkować ruchy na podstawie wyników uzyskanych dla głębokości *d*
- W większości gier słuszne jest założenie, iż najlepszy pierwszy ruch w przeszukiwaniu na głębokość *d* stanowi dobre przybliżenie najlepszego ruchu w przeszukiwaniu na głębokość *d+1*
- Wzrasta prawdopodobieństwo wybrania właściwego pierwszego ruchu im bliżej ostatniej (najkosztowniejszej) iteracji

Iteracyjne pogłębianie: *zmiana kolejności*



Badania eksperymentalne wykazały, iż koszty wielokrotnego przeszukiwania przestrzeni stanów są niewspółmiernie niskie w stosunku do zysków wynikających z uporządkowania ruchów w korzeniu grafu.

Metody porządkowania następników (ruchów)

- Mechanizmy statyczne – wykorzystanie wiedzy przedmiotowej
  - Zdobycie takiej wiedzy może być trudne!
- Mechanizmy dynamiczne – wykorzystanie wiedzy zdobytej podczas wcześniejszego przeszukiwania
  - tablica przejść
  - tablica historii ruchów (heurystyka historyczna)
  - mechanizm ETC
  - ruchy „zabójcy”

Tablica przejść (ang. *transposition table*)

- Tablica przejść – obszar pamięci, przechowujący wszystkie dotychczas odwiedzone stany
- Informacje zapisane w tablicy przejść:
  - identyfikator stanu (albo stan)
  - wartość stanu
  - ograniczenia (α,β) – opcjonalnie!
  - typ wartości
  - głębokość
  - najlepszy ruch (następnik)
- Przeszukiwanie kolejnego stanu w grafie gry poprzedzone jest weryfikacją jego występowania w tablicy przejść – dalsze działania zależeć będą od tego jakie informacje znaleziono w tablicy

Algorytm *AlfaBeta* z tablicą przejść

```
int AlphaBetaFSTT(state s, int depth, int alpha, int beta){
   int prevalpha = alpha;
   if( is_terminal_node(s) || depth==0 ) return(Eval(s));
   ptr = TTLookup(s); /* Sprawdzanie tablicy przejść */
   ...
   best = -∞;
   for(child=1; child<=NumOfSucc(s); child++) {
      val = -AlphaBetaFSTT(Succ(s,child),depth-1,-beta,-alpha);
      if( val > best) best = val;
      if( best >= beta ) break;          // cutoff
      if( best > alpha ) alpha = best;
   } //endfor
    /* Zapis do tablicy przejść */
   SaveTT(s, best, depth, prevalpha, beta);
   return best;
}
```


Tablica przejść (ang. *transposition table*)

- Stosując *tablicę przejść* można w znacznym stopniu zredukować efekt powtarzania przeszukiwania pewnych obszarów przestrzeni stanów
- Dzięki tablicy przejść możemy:
  - poprawić mechanizm wyboru następników w każdym stanie (wierzchołku grafu gry)
  - wykryć alternatywne ścieżki prowadzące do tego samego stanu
  - wykryć cykle
- Tablica przejść to mechanizm niezależny od dziedziny zastosowania (rodzaju gry)

Tablica przejść - jeśli znaleźliśmy stan...

- Załóżmy, że aktualna głębokość przeszukiwania wynosi *d'*, a znaleziony w tablicy stan był już analizowany na głębokość *d**:
  - jeżeli $d' < d$, to mamy do dyspozycji wynik, dokładniejszy od tego jaki bylibyśmy w stanie aktualnie uzyskać
  - jeżeli $d' = d$, to mamy do dyspozycji wystarczająco dokładny wynik
  - jeżeli $d' > d$, to mamy do dyspozycji wynik, na którym nie możemy polegać

* wartość *d* maleje wraz z pogłębianiem przeszukiwania

Tablica przejść - interpretacja wartości...

- Wiemy, że w trakcie przeszukiwania przestrzeni stanów możliwe są trzy interpretacje wartości *v* analizowanego stanu :
  - jeżeli $v <= \alpha < \beta$, to *v* jest górnym ograniczeniem właściwej wartości
  - jeżeli $\alpha < v < \beta$, to *v* jest właściwą wartością
  - jeżeli $\alpha < \beta <= v$, to *v* jest dolnym ograniczeniem właściwej wartości
- Informacja o tym, która z tych sytuacji miała miejsce muszą zostać zapisane w tablicy przejść (pozycja: typ wartości) i wykorzystane przy ponownej analizie stanu

Tablica przejść - zapis do tablicy

```
void SaveTT(state s,int val,int depth,int alpha,int beta)
{
   if( val <= alpha )
      bound = UPPER;
   else
   if( val >= beta )
      bound = LOWER;
   else
      bound = ACCURATE;
   InsertToTT(s, val, bound, depth);
}
```


Tablica przejść - sprawdzanie w tablicy

```
...
ptr = TTLookup(state);    // seeking in TT
if( ptr != NULL  &&  ptr->depth >= depth ){
   if( ptr->bound == LOWER )
      alpha = max(alpha, ptr->val);
   if( ptr->bound == UPPER )
      beta = min(beta, ptr->val);
   if( ptr->bound == ACCURATE )
      alpha = beta = ptr->val;
   if( alpha >= beta )    // TT's cutoff
      return ptr->val;
}
if( ptr != NULL ){       // ptr<-depth can be <depth !
   /* wybierz ptr->bestmove jako pierwszy */
}
...
```

Tablica przejść - implementacja

- Szybka metoda odwzorowania szukanego stanu w indeks tablicy
- Najczęściej implementowana jako tablica haszowa (konieczny stały koszt wyszukiwania!)
- Uwaga na konflikty w tablicach haszowych!
- Najpopularniejsza metoda - tablica haszowa Zobrist'a [Zobrist'90]

Tablica przejść - skuteczność

- Rezultaty zależne od rodzaju gry
  - Warcaby: o około 89% zredukowany obszar przeszukiwania!
  - Szachy: około 75%
  - Otello(Riversi): około 33%
- Większe zyski w grach, w których pojedynczy ruch nie powoduje dużych zmian w aktualnym stanie gry
- Podstawowa i najważniejsza metoda poprawy dla algorytmu *AlfaBeta*
- Skuteczność uzależniona także od stanu początkowego, głębokości przeszukiwania i rozmiaru tablicy haszowej

Tablica historii ruchów (heurystyka historyczna)

- Obserwacja: wykonanie ruchu *m* jest korzystne w stanie *p* (najwyższa wartość albo odcięcie)
- Historia ruchu *m*: ruch *m* jest teraz ruchem zalecanym w różnych stanach (najczęściej ruchem najlepszym)
- Zasada: preferuj ruchy o „pozytywnej" historii, o ile są dopuszczalne

Tablica historii ruchów

- Tablica historii ruchów (HT) - oceny wszystkich możliwych ruchów ale bez wskazania stanów, w których je stosowano
- Porządkowanie na podstawie tablicy HT wszystkich następników danego stanu przed przystąpieniem do ich przeszukiwania (sortowanie ruchów wg ocen z HT)
- Aktualizacja tablicy HT po przeszukaniu wszystkich następników danego stanu
- Ocena ruchów powinna zależeć od głębokości przeszukiwania (mniejsza głębokość to mniejsza ewentualna różnica pomiędzy stanami, z których wykonujemy ruch i tym samym większe znaczenie ruchu) - np. (`HT[move]+=2^depth`)

Algorytm *AlfaBeta* i tablica historii ruchów

wywołanie: `result = AlphaBeta(s, MAXDEPTH, -∞, ∞)`

```
int AlphaBeta(state s, int depth, int alpha, int beta)
{
   ...
   /* Ocena i porządkowanie ruchów w stanie s */
   for(child=1; child<=NumOfSucc(s); child++)
      score[child] = HT[Succ(s, child)];
   sort(score);
   ...
   /* Przeszukiwanie następników zgodnie z oceną */
   ...
   HT[bestmove] += (1 << depth);     // 2^depth
   ...
}
```


Tablica historii ruchów a tablica przejść

Tablica historii ruchów:

- Prosta forma uczenia się
- Bezkontekstowa ocena pojedynczego ruchu (w przeciwieństwie do tablicy przejść)
- Możliwość rozszerzenia kontekstu oceny o informacje zawarte w stanie, w którym wykonano ruch (większa precyzja oceny)

Tablica historii ruchów - skuteczność

- Prosta i ogólna metoda heurystyczna mocno ograniczająca rozmiar przeszukiwanej przestrzeni
- Niskie wymagania zasobowe, zarówno czasowe, jak i pamięciowe (łatwe do oszacowania)
- Skuteczna w większości zastosowań (gier)
- Wyniki zależne od zastosowania:
  - Szachy – faktycznie najlepszy ruch analizowany jako pierwszy w 90% przypadków
  - Warcaby - j.w.
  - Otello – faktycznie najlepszy ruch analizowany jako pierwszy w 80% przypadków

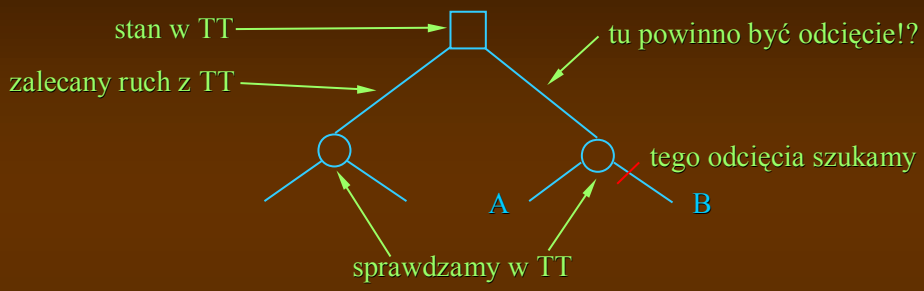
Heurystyka ruchów „zabójców” (ang. *killer heuristics*)

- Dla każdej głębokości przeszukiwania pamiętany jest niezależnie ruch, który spowodował największa liczbę odcięć (ruch „zabójca”)
- Kiedy przeszukiwanie dotrze po raz kolejny na daną głębokość stosujemy ruch ”zabójcę”, o ile jest to ruch dopuszczalny w aktualnym stanie
- Jeżeli inny ruch niż „zabójca” spowoduje odcięcie na danej głębokości, to staje się on nowym „zabójcą”
- Heurystyka ruchów „zabójców” - szczególny przypadek tablicy historii ruchów

ETC (ang. enhanced transposition cutoffs)

- Mechanizm bazujący na szerszym wykorzystaniu tablicy przejść
- Obserwacja: wybór najlepszego ruchu (bestmove) na podstawie tablicy przejść wcale nie musi być optymalny (jest tak tylko wtedy, gdy po faktycznym jego wykonaniu nastąpi odcięcie i był, to jedyny taki, powodujący odcięcie potomek)
- Po pierwsze: odcięcie nie musi wcale nastąpić!
- Po drugie: nawet jeśli nastąpi a inne ruchy też prowadzą do odcięć (albo nie nastąpi i żaden ruch nie prowadzi do odcięcia), to należy wybrać ten, poniżej którego z pewnością nastąpi odcięcie - otrzymamy wtedy jeszcze mniejsze poddrzewo przeszukiwania

ETC (ang. enhanced transposition cutoffs)



Zasada: zanim dokonamy właściwego przeszukiwania wierzchołków potomnych (wywołanie rekurencyjne, czyli faktyczny ruch gracza) sprawdzamy, czy którykolwiek z następników nie znajduje się już w tablicy przejść i czy jego zapamiętana wartość (najlepszy ruch) nie powoduje odcięcia.

ETC - implementacja

```
int AlphaBetaTTECT(state s, int depth, int alpha, int beta){
   int prevalpha = alpha;
   if( is_terminal_node(s) || depth==0 ) return(Eval(s,depth))
   ptr = TTLookup(s); /* Sprawdz. tab. przejść (aktual. stan) */
   ...                                  // TT's cutoff of state s
   for(child=1; child<=NumOfSucc(s); child++) {
      ptrch = TTLookup(Succ(s, child));
      if( ptrch != NULL && ptrch->depth >= depth-1 ){
         if( ptrch->val >= beta )  // TT's cutoff of child node
            return ptrch->val;
      }
   } //endfor
   best = -∞;
   for(child=1; child<=NumOfSucc(s); child++) {
      val = -AlphaBetaTTECT(Succ(s,child),depth-1,-beta,-alpha);
      if( val > best) best = val;
      if( best >= beta ) break;   // cutoff
      if( best > alpha ) alpha = best;
   } //endfor
    /* Zapis do tablicy przejść (aktualny stan) */
   SaveTT(s, best, depth, prevalpha, beta,);
   return best;
}
```


ETC (ang. enhanced transposition cutoffs)

- Wykorzystanie tablicy przejść jest kluczowe dla tego rozszerzenia algorytmu *AlfaBeta*
- Skuteczność: o 20-25% mniejszy graf przeszukiwania
- Ale: dodatkowe koszty sprawdzania potomków w tablicy przejść (zależne do gry: najczęściej 5%)
- Wskazówki: stosować tam gdzie możliwa jest największa redukcja koszów (największy zysk), czyli blisko korzenia grafu

Sterowanie zakresem α-β

- Metody manipulowania zakresem odcięć
  - Aspiration Search [Slate&Atkin'77]
  - Metody z minimalnym zakresem
    * NegaScout (PVS) [Reinefeld'83]
    * Rodzina algorytmów MTD [Plaat'96]

Zakresy α-β: *Aspiration Search*

- Tradycyjny zakres przeszukiwania $(-\infty, \infty)$
- Co w sytuacji , gdy jesteśmy w pewnym stopniu przewidzieć rezultat przeszukiwania?
- Przeszukiwanie z zakresem $(v - \Delta, v + \Delta)$, gdzie:
  v – spodziewany rezultat
  Δ – zakładane odchylenie od tej wartości ($\Delta > 0$)
- Mniejszy zakres alfa-beta oznacza więcej odcięć i mniejszy graf przeszukiwania
- Kiedy przewidywania się nie sprawdziły *(failing-low* lub *failing-high*), konieczność powtórzenia przeszukiwania z większym zakresem

Zakresy α-β: Algorytm *Aspiration Search*

```
int IDAspirationSearch(state s, deviation Δ)
{
   guess = 0;
   for(depth=1; !resources_up(); depth++) {
      alpha = guess-Δ; beta = guess+Δ;
      score = AlphaBetaFS(s, depth, alpha, beta);
      if( score >= beta ) {       // failing high
         alpha = score; beta = ∞;
         score = AlphaBetaFS(s, depth, alpha, beta);
      } else
      if( score <= alpha ) {      // failing low
         alpha = -∞; beta = score;
         score = AlphaBetaFS(s, depth, alpha, beta);
      }
      guess = score;
   } //endfor
   return(guess);
}
```


Zakresy α-β: Własności *Aspiration Search*

- Problem wyboru początkowej wartości zakresu alfa-beta (wartości ***guess*** oraz Δ)
- Najczęściej stosowany w wersji z iteracyjnym pogłębianiem (automatyzacja wyboru zakresu α-β: wartość korzenia wyznaczona w poprzedniej iteracji jest najlepszym przybliżeniem środka przedziału w następnej iteracji)
- Mniej efektywny niż metody z minimalnym zakresem α-β (omówione dalej)

Minimalny zakres α-β (ang. *null window search*)

- *AspirationSearch* nie może korzystać ze zbyt małego zakresu α-β, bo koszty powtórzeń przeszukiwania byłyby zbyt duże!
- Co oznacza zredukowanie zakresu do minimum, czyli $(v, v+1)$?
- Minimalny zakres („puste” okno α-β):
  - test logiczny: „Czy wynik $\leq v$, czy $> v$?”

[Pearl'80]

Minimalny zakres α-β: PVS (NegaScout)

- *AspirationSearch* ogranicza się tylko do manipulacji zakresem w korzeniu grafu, dlaczego nie robić tego w każdym wierzchołku grafu?
- Wykorzystywanie mechanizmów porządkowania ruchów (np. tablica przejść) gwarantuje z wysokim prawdopodobieństwem wybór najlepszego następnika
- Jeżeli najlepszy ruch w każdym wierzchołku znajdowany jest dość szybko, to resztę stanów potomnych lepiej przeszukiwać ze zredukowanym zakresem, tak aby odcięcia następowały jak najwcześniej (ang. *bad move proof*)

Minimalny zakres α-β: PVS (NegaScout)

- Ścieżka krytyczna (ang. *principal variation*) – ścieżka w grafie przeszukiwania od korzenia do najlepszego liścia (od którego pochodzi wynik)
- Obserwacja: gdyby mechanizmy porządkowania ruchów zawsze gwarantowały wybór najlepszego następnika, wszystkie pozostałe ruchy spoza ścieżki krytycznej miałyby gorszą wartość
- Przeszukiwanie z minimalnym zakresem („pustym" oknem α-β) jako test potwierdzający, że inne ruchy są gorsze (sytuacja *failing low*)
- Jeżeli wybór w oparciu o mechanizm porządkowania ruchów okaże się błędny (sytuacja *failing high*), należy powtórzyć przeszukiwanie z rozszerzonym zakresem α-β

Algorytm z min. zakresem α-β: PVS (NegaScout)

W każdym wierzchołku:

- Przeszukiwanie pierwszego ruchu z aktualnym zakresem
- Przeszukiwanie reszty ruchów z minimalnym zakresem w celu wykazania wyższości pierwszego ruchu:
  - Możliwość popełnienia błędu i konieczność powtórzenia przeszukiwania z szerszym zakresem
  - Ograniczenie kosztów ponownego przeszukiwania po błędzie poprzez zastosowanie tablicy przejść
  - Odpowiednia kolejność następników redukuje liczbę koniecznych powtórzeń - dysponując mechanizmem porządkowania następników i wybierania najlepszych ruchów szybko znajdujemy najlepszy następnik w każdym wierzchołku

Minimalny zakres α-β: PVS (NegaScout)

wywołanie: result = NegaScout(s, depth, -∞, ∞)

```
int NegaScout(state s, int depth, int alpha, int beta)
{
   if( is_terminal_node(s) || depth==0 ) return(Eval(s));
   best = -NegaScout(Succ(s,child),depth-1,-beta,-alpha);
   if( best >= beta ) return best;  // cutoff
   for(child=2; child<=NumOfSucc(s); child++) {
      if( best > alpha ) alpha = best;
      val = -NegaScout(Succ(s,child),depth-1,-alpha-1,-alpha);
      if( val >= beta ) return val;// cut off(failing too high)
      if( val > alpha)              //failing-high
         best = -NegaScout(Succ(s,child),depth-1,-beta,-val);
      else
         if( val >= best ) best = val;
   } //endfor
   return best;
}
```


Minimalny zakres α-β: PVS (NegaScout)

- Warunek stosowalności: grafy gry o głębokości co najmniej 3 [Reinefeld'83] - uważać pod koniec gry!
- *NegaScout* nigdy nie przeszukuje wierzchołków, odciętych przez standardowy algorytm *AlfaBeta* [Reinefeld'89]
- Jeżeli ma być skuteczna musi być używana razem z tablicą przejść (lub innym mechanizmem wyboru najlepszego ruchu)
- Skuteczność: zmniejszenie rozmiaru przeszukiwanej przestrzeni o około 10%

Minimalny zakres α-β: pozostałe metody

- Rozwiązanie ekstremalne: całe przeszukiwanie opiera się na serii testów logicznych z minimalnym zakresem
- Ruch w korzeniu określany jest podstawie zbieżnych do jednej wartości wielokrotnych wywołań algorytmu z minimalnym zakresem
- Rodzina algorytmów:
  - SSS* [Stockman'79]
  - DUAL* [Marsland i in.'87]
  - C* [Weill'91]
  - MTD(f) [Plaat'96]

Minimalny zakres α-β: algorytm SSS*

```
int SSS*(state s, int depth)
{
  g = +∞;
  {
    γ = g;
    g = AlphaBetaTT(s, depth, γ-1, γ);
  } while( g != γ );
  return g;
}
```

Metoda zbieżna do jednej wartości „od góry"

Minimalny zakres α-β: algorytm DUAL*

```
int DUAL*(state s, int depth)
{
  g = -∞;
  {
    γ = g;
    g = AlphaBetaTT(s, depth, γ, γ +1);
  } while( g != γ );
  return g;
}
```

Metoda zbieżna do jednej wartości „od dołu”

Minimalny zakres α-β: algorytm C*

```
int C*(state s, int depth)
{
  f+ = +∞; f- = -∞;
  {
    γ = average(f-, f+);
    g = AlphaBetaTT(s, depth, γ -1, γ);
    if( g < γ )    // failing low
      f+ = g;
    else           // failing high
      f- = g;
  } while( f+ != f- );
  return g;
}
```

Metoda „połowienia przedziału” ograniczeń

Minimalny zakres α-β: algorytm MTD(f)

- Metoda „połowienia przedziału” zbyt wolna
- Idea: zaczynać od przybliżenia faktycznej wartości
- Wykorzystanie wyników poprzednich iteracji w celu określenie minimalnego zakresu początkowego dla następnej iteracji

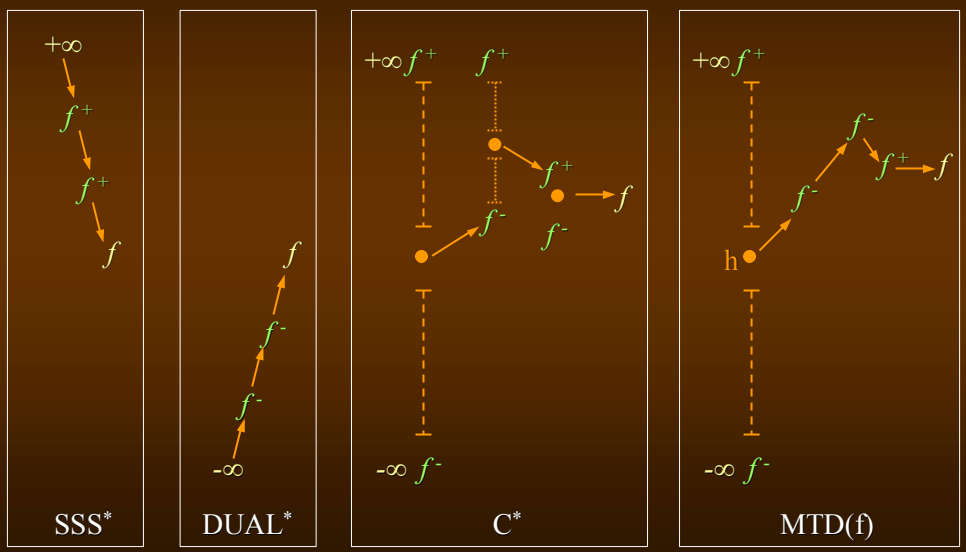
Minimalny zakres α-β: algorytm MTD(f)

```
int MTD(state s, int f, int depth)
{
  g = f;
  f+ = +∞; f- = -∞;
  {
    if( g == f- ) γ = g+1;  else  γ = g;
    g = AlphaBetaTT(s, depth, γ-1, γ);
    if( g < γ )
      f+ = g;      // failing low
    else
      f- = g;      // failing high
  } while( f+ != f- );
  return g;
}
```

Minimalny zakres α-β: algorytm MTD(f)

- Wszystkie iteracje przeszukiwania „idą w dół” (oprócz ostatniej) albo wszystkie „idą w górę” (oprócz ostatniej)
- Stosowanie MTD(f) ma sens (efektywność!), jeśli zostanie zastosowana tablica przejść, umożliwiająca wykorzystanie wyników poprzednich iteracji (odnosi się to również do innych algorytmów z tej klasy)
- Stanowi uogólnienie algorytmów: SSS* i DUAL*
- Wyniki eksperymentów wykazują przewagę MTD(f) nad algorytmem NegaScout (PVS)
- Skuteczność: 5-15% mniej przeszukanych wierzchołków

Zakresy α-β: pozostałe metody z minimalnym zakresem



Sterowanie głębokością przeszukiwania

- Dotychczas wszystkie algorytmy przeszukiwały do zadanej (z góry ustalonej) głębokości
- Założenie to pozwala:
  - porównywać grafy przeszukiwania i osiągnięte rezultaty
  - w pewnym stopniu przewidywać rezultaty przeszukiwania (skuteczność albo zasadność)
- Założenie to jest sztuczne! Czyż nie lepiej byłoby analizować głębiej podgrafy rokujące osiągnięcie dobrych rezultatów, zaś płycej te o gorszych prognozach?
- Zmienna głębokość przeszukiwania to próba lepszego rozłożenia nakładów obliczeniowych w celu pozyskania dokładniejszych informacji z procesu przeszukiwania

Sterowanie głębokością przeszukiwania

- Metoda „pustego ruchu” [Beal’90,Donninger’93]
- ProbCut [Buro’95] – (nie omówiony!)
- Metody pogłębiania przeszukiwania
  - Poszukiwanie stanów stabilnych [Slate&Atkin’77]
  - Pogłębianie pojedynczych stanów [Anantharaman i in.’90]

Sterowanie głębokością: metoda „pustego ruchu” (ang. null move search)

- W większości gier opcja braku ruchu („pusty ruch”) jest niedozwolona
- Wykonujemy przeszukiwanie z „pustym ruchem” i traktujemy otrzymany rezultat jako dolne ograniczenie wartości jaka faktycznie może być osiągnięta
  - Opiera się na założeniu, że wykonanie jakiegoś konkretnego ruch („niepustego”) dałoby niegorszy wynik niż brak ruchu
  - Najczęściej każdy ruch poprawia sytuację gracza, więc nie może być nic gorszego niż brak jakiegokolwiek ruchu!

Algorytm *AlphaBeta* z „pustym ruchem”

wywołanie: `result = AlphaBetaNMS(s, MAXDEPTH, -∞, ∞)`

```
#define  R   2
int AlphaBetaNMS(state s, int depth, int alpha, int beta)
{
  if( is_terminal_node(s) || depth==0 ) return(Eval(s));
  if( depth-1-R > 0 )
    score = -AlphaBetaNWS(s,depth-1-R,-beta,-beta+1);
  if( score >= beta ) return score;

  best = -∞;
  for(child=1; child<=NumOfSucc(s); child++) {
    val = -AlphaBetaNMS(Succ(s,child),depth-1,-beta,-alpha);
    if( val > best ) best = val;
    if( best >= beta ) break; // cutoff
    if( best > alpha) alpha = best;
  } //endfor
  return best;
}
```

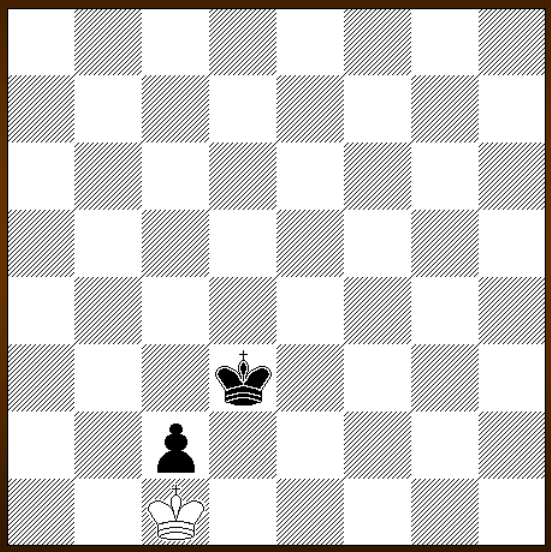
Algorytm *AlphaBeta* z „pusty ruchem”

- Metoda heurystyczna pomijania przeszukiwania tych stanów w grafie przestrzeni stanów, których ocena jest wystarczająco dobra (jeżeli oddanie ruchu nie poprawia sytuacji przeciwnika, to dany stan jest wystarczająco dobry)
- Testowanie odbywa się w oparciu o przeszukiwanie z minimalnym zakresem α-β i ograniczoną głębokością (współczynnik R) prowadzonym na tym samym stanie (brak ruchu)
- Jeśli otrzymany rezultat jest większy od β, to nie ma potrzeby dalszego przeszukiwania; w przypadku przeciwnym realizowane jest normalne przeszukiwanie

Algorytm *AlphaBeta* z „pusty ruchem”

- Heurystyka w znacznym stopniu ograniczająca przestrzeń przeszukiwania (można pogłębić całkowite przeszukiwanie o 1 lub 2 poziomy)
- Zawodna w niektórych grach (np. warcaby, końcówki szachowe)
  - kolejny drugi ruch stwarza często przeciwnikowi możliwość “ucieczki” z pozycji przegranej,co nie byłoby możliwe, gdyby miał do dyspozycji tylko jeden ruch
  - stany *zugzwang* - w niektórych grach zachodzą sytuacje, w których żaden z możliwych ruchów nie przynosi poprawy, a wręcz przeciwnie, pogarsza jeszcze sytuację (brak ruchu jest wtedy lepszy niż jakikolwiek ruch)
- Dobór wartości R: co najmniej 1, najczęściej 2

Stany *zugzwang* - przykład



Metody pogłębiania przeszukiwania (ang. *search extensions*)

- Jeśli istnieje interesująca, obiecująca lub niestabilna ścieżka być może należy pogłębić jej analizę?
- Jaki czynniki decydują o tym, że ruch jest „interesujący” ?
- Być może trzeba skorzystać z wiedzy specyficznej dla danego zadania (np. szach w szachach)?

Poszukiwanie stanów stabilnych (ang. *quiescence search*)

- Efekt horyzontu!
- Poszukiwanie stanów, których oceny jesteśmy pewni, nawet jeśli możliwe są dalsze ruchy
- Ocena heurystyczna liścia jest pewniejsza, jeżeli stan gry, któremu on odpowiada jest stanem stabilnym
- Stany, w których pojawiają się ruchy krytyczne (np. bicia, szach, promocje) wymagają dokładniejszej analizy
- Metoda zależna od dziedziny zastosowania (gry)

Poszukiwanie stanów stabilnych: kiedy?

- Po osiągnięciu głębokości 0, zamiast oceny heurystycznej stanu, dodatkowe pogłębienie przeszukiwania
- Pogłębianie przeszukiwania (ang. *quiescence search*) to rodzaj oceny oparty nie na własnościach statycznych (funkcja heurystyczna oceny stanu), lecz własnościach dynamicznych, wynikających z dodatkowego przeszukiwania

Poszukiwanie stanów stabilnych: jakie ruchy?

- Dodatkowe przeszukiwanie: najczęściej analizuje się tylko i wyłącznie ruch bicia (pogłębianie przeszukiwania odbywa się na nieznaną z góry liczbę poziomów, jeśli wystąpią sekwencję bić!)
- Problem: jaka kolejność przeszukiwania, jeżeli w ocenianym stanie można wykonać kilka bić?
- Uwaga! Koszt pogłębiania może przekroczyć koszty regularnego przeszukiwania (ang. *quiescence search explosion*)!

Poszukiwanie stanów stabilnych: algorytm

http://www.seanet.com/~brucemo/topics/quiescent.htm

Pogłębianie pojedynczych ruchów (ang. *singular extensions*)

- Głębsza analiza ruchów interesujących - heurystyka wykorzystywana przez ludzi (np. w szachach)
- Identyfikacja ruchów osobliwych (wyjątkowych) i pogłębienie ich przeszukiwania w celu dokładniejszej analizy
- Jedna z możliwych interpretacji ruchu wyjątkowego: ruch o wartości znacząco wyższej od innych ruchów alternatywnych (rodzeństwa)
- Manipulowanie zakresem α-β w celu potwierdzenia wyjątkowości ruchu
- Metoda dynamiczna (uniwersalna) - niezależna od dziedziny zastosowania (gry)

Pogłębianie pojedynczych ruchów: kiedy?

- Ruch wyjątkowy: ten, którego wartość jest co najmniej Δ większa od wartości ruchów alternatywnych
- Wartość najlepszego (wyjątkowego) ruchu: v
- Przeszukiwanie pozostałych ruchów z zakresem (v- Δ, v-Δ+1)
- Jeśli którykolwiek ruch alternatywny zwróci wartość $\geq v$-Δ+1 (*failing high*), konieczny jest powrót do normalnego przeszukiwania
- Jeśli wszystkie ruchy alternatywne zwrócą wartość $\leq v$-Δ, oznacza to, iż został znaleziony ruch wyjątkowy

Pogłębianie pojedynczych ruchów: jak?

- Kiedy ruch wyjątkowy zostanie już znaleziony należy pogłębić jego przeszukiwanie o kilka dodatkowych poziomów
- Uwaga! Efekt bardzo głębokiego przeszukiwania - wielokrotne rekurencyjne wykonanie pogłębiania pojedynczego ruchu
- Przechowywanie informacji o ruchach wyjątkowych w tablicy przejść

Metody pogłębiania przeszukiwania: skuteczność

- Wyniki eksperymentów obliczeniowych potwierdzają wyższość algorytmów przeszukiwania z mechanizmem pogłębiania pojedynczych ruchów nad standardowym przeszukiwaniem
- Szachy
  DeepBlue dla głębokości 12 przeszukiwano sekwencje prowadzące do wygranej w 40-ruchach!
- Warcaby
  Chinook przy standardowej głębokości 19, średnia głębokość analizy wynosiła 26, a maksymalna 45!

Sterowanie głębokością przeszukiwania: wnioski

- Przeszukiwanie z zadaną (stałą) głębokością to mało efektywna strategia sterowania głębokością przeszukiwania
- Metoda „pustego ruchu” jest prosta w implementacji i zwykle bardzo skuteczna (ale nie uniwersalna!)
- Brak dobrych uniwersalnych metod pogłębiania przeszukiwania:
  - Duże wymagania zasobowe metody pogłębiania pojedynczego ruchu!

Podsumowanie, czyli algorytmy przeszukiwania w praktyce

- Ogromny rozmiar przestrzeni stanów:
  - Warcaby - 10^{40}
  - Szachy - 10^{120}
  - Go – 10^{350}
- Podstawowe algorytmy przeszukiwania mają krótki kod (z reguły ok. 20 linii) i dobrze opracowana teorię
- Sam wybór właściwego algorytmu dla konkretnego problemu zadaniem trywialnym (internet!)
- Zapewnienie efektywności w realnych zastosowaniach możliwe jedynie po udoskonaleniu podstawowych wersji algorytmów (możliwy zysk rzędu 90% i więcej!)
- Obecnie stosowane algorytmy: NegaScout i MTD(f)

Podsumowanie, czyli algorytmy przeszukiwania w praktyce

Wniosek

Efektywne algorytmy dla gier to poważne wyzwanie programistyczne. Ogromna część wysiłku włożonego w implementację algorytmów dla gier skierowana jest na testowanie, poprawki kodu i doskonalenie algorytmu.

Literatura (1)

[1] Thomas Anantharaman, Murray Campbell and Feng-hsung Hsu. "Singular Extensions: Adding Selectivity to Brute-Force Searching", Artificial Intelligence, vol. 43, no. 1, pp. 99-109, 1990.

[2] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. "Learning to Play Chess with Temporal Differences", Machine Learning, vol. 40, no. 3, pp. 243-263, 2000.

[3] Don Beal. "A Generalized Quiescence Search Algorithm", Artificial Intelligence, vol. 43, no. 1, pp. 85-98, 1990.

[4] H. Berliner. "The B* Tree Search Algorithm: A Best First Proof Procedure", Artificial Intelligence, vol. 12, pp. 23-40, 1979.

[5] H. Berliner and C. McConnell. B* probability based search. Artificial Intelligence, vol. 86, pp. 97-156, 1996.

[6] Michael Buro."ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm", ICCA Journal, vol. 18, no. 2, pp. 71-76, 1995.

Literatura (2)

[7] Michael Buro. “From Simple Features to Sophisticated Evaluation Functions”, Computers and Games, Springer-Verlag, LNCS 1558, 1998.

[8] M. Buro. “Toward Opening Book Learning”, ICCA Journal, vol. 22, no. 2, pp. 98-102, 1999.

[9] Murray Campbell and Tony Marsland. “A Comparison of Minimax Tree Search Algorithms”, Artificial Intelligence, vol. 20, pp. 347-367, 1983.

[10] Chrilly Donninger. “Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs”, ICCA Journal, vol. 16, no.3, pp. 137-143, 1993.

[11] Fishburn J. „Three optimizations of alfa-beta search”, Computer Science Department, University of Wisconcin-Madison, Ph.D. Thesis, 1981.

Literatura (3)

[12] Richard E. Korf and David W. Chickering. Best-first minimax search: Othello results. In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94), volume 2, pages 1365—1370, 1994.

[13] D. Knuth and R. Moore. An analysis of alpha-beta pruning. Artificial Intelligence, vol.6(4), pp.293-326, 1975.

[14] R. Lake, J. Schaeffer and P. Lu. “Solving Large Retrograde Analysis Problems on a Network of Workstations”, Advances in Computer Chess VII, pp. 135-162, 1994.

[15] T. Lincke. “Strategies for Automatic Construction of Opening Books”, Computers and Games, T. Marsland and I. Frank (eds), pp. 74-86, Springer Verlag, 2002.

[16] D. McAllester. “Conspiracy Numbers for Min-Max Search”, Artificial Intelligence, vol. 35, pp. 287-310, 1988.

Literatura (4)

[17] T. Anthony Marsland, Alexander Reinefeld, and Jonathan Schaeffer. Low over-head alternatives to SSS*. Artificial Intelligence, vol. 31, pp.185-199, 1987.

[18] www.cs.vu.nl/~aske/mtdf.html

[19] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. "Best-first Fixed-depth Minimax Algorithms", Artificial Intelligence, vol. 87, no. 1-2, pp. 1-38, 1996.

[20] A. Plaat. Research Re: Search and Re-Search. PhD thesis, Erasmus University, The Netherlands, 1996.

[21] A. Palay. "The B* Tree Search Algorithm -- New Results", Artificial Intelligence, vol. 19, pp. 145-163, 1982,

[22] J. Pearl. Scout: A simple game-searching algorithm with proven optimal properties. In AAAI National Conference, pp. 143-145, 1980.

Literatura (5)

[23] Alexander Reinefeld. "An Improvement to the Scout Tree Search Algorithm", ICGA Journal, vol. 6, no.4, pp. 4-14, 1983.

[24] A. Reinefeld. Spielbaum Suchverfahren. Informatik-Fachberichte 200, Springer Verlag, 1989.

[25] A. Reinefeld and T.A. Marsland. "A Quantitative Analysis of Minimax Window Search", IJCAI, pp. 951-954, 1987.

[26] J.Schaeffer. "The History Heuristic and the Performance of Alpha-Beta Enhancements in Practice", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 11, no. 11, pp. 1203-1212, 1989.

[27] D. Slate and L. Atkin. Chess 4.5 - The Northwestern University chess program. In P. Frey, editor, Chess Skill in Man and Machine, pages 82-118. Springer Verlag, 1977.

[28] George C. Stockman. A minimax algorithm better than alpha-beta? Artificial Intelligence, vol. 12, pp.179-196, 1979.

[29] K. Thompson. Computer chess strength. In M. Clarke, editor, Advances in Computer Chess 3, pp. 55-56. Pergamon Press, 1982.

Literatura (6)

[30] K. Thompson. “Retrograde Analysis of Certain Endgames”, ICCA Journal, vol. 9, no.3, pp. 131-139, 1986.

[31] V. Allis, M. van der Muellen, and J. van den Herik. “Proof-number Search”, Artificial Intelligence, vol. 66, pp. 91-124, 1994.

[32] Weill J.C., Experiments With the NegaC* Search. In Heuristics Programming in Artificial Intelligence 2, D.N.L. Levy, D.F. Beal (Eds) 1991, pp. 174-187.

Strony WWW

http://www.xs4all.nl/~verhelst/chess/search.html

http://peg.it.uu.se/~saps01/FersmanMokrushin/

http://www.seanet.com/~brucemo/topics/topics.htm

http://www.zib.de/reinefeld/Research/nsc.html

http://www.fierz.ch/strategy.htm

http://www.brillianet.com/programming/artificial_intelligence/tutorials/index.htm

http://www.maths.nott.ac.uk/personal/anw/G13GT1/compch.html

http://sern.ucalgary.ca/courses/CPSC/533/W99/presentations/L1_5B_McCullough_Melnyk/

http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11/

http://www.cs.biu.ac.il/~davoudo/intro.html