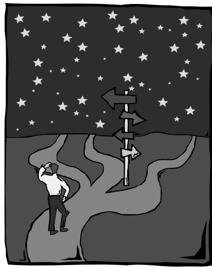
Wnioskowanie jako przeszukiwanie przestrzeni stanów

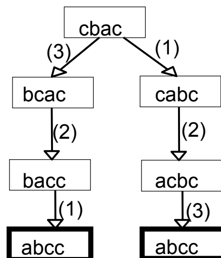


Plan wykładu

- Rozwiązywanie problemów jako poszukiwanie ścieżki rozwiązania
- Przestrzeń stanów jako graf skierowany
- Dokładne metody przeszukiwania przestrzeni stanów
- Przybliżone metody przeszukiwania przestrzeni stanów

Systemy produkcyjne

ba→ab	(1)
ca→ac	(2)
cb→bc	(3)



Przestrzeń stanów jest to czwórka uporządkowana **[N, A, S, GD],** gdzie:

N jest zbiorem wierzchołków odpowiadających stanom w procesie rozwiązywania problemu

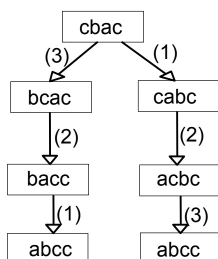
A jest zbiorem łuków, odpowiadających krokom w procesie rozwiązywania problemu

S jest niepustym podzbiorem N, zawierającym stany początkowe problemu

GD jest niepustym podzbiorem N, zawierającym stany docelowe problemu.

N – zbiór stanów

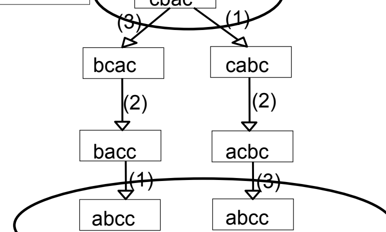
Systemy produkcyjne



N – zbiór stanów

A – zbiór kroków

S – zbiór stanów początkowych

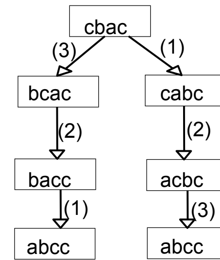


GD – zbiór stanów docelowych

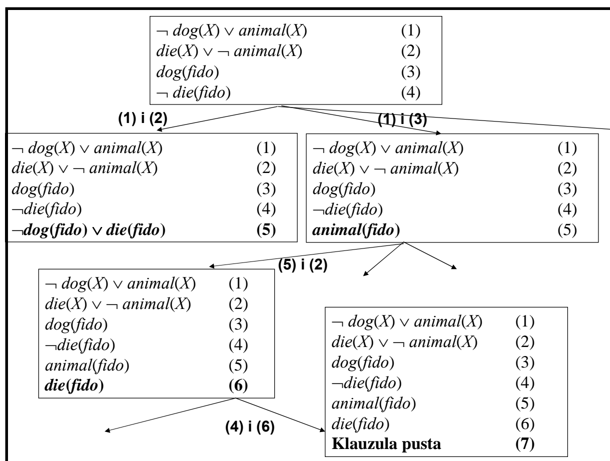
Stany w GD są opisane:

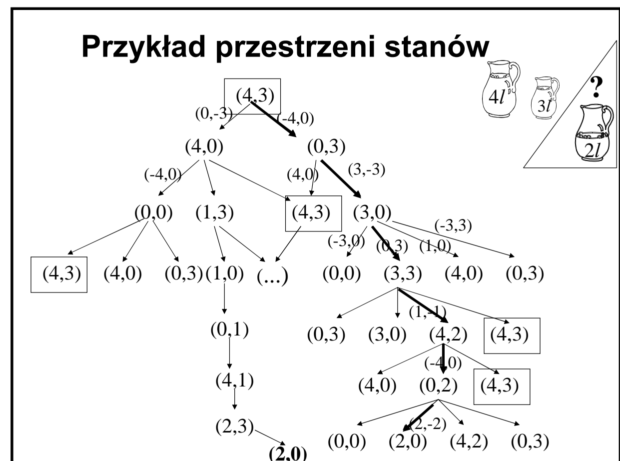
1. przez podanie własności stanów występujących w przeszukiwaniu
2. przez podanie własności ścieżki tworzonej podczas przeszukiwania

Ścieżką rozwiązania nazywamy ścieżkę wiodącą przez ten graf z wierzchołka z S do wierzchołka w GD.



Dwie ścieżki rozwiązania





Metody rozwiązywania problemów

- Czy metoda gwarantuje znalezienie rozwiązania?
- Czy algorytm zakończy się w każdym przypadku, czy może wpaść w pętlę nieskończoną?
- Czy jeśli rozwiązanie zostanie znalezione, to mamy gwarancję, że jest ono optymalne?
- Jaka jest czasowa i pamięciowa złożoność obliczeniowa algorytmu?
- W jaki sposób interpreter może najlepiej zredukować złożoność przeszukiwania?
- Jak można zaprojektować interpreter najbardziej efektywny z punktu widzenia wykorzystanego języka reprezentacji wiedzy?

Algorytmy przeszukiwania

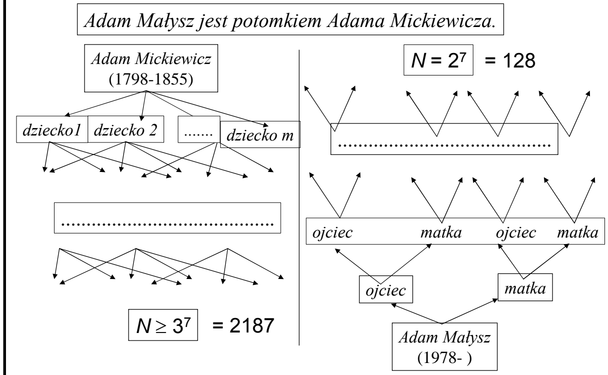
Zadaniem algorytmów przeszukiwania jest znalezienie ścieżki rozwiązania w przestrzeni problemu.

Strategie przeszukiwania:

- wszerz
- w głąb
- w przód
- w tył

Implementacja przeszukiwania: rekurencja.

Strategia przeszukiwania a złożoność



Kiedy stosować przeszukiwanie w tył?

- Gdy cel lub hipoteza jest dana w sformułowaniu problemu, np. w dowodzeniu twierdzeń matematycznych, systemach diagnostycznych.
- Gdy liczba reguł możliwych do zastosowania rośnie szybko, a wczesna eliminacja celów może wyeliminować przeszukiwanie pewnych gałęzi.
- Gdy stan początkowy nie jest dany explicite, ale musi być rozpoznany. Przeszukiwanie wstecz może pomóc w pokierowaniu pozyskiwaniem danych (np. diagnostyka medyczna).

Kiedy stosować przeszukiwanie w przód?

- Gdy wszystkie lub większość danych jest zawartych w sformułowaniu problemu (np. interpretacja).
- Gdy występuje duża liczba potencjalnych celów, ale jest tylko kilka możliwości zastosowania faktów i informacji wejściowych do konkretnej instancji problemu.
- Gdy trudno jest sformułować hipotezę docelową.

Przeszukiwanie heurystyczne

Funkcja oceny heurystycznej

$$f(n) = g(n) + h(n)$$

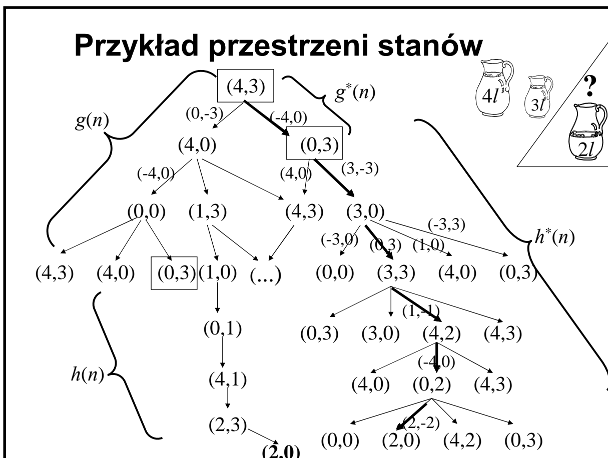
gdzie:

$g(n)$ mierzy aktualną długość ścieżki od stanu n do stanu początkowego

$h(n)$ jest heurystycznym oszacowaniem odległości stanu n od celu.

Jeżeli taka funkcja jest realizowana ze strategią przeszukiwania *best-first-search*, to algorytm nazywa się algorytmem A.

Przykład przestrzeni stanów



Algorytm A*

Jeżeli algorytm A wykorzystuje funkcję oceny heurystycznej taka, że $h(n) \leq h^*(n)$, to otrzymany w ten sposób algorytm nazywa się algorytmem A*.

Dopuszczalność

Algorytm przeszukiwania jest dopuszczalny (*admissible*), gdy gwarantuje znalezienie najkrótszej ścieżki do rozwiązania, jeżeli taka ścieżka istnieje.

Wszystkie algorytmy A^* są dopuszczalne.

Algorytm przeszukiwania wszerz jest algorytmem A^*.

$f(n) = g(n) + 0$

Monotoniczność

Funkcja $h(n)$ jest monotoniczna, gdy:

1. $h(n_i) - h(n_j) \le cost(n_i, n_j)$

gdzie $cost(n_i, n_j)$ jest rzeczywistym kosztem (liczbą kroków) przejścia od stanu n_i do stanu n_j

2. Ocena heurystyczna stanu docelowego $h(goal) = 0$.

Heurystyka monotoniczna osiąga każdy stan najkrótszą ścieżką.

Twierdzenie 1

Każda heurystyka monotoniczna jest dopuszczalna.

DOWÓD

Niech $s_1, s_2, ..., s_g$ będzie dowolną ścieżką od jakiegoś stanu początkowego s_1 do celu s_g.

Na podstawie definicji monotoniczności, dla każdej pary kolejnych stanów na tej ścieżce zachodzi:

$h(s_1) - h(s_2) \le cost(s_1, s_2)$

$h(s_2) - h(s_3) \le cost(s_2, s_3)$

....

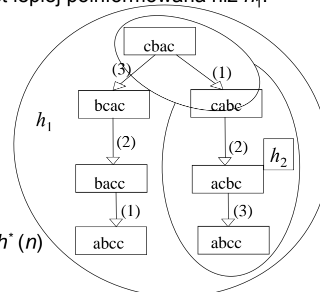
$h(s_{g-1}) - h(s_g) \le cost(s_{g-1}, s_g)$

$h(s_1) - h(s_g) \le cost(s_1, s_g) = h^*(s_1)$ $h(s_g) = 0$

$h(s_1) \le h^*(s_1)$

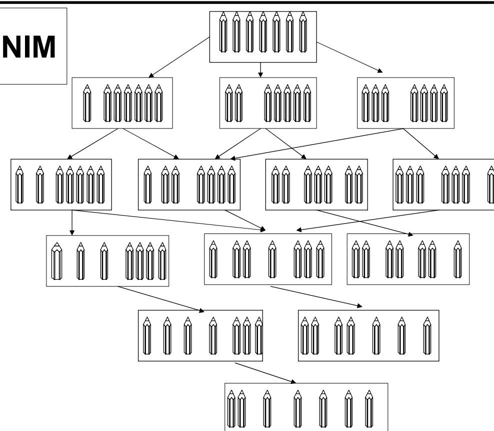
Poinformowanie

Dla dwóch heurystyk h_1 i h_2 typu A^*, jeżeli $h_1(n) \le h_2(n)$ dla wszystkich stanów w przeszukiwanej przestrzeni, to o h_2 mówimy, że jest lepiej poinformowana niż h_1.

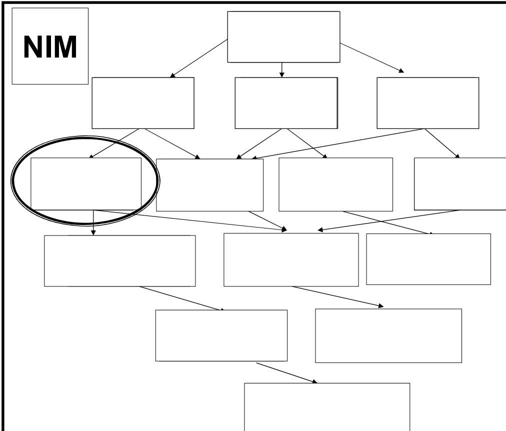


$h_1(n) \le h_2(n) \le h^*(n)$

NIM



NIM



Algorytm MIN-MAX

Graczy oznaczamy MIN i MAX

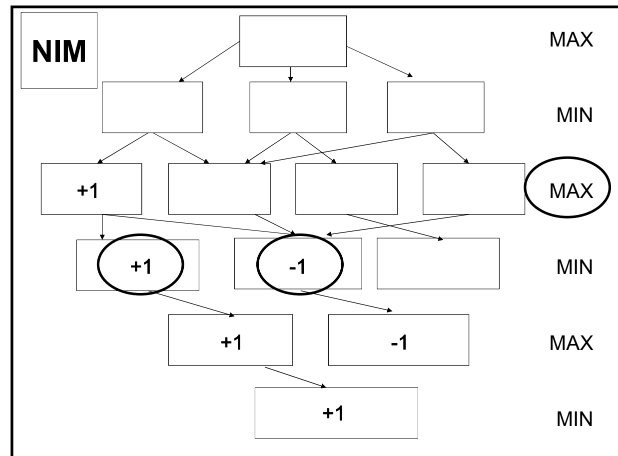
Liczba punktów wygrana przez gracza MAX jest równocześnie liczbą punktów przegraną przez gracza MIN. Zatem suma wygranych jest równa zero: gra o sumie zerowej.

Wartością gry nazywamy wygraną gracza MAX.

Gracz MAX stara się zmaksymalizować wartość gry.

Gracz MIN stara się zminimalizować wartość gry.

Wartość gry	MAX	MIN
5	5	– 5
– 4	– 4	4
0	0	0



Procedura MINiMAX

Przeciwnicy posiadają taką samą wiedzę o przestrzeni stanów i stosują tę wiedzę konsekwentnie w celu wygrania gry.

Jeżeli ojcem jest MIN, to nadaj mu najmniejszą wartość spośród dzieci.

Jeżeli ojcem jest MAX, to nadaj mu największą wartość spośród dzieci.

Podsumowanie

- Przeszukiwanie przestrzeni stanów jest ogólnym modelem rozwiązywania problemów
- Algorytmy dokładne przeszukiwania są zwykle zbyt pracochłonne
- Algorytmy heurystyczne wymagają dostosowania do charakterystyki rozwiązywanego problemu
- Można zdefiniować ogólne własności heurystyk: dopuszczalność, monotoniczność i poinformowanie

```
evalutemin(u,  B) //u is a min node
   {
              Alpha=+infinity;
              if u =leaf return the score;
              else
                     for all children v of u
                     {
                           Val = evalutemax(v,  B);
                           alpha= Min{alpha, Val};
                           if  Alpha<=beta  then exit
loop;
                      }
              Return alpha;
}
```

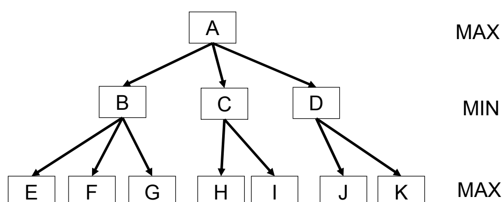
```
evalutemax(u,B)   // u is a Max node
   {
                    alpha=-infinity;
                     if u=leaf return the score;
                     else
                          for all children v of u
                          {
                             Val = evalutemin(v, B);
                             Alpha = Max{Alpha, Val};
                             if Alpha >= Beta then exit
loop;
                          }
                     Return Alpha;
    }
```

Heurystyczne metody przeszukiwania przestrzeni stanów

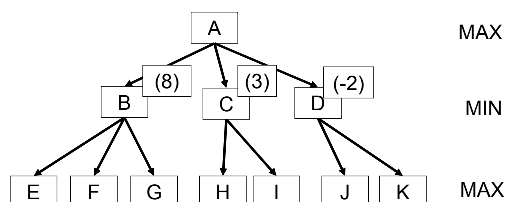
Plan wykładu

- Reprezentacja gry w postaci mini-max i nega-max
- Zasady implementacji algorytmu minimax
- Algorytm alfa-beta
- Algorytm B*
- Podsumowanie

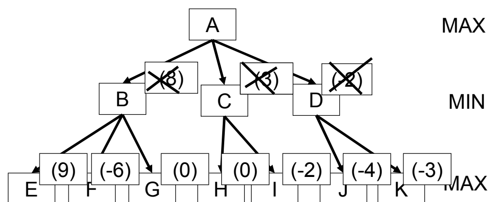
Algorytm mini-max



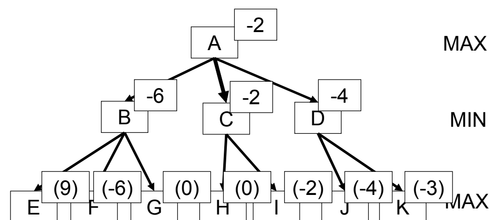
Algorytm mini-max



Algorytm mini-max



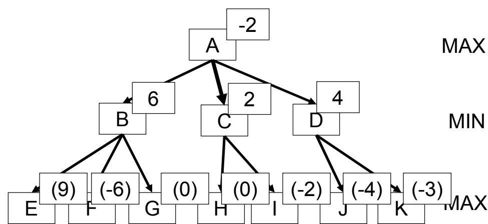
Algorytm mini-max



bestmin(X) = min{Y∈ succ(X): Evaluate(Y)}

bestmax(X) = max{Y∈ succ(X): Evaluate(Y)}

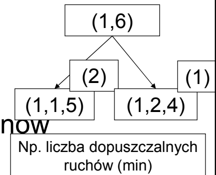
Algorytm nega-max



best(X) = max{Y∈ succ(X): (– Evaluate(Y))}

Implementacja przeszukiwania

- Reprezentacja aktualnego stanu w grze
  - NIM
  - Kółko i krzyżyk
  - Szachy
- Generowanie kolejnych stanów
  - ruchy odwrotne (undo)
- Funkcja oceny heurystycznej
  - szybkość (złożoność obliczeniowa)
  - dokładność



Implementacja przeszukiwania przestrzeni stanów

pos	pozycja w grze
depth	liczba poziomów w grafie do przeszukania
Evaluate	funkcja obliczająca wartość gry z punktu widzenia gracza, który wykonuje ruch
best	najlepsza wartość znaleziona na kolejnym poziomie
Successors	funkcja generująca zbiór pozycji osiągalnych w jednym ruchu z aktualnej pozycji
succ	zbiór pozycji osiągalnych w jednym ruchu z aktualnej pozycji

NegaMax

Jeżeli został osiągnięty ostatni poziom, to oblicz watrość funkcji oceny heurystycznej dla aktualnego stanu

```
int NegaMax (pos, depth)
{
        if (depth == 0) return Evaluate(pos);
        best = -INFINITY;
        succ = Successors(pos);
        while (not Empty(succ))
        {
                pos = RemoveOne(succ);
                value = -NegaMax(pos, depth-1);
                if (value > best) best = value;
        }
        return best;
}
```

NegaMax

Wygeneruj zbiór następników aktualnego stanu

```
int NegaMax (pos, depth)
{
        if (depth == 0) return Evaluate(pos);
        best = -INFINITY;
        succ = Successors(pos);
        while (not Empty(succ))
        {
                pos = RemoveOne(succ);
                value = -NegaMax(pos, depth-1);
                if (value > best) best = value;
        }
        return best;
}
```

NegaMax

wybiera stan ze zbioru succ

```
int NegaMax (pos, depth)
{
        if (depth == 0) return Evaluate(pos);
        best = -INFINITY;
        succ = Successors(pos);
        while (not Empty(succ))
        {
                pos = RemoveOne(succ);
                value = -NegaMax(pos, depth-1);
                if (value > best) best = value;
        }
        return best;
}
```

NegaMax

```
int NegaMax (pos, depth)
{
        if (depth == 0) return Evaluate(pos);
           = -INFINITY;
           Successors(pos);
           (not Empty(succ))

            po  = RemoveOne(succ);
            value = -NegaMax(pos, depth-1);
            if (value > best) best = value;
        }
        return best;
}
```

Rekurencyjne wywołanie procedury NegaMax z odpowiednio zaktualizowaną wartością depth

NegaMax

```
int NegaMax (pos, depth)
{
        if (depth == 0) return Evaluate(pos);
        best = -INFINITY;

        {
            pos = RemoveOne(succ);
            value = -NegaMax(pos, depth-1);
            if (value > best) best = value;
        }
        return best;
}
```

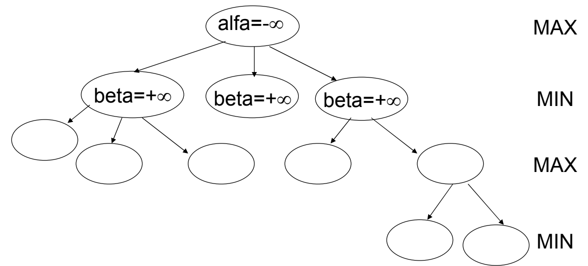
Wykładnicza złożoność obliczeniowa !

Procedura alfa-beta

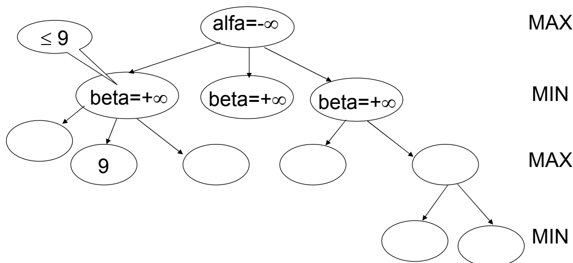
Założenia:

1. Reguły gry nie pozwalają na wygenerowanie nieskończonej ścieżki.
2. Z każdego stanu można osiągnąć tylko skończoną liczbę stanów.
3. (Lemat nieskończoności): dla każdego stanu *p* istnieje liczba N(*p*) taka, że żadna rozgrywka startująca ze stanu *p* nie będzie trwała dłużej niż N(*p*) ruchów.

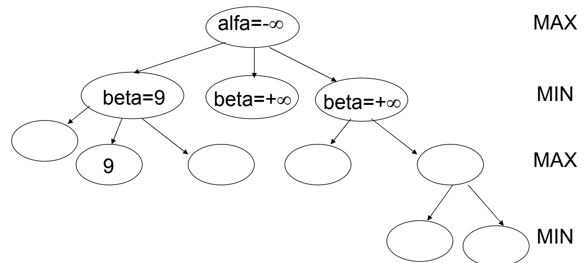
Algorytm alfa beta

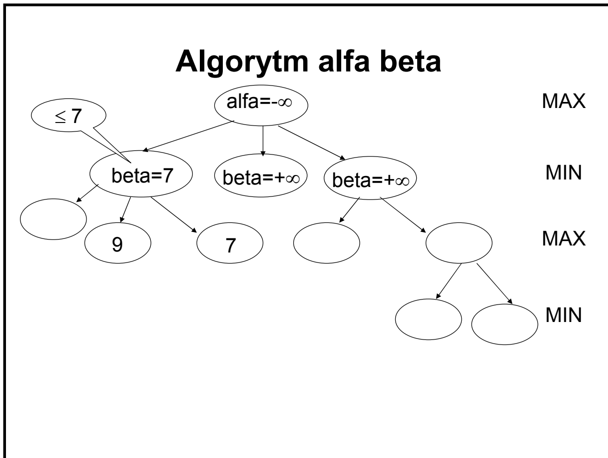


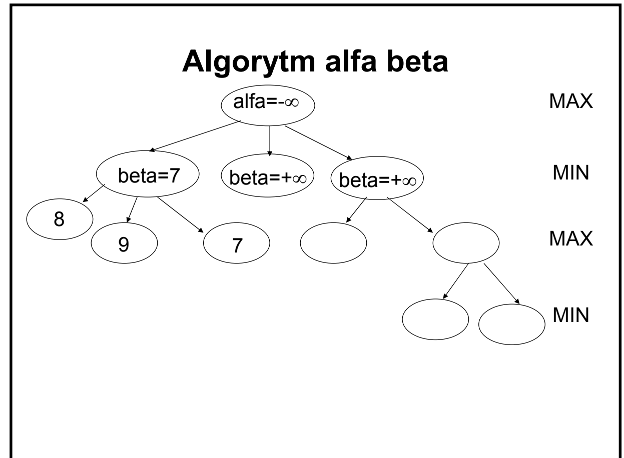
Algorytm alfa beta

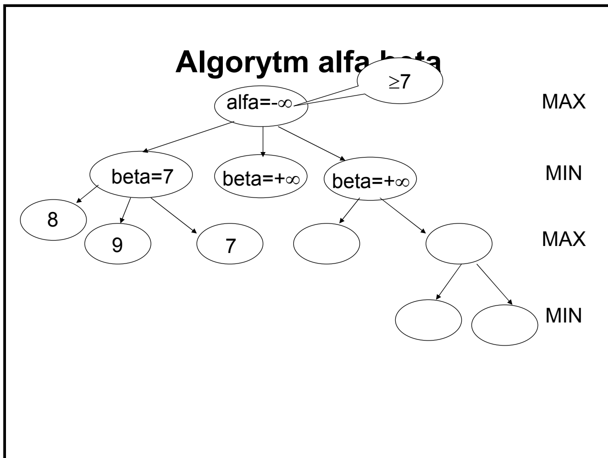


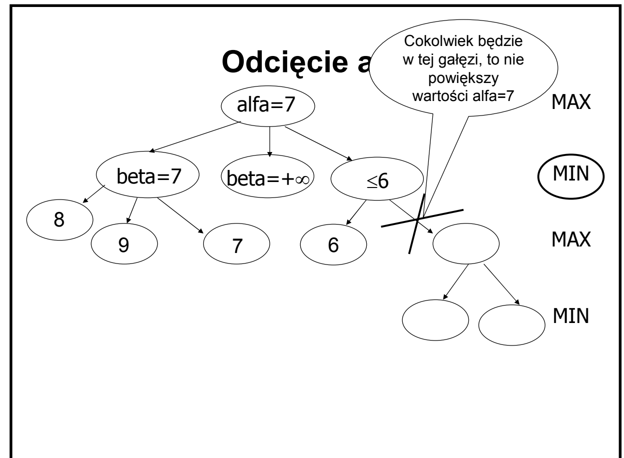
Algorytm alfa beta





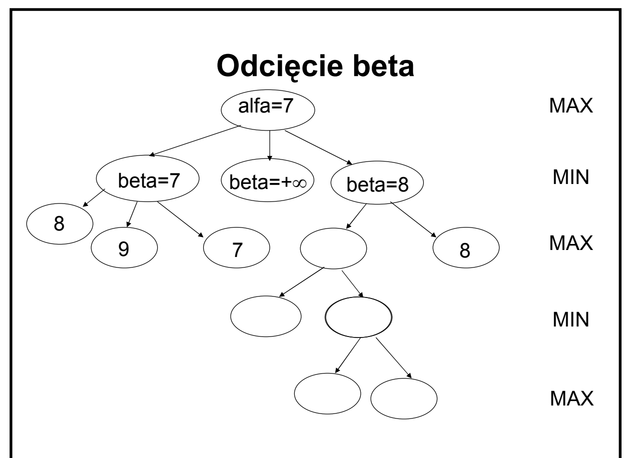




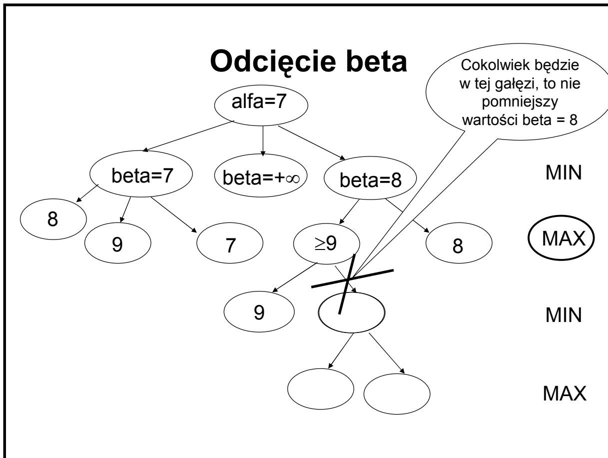


Odcięcie alfa

Przeszukiwanie można zakończyć poniżej dowolnego wierzchołka typu MIN o wartości mniejszej lub równej wartości alfa dowolnego z jego poprzedników (typu MAX).



Odcięcie beta



Odcięcie beta

Przeszukiwanie można zakończyć poniżej dowolnego wierzchołka typu MAX o wartości większej lub równej wartości beta dowolnego z jego poprzedników (typu MIN).

```
int AlphaBeta (pos, depth, alpha, beta)
{
if (depth == 0) return Evaluate(pos);
best = -INFINITY;
succ = Successors(pos);
while (not Empty(succ) && best < beta)
        {
        pos = RemoveOne(succ);
        if (best > alpha) alpha = best;
        value = -AlphaBeta(pos, depth-1, -beta, -alpha);
        if (value > best) best = value;
        }
return best;
}
```

Algorytm alfa-beta

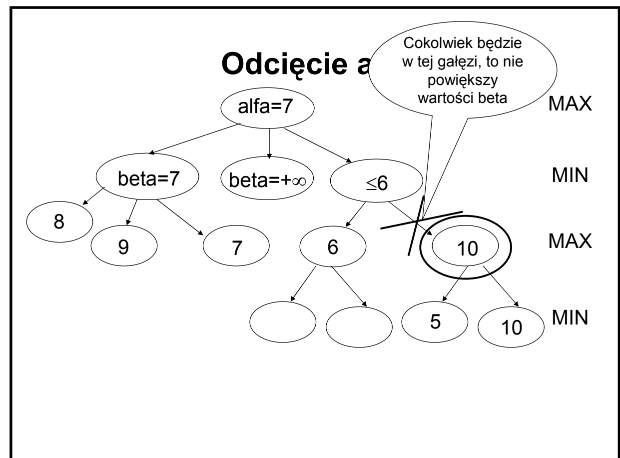
- Jest algorytmem dokładnym
- W najgorszym razie przeszukuje całą przestrzeń (tak jak MINIMAX, $O(N^D)$
- W najlepszym razie przeszukuje $N^{((D+1)/2)}+N^{(D/2)} – 1$ stanów
- W najlepszym razie pozwala dwukrotnie zwiększyć głębokość przeszukiwania

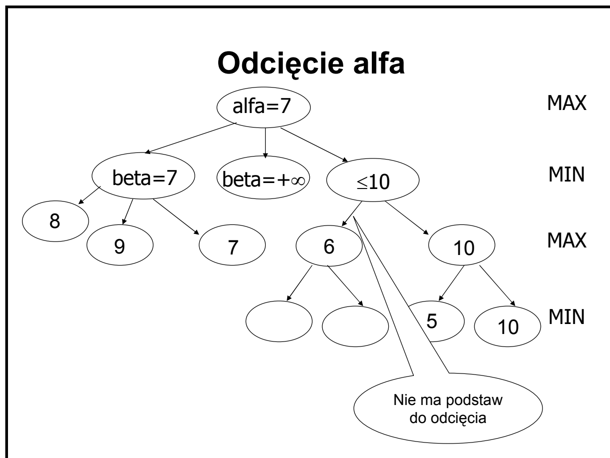
N – braching factor; D - głębokość

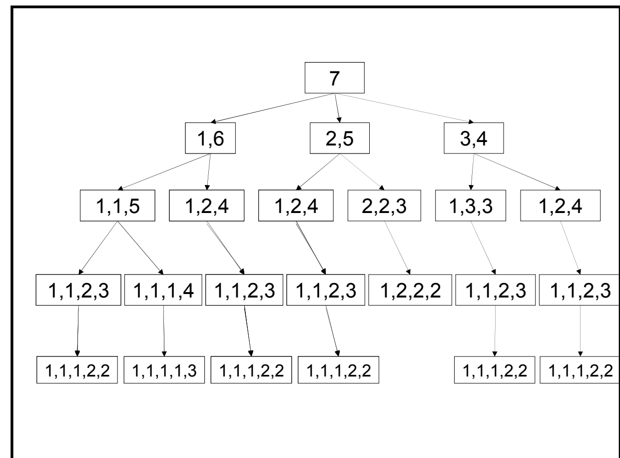
Modyfikacje algorytmu alfa-beta

- Kolejność ruchów (Move ordering)
  - Rezultaty wcześniejszych przeszukiwań
  - Dynamiczne porządkowanie ruchów
  - Statyczne porządkowanie ruchów

Odcięcie a







Move ordering

- Hashtables
- Killer moves
- History heuristics
- Static ordering

Move ordering

• Transposition table (Hashtable)

Polega na haszowaniu identycznych stanów w różnych gałęziach. Może skrócić przeszukiwanie nawet czterokrotnie (co oznacza jeden poziom w grafie więcej w tym samym czasie).

Move ordering

• Killer moves

Polega na zapamiętywaniu ruchów, które we wcześniejszych fazach przeszukiwania doprowadziły do odcięcia. Takie ruchy są wykonywane jako pierwsze podczas eksploracji stanu.

Po znalezieniu innego ruchu odcinającego, poprzedni jest zastępowany nowym.

Move ordering

• History heuristics

Jest rozszerzeniem poprzedniej heurystyki, polegającym na zapamiętywaniu wszystkich „dobrych” ruchów. Każde skuteczne zastosowanie ruchu zwiększa jego priorytet.

W ten sposób tworzy się pewne uporządkowanie ruchów.

Move ordering

• Static ordering

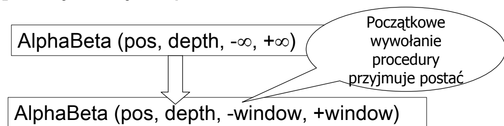
Jest heurystyką zależną od gry. Mimo, że preferowane są strategie niezależne od gry, to jednak w praktyce często korzysta się z informacji charakterystycznej dla rodzaju gry, aby zwiększyć efektywność heurystyki.

Np. w warcabach można preferować ruchy, które „zbijają” więcej pionów przeciwnika itd.

Modyfikacje algorytmu alfa-beta

• Aspiration search (windowing)

W korzeniu przeszukiwanego drzewa przyjmuje się ograniczony przedział [alfa, beta] nazywany aspiration window.



Jeżeli rozwiązanie leży poza oknem, to trzeba powtórzyć przeszukiwanie.

Modyfikacje algorytmu alfa-beta

• Principal variation search (PVS)

Polega na przeszukiwaniu gałęzi poza principal variation z oknem [localalfa, localalfa+1].

Zakłada się, że jest mało prawdopodobne, aby ruchy poza principal variation były dobre (przy założeniu *perfect ordering*)

Modyfikacje algorytmu alfa-beta

• MTD(f) (Memory enhanced test)

Polega na kilkakrotnym wywołaniu procedury alfa-beta ze zmieniającymi się oknami. Za każdym razem uzyskujemy dolne lub górne ograniczenie, które są zbieżne do prawdziwej wartości mini-max.

```
int mtdf(struct position p, int firstguess,int depth)
	{
	int g,lowerbound, upperbo
	g=firstguess;
	upperbound=INFINITY;
	lowerbound=-INFINITY;
	while(lowerbound<upperbound)
		{
		if(g==lowerbound) beta=g+1;
		else beta=g;
		g=alphabeta(p,depth,beta-1,beta);
		if(g<beta) upperbound=g;
		else lowerbound=g;
		}
	return g;
	}
```

wartość początkowa

Aktualizacja okna

Modyfikacje algorytmu alfa-beta

• Enhanced transposition cutoffs (ETC)

W algorytmie alfa-beta z tablicą haszującą przegląda się wszystkie następniki bieżącego stanu przed rekurencyjnym wywołaniem procedury alfa-beta.

W praktyce jest to koncepcja kosztowna i nakład zwraca się tylko w węzłach odległych od liści.

Modyfikacje algorytmu alfa-beta

• Quiescence search

Jest to procedura wywoływana często zamiast funkcji Evaluate na głębokości depth=0. Ma to zapobiec tzw. efektowi horyzontu, który polega na ukryciu rychłej przegranej przez fakt, że nie przeszukujemy dalej przestrzeni. Quiescence search ma zapewnić, że Evaluate będzie wywoływana tylko w stabilnych stanach (w których nie ma bezpośredniego zagrożenia przegraną). Przeszukuje się tylko takie stany, które zwykle wiążą się z zagrożeniem (np. utrata piona w szachach).

Modyfikacje algorytmu alfa-beta

• Iteracyjne pogłębianie

Polega na wielokrotnym wywoływaniu przeszukiwania na określoną głębokość, przy czym w kolejnych powtórzeniach głębokość przeszukiwania rośnie.

W kolejnych powtórzeniach można wykorzystać zdobytą wcześniej informację (zapisaną np. w tablicy transpozycji, czy w postaci okna aspiracji).

Modyfikacje algorytmu alfa-beta

• Null move heuristics

Polega na zaprzestaniu przeszukiwania w tych obszarach, gdzie znaleziono zadowalająco dobrą pozycję w grze. Wykrycie takiej pozycji polega na wykonaniu pustego ruchu i powtórzeniu przeszukiwania poziom niżej. Jeżeli wynik jest większy niż beta, to przeszukiwanie kończy się, w przeciwnym razie jest kontynuowane normalnie.

Algorytm B^*

Strategia **PROVEBEST** polega na tym, że próbuje się zwiększać dolne ograniczenie najbardziej lewego wierzchołka tak, aby była ona nie gorsza od górnego ograniczenia dowolnego wierzchołka spośród rodzeństwa.

Strategia **DISPROVEREST** polega na tym, że próbuje się zmniejszać górne ograniczenie wszystkich wierzchołków na głębokości 1 tak, aby żadna z nich nie była lepsza od dolnego ograniczenia najbardziej lewego wierzchołka.

Przykład algorytmu B*

[-∞,∞]

MAX

Przykład algorytmu B*

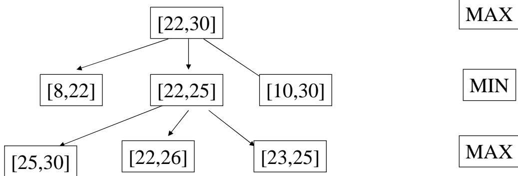
[15,30]

MAX

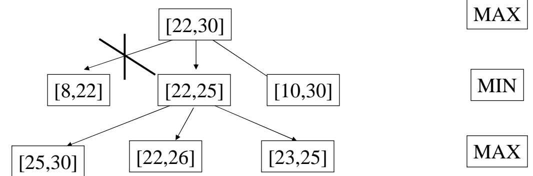
[8,22] [15,25] [10,30]

MIN

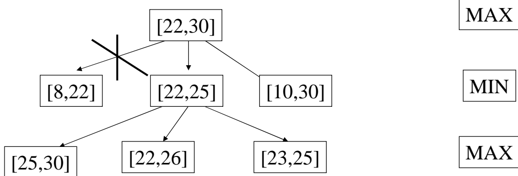
Przykład algorytmu B*



Przykład algorytmu B*



Przykład algorytmu B*



Deep Blue - podejście siłowe

Analizuje $2*10^8$ (200 mln) stanów w ciągu sekundy.

Przestrzeń stanów dla szachów wynosi $10^{120.}$

W czasie dozwolonym analizuje 6 ruchów do przodu.

Kasparow analizuje 3 ruchy do przodu.

Deep Junior

Analizuje mniej stanów, ale ma lepszą heurystykę.

Kasparow również nauczył się specyfiki gry z komputerem.

Deep Blue - podejście siłowe

The latest iteration of the Deep Blue computer is a 32-node IBM RS/6000 SP high-performance computer, which utilizes the new Power Two Super Chip processors (P2SC). Each node of the SP employs a single microchannel card containing 8 dedicated VLSI chess processors, for a total of 256 processors working in tandem.
Deep Blue's programming code is written in C and runs under the AIX operating system. Deep Blue is able to explore 200,000,000 positions per second. Incidentally, Garry Kasparov can examine approximately three positions per second.

Deep Junior vs. Deep Fritz

- Deep Fritz is running on an eight-core machine out of Hamburg Germany. The program is searching 13-14 million nodes per second, reaching a search depth of 20-21 ply.
- Deep Junior, playing out of London England, is employing the latest Intel Server technology with 16 cores. The program is running at 24 million nodes per second and consistently reaching search depths of 24 ply.
- http://www.chessbase.com/newsdetail.asp?newsid=3920