Politechnika Warszawska

OKNO

MSI

7. Poszukiwanie rozwiązania w przestrzeni stanów

Włodzimierz Kasprzak

Układ

1. Poszukiwanie celu
2. Losowe poszukiwanie celu
3. Przeszukiwanie lokalne
4. Symulowane wyżarzanie
5. Obliczenia ewolucyjne
6. Problem wymagający spełnienia ograniczeń (CSP)
7. Przeszukiwanie przyrostowe dla CSP
8. Przeszukiwanie ze stanem zupełnym dla CSP

Układ (c.d.)

9. Gra z przeciwstawnymi celami
10. Strategia „Mini-max”
11. „Cięcia alfa-beta”
12. Heurystyczna ocena – „obcięty „Mini-max”

1. Poszukiwanie celu

Rozpatrywane dotąd strategie przeszukiwania przestrzeni stanów poszukiwały **ścieżki** wiodącej do celu (mającego charakter globalnego optimum). Teraz zajmiemy się ogólnymi strategiami, które **poszukują** jedynie samego **celu**, gdy nieistotna jest przy tym ścieżka wiodąca do celu. Uwzględnimy strategie, które

1. korzystają z elementów losowości lub
2. mają charakter lokalnego przeszukiwania.

W przeciwieństwie do optymalnych strategii przeszukiwania (poszukiwania ścieżki), strategie losowe mogą dawać jedynie aproksymację optymalnego rozwiązania (celu), a strategie lokalnego przeszukiwania nie dawać gwarancji uzyskania optymalnego rozwiązania w ogóle.

Jednak zaletą strategii obu tych typów jest znacznie mniejsza złożoność obliczeń w porównaniu do strategii optymalnego przeszukiwania (poszukiwania optymalnej ścieżki).

Poszukiwanie celu

Dotychczas rozpatrywane optymalne strategie przeszukiwania (poszukiwania ścieżki) również znajdują cel. Jeśli proces przeszukiwania przestrzeni stanów jest „w pełni poinformowany” to z góry wiemy, jakie decyzje podejmować i nigdy nie zboczymy ze ścieżki optymalnego rozwiązania a liczba rozwijanych węzłów będzie liniową funkcją długości ścieżki – złożoność czasowa i pamięciowa takiej strategii wyniesie $O(m)$, gdzie m jest długością ścieżki rozwiązania.

Niejako „na drugim biegunie” są strategie przeszukiwania, w których decyzje podejmowane są w losowy sposób lub w oparciu jedynie o lokalną obserwację. Siłą rzeczy te decyzje jedynie z pewnym prawdopodobieństwem (a nie pewnością) będą globalnie optymalne.

2. Losowe poszukiwanie celu

Najprostsza losowa strategia poszukiwania celu to **losowe próbkowanie globalne :**

function LosowePróbkowanie(problem, k)

returns stan końcowy

{ $V = \varnothing$;

for i = 1 **to** k

{ generuj losowo $s_i \in$ Stany(problem);

$V = V \cup \{s_i\}$;

if (WarunekStopu(s_i)) return s_i;

}

return $\varnothing$;

}

Losowe generowanie następnika

function LosoweGenerowanieNastepnika(*problem*)
return stan końcowy
{ i = 0; $V_0 = s_0$ = StanPoczatkowy(problem);
 A_0 = Nastepniki(s_0, *problem*) ;
  repeat
  { wybierz losowo s_{k+1} z A_k;
   $V_{k+1} = V_k \cup \{s_{k+1}\}$;
   $A_{k+1} = A_k \cup$ Nastepniki(s_{k+1}, problem) – V_k;
   k = k+1;
  } **until** (s_k nie jest stanem końcowym, tzn.
   if (WarunekStopu(problem, s_k))
 return s_k;
}

Błądzenie przypadkowe – losowe, lokalne

function BladzeniePrzypadowe(*problem*, k)
returns stan końcowy
{ s_0 = StanPoczatkowy(*problem*);
V = { s_0};
for i = 1 **to** k
{ generuj losowo $s_i \in$ Nastepniki(s_{i-1});
V = V $\cup$ {s_i};
if (WarunekStopu(s_i)) **return** s_i;
}
return $\varnothing$;
}

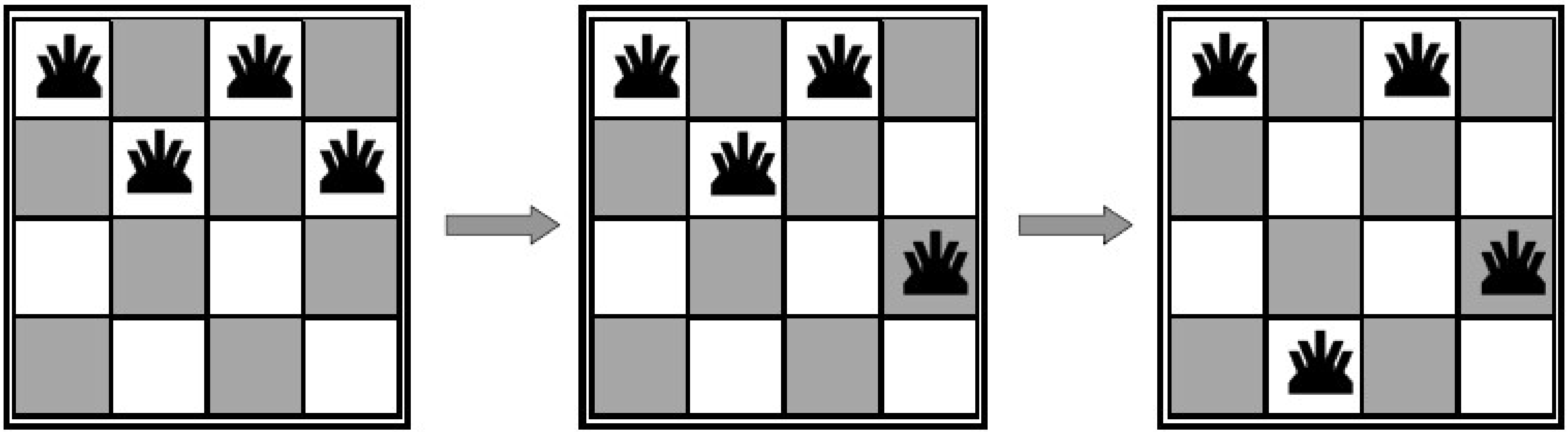
3. Przeszukiwanie lokalne

- W wielu problemach ścieżka prowadząca do celu nie jest taka ważna, jak sam fakt osiągnięcia celu.
- Np. w grach przestrzeń stanów definiowana jest jako zbiór „pełnych" konfiguracji pionków. Celem (szukanym rozwiązaniem) jest znalezienie konfiguracji spełniającej pewne warunki, np. w problemie *n królowych*.
- W takich sytuacjach możemy zastosować strategie przeszukiwania lokalnego, które zapamiętują jedynie pojedynczy stan „aktualny" i starają się go w sposób iteracyjny poprawiać.
- Problem: znalezione rozwiązanie lokalnie optymalne nie musi być globalnie optymalne.

Przykład: *n*-hetmanów

Problem: należy ustawić *n* hetmanów na szachownicy o rozmiarze *n* × *n* tak, aby żadni dwaj hetmani nie znaleźli się w tym samym wierszu, kolumnie i przekątnej (w zasięgu bicia).

Stan początkowy → Stan następny → Stan końcowy (nieoptymalny)



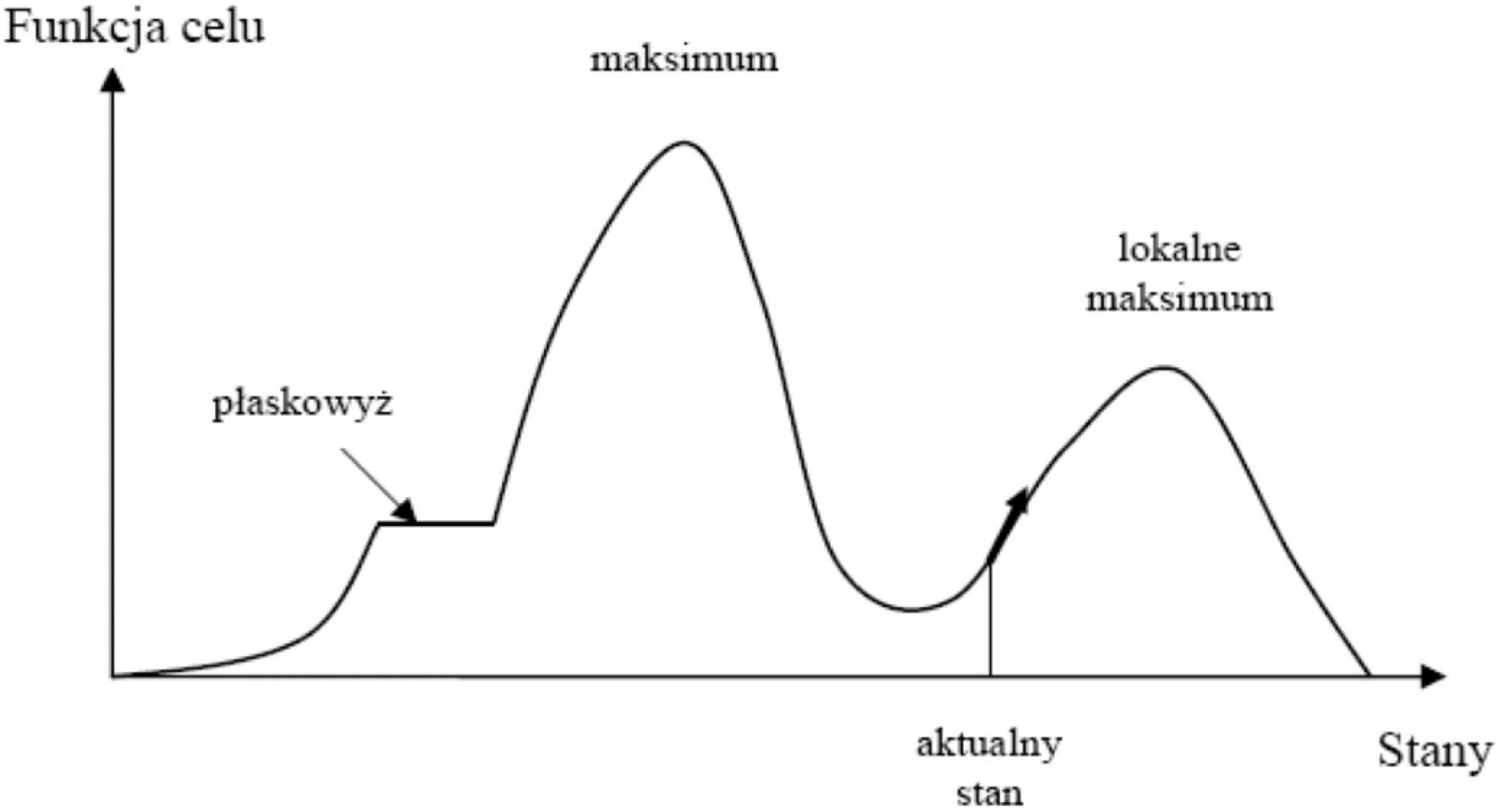
„Przeszukiwanie przez wspinanie”

„Wspinanie się na Mt. Everest w gęstej mgle będąc dotkniętym amnezją”.

function Hill-Climbing (*problem*) **returns** *stan*;
{
węzeł-aktualny ←Węzeł(Stan-Początkowy([*problem*]));
while (true)
{
sąsiad ← Najlepszy-następca(*węzeł-aktualny*);
if (Ocena(*sąsiad*) ≤ Ocena(*węzeł-aktualny*)
then **return** Stan(*węzeł-aktualny*);
węzeł-aktualny ←*sąsiad*;
}
}

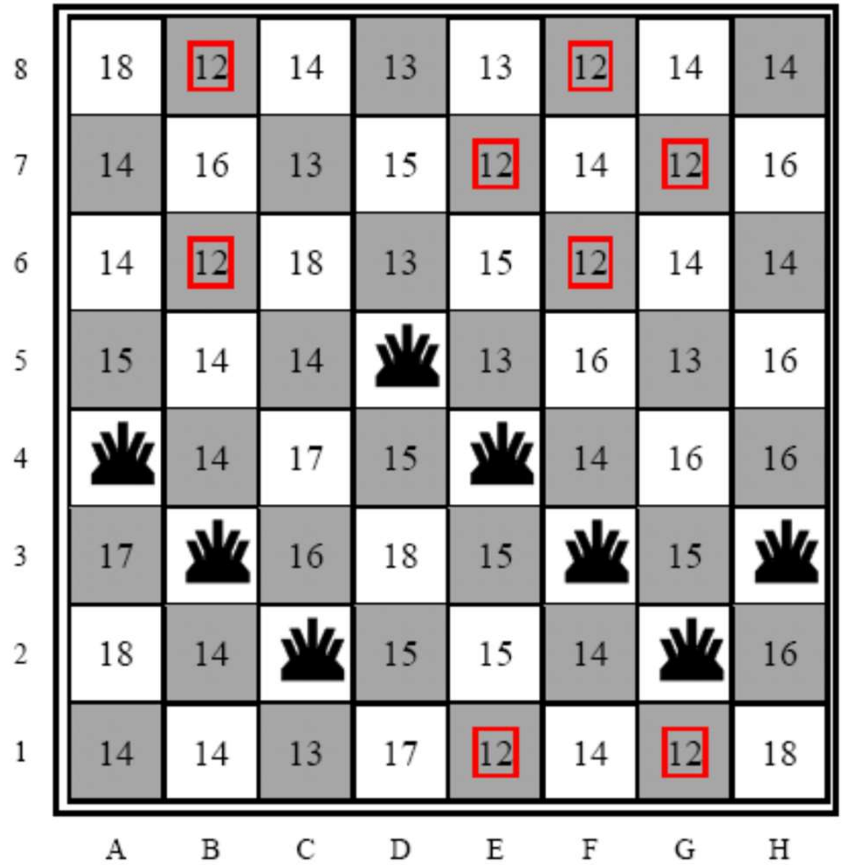
Przeszukiwanie przez „wspinanie”

- Problem: można utknąć w lokalnym maksimum.
- Problem: można utknąć na „płaskowyżu”.



Przeszukiwanie przez „wspinanie” – problem 8-hetmanów

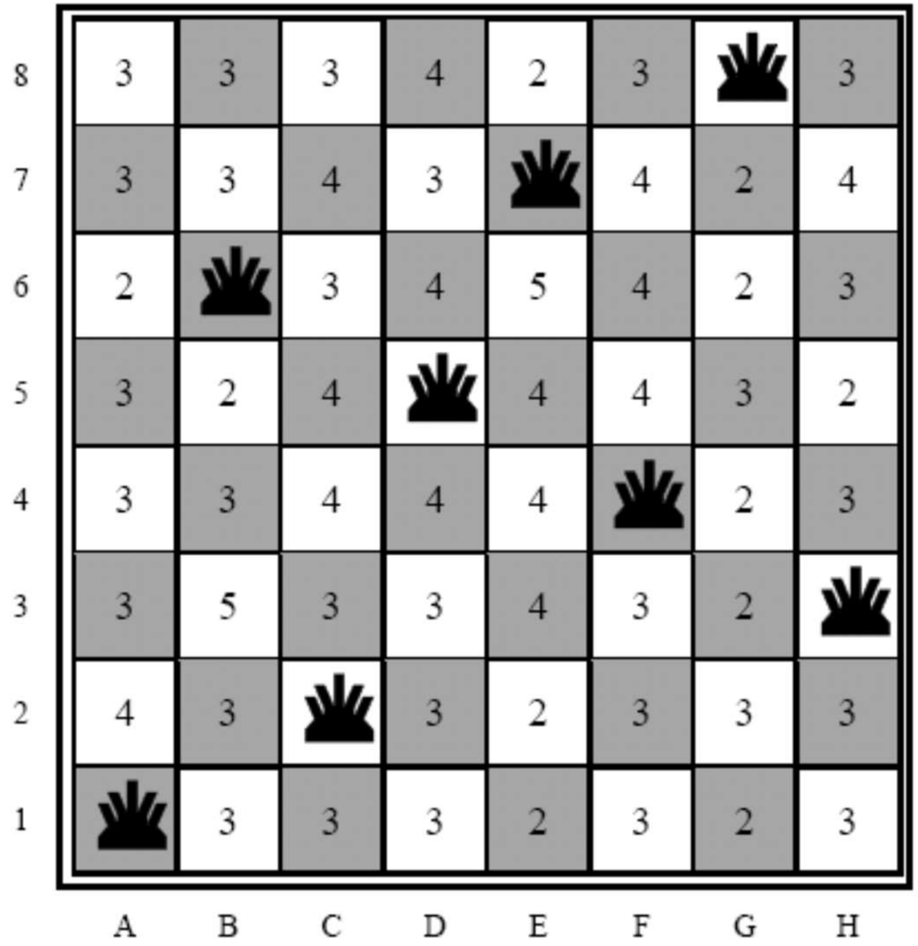
- Niech *h* oznacza liczbę atakujących się wzajemnie par hetmanów (bezpośrednio lub pośrednio).
- Dla podanego obok stanu początkowego: *h* = 17 .
- Liczba w każdym wolnym polu podaje ocenę *h* dla stanu powstałego po przesunięciu do niego hetmana znajdującego się w danej kolumnie.
- Wybierana jest akcja przesunięcia hetmana na najlepszą pozycję, czyli jedną z tych o ocenie *h* = 12.
- Itd. powtarzamy (generowanie następników, ich ocen, wybór) dopóki możliwa jest poprawa funkcji oceny.



Wynik „wspinania” dla problemu 8-hetmanów

- Jak wiemy, strategia lokalna nie gwarantuje osiągnięcia globalnego optimum.
- W naszym przykładzie dla 8 hetmanów końcowy wynik, odpowiadający lokalnemu minimum funkcji oceny, jakie można osiągnąć z zadanego stanu początkowego to:

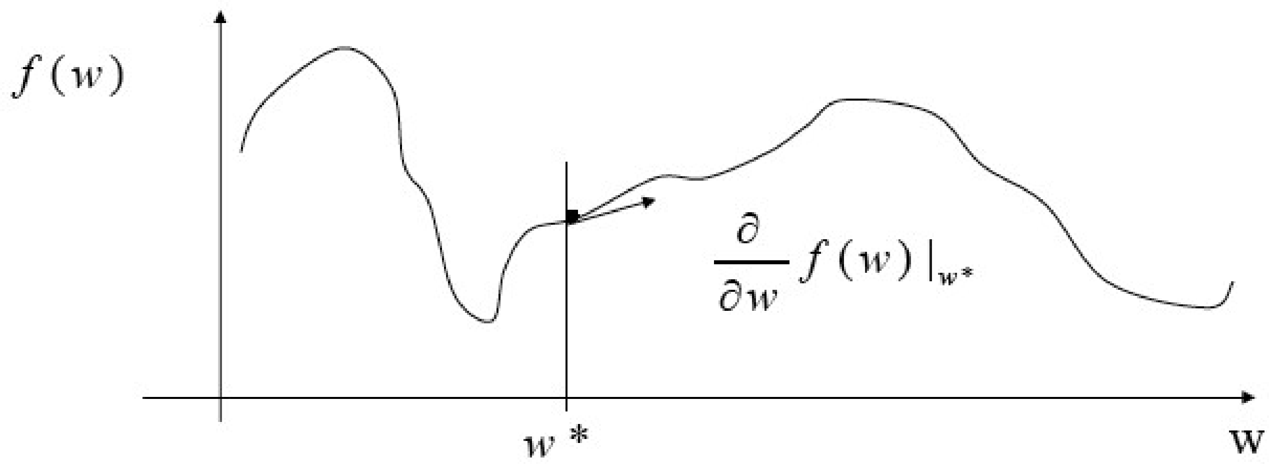
  $h = 1$.



Przeszukiwanie lokalne w dziedzinie ciągłej

Przeszukiwanie z dodatnim gradientem (ang. „gradient ascent search”) jest odpowiednikiem „przeszukiwania przez wspinanie” w dziedzinie ciągłych wartości (w ogólności: wektora) parametrów $\boldsymbol{w}$ funkcji celu $f(\boldsymbol{w})$:

- iteracyjnie modyfikuj wektor $\boldsymbol{w}$: $w \leftarrow w^* + \alpha \frac{\partial}{\partial w} f(w)|_{w^*}$



Przeszukiwanie z ujemnym gradientem (ang. gradient descent search) – dualny problem poszukiwania lokalnego minimum funkcji celu. $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha \frac{\partial}{\partial \boldsymbol{w}} f(\boldsymbol{w})|_{w_i}$

4. Symulowane wyżarzanie

- Pomysł: wydostać się z lokalnego maksimum wykonując „złe” ruchy, ze stopniowym zmniejszaniem ich „amplitudy”.
- Algorytm „symulowanego wyżarzania” realizuje niedeterministyczne przejścia pomiędzy stanami. Jeśli losowe przejście poprawia sytuację to jest na pewno wykonywane, w przeciwnym razie wykonywane jest z pewnym prawdopodobieństwem mniejszym niż 1, zależnym od aktualnej wartości parametru T(>0) (globalnej „temperatury”). Wartość T stale maleje, co czyni takie przejścia coraz mniej prawdopodobnymi.
- Można pokazać, że jeśli T zmniejsza się wystarczająco wolno to symulowane wyżarzanie znajduje globalne optimum z prawdopodobieństwem bliskim 1.

Funkcja symulowanego wyżarzania

```
function SYMULOWANE-WYŻARZANIE(problem, T0, dT, eps)
   return stan_wynikowy;
{ T ← T0;
  aktualny ←Węzeł(PoczątkowyStan[problem]);
  for t←1 to ∞ do
  { if (T < eps) then return aktualny; // Koniec przeszukiwania
    nastepny ←losowo wybrany następca dla aktualny;
    dE = Energia[nastepny] – Energia[aktualny];
    if (dE >0 ) then aktualny ← nastepny;
    else { v ← losowa wartość z zakresu [0, 1];
           if  exp(dE/T) > v then aktualny ← nastepny;
         }    // Gdy dE<0 i T→ 0+ to exp(dE/T) → 0
    T ← T - dT; // T jest coraz mniejsze
  }
}
```

5. Obliczenia ewolucyjne

Przeszukiwanie wiązki („beam search”):

- Jednoczesny wybór k stanów zamiast jednego; wybór k najlepszych następników.
- Nie jest to tożsame z równoległymi k pojedynczymi przeszukiwaniami, gdyż rozwijanych jest tylko k stanów w skraju.
- Problem: częsty przypadek, że wszystkie k stanów prowadzi do tego samego lokalnego optimum.
- Pomysł: wybierać losowo k następników, z tendencją do wyboru „dobrych” następników.
- Zauważmy pewną analogię z naturalną selekcją:
  - Następcy są podobni do swoich rodziców, zdrowsi osobnicy z większą pewnością mają dzieci, czasami zdarzają się przypadkowe mutacje.

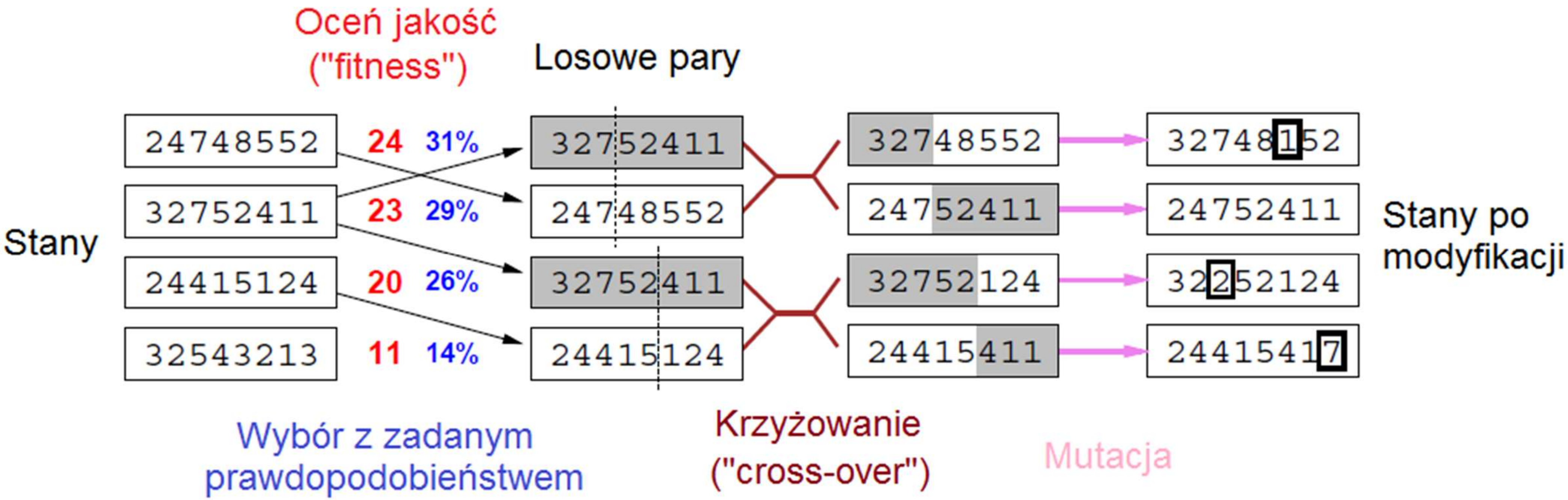
Algorytmy genetyczne

Algorytm genetyczny stanowi połączenie stochastycznego lokalnego przeszukiwania wiązki i metody generowania następników z połączenia par stanów:

- Pojedyncze rozwiązanie (stan) jest postaci sekwencji „genów”.
- Wybór następników ma charakter losowy, z prawdopodobieństwem proporcjonalnym do ich jakości.
- Stany wybrane do reprodukcji są parowane w sposób losowy, niektóre geny są mieszane a niektóre mutują.

Algorytmy genetyczne

- Przykład



Algorytm GA

function GA (*próg-oceny, p, r, m*) **return** stan
{ $\boldsymbol{P} \leftarrow$ { *p* losowo wybranych stanów }
for (każdy stan *h* w ***P***) **do** *OBLICZ-OCENĘ* (*h*)
while [$\max_h$ *ocena*(*h*)] < *próg-oceny* **do**
{ 1. Losowy wybór : *dodaj* (1-*r*)$\cdot$ *p stanów do* $\boldsymbol{P}_s$
$Pr(h_i)$= OCENA(h_i)/ *Suma ocen wszystkich stanów z* $\boldsymbol{P}_s$
2. Krzyżowanie: *losowo wybierz r·p/2 par stanów z* $\boldsymbol{P}_s$.
Dla każdej pary (h_j, h_k), zastosuj operator krzyżowania.
Dodaj wyniki do $\boldsymbol{P}_s$
3. Mutacja: inwersja losowo wybranego bitu w (*m · p)* losowo wybranych stanach *z* $\boldsymbol{P}_s$
4. Podstaw*:* $\boldsymbol{P} \leftarrow \boldsymbol{P}_s$
5. **for** każdy stan *h* w ***P*** **do**: OBLICZ-OCENĘ (*h*)
} **return** stan z ***P*** *o najwyższej ocenie*
}

Strategie ewolucyjne

Osobnik jest reprezentowany przez parę wektorów (X, σ), gdzie X odpowiada umiejscowieniu osobnika w n-wymiarowej przestrzeni ciągłej rozwiązań a jest ciągiem parametrów metody.

6. Problem spełniania ograniczeń (CSP)

Dotychczas rozpatrywaliśmy standardowy problem przeszukiwania - stan był „czarną skrzynką" – czyli strukturą danych zawierającą odniesienie do następnika, wartość heurystyki i własności dla warunku stopu.

Problem wymagający spełnienia ograniczeń:

- stan jest definiowany przez zmienne $\{X_i, | i=1,2,..,n\}$, których wartości należą do dziedzin $\{D_i\}$
- warunek stopu to zbiór ograniczeń określający dopuszczalne kombinacje wartości dla zmiennych
- rozwiązaniem jest dowolny stan spełniający te ograniczenia
- rozwiązanie generowane jest przyrostowo.

Można tu stosować strategie przeszukiwania z ograniczeniami.

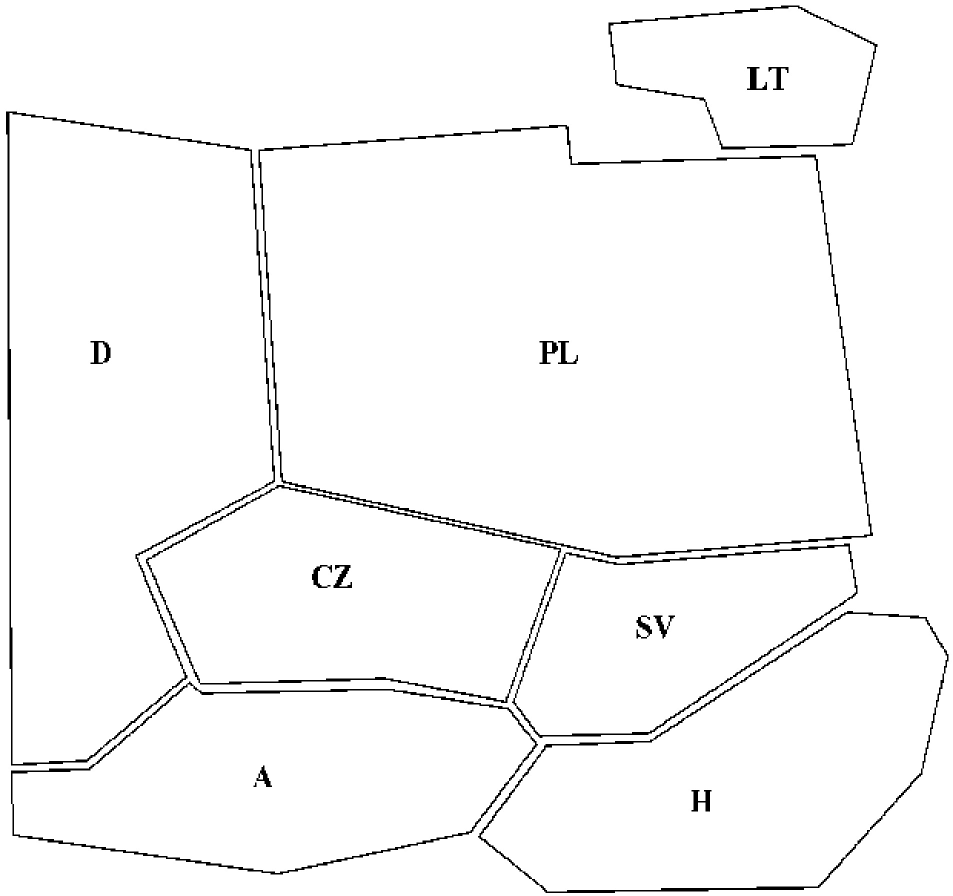
Przykład: kolorowanie mapy

- Zmienne: *PL, D, LT, CZ, SV, A, H*
- Dziedziny: $D_i = \{r, g, b\}$
- Ograniczenia: przylegające regiony muszą być pomalowane różnymi kolorami.

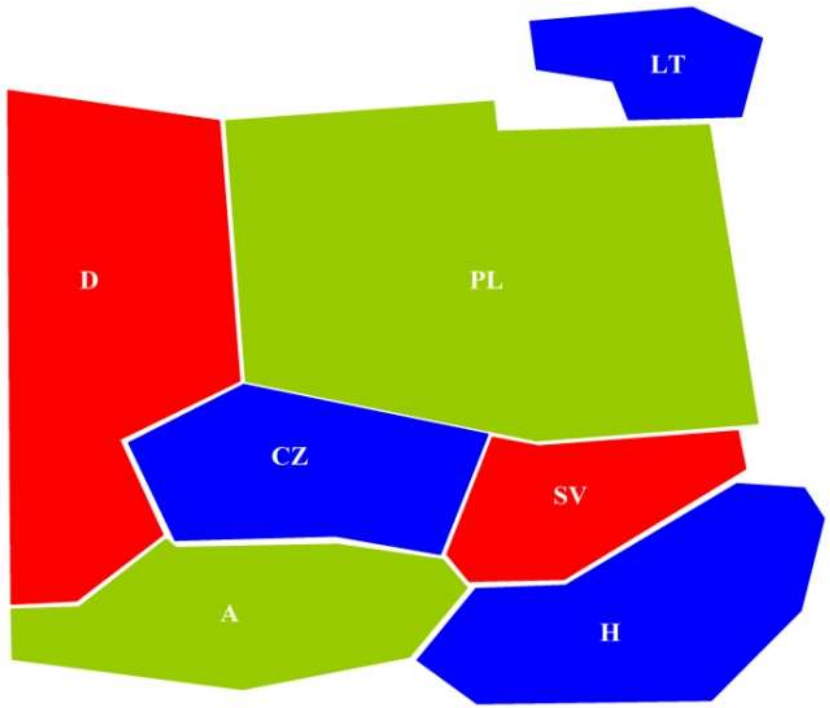
  Np.:

  $(D \neq PL)$ lub

  $(D, PL) \in \{(r, g), (r, b), (g, r), (g, b), (b, r), (b, g)\}$



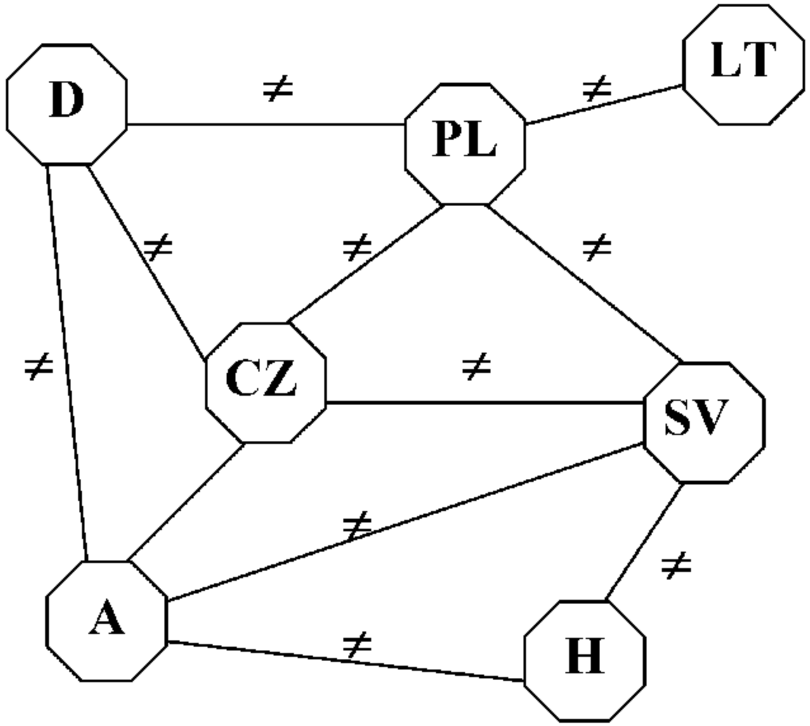
Przykład: kolorowanie mapy (2)



- Rozwiązania są zupełnymi (wszystkim zmiennym nadano wartości) i spójnymi (niesprzecznymi z ograniczeniami) podstawieniami wartości pod zmienne.
- Np.: $\{D=r, PL=g, LT=b, A=g, CZ=b, SV=r, H=b\}$

Graf ograniczeń

- Binarny problem CSP: każde ograniczenie stanowi związek pomiędzy dwiema zmiennymi.
- Graf ograniczeń: zmienne są węzłami, łuki są ograniczeniami (etykieta łuku wyznacza rodzaj ograniczenia).



Rodzaje problemów CSP

- Zmienne **dyskretne**
  - Dziedziny skończone:
    - *dla* n zmiennych i mocy dziedziny d → $O(d^n)$ *podstawień*
  - Dziedziny nieskończone
    - Liczby całkowite, napisy, etc.
    - Np.: harmonogramowanie - zmienne są datami początku/końca każdego zadania
    - Potrzebny jest język zapisu ograniczeń.

    Np.: $StartJob_1 + 5 \leq StartJob_3$
- Zmienne **ciągłe**
  - Np. czas rozpoczęcia / zakończenia obserwacji przeprowadzanych przez Teleskop Hubble'a
  - CSP z ograniczeniami liniowymi rozwiązywane są przez algorytmy *programowania liniowego*.

Rodzaje ograniczeń

- Unarne ograniczenia dotyczą pojedynczej zmiennej,
  – Np. $PL \neq g$
- Binarne ograniczenia dotyczą par zmiennych,
  – Np. $D \neq PL$
- Wyższego rzędu ograniczenia dotyczą 3 lub większej liczby zmiennych.
  – Np. ograniczenia kolumnowe w krypto-arytmetyce

Przykład CSP: kryptoarytmetyka

- Zmienne: $F, T, U, W, R, O.$

  Dziedziny :

  D = $\{0,1,2,3,4,5,6,7,8,9\}$

$$
\begin{array}{r}
T\ W\ O \\
+\,T\ W\ O \\
\hline
F\ O\ U\ R
\end{array}
$$

- Ograniczenia dla zmiennych i wartości przeniesień X_1, X_2 :
- $Wszystkie\_różne(F,T,U,W,R,O)$,

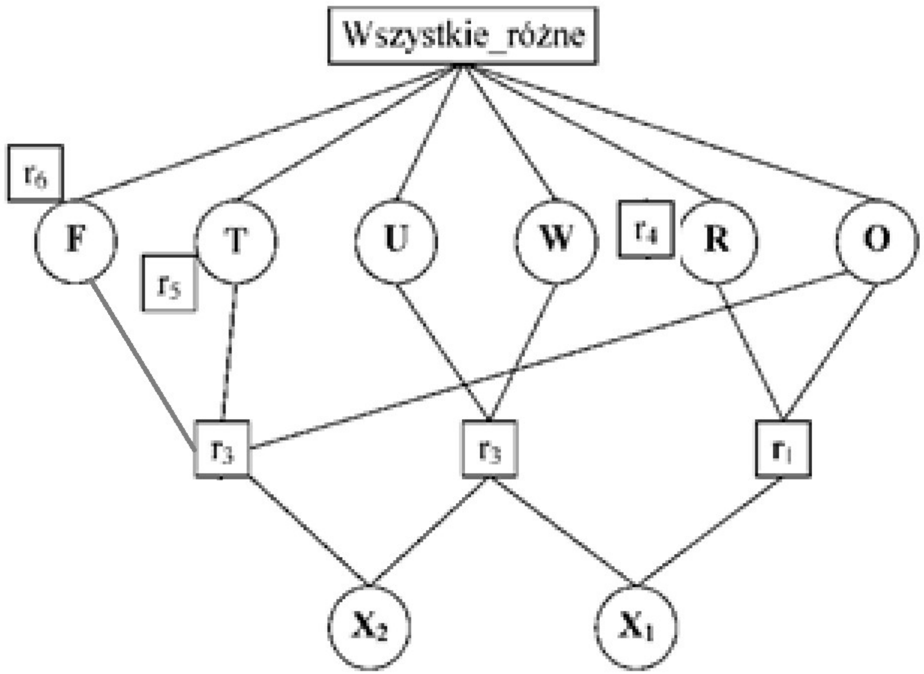
  $r1: O + O = R + 10 \cdot X_1$

  $r2: X_1 + W + W = U + 10 \cdot X_2$

  $r3: X_2 + T + T = O + 10 \cdot F$

  $r4: Parzyste\_niezerowe(R)$,

  $r5: T \neq 0$, $r6$: Dziedzina$(F) = 1$



Przykłady realnych CSP

- Problemy przyporządkowania
  - Np. nauczycieli do lekcji
- Rozkłady zajęć
  - Np. wykład, w której sali i o jakiej godzinie?
- Harmonogramowanie transportowe
- Harmonogramowanie linii produkcyjnej

Przeszukiwanie **przyrostowe** lub ze stanem **kompletnym** dla CSP

Uniwersalne własności problemów CSP

- **Stan niekompletny** - przyrostowe definiowanie przyporządkowania wartości do zmiennych:

  → rozwiązanie pojawia się na głębokości n, gdzie n jest liczbą wszystkich zmiennych,

  → można użyć przeszukiwania w głąb (z nawrotami).

  → liczba następników węzła w drzewie decyzyjnym zależy od głębi położenia l i liczby możliwych wartości zmiennych d: $b_l = (n - l)\, d$ na głębokości l, a więc będzie $n! \cdot d^n$ liści.

- Ponieważ sekwencja akcji prowadząca do rozwiązania jest nieistotna, więc alternatywnie można stosować przeszukiwanie ze **stanem kompletnym**:

  → zastosować przeszukiwanie z heurystyką.

7. Przeszukiwanie przyrostowe dla CSP

Przeszukiwanie przyrostowe to standardowy sposób ślepego (niepoinformowanego) przeszukiwania przestrzeni problemu (stanów) dla problemów z ograniczeniami:

- Węzły (to zwykle niekompletne stany) są zdefiniowane przez wartości dotychczas przypisane zmiennym.
- Węzeł początkowy: przyporządkowanie puste { }.
- Funkcja następnika: nadaj wartość zmiennej, która jeszcze nie ma wartości, w taki sposób, aby nie wywołać konfliktu (nie naruszyć ograniczeń) z dotychczasowym przyporządkowaniem. → Zrezygnuj ze ścieżki (wykonaj „nawrót”), jeżeli nie ma takiego przyporządkowania
- Warunek stopu: bieżące przyporządkowanie jest kompletne (stan zupełny).

Przeszukiwanie „z nawrotami”

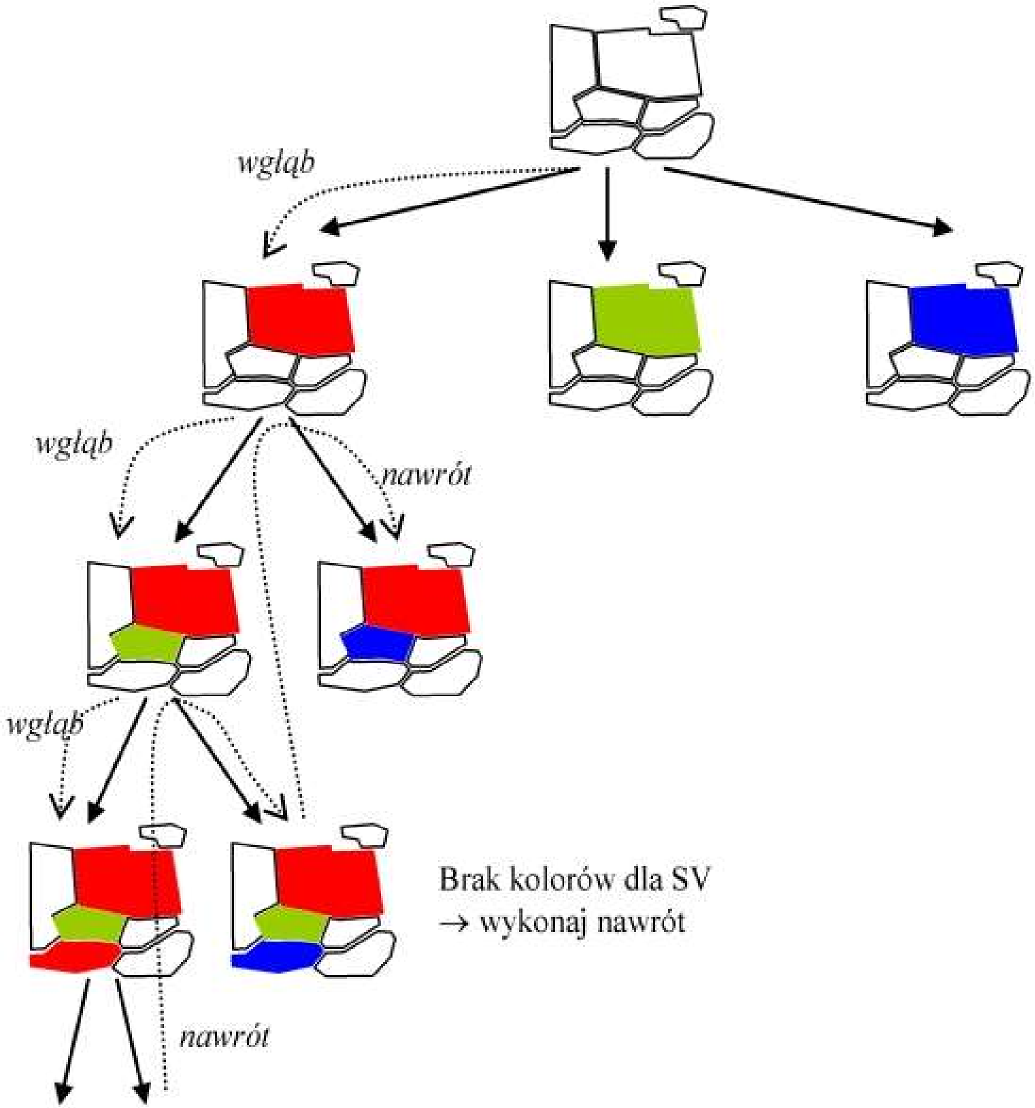
- Przyporządkowania zmiennych są przemienne. Np.: wpierw [$WA = red$] a potem [$NT = green$] jest tym samym co wpierw [$NT = green$] a potem [$WA = red$].
- Dlatego wystarczy rozpatrywać przyporządkowania tylko jednej zmiennej na każdym poziomie drzewa decyzji.

  → wtedy $b = d$ (niezależnie od l) i mamy d^n liści
- Przeszukiwanie w głąb dla CSP, w którym pojedyncza akcja odpowiada przyporządkowaniu wartości pojedynczej zmiennej, nazywane jest **przeszukiwaniem z nawrotami**.
- Przeszukiwanie z nawrotami jest podstawową formą ślepego poszukiwania dla CSP.

  Np. dzięki niemu można realnie rozwiązać problem *n*-hetmanów dla $n \approx 25$.

Przykład przeszukiwania z nawrotami

Przykładowe drzewo decyzyjne dla problemu kolorowania mapy:



Usprawnienia przeszukiwania „z nawrotami

- Możemy wykorzystać **ograniczenia** charakteryzujące dany problem CSP aby **usprawnić** podstawowy algorytm przeszukiwania z nawrotami. Usprawnienia te mają ogólny charakter dla **dyskretnych CSP**.
- Podczas przeszukiwania możemy kierować się informacją:

(A) Której zmiennej jako następnej powinno się przyporządkować wartość?

→ (A1) najbardziej ograniczona zmienna, lub

(A2) najbardziej ograniczająca zmienna;

(B) W jakiej kolejności powinno się nadawać wartości?

→ (B1) najmniej ograniczająca wartość

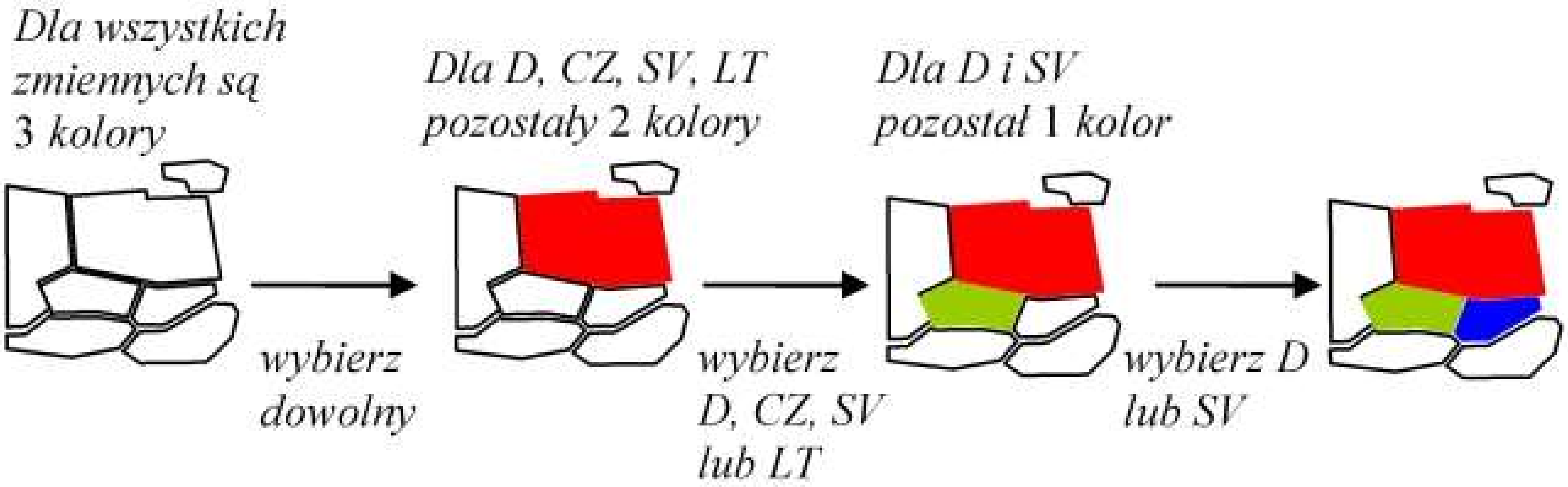
(C) Czy już możemy zdecydować o porażce aktualnej ścieżki?

→ (C1) sprawdzanie wprzód, (C2) spójność łuków.

A1) Najbardziej ograniczona zmienna

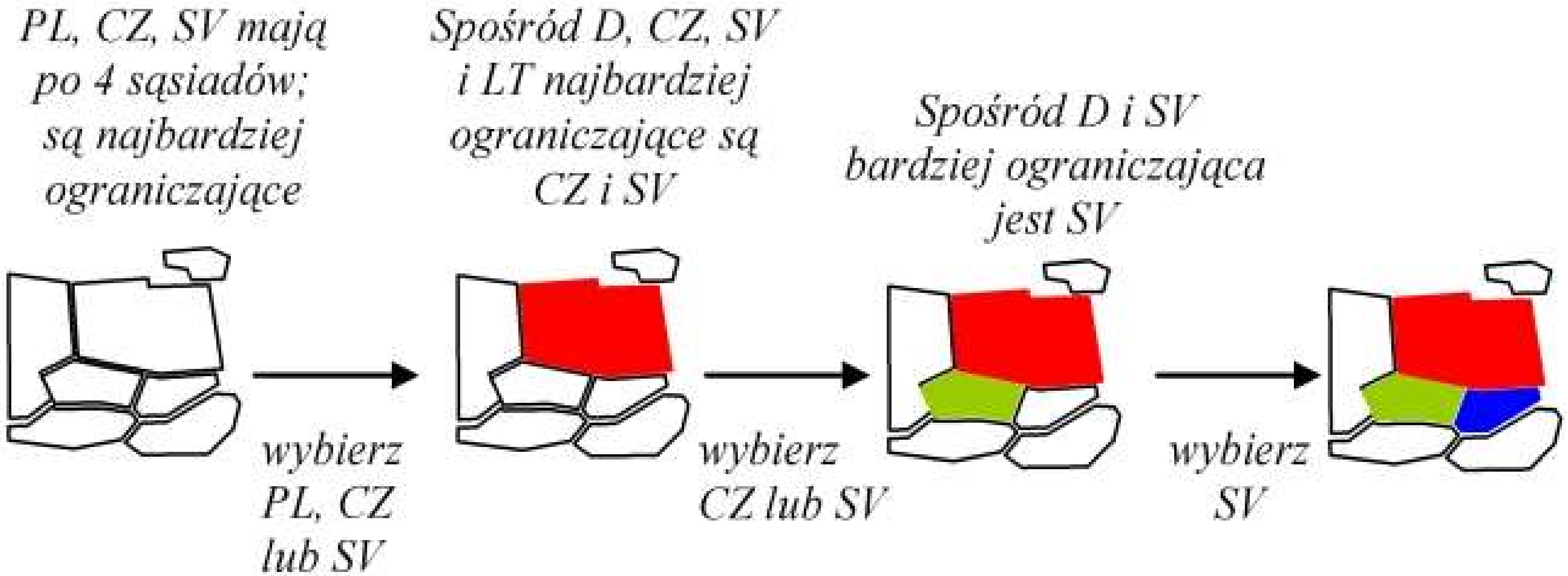
- Wybrać zmienną o najmniejszej liczbie możliwych pozostałych jeszcze wartości.

  Np.:



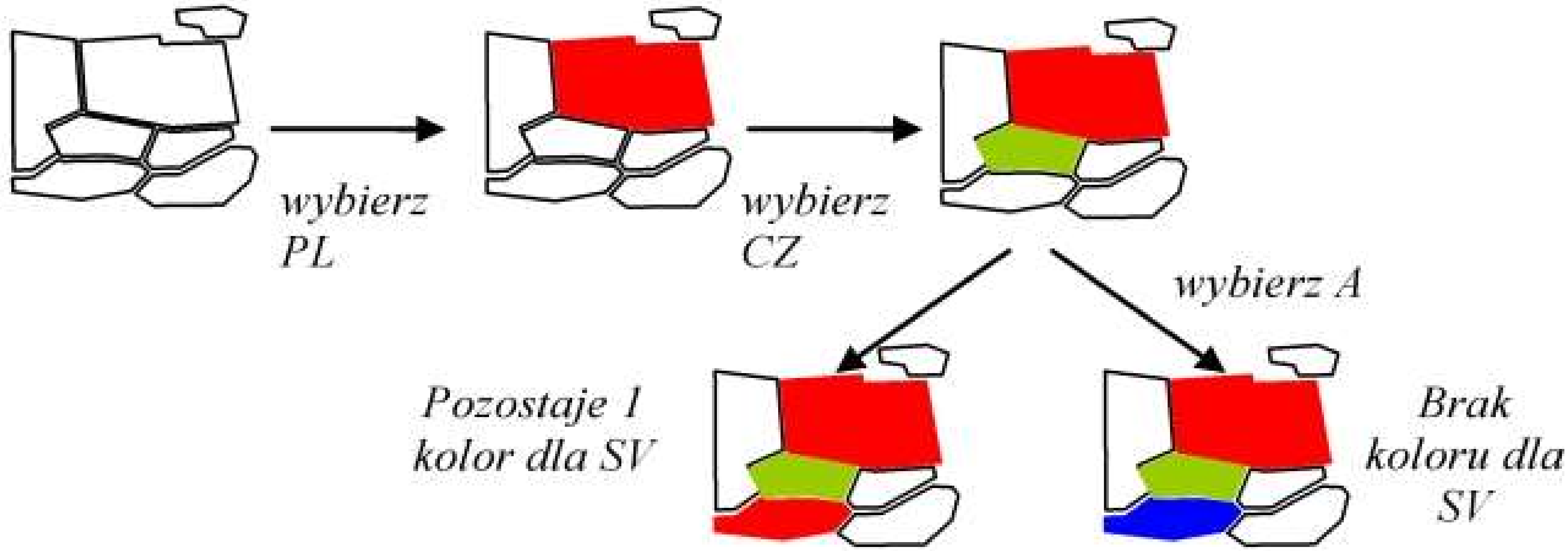
A2) Najbardziej ograniczająca zmienna

- Dodatkowe kryterium do A1) rozstrzygające pomiędzy równymi sobie najbardziej ograniczonymi zmiennymi
- Najbardziej ograniczająca zmienna:
  - Wybrać zmienną, która narzuca najwięcej ograniczeń na pozostałe zmienne.



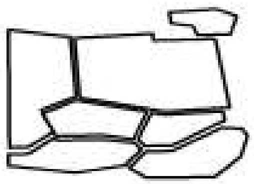
B1) Najmniej ograniczająca wartość

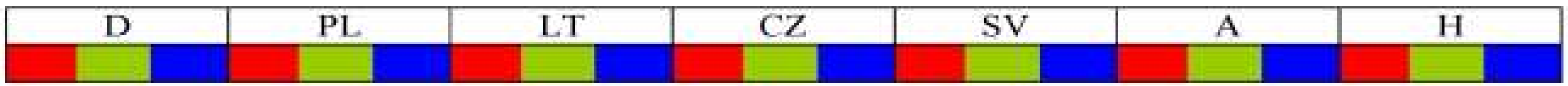
- Po wybraniu zmiennej wybierz dla niej wartość najmniej ograniczającą – taką, która wyklucza najmniej wartości pozostałych zmiennych.



C1) Sprawdzanie w przód

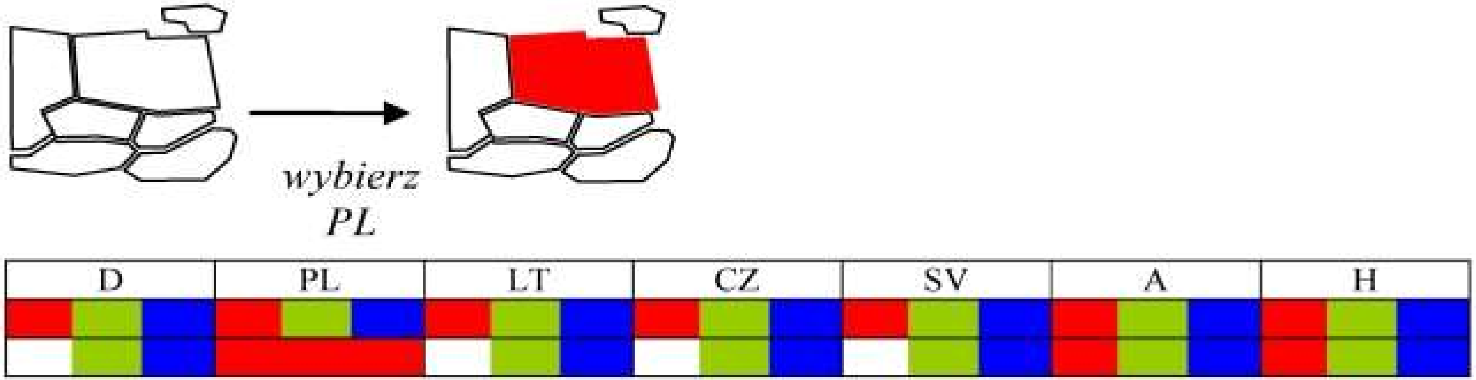
- Idea:
  - Należy śledzić pozostałe dopuszczalne wartości jeszcze niezwiązanych zmiennych.
  - Należy przerwać poszukiwania wtedy, gdy jakaś zmienna nie ma już żadnych dopuszczalnych wartości.



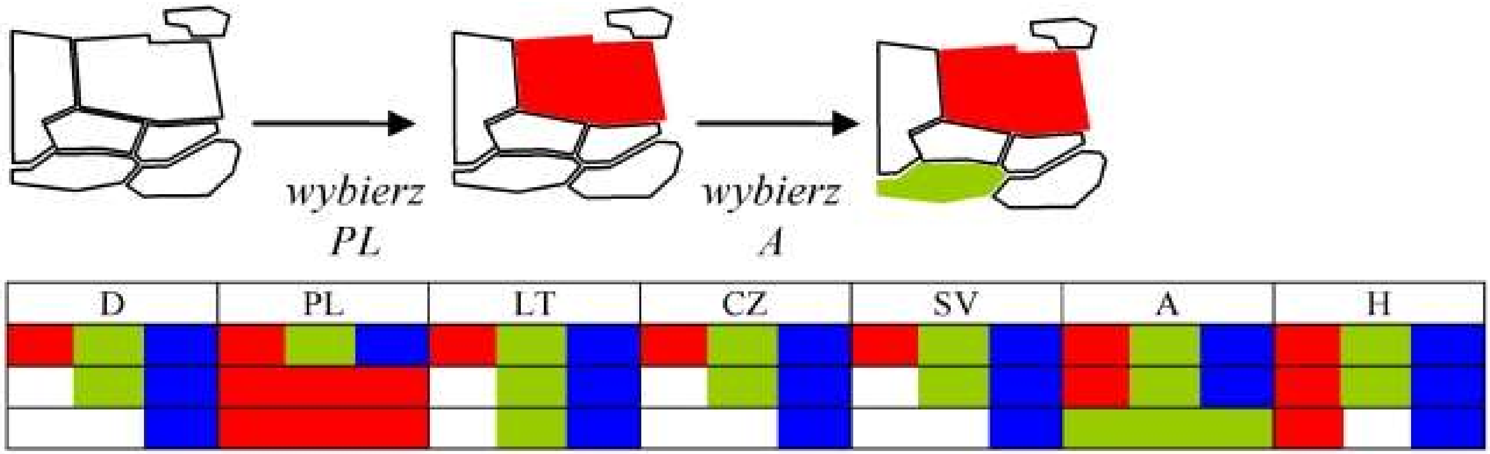


W tym stanie każdej zmiennej można przyporządkować jeszcze każdy kolor.

Sprawdzanie wprzód (2)

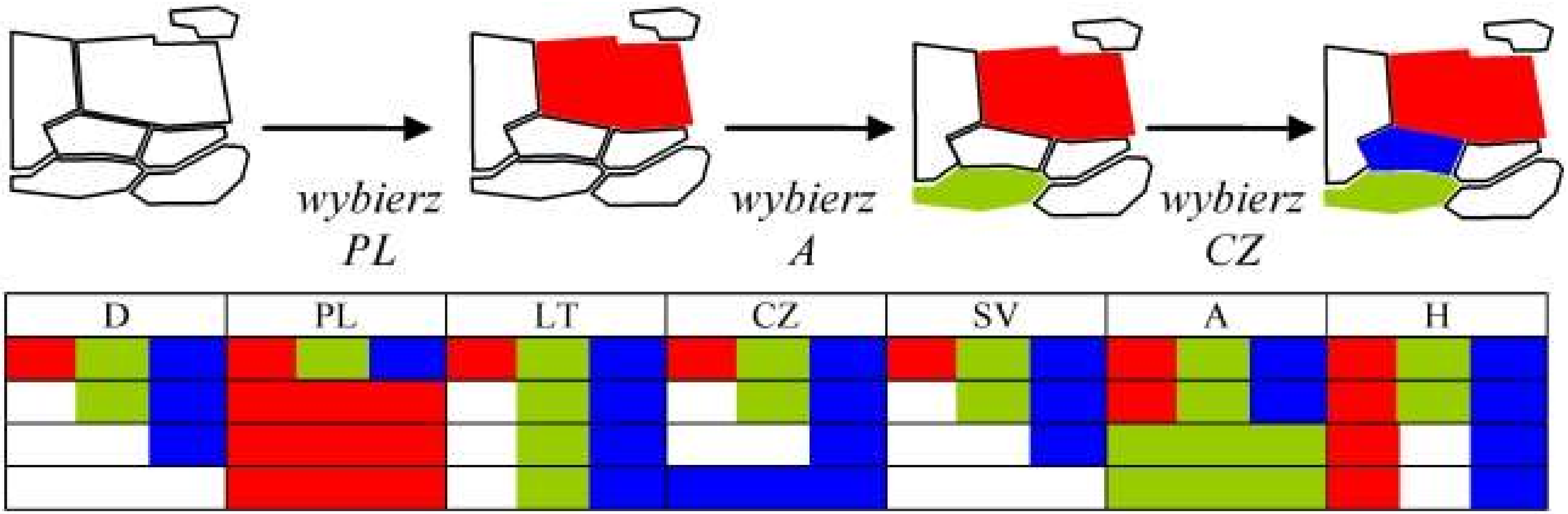


Przypisanie (*PL=r*) zmniejsza legalne wartości dla *D,CZ,SV, LT;*



Przypisanie (*A=g*) sprawia, że *D, CZ, SV i H* tracą kolor *g*; sprawdzanie w przód nie wykryje, że *D* i *CZ* są już w konflikcie;

Sprawdzanie wprzód (3)



Przypisanie (*CZ=b*) kasuje już wszelkie możliwe wartości *D* i *SV*, więc należy wykonać nawrót

Analiza spójności ograniczeń

Sprawdzanie w przód dzięki istniejącym ograniczeniom propaguje informację o dopuszczalnych wartościach od związanych do niezwiązanych zmiennych, ale nie umożliwia wcześniejszego wykrycia porażki zanim zabraknie wartości dla pewnej zmiennej.

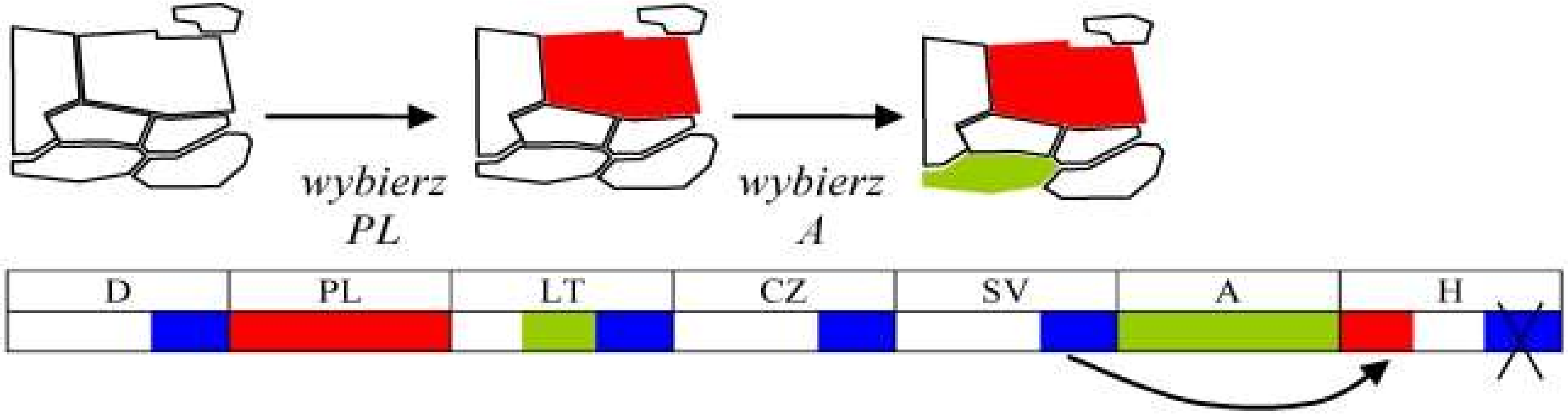
Np.: w poprzednim przykładzie po 2 krokach dla *D* i *CZ* pozostała już tylko jedna legalna wartość niebieska. Z analizy ograniczeń wynika jednak, że *D* i *CZ* nie mogą jednocześnie być niebieskie!

→ Pomocna będzie **analiza spójności ograniczeń** dla stanu.

C2) Spójność łuków (ograniczeń)

- Analizujemy spójność każdego łuku w grafie ograniczeń.
- Łuk *X* → *Y* jest spójny wtedy i tylko wtedy gdy dla każdej wartości *x* przypisanej do *X* istnieje jeszcze jakaś dopuszczalna wartość *y* dla *Y.*

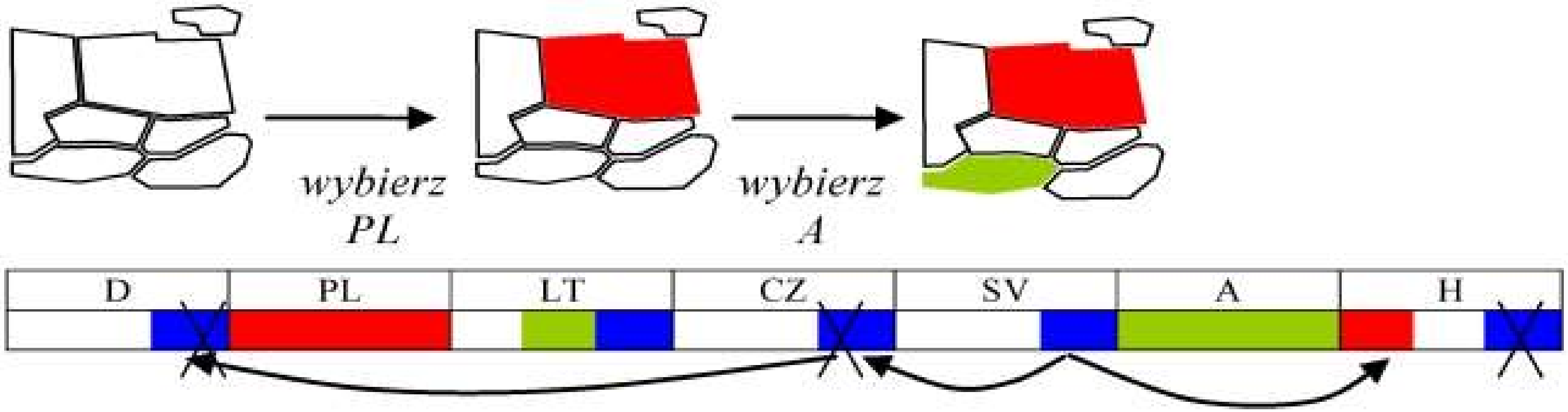
Np.



Łuk określa badane ograniczenie pomiędzy zmiennymi. W tym przypadku dla zapewnienia rozwiązania H straci wartość b. Ale łuk pomiędzy SV i H jest nadal spójny.

Spójność łuków – propagacja (2)

- Jeżeli *X* straci wartość, sąsiedzi *X* muszą być sprawdzeni ponownie.



Np. Po przypisaniu, *A=g,* sprawdzane są łuki w grafie ograniczeń „wychodzące" z *A*. Takim sąsiadem jest zmienna *SV* - traci ona kolor *r*. Następnie sprawdzani są jej sąsiedzi, *H*, *CZ* i *PL*.

Łuki pomiędzy *SV* i *CZ* oraz *CZ* i *D* przestają być spójne.

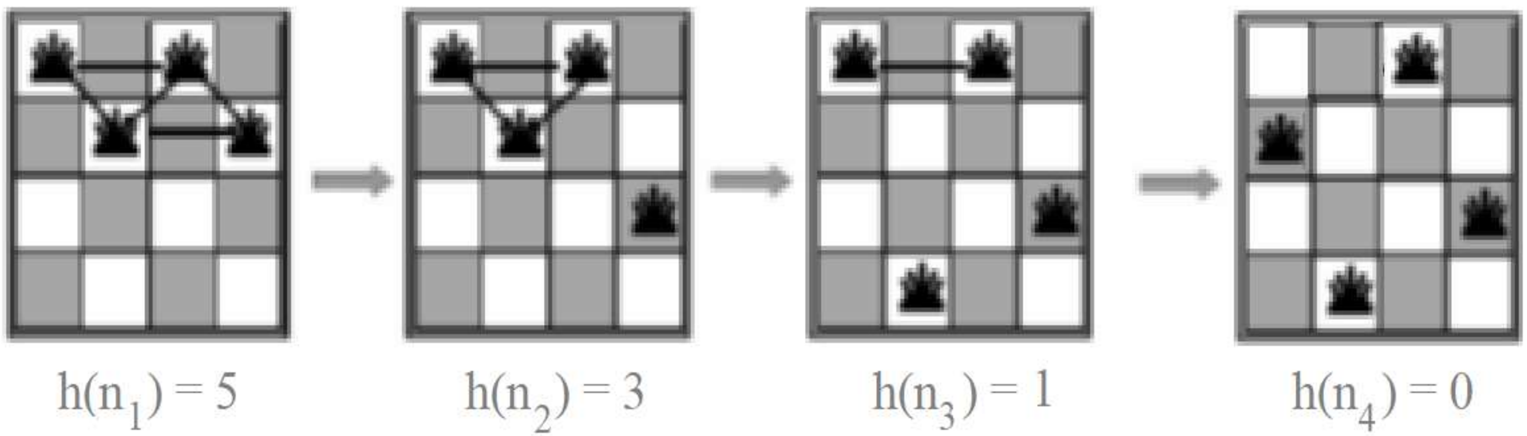
8. Przeszukiwanie ze stanem zupełnym dla CSP

- Przeszukiwanie stanów „zachłanne z heurystyką”, „przez wspinanie” czy „symulowane wyżarzanie” zakładają „kompletne" konfiguracje problemu (**stany kompletne**). W przypadku CSP stan zupełny to taki, gdy wszystkie zmienne mają przypisane wartości.
- Aby zastosować te metody dla rozwiązania problemu CSP:
  - Dopuścimy stany (kompletne) z konfliktami zmiennych.
  - Operatory przestrzeni (akcje) dokonują zmian wartości zmiennych.
- Selekcja zmiennych: losowo wybrać dowolną ze zmiennych znajdujących się w konflikcie.
- Selekcja wartości: poprzez heurystykę podającą liczbę konfliktów - wybrać wartość, która narusza najmniejszą liczbę ograniczeń – przeszukiwanie przez wspinanie się i $h(n)$ = całkowita liczba naruszonych ograniczeń.

Przykład: problem 4 królowych jako CSP

- Stany: 4 hetmanów w 4 kolumnach ($4^4 = 256$ stanów)
- Akcje: przesuń hetmana w kolumnie
- Warunek stopu: brak wzajemnych ataków (konfliktów)
- Ocena: $h(n)$ = liczba wzajemnych ataków (konfliktów)

  Np. Ścieżka w drzewie decyzyjnym prowadząca do rozwiązania (stanu końcowego) spełniającego wszystkie ograniczenia:



9. Gra z przeciwstawnymi celami

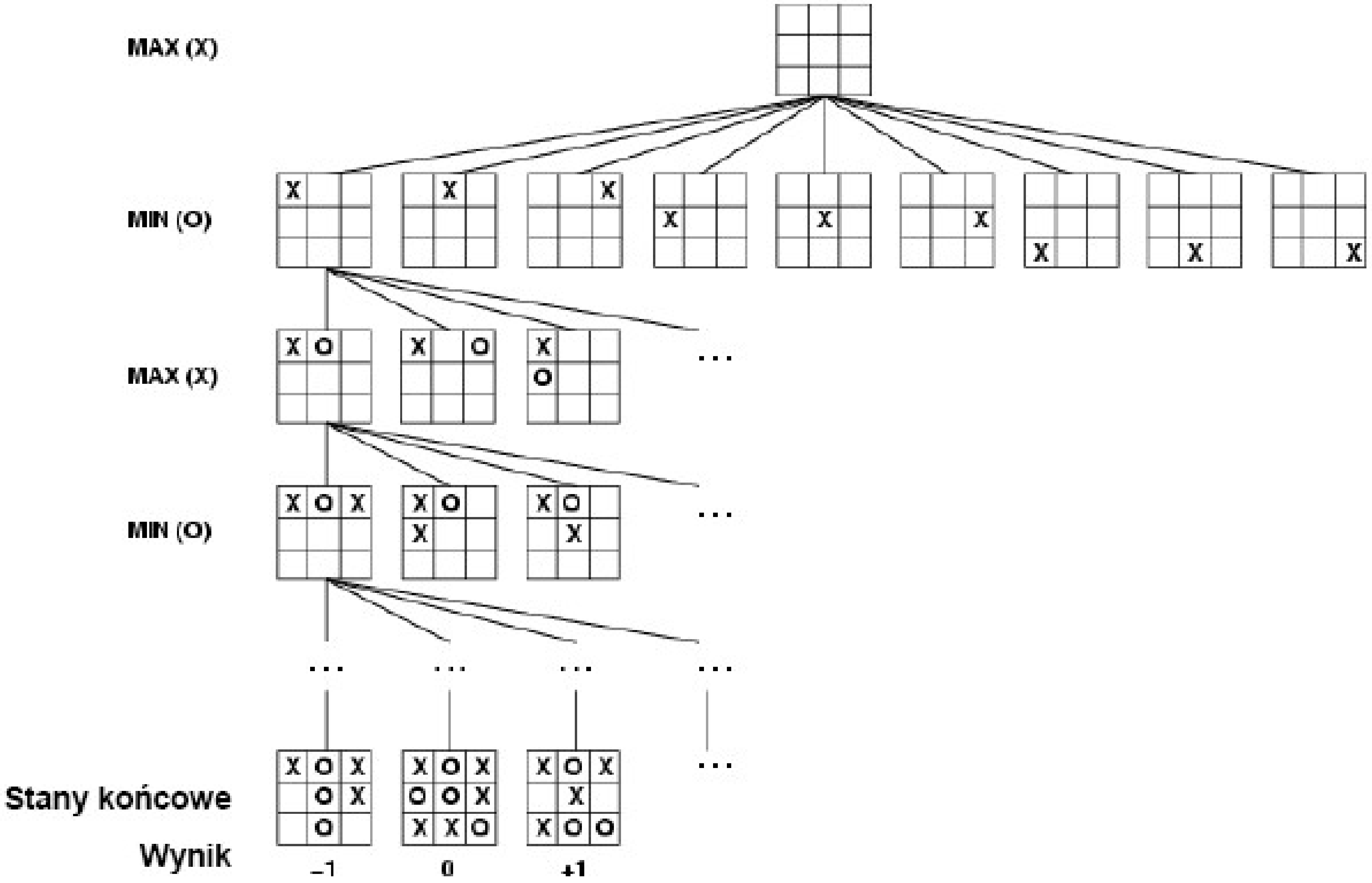
- W nietrywialnych grach zagadnienie optymalnego wyboru następnego ruchu wymaga rozważenia wszystkich możliwych sekwencji ruchów własnych i przeciwnika. W praktyce wybór ruchu w grach dwuosobowych (o przeciwstawnych celach), tak jak rozwiązywanie innych złożonych zadań, wymaga odpowiednich strategii przeszukiwania.
- „Nieprzewidywalny" przeciwnik → należy określić własny ruch przewidując „najgorszą dla nas” decyzję przeciwnika.
- Ze względu na ograniczenie czasu gry → znalezienie optymalnego ruchu metodą przejrzenia wszystkich możliwych sekwencji ruchów prowadzących do celu może nie być możliwe → trzeba przyjąć pewne ograniczenia.

Drzewo gry

Ograniczmy nasze rozważania do gier dwuosobowych (o przeciwstawnych celach), gdy akcje wykonywane są na przemian przez obu graczy. Rozważamy model gry w postaci przeszukiwania specyficznego drzewa typu „Mini-max”:

- Występują 2 rodzaje węzłów: **Min** i **Max**,
  - węzeł **Min** reprezentuje stan gry, w którym ruch należy do przeciwnika, węzeł **Max** odpowiada decyzji naszego gracza,
  - ocena stanu gry propagowana jest „od dołu na górę” (od liści do korzenia drzewa) - ocenę węzła rodzica typu **Max** ustawiamy na wartość maksymalną spośród jego węzłów potomnych (następników), ocena węzła rodzica typu **Min** jest minimalną spośród jego następników;
- Akcje: „nasz” gracz wybiera akcję odpowiadającą przejściu do **najlepszego** spośród jego następców; „przeciwnik” wykonuje ruch odpowiadający przejściu do (dla nas) najgorszego następnika.

Drzewo gry typu Mini-max



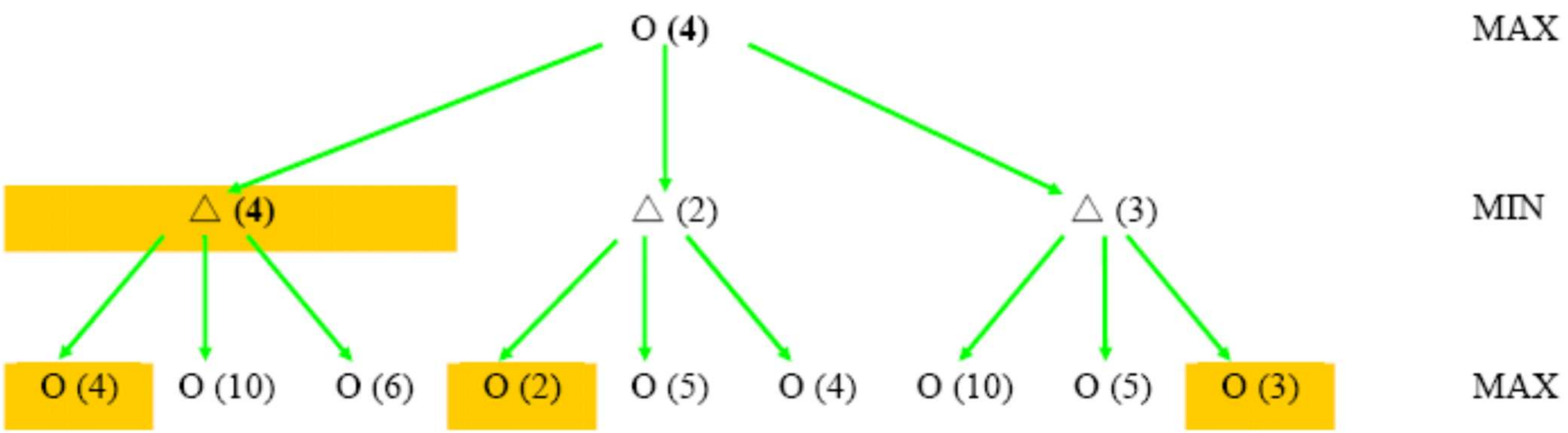
Wynik gry

Ocena węzła odpowiada możliwemu do osiągnięcia wynikowi gry, reprezentowanemu przez liść drzewa przeszukiwania. Np. możliwy wynik w grze „kółko i krzyżyk”:

+1 = wygrana, 0 = remis, -1= porażka.

Niektóre gry mogą jednak kończyć się z różnymi wynikami punktowymi. W ogólności funkcja Wynik(*stan*) dostarcza oceny liczbowej każdego stanu końcowego gry.

Np.



10. Strategia „Mini-max”

„Mini-max” zakłada **idealną** rozgrywkę dla **deterministycznych** 2-osobowych gier naprzemiennych.

Idea: wybieramy ruch do pozycji z najwyższą wartością, a przeciwnik wybiera dla nas najgorszy ruch.

Strategia „Mini-max” : uzyskać najlepszy możliwy wynik w grze z najlepszym przeciwnikiem.

Implementacja strategii „Mini-max”

- Funkcja MiniMaks() i jej podfunkcje MaksOcena() i MinOcena() .

Implementacja strategii „Mini-max”

```
function MiniMaks(stan) : returns akcja
{
      v := MaksOcena(stan);
      wybierz akcja = (stan, stan_N),
                  gdzie: stan_N ∈ Następne(stan), Wynik(stan_N) == v ;
      return akcja;
}
function MaksOcena(stan) : returns najlepszy_możliwy_wynik
{
      if (TerminalTest(stan)) then return Wynik(stan);
      v := - ∞ ;
      for (s ∈ Następne(stan)) do v := max(v, MinOcena(s));
      return v;
}
```

Implementacja strategii „Mini-max” (c.d.)

function **MinOcena(*stan*)** : **returns** *najgorszy_możliwy_wynik*
{
    if (TerminalTest(*stan*)) **then return** Wynik(*stan*);
    v **:=** ∞ ;
    for ($s \in$ Następne(*stan*)) **do** v := min(v, MaksOcena(s));
    return v;
}

Charakterystyka przeszukiwania Mini-max

- Zupełność? Tak (jeżeli drzewo jest skończone)
- Optymalność? Tak (jeżeli przeciwnik jest racjonalny)
- Czas? $O(b^m)$ (przeszukujemy całe drzewo)
- Pamięć? $O(bm)$ (stosujemy przeszukiwanie w głąb)

  gdzie: b – średni stopień rozgałęzienia drzewa, m – długość ścieżki rozwiązania.

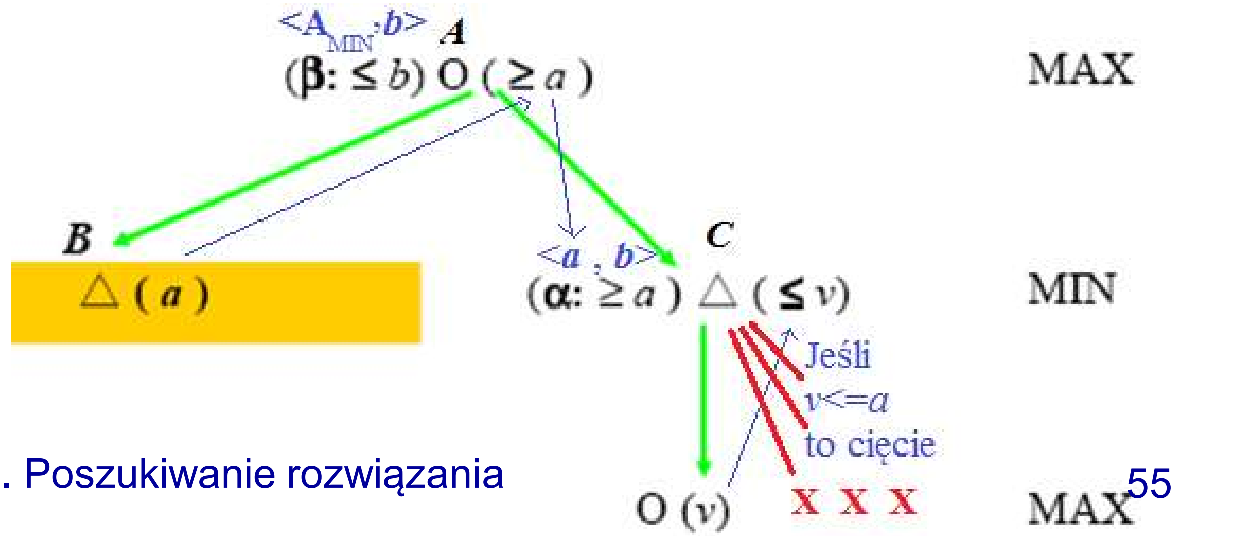
Dla szachów szacujemy: $b \approx 35$, $m \approx 100$, dla "*rozsądnych*" rozgrywek, tzn. wtedy gdy spotykają się gracze o podobnych wysokich umiejętnościach. W praktyce, przy tak dużych wartościach parametrów, dokładne rozwiązanie problemu strategią **Mini-max** jest niemożliwe.

→ Wymagane jest usprawnienie tej strategii.

11. „Cięcia α-β”

- Niech $a = v(B)$ jest oceną już przeanalizowanego poddrzewa o korzeniu B (typu MIN) dla korzenia całego drzewa A typu MAX.

  Odtąd w korzeniu A interesuje nas czy można jeszcze powiększyć a.
- Niech C będzie kolejnym następnikiem MIN korzenia A. Jeśli podczas analizy pierwszej lub kolejnej gałęzi O węzła C jego ocena v(O) byłaby nie lepsza niż a, ($v(O) \leq a$) to bez szkody dla optymalnego rozwiązania zrezygnujemy z dalszych gałęzi C (cięcie „alfa”).
- Analogicznie zrobilibyśmy cięcie „beta” kolejnych gałęzi MIN węzła A, po stwierdzeniu, że $v(C) \geq b$, gdy b jest ograniczeniem górnym dla oceny węzła A.



Cięcia α, β

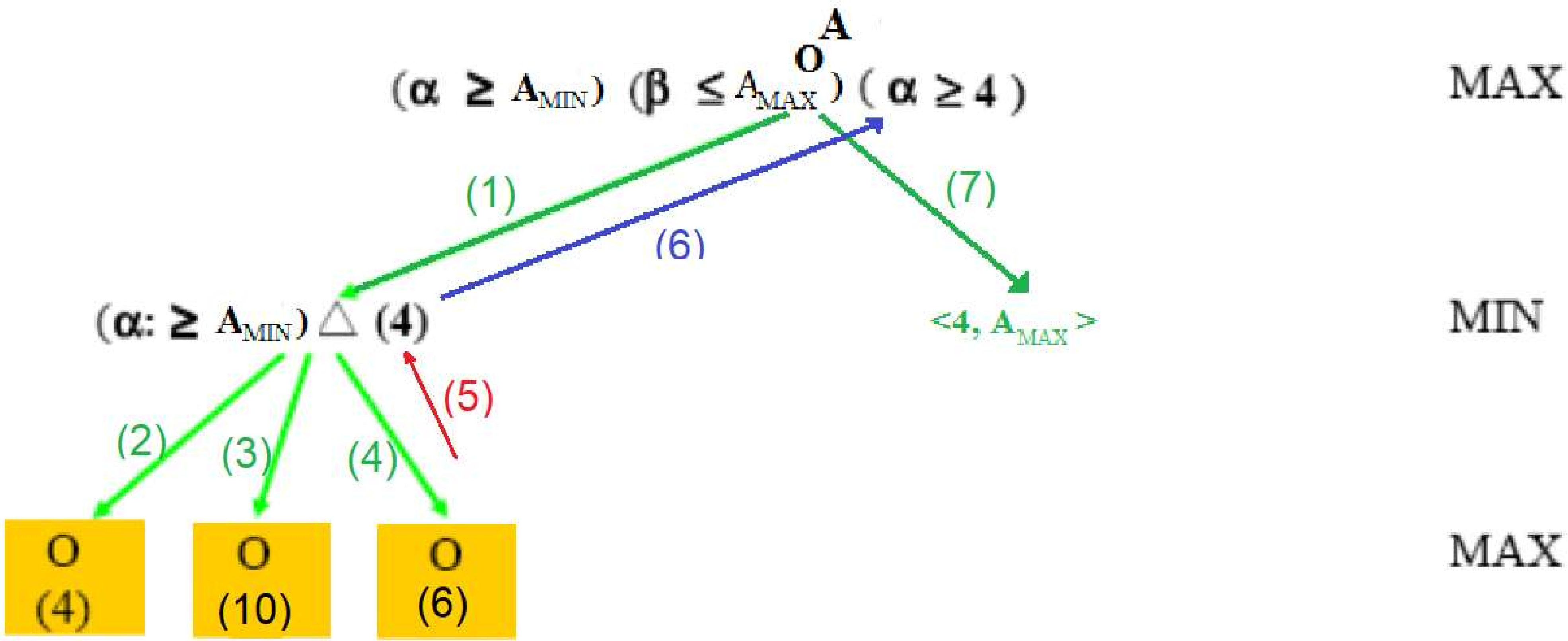
Ograniczenie dolne (alfa) i górne (beta) przekazywane są od węzła rodzica „w dół” do jego węzłów potomnych w drzewie decyzyjnym typu MiniMax.

Cięcie alfa: oceniając węzeł n typu MIN przez minimalizację ocen węzłów potomnych typu MAX możemy zakończyć wyznaczanie ocen węzłów potomnych natychmiast po stwierdzeniu, że ocena węzła MIN nie będzie wyższa niż jej ograniczenie dolne α, gdyż ocena sprawdzonego już następnika m: $v(m) \leq \alpha(n)$.

Cięcie beta: oceniając węzeł m typu MAX przez maksymalizację ocen węzłów potomnych typu MIN możemy zakończyć wyznaczanie ocen kolejnych węzłów potomnych natychmiast po stwierdzeniu, że ocena węzła MAX nie będzie niższa niż jej ograniczenie górne β, gdyż ocena sprawdzonego już następnika n: $v(n) \geq \beta(m)$.

Ograniczenie dolne jest maksymalizowane w węźle typu MAX, a górne – minimalizowane w węźle typu MIN.

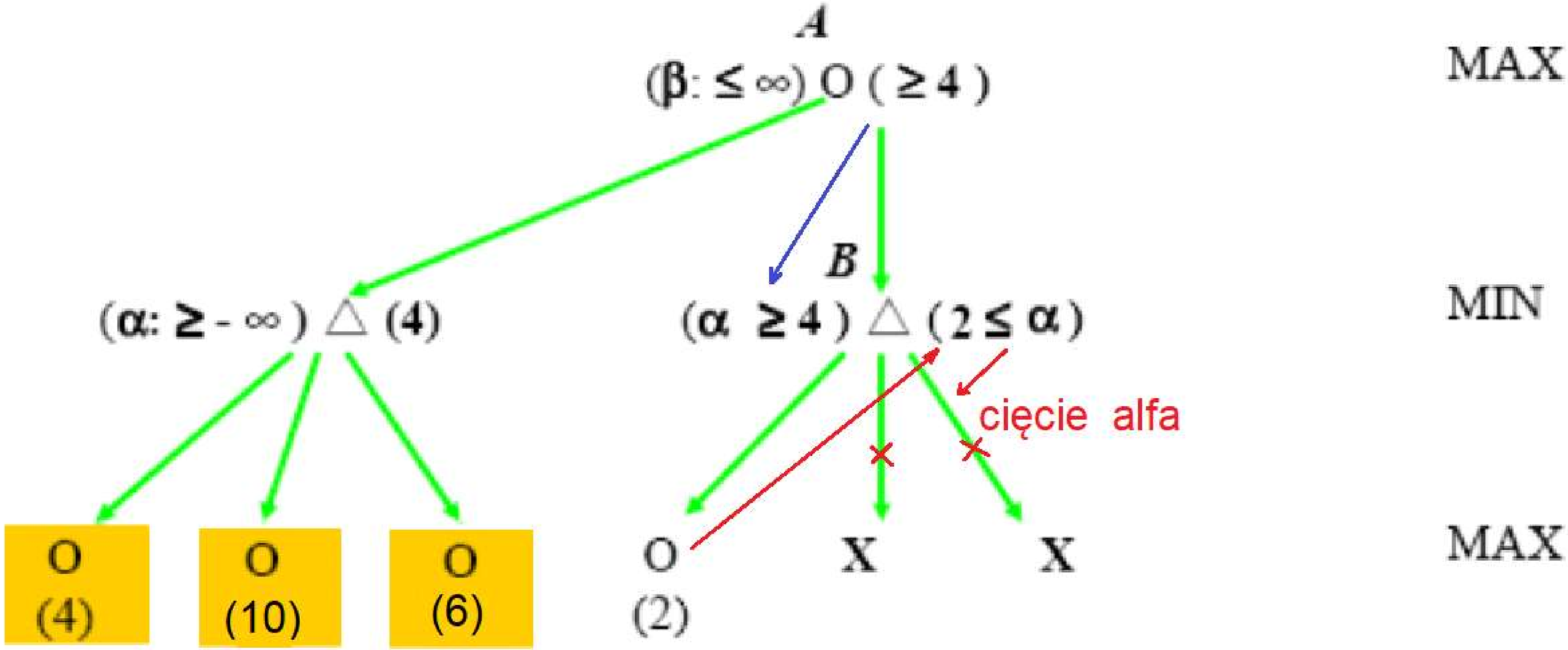
Przykład cięć α-β (1)



Modyfikacja ograniczenia górnego α w węźle *A* typu MAX:

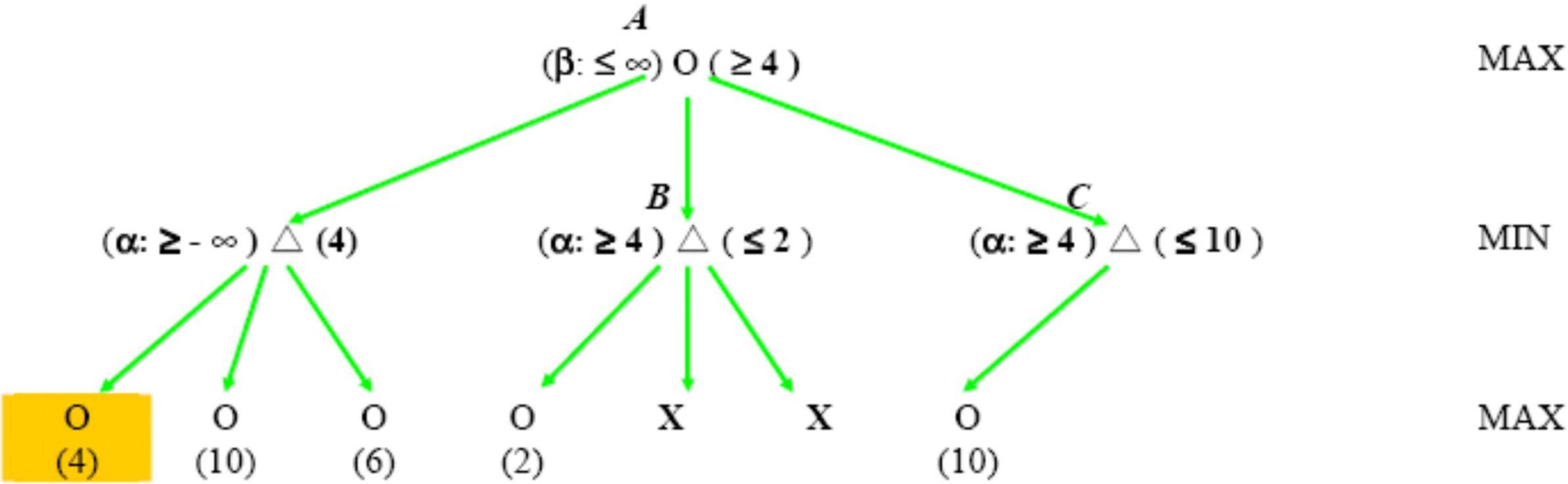
- zakładamy, że początkowo: $A_{MIN} < 4$, $A_{MIN} > 10$;
- pierwsza przeanalizowana gałąź daje ocenę 4;
- teraz ograniczenie dolne dla kolejnych gałęzi węzła *A*: $\alpha = 4$.

Przykład cięć α-β (2)



Dla węzła *B* typu MIN dotychczas przeanalizowana gałąź ma ocenę 2. W kontekście ograniczenia dolnego dla *B* wynoszącego 4 zachodzi $((\alpha=4) \geq 2)$, co „blokuje" rozpatrywanie pozostałych gałęzi węzła *B* - mamy *cięcie alfa.*

Przykład cięć α-β (3)

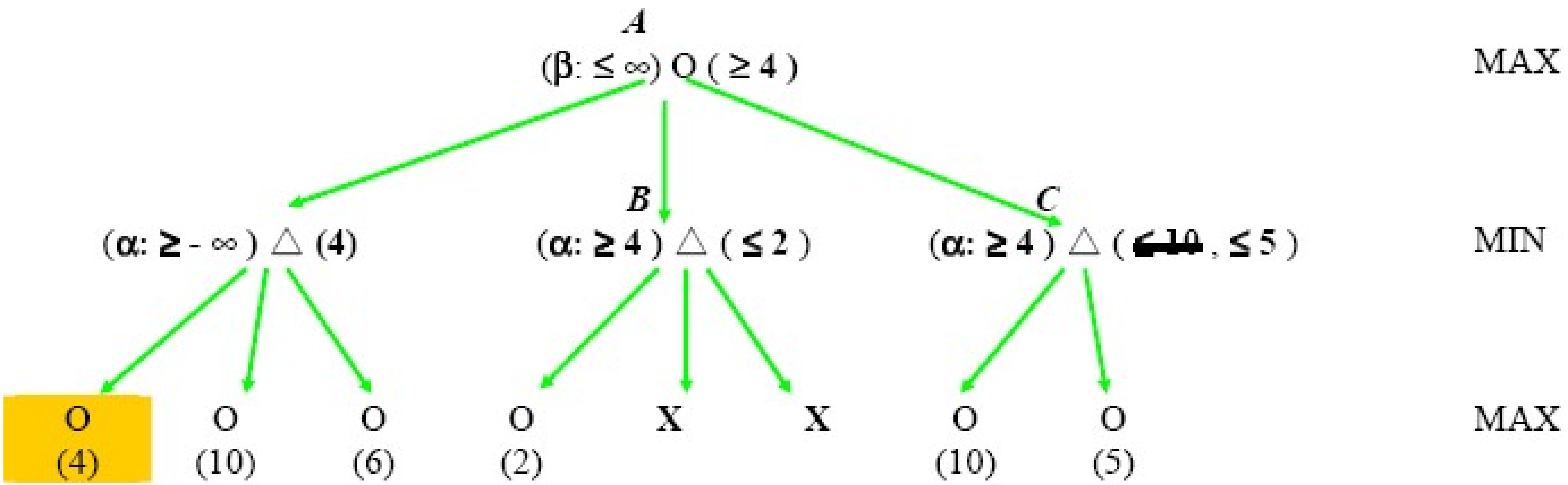


Modyfikacja ograniczenia górnego w węźle C typu MIN.

Dla węzła *C* typu MIN jej pierwsza gałąź ma ocenę 10.

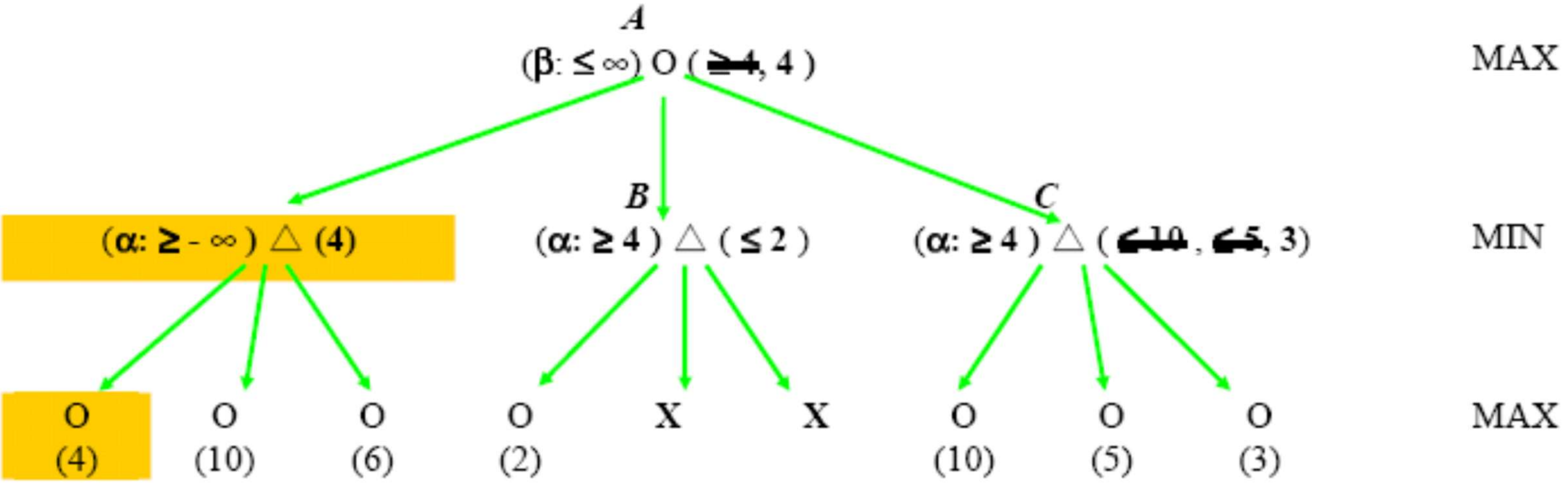
Nie ma „cięcia alfa" pozostałych gałęzi *C*, gdyż $(10 > \alpha)$, ale zmniejszamy ograniczenie $(\beta = 10)$ dla kolejnych potomków węzła *C*.

Przykład cięć α-β (4)



Dla węzła *C* druga gałąź daje ograniczenie górne 5. Nadal nie możemy wykonać przycinania pozostałej gałęzi *C*, gdyż warunek α nadal jest spełniony w węźle *C*, jednak ponownie ustawiamy nowe ograniczenie górne, ($\beta = 5$), dla potomków węzła *C*.

Przykład cięć α-β (5)



Ostatnia gałąź węzła C dała wynik 3.

W węźle A następuje maksymalizacja wyników jej gałęzi: $\max(4, 2, 3) = 4$. Wybierany jest ruch prowadzący do węzła typu MIN o ocenie 4. Pomimo wykonanego cięcia wynik jest ten sam co dla podstawowego algorytmu Mini-Max.

Implementacja „cięć alfa-beta”

Implementacja strategii „Mini-max z cięciami alfa-beta”

- Niech A_{MIN} oznacza najniższą możliwą ocenę w danej grze;
- Podobnie A_{MAX} niech oznacza najwyższy możliwy wynik.
- Jeśli nie można ustalić tych wartości to przyjmujemy:

$$A_{MIN} = -\infty \; ; \; A_{MAX} = \infty.$$

- Funkcja CięciaAlfaBeta() i jej podfunkcje MaksOcena() i MinOcena() podane są na następnych stronach.

Algorytm „cięć α-β”

```
function CięciaAlfaBeta(stan, A_MIN, A_MAX) : returns akcja {
    v := MaksOcena(stan, A_MIN, A_MAX);
    wybierz akcja = (stan, stan_N),
            gdzie: stan_N ∈Następne(stan), Wynik(stan_N) == v ;
    return akcja;
}
function MaksOcena(stan, α, β) : returns najlepszy_wynik {
    if (TerminalTest(stan)) then return Wynik(stan);
    v :=  α ;
    for (s ∈ Następne(stan)) do
    begin  v := max(v, MinOcena(s, α, β ));
        if v ≥ β  then return v;  // Cięcie beta
        α := max(α, v);
    end;
    return v;
}
```

Algorytm „cięć α-β”

Dodatkowe parametry:

α - największa wartość węzła Max dla ścieżki rozwiązania

β - najmniejsza wartość węzła Min dla ścieżki rozwiązania

function **MinOcena**(*stan*, α, β) : **returns** *najgorszy_wynik* {

    if (TerminalTest(*stan*)) **then return** Wynik(*stan*);

    $v := \beta$;

    for ($s \in$ Następne(*stan*)) **do**

    begin $v :=$ min(v, MaksOcena(s, α, β));

        if $v \leq \alpha$ **then return** v; // Cięcie alfa

        $\beta :=$ min(β, v);

    end;

    return v;

}

Własności „cięć α-β”

- Przycinanie nie ma wpływu na końcowy rezultat przeszukiwania. Osiągamy ten sam wynik, który dałoby zastosowanie podstawowego Mini-max-u.
- Sprzyjające uporządkowanie ruchów (następników węzła) poprawia efektywność przycinania.
- Przy "perfekcyjnym uporządkowaniu” następników :

  złożoność czasowa = $O(b^{m/2})$

  Dzięki temu możliwe staje się rozwiązanie za pomocą „Mini-max-u z cięciami α-β” problemów o dwa razy większej głębokości drzewa przeszukiwania niż w przypadku klasycznej strategii Mini-max.

12. Heurystyczna ocena – „obcięty MiniMax”

W praktyce występują ograniczenia zasobów. Załóżmy, że mamy 100 sekund czasu na wykonanie ruchu i możemy przeglądać węzły z prędkością 10^6 węzłów na sekundę → czyli mamy czas na przeglądanie do 10^8 węzłów dla wykonania jednego ruchu.

<u>Standardowa modyfikacja „Mini-max-u” w praktyce:</u>

1. Wykonujemy „test odcięcia” dla ścieżki (tu: funkcja C*utoff*):

   np.: Test polega na ograniczeniu głębokości drzewa do wartości zadanej parametrem D.

2. Wprowadzamy funkcję oszacowania stanu (heurystyka) (tu: Ocena()) – dokonuje ona szacunkowej oceny (niekońcowej) konfiguracji gry, reprezentowanej przez aktualny węzeł drzewa.

Przykład funkcji oceny stanu

- Dla warcabów, zazwyczaj stosujemy liniową sumę ważoną cech pionków w danej pozycji:

$$Ocena(s) = w_1\, f_1(s) + w_2\, f_2(s) + \ldots + w_n\, f_n(s)$$

- Np.: $w_1 = 9$ (szacunkowa moc damki),

$$f_1(s) = (\text{liczba białych damek}) - (\text{liczba czarnych damek})$$

  itp.

- W ocenie uwzględnia się również wagi poszczególnych pozycji na planszy i wagi dla wzajemnych położeń pionków obu graczy.

„Obcięty Mini-max”

Implementacja strategii „obciętego Mini-max-u” jest prawie identyczna z podstawowym algorytmem „Mini-max”:

1. Po sprawdzeniu warunku stopu – TerminalTest() - mamy teraz dodatkowo sprawdzanie warunku obcinania ścieżki – funkcją Cutoff() - i ewentualne obcięcie z podaniem wartości heurystycznej oceny akualnego stanu *.*
2. Zamiast funkcji Wynik()*,* podającej *rzeczywistą ocenę końcowego stanu gry* dla pierwszego gracza, w sytuacji obcinania wywołujemy *funkcję oszacowania* stanu: Ocena().

Implementacja strategii „obciętego Mini-max-u”

```
function ObciętyMiniMaks(stan, D) : returns akcja {
    v := MaksOcena(stan, D);
    wybierz akcja = (stan, stan_N),
            gdzie: stan_N ∈Następne(stan), Wynik(stan_N) == v ;
    return akcja;
}
function MaksOcena(stan, d) : returns najlepszą_ocena
{
    if (TerminalTest(stan)) then return Wynik(stan);
    if (Cutoff(stan, d)) then return Ocena(stan);
    v := - ∞ ;
    for (s ∈ Następne(stan)) do v := max(v, MinOcena(s, d-1));
    return v;
}
```

Implementacja „obciętego Mini-max-u” (c.d.)

```
function MinOcena(stan, d) : returns najgorsza_ocena
{
    if (TerminalTest(stan)) then return Wynik(stan);
    if (Cutoff(stan, d)) then return Ocena(stan);
    v := + ∞ ;
    for (s ∈ Następne(stan)) do v := min(v, MaksOcena(s, d-1));
    return v;
}
```

„Obcięty Mini-max” w praktyce

Czy ta strategia dobrze działa w praktyce?

Dla szachów, niech liczba możliwych stanów (na początku gry), które mogą być wizytowane w zadanym czasie wynosi $b^d = 10^8$.

Przy typowym stopniu rozgałęzienia początkowego drzewa dla gry w szachy, $b = 35$, odcinanie musiałoby w takiej sytuacji nastąpić po, $d = 5$, posunięciach.

Przewidywanie przez gracza jedynie 5 ruchów do przodu jest w praktyce oznaką słabego gracza w szachy. Siła gry gracza w szachy oceniana jest bowiem według następującej skali:

– przewiduje 4 ruchy ≈ początkujący gracz,
– przewiduje 8 ruchów ≈ program szachowy na typowym komputerze PC, mistrz szachowy,
– 12 ruchów ≈ program na komputerze *Deep Blue*, arcymistrz szachowy.

Pytania

1. Przedstawić losowe, niepoinformowane algorytmy poszukiwania celu („próbkowanie”, „błądzenie”).
2. Przedstawić poinformowane przeszukiwanie lokalne („przez wspinanie”).
3. Przedstawić poinformowane losowe przeszukiwanie („symulowane wyżarzanie”).
4. Omówić algorytm genetyczny.
5. Jak reprezentujemy dyskretny problem z ograniczeniami (CSP)?
6. Wyjaśnić zasadę działania algorytmu przeszukiwania przyrostowego dla CSP ?

Pytania (c.d.)

7. Na czym polegają **usprawnienia** algorytmu przeszukiwania z nawrotami? Zilustrować odpowiedzi na przykładzie.
8. Na czym polega przeszukiwanie CSP ze stanem zupełnym?
9. Wyjaśnić cel stosowania i zasady działania strategii „przeszukiwanie MiniMax”
10. Omówić strategię „cięć α-β”. Czy jest ona optymalna?
11. Wyjaśnić cel stosowania i zasady działania strategii „obciętego Mini-max-u”. Czy jest ona optymalna - jeśli tak, to w jakich warunkach a jeśli nie - to dlaczego?