2021

Metody Sztucznej Inteligencji. 4. Poszukiwanie rozwiązania. Planowanie działań.

4. POSZUKIWANIE ROZWIĄZANIA. PLANOWANIE DZIAŁAŃ

WŁODZIMIERZ KASPRZAK

POSZUKIWANIE CELU, STRATEGIE DETERMINISTYCZNE, STRATEGIE PSEUDOLOSOWE, PROBLEM Z OGRANICZENIAMI, STRATEGIE MINI-MAX, STRIPS, GRAPHPLAN

Przedstawione zostaną podstawowe strategie decyzyjne polegające na poszukiwaniu celu (rozwiązania problemu) w dyskretnej „przestrzeni stanów” wyrażającej problem decyzyjny agenta Sztucznej Inteligencji. W tej uniwersalnej metodyce celem jest znalezienie stanu, który jest optymalny ze względu na funkcję użyteczności lub spełnia ograniczenia (oba te pojęcia zależą od problemu). Omówione zostaną algorytmy poszukiwania „przez wspinanie” (i jego wersja ciągła - „wzdłuż gradientu”), losowe i pseudo-losowe (np. „symulowane wyżarzanie”, algorytmy „genetyczne”). W tym przypadkach funkcja celu jedynie wartościuje każdy stan nie wnikając w ich wewnętrzne własności. Inne omawiane problemy i ich rozwiązania zakładają znajomość własności każdego stanu. Są nimi: dyskretny problem spełniania ograniczeń (CSP) i gry z przeciwstawnymi celami (Mini-Max). Na koniec przedstawione zostaną dwa rozwiązania problemu planowania działań agenta Szt. Int., w oparciu o własności stanów reprezentowane językiem predykatów – metoda STRIPS tworzenia planu częściowo uporządkowanego i metoda Graphplan.

Spis treści

1. Poszukiwanie rozwiązania (celu, stanu końcowego)

W wielu problemach obliczeniowych, ścieżka prowadząca do celu sama w sobie nie jest taka ważna, jak sam fakt osiągnięcia celu. Np. w grach, przestrzeń stanów definiowana jest jako zbiór „pełnych" konfiguracji pionków. Celem gry (poszukiwanym rozwiązaniem) jest znalezienie konfiguracji spełniającej pewne warunki.

1.1. Poszukiwanie ścieżki a poszukiwanie celu

Rozpatrywane dotąd strategie przeszukiwania przestrzeni stanów poszukiwały optymalnej ścieżki dojścia do celu (stanu końcowego), przy czym mogło być wiele stanów końcowych i każdy z nich był równorzędny. Optymalność rozwiązania gwarantowały te strategie, które korzystały z horyzontu globalnego lub mogły wykonać nawrót do poprzednio wizytowanych stanowi. Złożoność obliczeniowa zależała w dużym stopniu od dokładności „poinformowania" strategii o odległości od celu. Przy niskiej dokładności takiej informacji, złożoność obliczeniowa rosła wykładniczo $O(b^m)$ wraz z długością ścieżki rozwiązania m, ale poprawiała się ona przy lepszej dokładności heurystyki kosztów resztkowych i zdążała do liniowej zależności od długości ścieżki O(bm).
Teraz zajmiemy się strategiami, w których sama ścieżka nie jest istotna a liczy się jedynie znalezione rozwiązanie (cel), a gdy występuje wiele rozwiązań to zazwyczaj poszukiwane jest najlepsze z nich. Wyróżnimy strategie:

- korzystające z elementów **losowości**
  lub
- mające charakter **lokalnego** przeszukiwania.

W odróżnieniu od deterministycznych strategii poszukiwania ścieżki korzystających z horyzontu globalnego i gwarantujących uzyskanie optymalnych ścieżek przy pewnych warunkach, strategie losowe i lokalne zazwyczaj nie gwarantują uzyskania globalnie optymalnego rozwiązania. Jednak ich zaletą jest znacznie mniejsza złożoność obliczeń (mniejsze wymagania na czas obliczeń i pamięć) w porównaniu do strategii globalnego przeszukiwania.

1.2. Losowość w poszukiwaniu celu

1.2.1. Losowe próbkowanie przestrzeni

Najprostsza losowa strategia przeszukiwania to **losowe próbkowanie globalne** (Tabela 1). Wraz z kolejnymi iteracjami algorytmu, zbiór odwiedzonych węzłów V będzie coraz większy, a prawdopodobieństwo, że pokryje się on ze zbiorem wszystkich węzłów grafu problemu, wzrasta do jedności.

Tabela 1. Algorytm losowego próbkowania przestrzeni stanów

```
function LosowePróbkowanie(problem, k)
returns stan_końcowy
{       V = ∅ ;
        for i = 1 to k
        {       generuj losowo s_i ∈ Stany(problem);
                V = V ∪ {s_i}; // zbiór CLOSED
                if (WarunekStopu(s_i)) return s_i;
        }
        return ∅;
}
```

1.2.2. Losowe próbkowanie skraju

W problemach, w których nie jesteśmy w stanie wygenerować a priori wszystkich stanów problemu, ale dla każdego węzła umiemy jedynie wygenerować jego najbliższych sąsiadów, znalezienie rozwiązania musi być wieloetapowe a wygenerowane następniki będą pochodzić z lokalnego sąsiedztwa aktualnego węzła. Czyli próbkowanie (losowy wybór następnika) ograniczy się do aktualnego skraju drzewa przeszukiwania, reprezentującego wygenerowane uprzednio węzły, zaś skraj jest każdorazowo rozszerzany o lokalne następniki wybranego węzła (Tabela 2).

Tabela 2. Strategia losowego wyboru następnika ze skraju drzewa przeszukiwania

```
function LosoweGenerowanieNastepnika(problem) return stan końcowy
{       k = 0; V0 = s0 = StanPoczatkowy(problem);
        A0 = Nastepniki(s0, problem) ; // zbiór OPEN
        repeat
        {       wybierz losowo sk+1 z Ak;
                Vk+1 = Vk ∪ {sk+1}; // zbiór CLOSED
                Ak+1 = Ak+1 ∪ Nastepniki(sk+1, problem) – Vk;
                k = k+1;
        } until (sk nie jest stanem końcowym, tzn.
                (WarunekStopu(problem, sk) = True)
        return sk;
}
```

1.2.3. Błądzenie przypadkowe – lokalny skraj

Metoda błądzenia przypadkowego charakteryzuje się l**okalnym** skrajem drzewa przeszukiwania. , Kolejne odwiedzane węzły pochodzą z lokalnych sąsiedztw swoich poprzedników (Tabela 3).

Tabela 3. Algorytm błądzenia przypadkowego

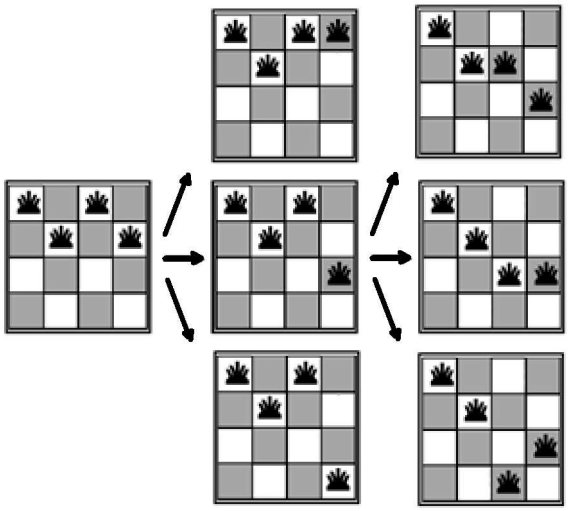
```
function BladzeniePrzypadkowe(problem, k) return stan końcowy
{       s0 = StanPoczatkowy(problem);
        V = { s0};
        for i = 1 to k
        {       generuj losowo si ∈ Nastepniki(si-1);
                V = V ∪ {si};
                if (WarunekStopu(si)) return si;
        }
        return ∅;
}
```

1.3. Przeszukiwanie lokalne

W sytuacji, gdy ze stanami przestrzeni związana jest funkcja oceny „jakości” (lub dualnie „kosztu”), możemy zastosować strategie przeszukiwania **lokalnego** sąsiedztwa. Zapamiętują one jedynie aktualny stan lub ścieżkę i starają się w sposób iteracyjny poprawiać ocenę aktualnego stanu dzięki przejściu do lepiej ocenionego stanu sąsiedniego.

Jak sama nazwa wskazuje, strategie lokalne operują na lokalnym skraju aktualnego drzewa decyzji, czyli na tych „liściach” drzewa, które odpowiadają bezpośrednim następnikom aktualnie rozszerzanego węzła. Rozwiązanie znalezione przez strategie przeszukiwania lokalnego nie musi być globalnie optymalne.

Przykład.
Problem „*n*-królowych”: należy ustawić *n* królowych na szachownicy o rozmiarze $n \times n$ tak, aby żadne dwie królowe nie znalazły się w tym samym wierszu, kolumnie i przekątnej (w zasięgu bicia) (Rysunek 1). Załóżmy, że każda konfiguracja pionków odpowiada stanowi przestrzeni przeszukiwania a ocena stanu (w tym przypadku – jego koszt) to liczba par pionków, które zagrażają sobie wzajemnie. Liczbę akcji ograniczymy do przesuwania jednego pionka w kolumnie szachownicy. Celem jest znalezienie stanu, którego koszt jest zerowy. Strategię przeszukiwania lokalnego charakteryzować będzie wybieranie najlepszego następnika aktualnego stanu i rozszerzanie aktualnej ścieżki w drzewie decyzji bez możliwości nawrotu i zmiany ścieżki.



Rysunek 1. Ilustracja stanów i tworzenia ścieżki w problemie 4 królowych.

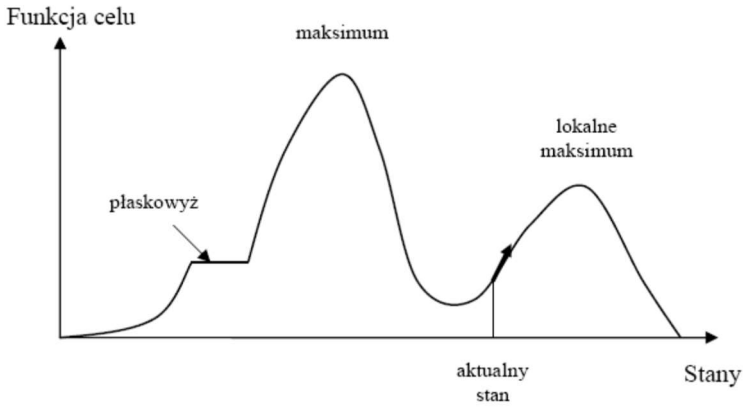
1.3.1. Przeszukiwanie przez „wspinanie”

Strategia lokalnego przeszukiwania z funkcją oceny stanów określana jest poglądowo zdaniem: „*Wspinanie się na Mt. Everest w gęstej mgle będąc dotkniętym amnezją*”. „Wspinanie się” oznacza wybór najlepiej ocenionego węzła następnika, „gęsta mgła” określa lokalny charakter skraju drzewa przeszukiwania, „amnezja” oznacza brak pamiętania przeszłych akcji i wcześniej odwiedzanych węzłów. Implementacja tej strategii nosi nazwę „przeszukiwania przez wspinanie” (ang. *hill climbing*) ().

Tabela 4. Algorytm „przeszukiwania przez wspinanie”.

```
function HillClimbing (problem) return stan;
{       węzełAktualny ← Węzeł(StanPoczątkowy([problem]));
        while (True)
        {       sąsiad ← NajlepszyNastępca(węzełAktualny);
                if (Ocena(sąsiad) ≤ Ocena(węzełAktualny)
                then    return Stan(węzełAktualny);
                węzełAktualny ←sąsiad;
        }
}
```

Na Rysunek 2 zilustrowano zasadniczą wadę każdej strategii lokalnej - możemy utknąć w lokalnym optimum lub na płaskowyżu funkcji oceny stanów.



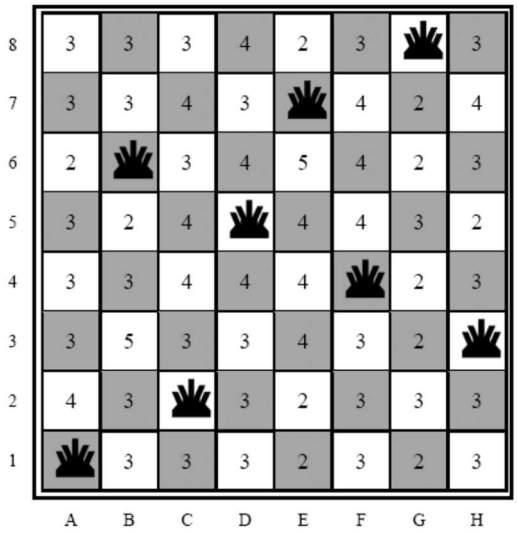
Rysunek 2. Ilustracja problemu „lokalnego optimum" w przeszukiwaniu lokalnym.

Przykład.
Przeszukiwanie przez „wspinanie" w zastosowaniu do problemu 8-królowych. Niech *h* oznacza liczbę atakujących się wzajemnie par hetmanów (bezpośrednio lub pośrednio). Załóżmy przykładowy stan początkowy taki, jak na Rysunek 3. Ponieważ stan ten zawiera już po jednej królowej w każdej kolumnie, więc możemy ograniczyć możliwe akcje do przesunięcia każdej królowej jedynie w ramach swojej kolumny. Ocena (koszt) stanu odpowiada liczbie zagrażających sobie par pionków – dla stanu początkowego ocena wynosi, *h* = 17. Liczba w każdym wolnym polu podaje ocenę dla stanu-następnika powstałego ze stanu początkowego po przesunięciu w to pole królowej znajdującej się w kolumnie tego pola. Czerwoną ramką zaznaczone zostały najlepsze oceny (najniższe koszty) następników stanu początkowego, *h* = 12. Dlatego w pierwszym kroku wybrana będzie akcja przesunięcia jednej z królowych na jedną z tych pozycji o ocenie 12.

	A	B	C	D	E	F	G	H
8	18	12	14	13	13	12	14	14
7	14	16	13	15	12	14	12	16
6	14	12	18	13	15	12	14	14
5	15	14	14	♛	13	16	13	16
4	♛	14	17	15	♛	14	16	16
3	17	♛	16	18	15	♛	15	♛
2	18	14	♛	15	15	14	♛	16
1	14	14	13	17	12	14	12	18

Rysunek 3. Przykładowy stan początkowy dla przeszukiwania lokalnego w problemie „8 królowych".

Iteracyjnie powtarzamy to postępowanie (generowanie następników stanu, ich ocena, wybór najlepszego następnika aktualnego stanu) dopóki możliwa jest poprawa funkcji oceny. Na Rysunek 4 podano stan końcowy osiągany w tym przykładzie z zadanego stanu początkowego stosując strategię „przez wspinanie". Stan końcowy ma ocenę *h* = 1. Nie jest to idealne rozwiązanie, które powinno mieć ocenę *h*=0. Nie ma już możliwości poprawy oceny, gdyż wszystkie stany następne mają gorszą ocenę od aktualnego stanu.

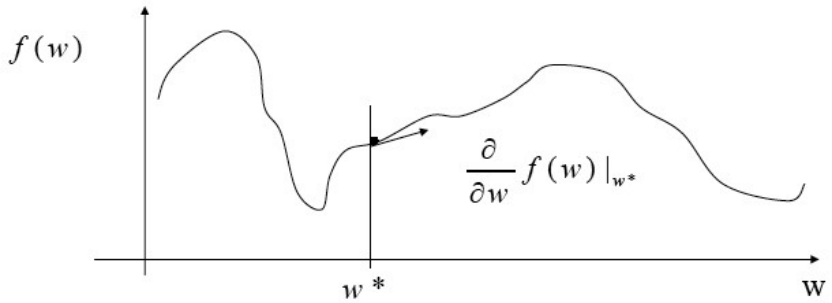


Rysunek 4. Wynik przeszukiwania lokalnego „przez wspinanie" dla stanu początkowego z Rysunek 3.

1.3.2. Przeszukiwanie lokalne w dziedzinie ciągłej

Przeszukiwanie „z dodatnim gradientem" (ang. *„gradient ascent search"*) jest odpowiednikiem „przeszukiwania przez wspinanie" w dziedzinie ciągłych wartości (w ogólności dla zmiennej wektorowej) parametrów **funkcji celu,** $y = f(x \mid \boldsymbol{w})$ (Rysunek 5). Jego zasada polega na iteracyjnej modyfikacji wektora parametrów $\boldsymbol{w}$ według następującej reguły:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \alpha \frac{\partial}{\partial \boldsymbol{w}} f(\boldsymbol{w})|_{w_i}$$



Rysunek 5. Ilustracja przeszukiwania wzdłuż dodatniego gradientu funkcji celu.

Przeszukiwanie „z ujemnym gradientem" (ang. *gradient descent search*) to dualny problem poszukiwania lokalnego minimum funkcji kosztu (błędu, straty):

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha \frac{\partial}{\partial \boldsymbol{w}} f(\boldsymbol{w})|_{w_i}$$

1.4. Symulowane wyżarzanie

Zasygnalizujemy tu inną grupę strategii rozwiązywania problemów, które zawierają niedeterministyczne (stochastyczne) elementy. Jedną z nich jest strategia „symulowanego wyżarzania". Można ją podsumować jako próbę poprawienia wad lokalnego przeszukiwania poprzez wysoce prawdopodobne wydostanie się z lokalnego optimum przez początkowo częste wykonywanie przypadkowych „złych" akcji, stopniowe zmniejszane wraz z obniżaniem się parametru „temperatury".

Strategia „symulowanego wyżarzania” realizuje niedeterministyczne przejścia pomiędzy stanami (Tabela 5). 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* (globalnej „temperatury”). Wartość ta stale maleje, co czyni takie przejścia coraz mniej prawdopodobnymi wraz z upływem czasu przeszukiwania. Można pokazać, że jeśli *T* zmniejsza się wystarczająco wolno to „symulowane wyżarzanie” znajduje globalne optimum z prawdopodobieństwem bliskim 1. W algorytmie zachowano tradycyjne określenie „Energia” dla oceny stanu.

Tabela 5. Algorytm „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
        }
}
```

1.5. Algorytmy genetyczne

1.5.1. Przeszukiwanie wiązki

Przeszukiwanie wiązki (ang. „*beam search*”) polega na jednoczesnym rozwijaniu pewnej liczby ścieżek. Kierujemy się tu zasadą:

- jednoczesny wybór *k* najlepszych stanów-następników zamiast jednego;
- w każdej iteracji rozwijanych jest *k* „najlepszych” stanów następników aktualnego skraju drzewa decyzyjnego.

Z uwagi na problem, że często wszystkie *k* stanów prowadzi do tego samego lokalnego optimum, pojawia się pomysł losowego wyboru następników. Stąd następująca idea:

- należy 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 większym prawdopodobieństwem mają dzieci, czasami zdarzają się przypadkowe mutacje.

1.5.2. Algorytm genetyczny

Algorytm genetyczny (Tabela 6) stanowi połączenie stochastycznego lokalnego przeszukiwania wiązki i metody generowania następników z połączenia par stanów:

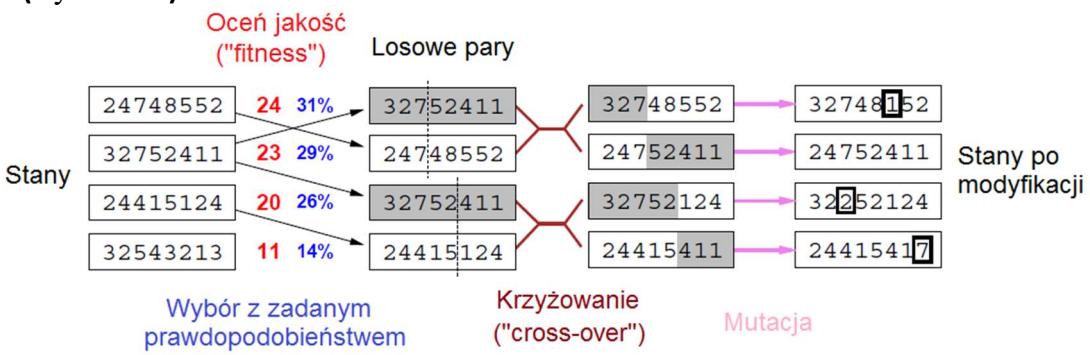
1. Pojedyncze rozwiązanie (stan) jest postaci sekwencji „genów”.
2. Wybór następników ma charakter losowy, z prawdopodobieństwem proporcjonalnym do oceny ich jakości, ang. *fittness*).

3. Stany wybrane do reprodukcji są parowane w sposób losowy, niektóre geny są mieszane a niektóre mutują.

Tabela 6. Funkcja „algorytm genetyczny.

funkcja GA (próg-oceny, p, r, m) **zwraca** stan
{ **P**← { 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)·p stanów do $\mathbf{P}_s$
$Pr(h_i)$= Ocena(h_i)/ SumaOcen (wszystkich stanów z $\mathbf{P}_s$)
2. Krzyżowanie: losowo wybierz (r·p/2) par stanów z $\mathbf{P}_s$.
Dla każdej pary (h_j, h_k), zastosuj operator krzyżowania.
Dodaj wyniki do $\mathbf{P}_s$
3. Mutacja: inwersja losowo wybranego bitu w (m· p) losowo wybranych stanach z $\mathbf{P}_s$
4. Podstaw: **P**← $\mathbf{P}_s$
5. **for** każdy stan h w **P do**: Oblicz-ocenę (h)
}
return stan z **P** o najwyższej ocenie
}

Przykład (Rysunek 6**)**



Rysunek 6. Ilustracja kroków algorytmu genetycznego.

2. Problemy spełniania ograniczeń

Dotychczas rozpatrywaliśmy ogólny problem przeszukiwania stanów, w którym z punktu widzenia stan stanowił zawsze „czarną skrzynką". Informacja o stanie miała jedynie „zewnętrzny" charakter i zawierała: związki stanu poprzez akcje z innymi stanami, wartość składowej heurystycznej dla funkcji oceny stanu i „nieznane" strategii własności stanu badane w warunku zatrzymania, zależnym od problemu. Teraz zajmiemy się specyficzną kategorią problemów, w których strategia przeszukiwania posiada pewną informację o „wewnętrznej" strukturze lub własnościach stanów.

2.1. Problem CSP

W przypadku specyficznej kategorii problemów przeszukiwania tworzonej przez „dyskretne problemy spełniania ograniczeń" (ang. *constraint satisfaction problem*, CSP) zakłada się następującą strukturę i informację o stanie:

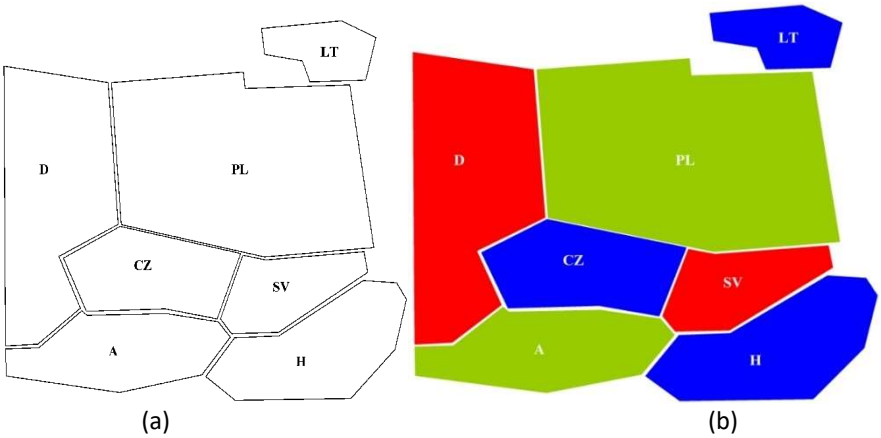
- stan jest definiowany przez zmienne $\{X_i, | i=1,2,..,n\}$, których wartości należą do odpowiednich dziedzin $\{D_i\}$;
- warunek zatrzymania to zbiór ograniczeń określający dopuszczalne kombinacje wartości dla zmiennych;
- rozwiązaniem jest dowolny stan spełniający te ograniczenia;
- rozwiązanie może być generowane przyrostowo, tzn. nie wszystkie zmienne muszą mieć od razu przydzielone wartości – w takiej sytuacji mówimy o stanie częściowym w odróżnieniu od stanu zupełnego, gdy wszystkie zmienne mają przydzielone wartości.

Dla zmiennych o skończonych dziedzinach możemy do rozwiązania problemu CSP zastosować strategie **przeszukiwania z ograniczeniami**.

2.1.1. Przykłady problemów spełniania ograniczeń

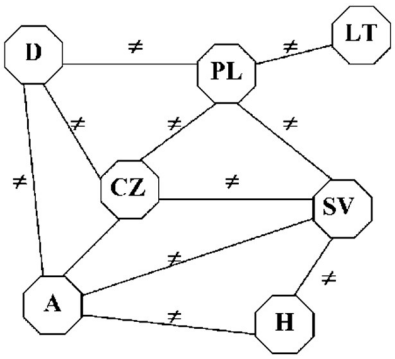
Przykład 1.

Problem kolorowania mapy (np. obejmującej kraje Europy Środkowo-Wschodniej) przy ograniczonej liczbie dostępnych kolorów (Rysunek 7). Wprowadzamy symbole zmiennych dla oznaczenia 7 krajów: *D, PL, CZ, SV, LT, A, H.* Załóżmy, że dysponujemy jedynie trzema kolorami, tzn. wszystkie dziedziny zmiennych są takie same i wynoszą: D_i = {*r*, *g*, *b*}. Ograniczenia polegają na tym, że przylegające obszary muszą być pomalowane różnymi kolorami, co zapiszemy w postaci, np. (D ≠ PL) lub (D, PL) ∈ {(*r, g*), (*r, b*), (*g, r*), (*g, b*), (*b, r*), (*b, g*)}. Poszukiwane rozwiązania są zupełnymi (wszystkim zmiennym nadano wartości) i spójnymi (niesprzecznymi z ograniczeniami) podstawieniami wartości pod zmienne. Np.: {*D=r, PL=g, CZ=b, SV=r, LT=b, A=g, H=b* }.



Rysunek 7. Przykład problemu spełniania ograniczeń – kolorowanie mapy: (a) zmienne - kraje, (b) zmienne z przydzielonymi wartościami – kolorami.

W tym przykładzie mamy do czynienia z ograniczeniami dwuargumentowymi (binarnymi), tzn. każde ograniczenie stanowi związek pomiędzy dwiema zmiennymi. Przedstawiamy je w postaci **grafu ograniczeń**, w którym zmienne są węzłami a łuki są relacjami ograniczeń (etykieta łuku wyznacza rodzaj ograniczenia) (Rysunek 8).



Rysunek 8. Przykład grafu ograniczeń dla problemu kolorowania mapy z Rysunek 7.

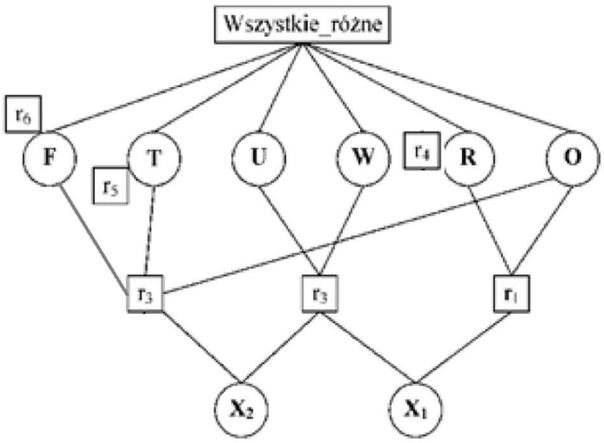
Ograniczenia mogą być zarówno jedno- i dwuargumentowe, jak i wyższego rzędu. Jednoargumentowe ograniczenia dotyczą pojedynczej zmiennej (np. PL ≠ g). Binarne ograniczenia dotyczą par zmiennych (np. D ≠ PL). Ograniczenia wyższego rzędu dotyczą 3 lub większej liczby zmiennych (np. ograniczenia kolumnowe w krypto-arytmetyce).

Przykład.
Krypto-arytmetyka jako problem CSP. Dana jest zaszyfrowana operacja matematyczna:

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

Należy znaleźć cyfry reprezentowane literami. Definiujemy zbiór zmiennych: *F, T, U, W, R, O.* Dziedziny wartości zmiennych wywodzą się ze zbioru D = {0,1,2,3,4,5,6,7,8,9}. Jednak dla niektórych zmiennych można zdefiniować ograniczenia jedno-argumentowe zmniejszające ich dziedzinę wartości (np. Dziedzina(*F*) = 1), Dziedzina(*R*) to cyfry parzyste bez zera).
Dla budowy ograniczeń potrzebne są też zmienne tymczasowe (wartości przeniesień), oprócz samych zmiennych – są to X_1 , X_2 . Graf ograniczeń (Rysunek 9):

Wszystkie_różne(*F, T, U, W, R, O*),
*r*1*: O + O = R +* 10 · X_1 ;
*r*2*:* X_1 *+ W + W = U +* 10 · X_2 *;*
*r*3*:* X_2 *+ T + T = O +* 10 *F*
r4: Parzyste_niezero(R), r5: $T \neq 0$, r6: Dziedzina(*F*) = 1



Rysunek 9. Przykład grafu ograniczeń dla przykładowego problemu krypto-arytmetyki.

2.1.2. Poszukiwanie przyrostowe lub ze stanem kompletnym dla CSP

Wyróżnimy dwie zasadnicze strategie przeszukiwania stanów dla rozwiązania problemu CSP: poszukiwanie **przyrostowe** ze stanem częściowym i poszukiwanie **ze stanem kompletnym**.

- Pierwsza z nich jest typową strategią dla tego typu problemów. Stan niekompletny oznacza, że tylko część zmiennych ma w nim przypisane wartości. Przeszukiwanie przyrostowe wyznacza ścieżkę rozwiązania, na której po kolei następują przyporządkowania wartości do kolejnych zmiennych. W takiej sytuacji rozwiązanie pojawia się na głębokości *n*, gdzie *n* jest liczbą wszystkich zmiennych. Typowa strategia przeszukiwania przyrostowego łączy niepoinformowane przeszukiwanie w głąb, ze sprawdzaniem ograniczeń i z możliwością wykonania nawrotu, czyli cofnięcia się na wyższy poziom drzewa przeszukiwania.
  Złożoność tego przeszukiwania możemy szacować w następujący sposób: liczba następników węzła w drzewie zależy od głębi położenia *l* i liczby możliwych wartości zmiennych *d*:
  $b_l = (n - l)\, d$ na głębokości *l*, a więc będzie $n!\, d^n$ liści.

- Ponieważ sekwencja akcji prowadząca do stanu kompletnego, czyli do rozwiązania, jest nieistotna, więc alternatywnie do przeszukiwania przyrostowego można stosować przeszukiwanie ze stanem kompletnym. Należy zainicjalizować problem w dowolnym stanie kompletnym, niespełniającym ograniczeń, zdefiniować akcje wymiany wartości nadanych uprzednio zmiennym i tak długo zmieniać stan, jak długo nie spełnia on zadanych ograniczeń. Do tego celu nadają się ogólne strategie przeszukiwania poinformowanego (np. strategia zachłanna z heurystyką), w których funkcja oceny stanu wyraża liczbę niespełnianych ograniczeń a stan końcowy spełnia wszystkie zadane ograniczenia.

2.2. Poszukiwanie przyrostowe dla CSP

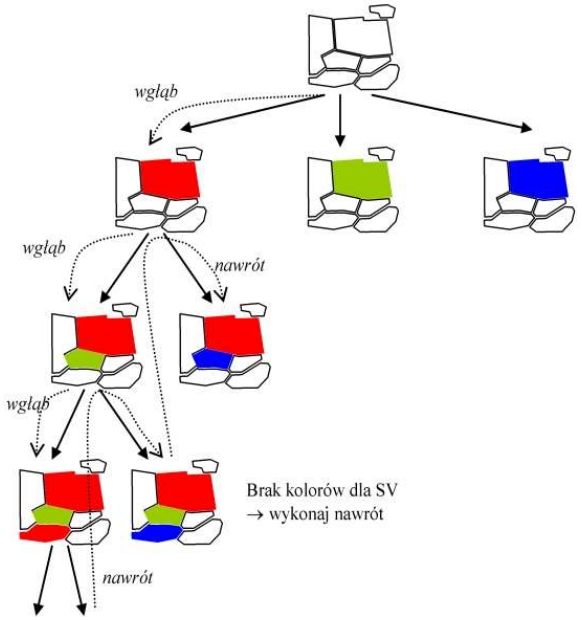
Przeszukiwanie przyrostowe stanowi standardowy sposób przeszukiwania przestrzeni problemu CSP. Węzły drzewa reprezentują zwykle niekompletne stany, które wyznaczone są przez wartości dotychczas przypisane zmiennym. Węzeł początkowy reprezentuje stan pusty - przyporządkowanie puste { }. Funkcja generowania następnika (akcja) polega wybraniu zmiennej w dotychczasowym stanie, która nie ma wartości i na nadaniu jej wartości w taki sposób, aby nie wywołać konfliktu (nie naruszyć ograniczeń) z dotychczas przyporządkowanym zmiennym. W sytuacji, gdy wybrany stan nie spełnia ograniczeń należy zrezygnować z niego i wykonać „nawrót” w drzewie przeszukiwania. Warunek zatrzymania jest znany z góry: stan jest zupełny i spełnia wszystkie ograniczenia.

2.2.1. Przeszukiwanie w głąb z nawrotami

Zauważmy, że przyporządkowania zmiennych są przemienne. Np. wykonamy najpierw podstawienie [*D = r*] a następnie [*PL = g*] i uzyskamy ten sam efekt co wykonanie najpierw [*PL = g*] a potem [*D = red*]. Dlatego wystarczy rozpatrzeć przyporządkowania tylko jednej zmiennej na każdym poziomie drzewa decyzji. Pozwala to znacznie zmniejszyć rozmiar drzewa przeszukiwania, gdyż nie musimy już rozpatrywać alternatyw wyboru kolejności zmiennych. Wtedy stopień rozgałęzienia drzewa wyniesie:

$b = d$ (niezależnie od poziomu l) i mamy najwyżej d^n liści.

Przeszukiwanie z nawrotami jest podstawową formą przyrostowego przeszukiwania dla CSP. Opiera się ono na ślepym przeszukiwaniu w głąb, w którym pojedyncza akcja odpowiada przyporządkowaniu wartości pojedynczej zmiennej i sprawdzaniu ograniczeń dla zmiennych danego stanu. W sytuacji, gdy nie są spełnione zadane ograniczenia (te, które można już sprawdzić na podstawie częściowo przypisanych wartości) jest wykonywany nawrót (aktualna ścieżka drzewa jest przycinana) przy niespełnianiu ograniczeń przez aktualny stan (Rysunek 10).



Rysunek 10. Ilustracja poszukiwania z nawrotami dla problemu CSP.

2.3. Usprawnienia poszukiwania z nawrotami

Usprawnienia podstawowej strategii mają na celu wczesne wykorzystanie wszelkiej informacji, która wynika z grafu ograniczeń dla problemu. Pozwoli to w praktyce zmniejszyć rozmiar drzewa poszukiwania i wcześnie wykrywać ścieżki nie prowadzące do rozwiązania. Omówimy kilka usprawnień, które mają ogólny charakter dla dyskretnych CSP, czyli mogą zawsze zostać zaimplementowane dla tego typu problemów.

Podczas poszukiwania możemy kierować się wynikami następującej analizy:
(A) Której **zmiennej** jako następnej powinno się przyporządkować wartość? Decyduje to o poziomach drzewa przeszukiwań. Mamy tu dwa kryteria, które są badane po kolei:

- (A1) najbardziej **ograniczona** zmienna, lub
- (A2) najbardziej **ograniczająca** zmienna;

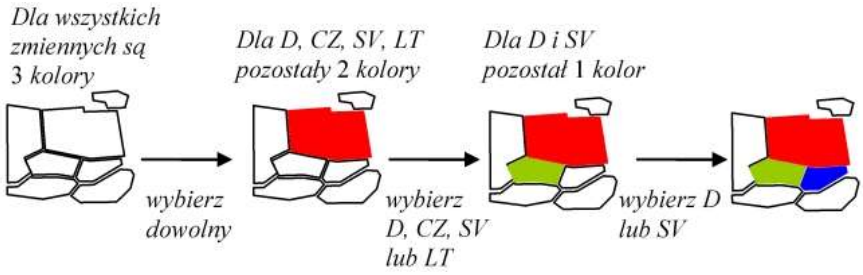
(B) W jakiej kolejności powinno się nadawać **wartości**? Decyduje to o kolejności następników – od lewej do prawej. Wiemy, że przy strategii przeszukiwania w głąb w tej właśnie kolejności będą wizytowane następniki aktualnego węzła. Mamy tu jedno kryterium:

- (B1) najmniej ograniczająca wartość

(C) Czy na podstawie **częściowego stanu** już możemy zdecydować o porażce aktualnej ścieżki?
- (C1) sprawdzanie wprzód,
- (C2) spójność łuków.

2.3.1. A1) Najbardziej ograniczona zmienna

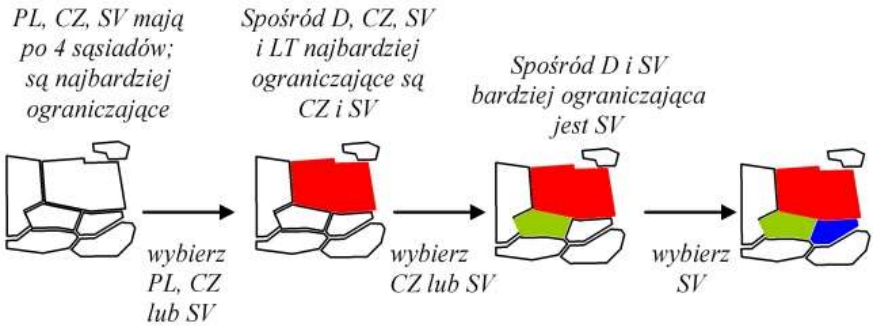
Należy wybrać zmienną o najmniejszej liczbie możliwych pozostałych jeszcze wartości (Rysunek 11), co minimalizuje liczbę następników aktualnego węzła w drzewie przeszukiwania.



Rysunek 11. Przykład korzystania z kryterium A1) „najbardziej ograniczona zmienna".

2.3.2. A2) Najbardziej ograniczająca zmienna

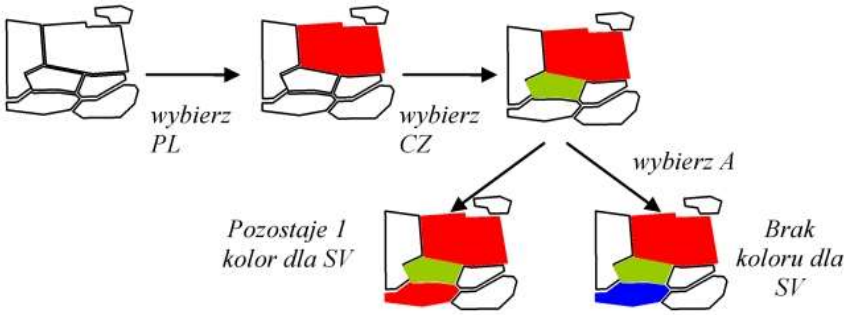
Jest to dodatkowe kryterium w stosunku do A1), które rozstrzyga o wyborze następnej zmiennej w sytuacji gdy dwie lub więcej zmiennych są równo ograniczone. Spośród takich najbardziej ograniczonych zmiennych należy wybrać zmienną, która narzuca najwięcej ograniczeń na pozostałe zmienne (Rysunek 12).



Rysunek 12. Przykład korzystania z kryterium A2) „najbardziej ograniczająca zmienna".

2.3.3. B1) Najmniej ograniczająca wartość

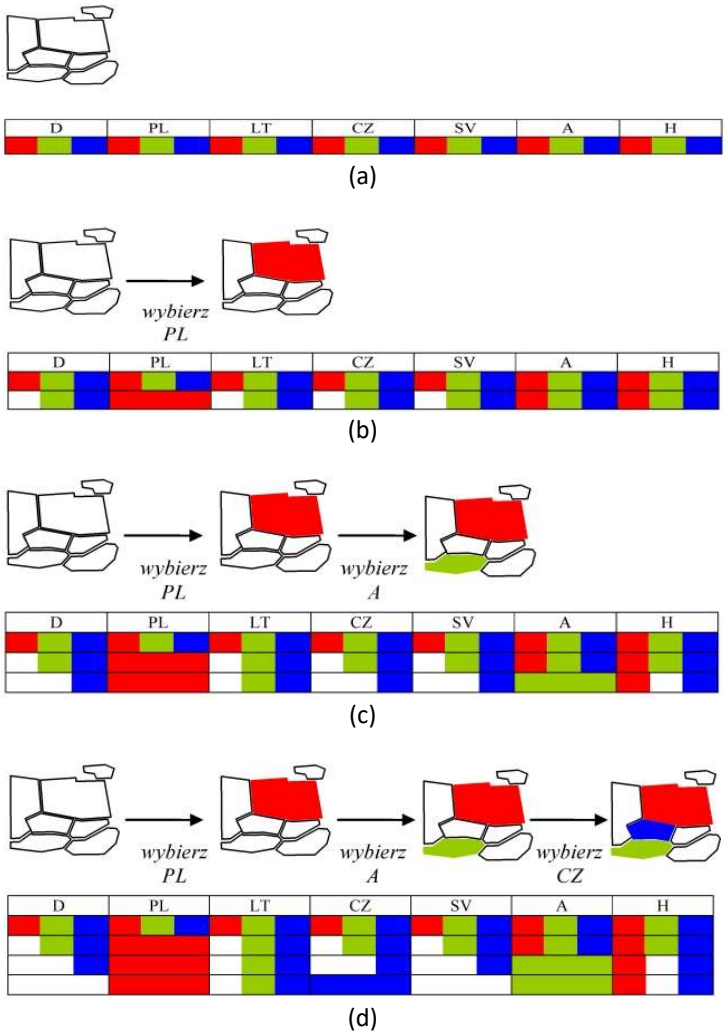
Po wybraniu zmiennej uporządkuj jej wartości zaczynając od wartości najmniej ograniczającej, czyli takiej, która wyklucza najmniej wartości pozostałych zmiennych (Rysunek 13).



Rysunek 13. Przykład korzystania z kryterium B1) „najmniej ograniczająca wartość".

2.3.4. C1) Sprawdzanie w przód

To kryterium wymaga wprowadzeniu dodatkowej pamięci - należy w ten sposób śledzić pozostałe, jeszcze legalne wartości dla pozostałych nieustalonych zmiennych. Pozwoli to przerwać poszukiwania na danej ścieżce wtedy, gdy jakaś zmienna nie ma już żadnych legalnych wartości.
Sprawdzanie w przód dzięki istniejącym ograniczeniom propaguje informację o dopuszczalnych wartościach od już związanych wartością zmiennych do jeszcze niezwiązanych zmiennych (Rysunek 14). Jednak nie umożliwia to wcześniejszego wykrycia porażki ścieżki dopóki nie zabraknie wartości dla pewnej zmiennej. W przykładzie na Rysunek 14(c) *D* i *CZ* mają jeszcze wartości niebieskie, ale z warunków zadania (ograniczenia) wiemy, że nie mogą jednocześnie mieć tej samej wartości. Pomocna będzie analiza spójności ograniczeń dla całego stanu.

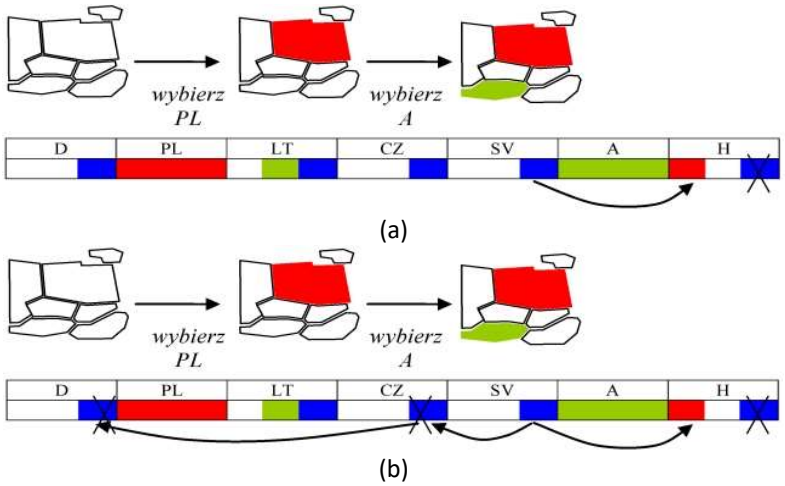


Rysunek 14. Przykład korzystania z kryterium C1) „sprawdzanie wprzód". (a) w stanie początkowym wszystkie wartości są legalne dla każdej zmiennej; (b) przypisanie (PL=r) zmniejsza legalne wartości dla D,CZ,SV i LT; (c) przypisanie (A=g) sprawia, że D, CZ, SV i H tracą kolor g; sprawdzanie w przód jeszcze nie wykryje, że już w tym stanie D i CZ są w konflikcie; (d) przypisanie (CZ=b) kasuje już wszelkie możliwe wartości D i SV, więc należy wykonać nawrót.

2.3.5. C2) Spójność łuków (ograniczeń)

W tym kryterium sprawdzamy czy dla każdego łuku w grafie ograniczeń istnieje jeszcze przynajmniej jedna kombinacja legalnych wartości jego zmiennych, spełniająca jego ograniczenie. Powiemy, że łuk *X* →*Y* jest spójny wtedy i tylko wtedy gdy dla każdej wartości *x* nadanej *X* istnieje jakaś dopuszczalna wartość *y* dla *Y.* Z kolei, jeśli dla wartości *x* lub *y* nie ma przynajmniej jednej wartości

drugiej zmiennej, to ta wartość przestaje być legalna (Rysunek 15(a)). Jeżeli *X* straci jakąś wartość, sąsiedzi *X* (w sensie istnienia między nimi ograniczeń) muszą być sprawdzeni ponownie. Mamy więc do czynienia z propagacją badania spójności łuków dla legalnych wartości w danym stanie. W naszym przykładzie to, że zmienna *H* straci wartość *b*, nie jest jeszcze krytyczne. Ale *CZ* straci *b*, i łuk między *SV* a *CZ* przestaje być spójny (Rysunek 15(b)). Teraz już po dwóch akcjach wykryjemy, że w tym stanie *SV* i CZ są już w konflikcie (podobnie jest z parą *CZ* i *D*).



Rysunek 15. Przykład korzystania z kryterium C2) „spójność łuków": (a) Po sprawdzeniu spójności łuku (SV, H) zmienna H traci wartość b. (b) Łuki pomiędzy SV i CZ oraz CZ i D przestają być spójne.

2.4. Przeszukiwanie ze stanem zupełnym dla CSP

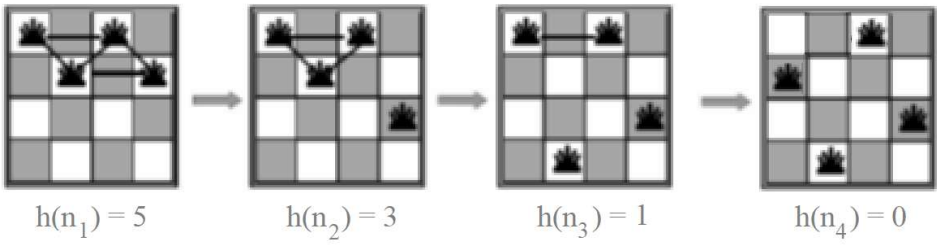
Pamiętamy, że strategie przeszukiwania stanów mające na celu znalezienie rozwiązania, takie jak „przez wspinanie" czy „symulowane wyżarzanie", operują na „kompletnych" stanach i ich ocenach. W przypadku problemu CSP „stan zupełny" oznacza, że wszystkie zmienne mają przypisane wartości a ocena stanu wyraża koszt niespełniania ograniczeń w danym stanie. Innymi słowa, określa koszt ścieżki od danego stanu do stanu docelowego. W związku z tym obok strategii lokalnego przeszukiwania można zastosować też dla rozwiązania problemu CSP strategię „zachłanną z heurystyką", w której „heurystyka" wyraża koszt niespełniania ograniczeń.

Podsumowując, strategia przeszukiwania ze stanem zupełnym dla CSP określona jest następująco:

1. Dopuszczamy stany kompletne z konfliktami wartości zmiennych (nie spełniające ograniczeń).
2. Operatory przeszukiwania (akcje) dokonują zmian wartości zmiennych, czyli przechodzimy z jednego stanu kompletnego w inny stan kompletny.
3. Wybór zmiennych i ich wartości wynika ze sprawdzenia niespełniania ograniczeń i konieczności minimalizacji naruszeń ograniczeń.

Przykład

Heurystyka wyrażająca liczbę konfliktów w problemie 4-królowych stanowi o koszcie danego stanu w problemie CSP. Liczba stanów wynosi: $4^4 = 256$. Wyrażając ten problem jako problem CSP mamy 4 zmienne, gdzie każdej z nich mogą być nadane 4 wartości. Pojedyncza akcja to przesunięcie królowej w kolumnie. Ocena stanu równa się liczbie par wzajemnych ataków (konfliktów). Warunkiem zatrzymania procesu przeszukiwania powinien być brak wzajemnych ataków (konfliktów). W przypadku przeszukiwania „zachłannego z heurystyką" operującego na globalnym skraju drzewa przeszukiwania rozwiązanie (jeśli istnieje) zostanie znalezione (Rysunek 16).



Rysunek 16. Przykład ścieżki w drzewie decyzji minimalizującej liczbę konfliktów w problemie „4 królowych”.

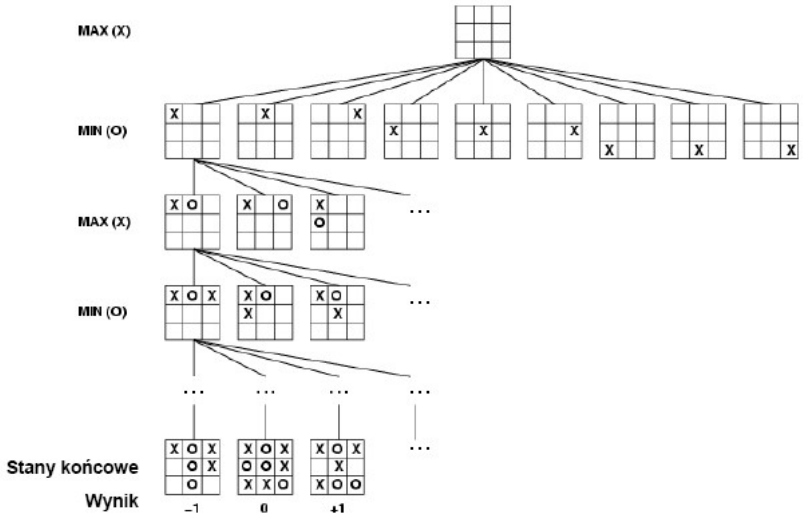
3. Gry dwuosobowe

W nietrywialnych grach nie da się rozwiązać zagadnienia wyboru następnego ruchu ekstensywną metodą rozważenia wszystkich możliwych sekwencji ruchów własnych i przeciwnika. Oznacza to, że wybór ruchu w grach dwuosobowych, tak jak rozwiązywanie innych złożonych zadań, wymaga odpowiednich strategii przeszukiwania przestrzeni stanów.
Ponieważ przeciwnik jest „nieprzewidywalny" należy określić własny ruch przewidując „najgorszą dla nas" decyzję przeciwnika. Prowadzi to do strategii określanej jako „Mini-max" lub „Mini-max z cięciami alfa-beta". W sytuacji, gdy przejrzenie wszystkich możliwych sekwencji ruchów prowadzących do celu nie jest w praktyce możliwe, należy aproksymować koszty rozwiązań stosując strategię „obcięty Mini-max".

3.1. Drzewo gry typu „Mini-max"

Ograniczmy nasze rozważania do gier dwuosobowych o przeciwstawnych celach, gdy obaj gracze wykonują ruchy na przemian. Zakładamy problem decyzyjny o postaci przeszukiwania specyficznego drzewa typu „Mini-max" (Rysunek 17):

- Występują 2 rodzaje węzłów: Min i Max,
  - węzeł Min reprezentuje stan gry, w którym ruch należy do przeciwnika,
  - węzeł Max odpowiada decyzji naszego gracza.
- Liście drzewa reprezentują stany końcowe i zawierają liczbową ocenę wyniku gry.
- Ocena stanu gry propagowana jest „od dołu na górę" (od liści do korzenia drzewa), przy czym ocenę węzła rodzica typu „Max" ustawiamy na wartość maksymalną spośród jego węzłów potomnych (następników), ocena węzła rodzica typu „Min" jest minimalną spośród jego następników;
- Przewiduje się wykonywanie akcji przez obu graczy:
  - „nasz" gracz wybiera akcję odpowiadającą przejściu do najlepszego spośród jego następców;
  - „przeciwnik" wykonuje ruch odpowiadający przejściu do najgorszego następnika.

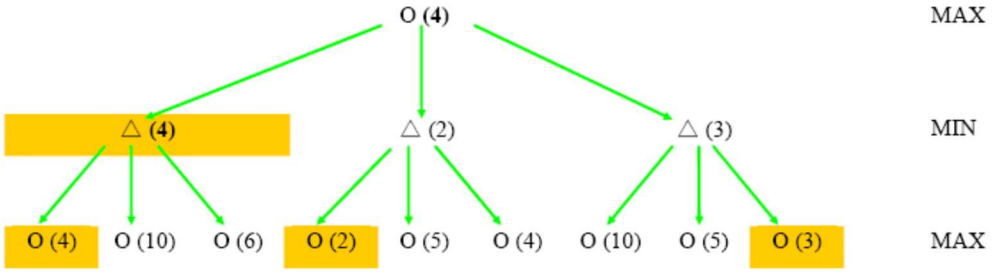


Rysunek 17. Ilustracja drzewa Mini-max dla gry w „kółko i krzyżyk".

Ocena węzła odpowiada możliwemu do osiągnięcia wynikowi gry, reprezentowanemu przez liść drzewa przeszukiwania (rys. 11.2). Np. możliwy wynik w grze „kółko i krzyżyk" to:

+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.



Rysunek 18. Ilustracja wyznaczenia wyniku gry w liściach drzewa i jego propagacji w górę drzewa do korzenia.

3.2. Strategia Mini-max

„Mini-max" zakłada **idealną** rozgrywkę dla **deterministycznych** 2-osobowych gier naprzemiennych. Strategia polega na wybieraniu przez „naszego gracza" ruchu do pozycji z najwyższą wartością i na przyjęciu założenia, że przeciwnik wybiera dla nas najgorszy ruch. Tym samym strategia „Mini-max" realizuje cel: *uzyskać najlepszy możliwy wynik w grze z najlepszym przeciwnikiem*. Nasza implementacja strategii „Mini-max" składa się z funkcji **MiniMaks()** (Tabela 7) i jej podfunkcji **MaksOcena()** (Tabela 8) i **MinOcena()** (Tabela 9).

*Tabela 7. Funkcja **Mini-Maks.***

```
function MiniMaks(stan) : returns akcja
{
        v := MaksOcena(stan);
        wybierz akcja = (stan, stan_N),
                gdzie: stan_N ∈ Następne(stan), Wynik(stan_N) == v ;
        return akcja;
}
```

*Tabela 8. Podfunkcja **MaksOcena**.*

```
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;
}
```

*Tabela 9. Podfunkcja **MinOcena**.*

```
function MinOcena(stan) : returns najgorszy_możliwy_wynik
{
        if (TerminalTest(stan)) then return Wynik(stan);
        v := ∞ ;
        for (s ∈ Następne(stan)) do v := min(v, MaksOcena(s));
```

```
        return v;
}
```

Własności 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.

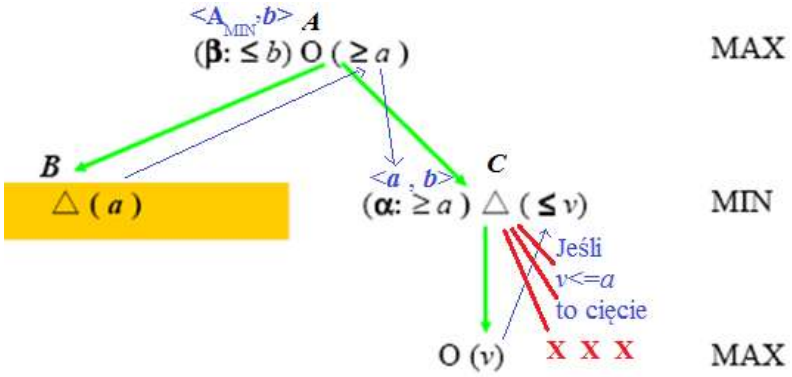
Dla szachów szacujemy: b≈35, m≈100, dla "*rozsądnych*" rozgrywek, tzn. wtedy gdy spotykają się gracze o podobnych wysokich umiejętnościach. W praktyce, przy tak dużych wartościach parametrów, dokładne rozwiązanie problemu strategią Mini-max jest niemożliwe. Wymagane jest usprawnienie tej strategii. Taką formą jest strategia przeszukiwania drzewa Min-Max zwana „cięciami alfa-beta".

3.3. Cięcia alfa-beta

3.3.1. Zasada przycinania drzewa

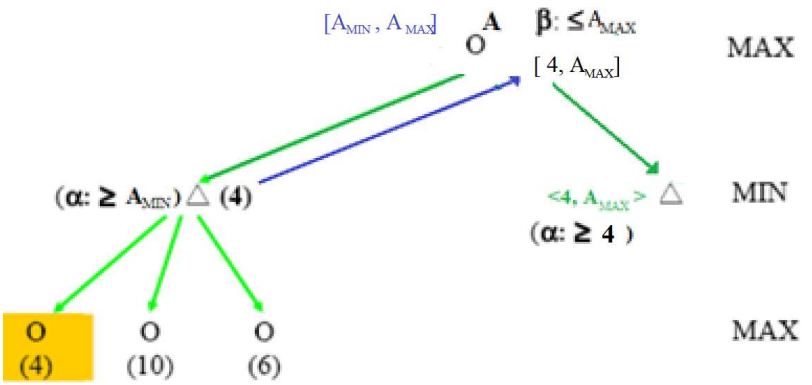
Niech *a* jest oceną najlepszego wyniku w dotychczas przeanalizowanym poddrzewie *B* dla korzenia *A* typu MAX. Jeśli podczas analizy kolejnych gałęzi dla węzła *C* jego ocena *v* jest gorsza niż *a*, to bez szkody dla optymalnego rozwiązania zrezygnujemy z *v*, czyli przycinamy gałąź *C* (cięcie „alfa").

Podobnie zrobimy dla gałęzi węzła *B* typu MIN - gdy stwierdzimy, że jakaś gałąź *B* będzie miała ocenę *w* i zachodzi, $w \geq b$, to możemy przyciąć pozostałe gałęzie węzła *B* (cięcie „beta") (Rysunek 19).

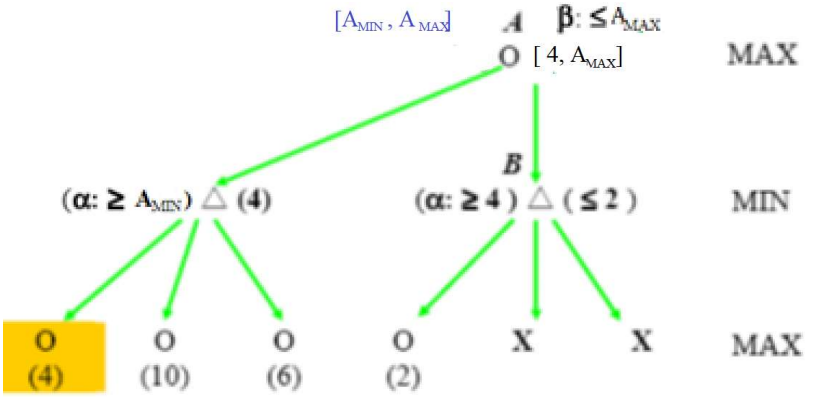


Rysunek 19. Ilustracja „cięcia alfa".

Przykład „cięcia alfa".
Węzeł A jest korzeniem drzewa, więc jest typu MAX (Rysunek 20(a)). Jego pierwsza gałąź dostarcza ocenę 4. Załóżmy, że początkowe ograniczenie dolne było niższe, tzn. ($4 > A_{MIN}$). Odtąd ciekawe będą jedynie te gałęzie węzła A, które dają wyższą ocenę niż 4. Dla nich (na poziomie MIN) mamy nowe ograniczenie: $\alpha \geq 4$. Dla węzła *B* typu MIN pierwsza przeanalizowana gałąź dostarcza ocenę 2 (Rysunek 20(b)). Jest to nowe ograniczenie górne dla węzła B (i jego poddrzewa). W kontekście już posiadanej oceny 4 węzła nadrzędnego *A,* warunek ((α=4) > (β=2)) „przycina" pozostałe gałęzie węzła *B*. Ocena węzła B nie będzie wyższa niż 2, a węzeł A „zainteresowany" jest jedynie ocenami co najmniej równymi 4.



(a)

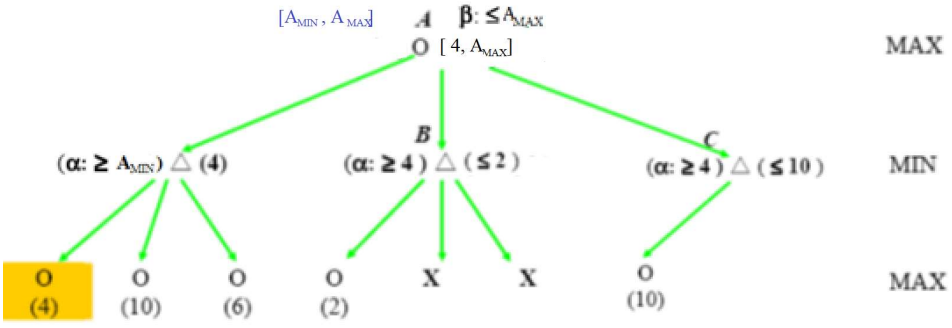


(b)

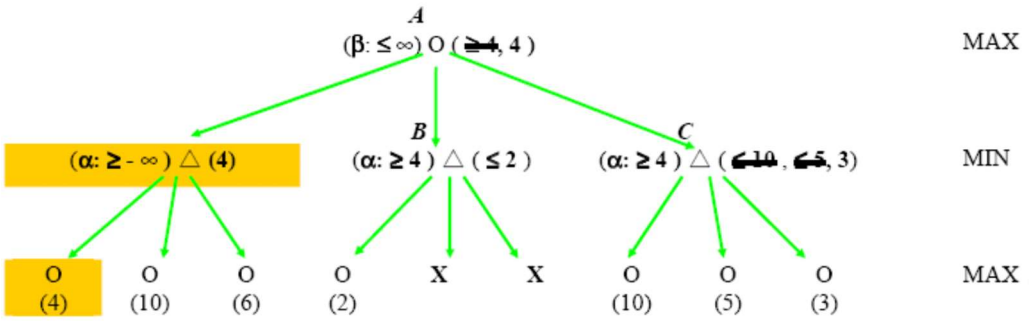
Rysunek 20. Przykład „cięcia alfa”: (a) Pierwsza gałąź węzła A (typu MAX) wyznacza nowe ograniczenie dolne dla A (alfa = 4); (b) pierwsza gałąź węzła B (typu MIN) daje ograniczenie górne dla B (beta=2).

Przykład „cięcia alfa” (c.d.)

Przeszukujemy kolejną gałąź dla korzenia A, czyli poddrzewo o korzeniu C typu MIN (Rysunek 21). Przedział „dozwolonych” wartości węzła C wynosi [(α=4), (β=A_{MAX})]. Niech pierwsza gałąź węzła *C* dostarcza ocenę 10. Jest to nowe ograniczenie górne β. Nie możemy wykonać „cięcia alfa” pozostałych gałęzi *C* (tak jak to ma miejsce dla węzła B), gdyż nowy przedział „dozwolonych” wartości nadal jest niepusty [4, 10]. Przeszukanie drugiej gałęzi węzła *C* daje nowe, niższe ograniczenie górne 5, czyli nowy przedział „dozwolonych” wartości wynosi [4, 5]. Warunek ($\alpha \leq \beta$) nadal jest spełniony w węźle *C*. Niech kolejna gałąź węzła C daje wynik 3, co zmniejsza ograniczenie górne poniżej ograniczenia dolnego ($\alpha > \beta$). Można już wykonać cięcie pozostałych gałęzi węzła C.



(a)



(b)

Rysunek 21. W węźle C typu MIN ustawiane jest nowe ograniczenie górne dla gałęzi tego węzła: (a) warunek ($\alpha \leq \beta$) jest spełniany – przeszukiwanie dla poddrzewa C jest kontynuowane, (b) zachodzi ($\alpha > \beta$) dla węzła C i przycinane są kolejne gałęzie tego węzła.

Podsumowując powyższy przykład możemy podać ogólne zasady cięć alfa-beta.

- **Cięcie alfa:** oceniając węzeł 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 (pochodzące z nadrzędnego węzła MAX).
- **Cięcie beta:** oceniając węzeł MAX przez maksymalizację ocen węzłów potomnych typu MIN możemy zakończyć wyznaczanie ocen węzłów potomnych natychmiast po stwierdzeniu, że ocena węzła MAX nie będzie niższa niż jej ograniczenie górne β (pochodzące z nadrzędnego węzła MIN).

3.3.2. 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 w Tabela 10. Parametry: α – ograniczenie dolne = największa wartość węzła Max dla ścieżki rozwiązania; β – ograniczenie górne = najmniejsza wartość węzła Min dla ścieżki rozwiązania.

Tabela 10. Implementacja przeszukiwania „Mini-Max z cięciami alfa-beta”.

```
function CięciaAlfaBeta(stan, AMIN, AMAX) : returns akcja {
        v := MaksOcena(stan, AMIN, AMAX);
        wybierz akcja = (stan, stanN), gdzie: stanN ∈Następne(stan), Wynik(stanN) == 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;
```

```
}
```

```
function MinOcena(stan, α, β) : returns najgorszy_wynik
{
        if (TerminalTest(stan)) then return Wynik(stan);
        v := β;
        for (s ∈ Następne(stan)) do
        begin  v := min(v, MaksOcena(s, α, β));
                        if v ≤ α then return v; // Cięcie alfa
                        β := min(β, v);
        end;
        return v;
}
```

3.3.3. Własności strategii „cięć alfa-beta”

Przycinanie nie ma wpływu na końcowy rezultat przeszukiwania. Osiągamy **ten sam wynik**, który dałoby zastosowanie podstawowej strategii Mini-Max. Sprzyjające uporządkowanie ruchów (następników węzła) poprawia efektywność przycinania. Przy "perfekcyjnym uporządkowaniu” następników złożoność czasowa algorytmu wynosi $O(b^{m/2})$. Dzięki temu możliwe staje się rozwiązanie za pomocą „Mini-max z cięciami α-β” problemów o dwa razy większej głębokości drzewa przeszukiwania niż w przypadku klasycznej strategii Mini-Max.

3.4. Heurystyczna ocena – „obcięty Mini-Max”

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. W efekcie mamy czas na przeglądanie do 10^8 węzłów dla wykonania jednego ruchu.

3.4.1. Zasada modyfikacji „Mini-Max”-u w praktyce

Wykonujemy „test odcięcia” dla ścieżki (w postaci funkcji C*utoff*). Test polega na ograniczeniu głębokości drzewa do wartości zadanej parametrem D. Wprowadzamy heurystykę oceny stanu gry (w postaci funkcji *Ocena* – podaje ona szacunkową wartość (niekońcowej) konfiguracji gry, reprezentowanej przez dany węzeł drzewa Mini-max.

Przykład oceny stanu gry.

Dla warcabów możemy zastosować 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), $f_1(s)$ = (liczba białych damek) – (liczba czarnych damek).
W ocenie można także uwzględnić wagi poszczególnych pozycji na planszy i wagi dla wzajemnych położeń pionków obu graczy.

3.4.2. Implementacja obciętego „Mini-Max”

Implementacja strategii „obciętego Mini-Max” 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 aktualnego stanu.
2. Zamiast funkcji Wynik(), podającej *rzeczywistą* ocenę końcowego stanu gry dla pierwszego gracza, w sytuacji obcinania wywołujemy *funkcję oszacowania* stanu: Ocena().

Czy ta strategia dobrze działa w praktyce? Odpowiedzmy na to na przykładzie. 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 kiepskiego 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,
- przewiduje 12 ruchów ≈ program na komputerze *Deep Blue* lub arcymistrz szachowy.

Tabela 11. Implementacja strategii „obciętego Mini-Max"

```
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;
}
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;
}
```

4. Klasyczne planowanie działań

4.1. Agent realizujący cel

Znamy trzy sposoby wyboru akcji (działania) logicznego agenta realizującego cel: przez **wnioskowanie**, **przeszukiwanie** lub **planowanie**.

Pierwszy z tych sposobów (przez wnioskowanie) nie wymagał dodatkowych mechanizmów, wykraczających poza system logicznego wnioskowania. Wprawdzie w nieefektywny sposób, poprzez dużą liczbę reguł wyboru akcji zawartych w bazie wiedzy, ale możliwe było zrealizowanie takiego agenta przez system logicznego wnioskowania.

Następnie poznaliśmy uniwersalne metodyki przeszukiwania przestrzeni stanów, które można zastosować dla wyboru akcji agenta. Wymagamy wtedy jedynie, aby problem działania agenta został przedstawiony w postaci: przestrzeni stanów, możliwych akcji i ich kosztów, warunku celu i ewentualnych ocen heurystycznych dla stanów. W wyniku zastosowania strategii przeszukiwania przestrzeni stanów agent znajdzie sekwencję akcji wiodącą od stanu początkowego do stanu końcowego, w którym spełniony jest warunek celu.

W tym rozdziale przedstawimy metodykę realizacji celu za pomocą konstruowania planu działania (rozumianego jako sekwencja akcji w przestrzeni stanów). W szczególności sam proces tworzenia planu może być realizowany w postaci przeszukiwania przestrzeni (częściowych) planów.

4.1.1. Przeszukiwanie przestrzeni planów

Jak dotąd zakładaliśmy, że stan problemu reprezentowany jest zwykle prostą strukturą danych potrzebnych dla *generacji następnika, funkcji oceny* i sprawdzenia *warunku zatrzymania.* W istocie taki stan ma charakter „czarnej skrzynki" i nie daje agentowi żadnych wskazówek co do wyboru akcji. Dla problemów, w których występuje bardzo duża liczba potencjalnie możliwych akcji, powoduje to generowanie bardzo dużej liczby następników, z których większość nie jest istotna.

Przykład.

Zwykłe przeszukiwanie stanów zawiedzie z powodu dużego stopnia rozgałęzienia drzewa potrzebnego dla realnych, życiowych problemów. Dany jest prosty cel dla człowieka: „*kup karton mleka, bochen chleba i kwiaty oraz wróć z nimi do domu".* Ponieważ człowiek może wykonywać wiele czynności, z których większość nie jest związana z zadanym celem, próba zastosowania przeszukiwania doprowadzi do drzewa o wysokim współczynniku rozgałęzienia (Rysunek 22). Funkcja oceny, w tym składowa heurystyczna nie pozwala eliminować zbędnych akcji a jedynie wybierać stany.

Powyższy przykład wskazuje, że w wielu rzeczywistych problemach potrzebny jest silniejszy mechanizm niż przeszukiwanie, który wyznaczałby sekwencję akcji dla agenta prowadzącą do celu, korzystając z następujących założeń:

- stan przestaje być czarną skrzynką i posiada własności warunkujące stosowanie dla nich akcji;
- akcje wyznaczane są po to aby spełnić własności stanu docelowego,

Ten mechanizm nazywamy planowaniem działań.

Rysunek 22. Bardzo wysoki stopień rozgałęzienia drzewa przeszukiwania.

4.1.2. Planowanie działań

Planowanie ma na celu „usprawnienie" przeszukiwania poprzez „otwarcie" reprezentacji stanów, celów i akcji. Algorytmy planujące korzystają z formalnych języków dla opisu stanów, celów i akcji – zwykle jest to logika predykatów lub jej podzbiór. Stany, w tym stan końcowy (cel) są reprezentowane poprzez zbiory formuł, które są spełnialne.

- Akcje są charakteryzowane poprzez logiczne opisy warunków wykonania i efektów akcji. Pozwala to na określenie bezpośrednich związków pomiędzy stanami a akcjami i na pominięcie bezzasadnych akcji w danym stanie.
- Akcje są dodawane do planu w dowolnej kolejności wtedy, gdy są potrzebne, a nie w wyniku sztywnej sekwencji prowadzącej od stanu początkowego do końcowego (wykonując *oczywiste* i *ważne* decyzje w pierwszej kolejności, prowadzimy najpierw do ograniczenia przestrzeni rozwiązania a następnie stopniowo uszczegóławiamy plan).

Plan ma postać sekwencji operatorów (operator jest prototypem akcji wraz z warunkami i efektami jej wykonania) i powstaje w wyniku przeszukiwania przestrzeni planów. Możliwe jest:

a) przeszukiwanie z planem częściowym, które rozszerza plan początkowy (prosty, niepełny plan) do końcowego; lub (
b) przeszukiwanie z planem zupełnym, które modyfikuje pełny, ale błędny plan do końcowego.

W porównaniu do przeszukiwania przestrzeni stanów mamy tu znacznie mniejszą wariantowość i dowolność kolejności operatorów dla planów niż akcji dla stanów.

Problem jest często formułowany jako koniunkcja pod-problemów. Możemy wtedy wyróżnić podcele i wyznaczać osobne plany dla osiągnięcia kolejnych podcelów.

Tworzenie planu jest wywoływane przez funkcję (sterowanie) agenta planującego (Tabela 12).

4.2. Klasyczne planowanie

4.2.1. Planowanie w języku logiki

Omówimy tu planowanie wyrażone w rachunku sytuacyjnym logiki predykatów. Stan początkowy reprezentowany jest jako koniunkcja predykatów, zwykle działających na stałych.

Np. $W(\text{dom}) \wedge \neg \textit{Mieć}(\text{mleko}) \wedge \neg \textit{Mieć}(\text{chleb}) \ldots$

Cele (stany końcowe) reprezentowane są jako koniunkcje predykatów i mogą one zawierać zmienne.

Np. $\exists x\, At(x) \wedge \textit{Sprzedaje}(x, \text{mleko})$

Zmienne w formule celu są kwantyfikowane egzystencjalnie i jest to formuła zapytania.

Tabela 12. Funkcja prostego agenta planującego.

```
funkcja ProstyAgentPlanujący(percepcja); wynik: akcja;
{
  Dane statyczne (trwałe): KB – baza wiedzy (zawiera opisy akcji);
       p – plan, początkowo = BrakPlanu;
       t – licznik czasu, początkowo = 0;
  Lokalne zmienne: cel – aktualny cel; stan – aktualny stan;
  TELL(KB, UtwórzFormułęPercepcji(percepcja, t));
  stan := STAN(KB, t);
  if ((p = = BrakPlanu)
       then {
       cel := ASK(KB, UtwórzFormułęZzapytania(t));
        p := IdealnyPlaner(stan, cel, KB);
       }
  if ((p = = BrakPlanu) or (p = = ∅))
       then akcja := BrakAkcji;
       else {
                akcja := PIERWSZY(p);
                p := RESZTA(p);
                }
  TELL(KB, UtwórzFormułęAkcji(akcja, t);
   t := t+1;
  return akcja;
}
```

Operatory są formułami o postaci *aksjomatu następnika stanu* i są powiązane z akcjami (np. predykatem *Rezultat*). Np.

$$\forall a,s\ \textit{Mieć}(\text{mleko}, \textit{Rezultat}(a,s)) \Leftrightarrow [(a = \textit{Kupić}(\text{mleko}) \wedge \textit{W}(\text{Supermarket}, s) \vee (\textit{Mieć}(\text{mleko}, s) \wedge a \neq \textit{Stracić}(\text{mleko}))]$$

4.2.2. STRIPS

„Klasyczna" budowa planu odwołuje się do języka STRIPS (*Stanford Research Institute Problem Solver*) będącego ograniczoną formą rachunku sytuacyjnego. W tym podejściu stany niekońcowe są reprezentowane jako koniunkcje literałów bez zmiennych, czyli predykatów zawierających jedynie stałe. Stany końcowe (cele) są również koniunkcjami predykatów, ale obok stałych mogą też zawierać zmienne, dla których zakłada się kwantyfikację egzystencjalną. Operator składa się z 3 części (Tabela 13):

1. Warunek operatora (PRECOND) – koniunkcja pozytywnych predykatów atomowych warunkująca wykonalność operatora.
2. Akcja (ACTION) – jej wykonanie przez agenta przeprowadzi środowisko do nowej sytuacji.
3. Następnik – efekt (EFFECT) – efekt operatora opisany za pomocą koniunkcji predykatów atomowych (pozytywnych lub zanegowanych).

Operator przekształca sytuację opisaną w jej warunku w sytuację opisaną w jej następniku i podaje związaną akcję. Pełny opis stanu nie jest konieczny – dla poznania stanu w aktualnej sytuacji wystarczy śledzić zmiany zachodzące od sytuacji początkowej. Brak jest też jawnych indeksów sytuacji, gdyż ze struktury operatora jawnie wynika co jest sytuacją poprzednią a co następną, a poza tym operator ma charakter generyczny i nie zależy od indeksu sytuacji.

Przykład

Op(ACTION*: Pójść*(*tam*),
PRECOND*: W*(*tu*) ∧ *Droga*(*tu*, *tam*),
EFFECT: *W*(*tam*) ∧ ¬*W*(*tu*)
)

Tabela 13. Przykład operatora w języku STRIPS.

W(*tu*), Droga(*tu*, *tam*)
Pójść(tam)
W(*tam*), ¬W(*tu*)

Operator jest **stosowalny** w stanie *S*, jeżeli warunek operatora jest spełniony w tym stanie. Operator zawierający symbole zmiennych reprezentuje rodzinę akcji. Wykonany może być tylko w pełni wartościowany operator (po podstawieniu wartości za wszystkie zmienne). Procedura planująca musi zapewnić dostępność wartości dla zmiennych w danym stanie. W stanie powstałym dzięki zastosowaniu operatora spełnione są:

- wszystkie predykaty zawarte w następniku operatora,
- wszystkie predykaty zawarte w warunku operatora, które nie zostały zanegowane przez następnik operatora.

Stan docelowy jest reprezentowany w postaci operatora GOAL nie posiadającego części „następnika", a INIT jest operatorem dla stanu początkowego i z kolei nie posiada „warunku". Operatory GOAL i INIT nie posiadają zmiennych.

Przykład - plan w STRIPS.

Problem transportu przesyłek lotniczych (cargo) pomiędzy dwoma lotniskami może być następująco reprezentowany w STRIPS:

INIT(W(C_1,WAW) ∧ W(C_2, FRA) ∧ W(P_1, WAW) ∧ W(P_2, FRA) ∧ Cargo(C_1) ∧ Cargo(C_2)
∧ Samolot(P_1) ∧ Samolot(P_2) ∧ Lotnisko(WAW) ∧ Lotnisko(FRA))
GOAL(W(C_1,FRA) W(C_2, WAW))
ACTION(Załadunek(c, p, a),
PRECOND: W(c,a) ∧ W(p,a) ∧ Cargo(c) ∧ Samolot(p) ∧ Lotnisko(a)
EFFECT: ¬W(c,a) ∧ W(c,p))
ACTION(Rozładunek(c, p, a),
PRECOND: W(c,p) ∧ W(p,a) ∧ Cargo(c) ∧ Samolot(p) ∧ Lotnisko(a)
EFFECT: W(c,a) ∧ ¬W(c,p))
ACTION(Lot(p, od, do),
PRECOND: W(p,od) ∧ Samolot(p) ∧ Lotnisko(od) ∧ Lotnisko(do)
EFFECT: ¬W(p,od) ∧ W(p,do))

Rozwiązaniem podanego problemu może być następujący plan:
[Załadunek(C_1; P_1; WAW); Lot(P_1; WAW; FRA); Załadunek(C_2; P_2; FRA); Lot(P_2; FRA; WAW)] .

Jednak powyższy plan nie jest idealny. Wadą reprezentacji problemu w STRIPS jest to, że generowany może być plan, który zezwala na przelot samolotu z lotniska do tego samego lotniska docelowego.
Po pojawieniu się STRIPS w kolejnych latach okazało się, że jego siła wyrazu nie wystarczy dla opisu wielu praktycznych dziedzin. Dlatego też upowszechniła się odmiana tego języka nazwana „Action Description Language" (ADL) (Tabela 14).

Tabela 14. Porównanie języków STRIPS i ADL.

STRIPS	ADL
Tylko pozytywne literały w warunkach, np. $W(\text{dom}) \wedge Ma(\text{mleko})$	Pozytywne i zanegowane literały w warunkach: np. $\neg W(\text{dom}) \wedge \neg Ma(\text{mleko})$
Założenie „zamkniętego świata": brakujące literały są niespełnione (False).	Założenie „otwartego świata": spełnianie brakujących literałów nie jest znane.
$(P \wedge \neg Q)$ oznacza: *dołącz P i usuń Q*	$(P \wedge \neg Q)$ oznacza: *dołącz P i $\neg Q$, oraz usuń $\neg P$ i Q*
Tylko bazowe literały (predykatowe) w operatorze „cel". Np. $W(\text{P1,WAW}) \wedge W(\text{P2,WAW})$	Kwantyfikowane zmienne w operatorze „cel". Np. $\exists x\, At(\text{P1},x) \wedge At(\text{P2},x)$
Cele są koniunkcyjnej postaci. Np. $W(\text{dom}) \wedge Ma(\text{mleko})$	Cele są koniunkcyjne i alternatywne. Np. $W(\text{dom}) \wedge (Ma(\text{mleko}) \vee Ma(\text{banany}))$
Brak wbudowanej semantyki dla równości (=). Brak typów.	Wbudowany predykat równości (np. x=y). Zmienne są określonych typów. Np. $(x: \text{Miasto})$

Przykład.
W języku ADL akcję *Lot* z poprzedniego przykładu reprezentować możemy w następujący sposób:
ACTION(Lot(p:Samolot, od:Lotnisko, do:Lotnisko),
PRECOND: W(p,od) $\wedge$ (od $\neq$ do)
EFFECT: $\neg$W(p,od) $\wedge$ W(p,do))
Teraz w warunku operatora występuje dodatkowo wyrażenie (od $\neq$ do), które wyklucza planowanie wykonania lotu z dowolnego lotniska na to samo lotnisko. W języku STRIPS należy zdefiniować własny predykat dla wyrażenia tej własności. Możliwość natychmiastowego określenia typu zmiennej w ADL zastępuje definiowanie odrębnego predykatu w STRIPS, sprawdzającego obiekt podstawiany w miejsce zmiennej.

4.3. Przestrzeń planów częściowych

Problem wyznaczenia planu rozwiążemy poprzez przeszukiwanie przestrzeni możliwych planów częściowych dla zadanego problemu agenta. To przeszukiwanie będzie realizowane w postaci strategii specyficznej dla zagadnienia planowania działań. Jeszcze raz zwróćmy uwagę na zasadniczą różnicę dwóch zagadnień przeszukiwania:
1. dla rozwiązania problemu agenta wykonuje się przeszukiwanie przestrzeni stanów dla problemu,
2. dla znalezienia planu przeszukiwania będą prowadzone w przestrzeni planów.

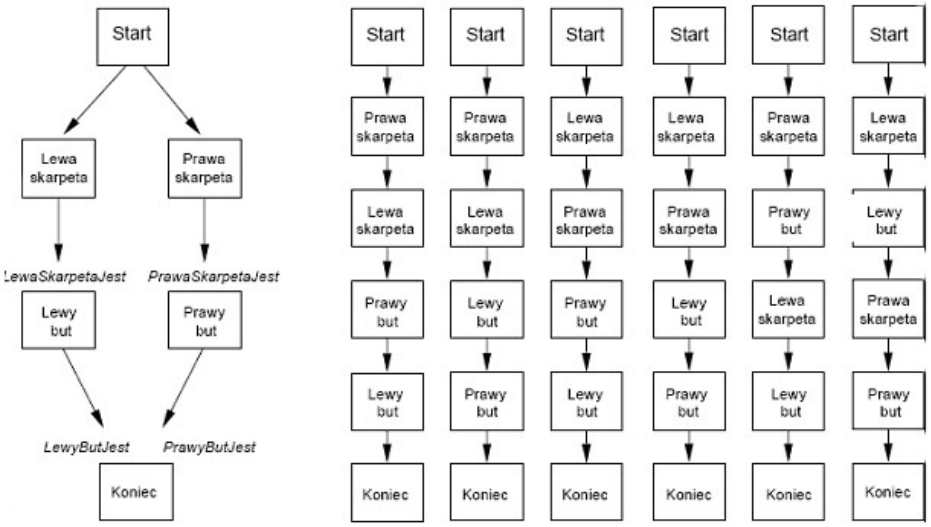
Idea poszukiwania planu:
- rozpoczynamy z planem początkowym (niedużym),
- generujemy kolejne (coraz dokładniejsze) plany częściowe, dopóki nie znajdziemy pełnego planu, wyznaczającego akcje stanowiące rozwiązanie problemu agenta.

Plan składa się z instancji operatorów zdefiniowanych dla problemu. Kolejność kroków planu (zastosowanych operatorów) musi być taka, aby były spełnione warunki wstępne (poprzedniki) dla każdej z nich.

4.3.1. Plan częściowo uporządkowany

Plan częściowo uporządkowany (PczU) to plan w którym kolejność niektórych kroków względem siebie jest ustalona, ale niekoniecznie wszystkich kroków. Plan w pełni uporządkowany stanowi sekwencję kroków.
Zamiana planu częściowo uporządkowanego na w pełni uporządkowany jest to tzw. **linearyzacja** planu - plan w pełni uporządkowany uzyskany po dodaniu w nim ograniczeń porządkujących. W ogólności możemy otrzymać wiele alternatywnych wyników linearyzacji planu.

Przykład.
Plan częściowo uporządkowany lub w pełni uporządkowany (Rysunek 23).



Rysunek 23. Plan częściowo uporządkowany i alternatywne sposoby jego linearyzacji.

Reprezentacja planu PczU obejmuje elementy:

- Zbiór kroków *S*, czyli zbiór zastosowanych operatorów.
- Zbiór ograniczeń porządkujących kroki planu, PORZĄDEK(*plan*), zapisanych w postaci $S_i < S_j$ (tzn. S_i poprzedza S_j, ale niekoniecznie bezpośrednio).
- Zbiór ograniczeń narzuconych na zmienne, o postaci *v=x*, gdzie *v* jest zmienną, a *x* - stałą lub inną zmienną, PRZYPISANIA(*plan*).
- Zbiór związków przyczynowo-skutkowych, ZWIĄZKI(*plan*), o postaci $S_i \rightarrow^c S_j$; krok S_i tworzy warunek *c* dla S_j ; taki związek określa powód stosowania kroku S_i .

Plan częściowo uporządkowany może posiadać wiele linearyzacji (każda jest równie dobra, gdyż interesuje nas dowolna sekwencja akcji prowadząca do celu). Ale taki plan może być tez wykonywany częściowo równolegle.

Rozbudowa planu jest sterowana przeszukiwaniem przestrzeni planów. Warunkiem zakończenia jest uzyskanie **planu kompletnego**, czyli planu, w którym warunek każdej jego akcji jest spełniony:
$\forall(c, S_j)$: $S_i < S_j$
$c \in$ EFEKTY(S_i)

$\neg \exists S_k : \neg c \in \text{EFEKTY}(S_k) \wedge (S_i < S_k < S_j)$

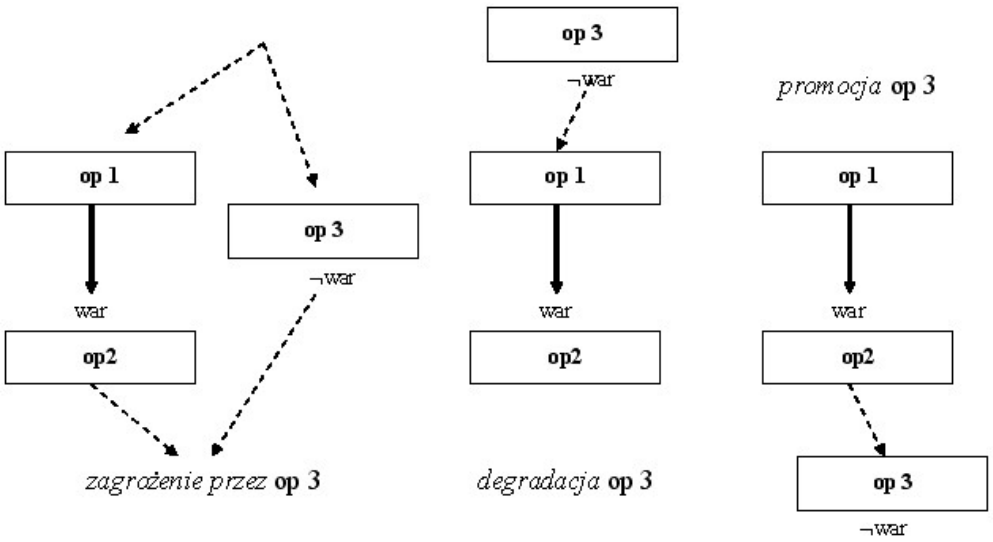
Powyższy zapis mówi o tym, że dla każdego warunku wstępnego każdej instancji operatora istnieje inna instancja operator powodująca spełnienie tego warunku i nie istnieje jakakolwiek instancja operatora pomiędzy nimi, która kasowałaby ten warunek.

4.3.2. Tworzenie planu częściowo uporządkowanego

Strategię przeszukiwania przestrzeni planów określimy następująco kroki (patrz Tabela 15 i Tabela 16):

1. Tworzymy plan początkowy zawierający jedynie akcje INIT i GOAL
2. W każdym kolejnym kroku albo dodajemy nową operację albo wykorzystujemy już obecną (poprzez dodanie połączenia przyczynowo-skutkowego), aby zapewnić spełnienie jednego z warunków wstępnych którejś z operacji już utworzonego planu.
3. Powtarzamy krok 2, dopóki nie uzyskamy planu końcowego (wtedy jest koniec) albo operację niewykonalną. W drugim przypadku wykonujemy:
   3.a. próbujemy zmienić kolejność operacji (degradacja, promocja), lub
   3.b. wykonujemy nawrót (w ramach decyzji podjętych wcześniej w kroku 2) i zmieniamy warunki wykonania operacji.

Rozwiązywanie zagrożeń jest możliwe na dwa sposoby – przez **degradację** (cofnięcie kroku stanowiącego zagrożenie) lub **promocję** (przesunięcie kroku zagrożenia do przodu (Rysunek 24).



Rysunek 24. Idea dwóch sposobów rozwiązywania zagrożeń.

Tabela 15. Funkcja tworzenie planu częściowo uporządkowanego.

```
function AgentPCzU(init, cel, operatory) : returns plan
{       plan := UtwórzPlanMinimalny(init, cel);
        while(true)
        {       if (KOMPLETNY(plan) == true) then return plan;
                [krok_wym, c] := WybierzPodcel(plan);
                WybierzOperator(plan, operatory, krok_wym, c);
                krok_zagrożenie := RozwiążZagrożenia(plan);
                if (krok_zagrożenie ≠ ∅)  NawrótNapraw(krok_zagrożenie);
        }
}
```

Tabela 16. Podfunkcje dla tworzenia PCzU.

```
function WybierzPodcel(plan) : returns [krok_wym, c]
{       wybierz krok planu (krok_wym) ze zbioru KROKI(plan) o warunku wstępnym c,
        który nie został jeszcze spełniony;
        return [krok_wym, c];
}
procedure WybierzOperator(plan, operatory, krok_wym, c)
{       wybierz krok_dod ze zbioru operatory lub KROKI(plan), taki, że jego efektem jest c;
        if (brak krok_dod ) then return;
        Dodaj związek przyczynowy (krok_dod →c krok_wym) do ZWIĄZKI(plan);
        Dodaj porządek ( krok_dod  <  krok_wym ) do PORZĄDEK(plan);
        if (krok_dod jest nowo dodanym krokiem ze zbiory operatory) then
        {  dodaj krok_dod do KROKI(plan);
           dodaj (Start < krok_dod < Finish ) do PORZĄDEK(plan);
        }
}
function RozwiążZagrożenia(plan) : returns krok_zagrożenie
{
        for (każdy krok_zagrożenie, który zagraża związkowi (krok_A →^c krok_B) ze zbioru ZWIĄZKI(plan)
        do
        {
           Wybierz {    Degradacja: dodaj krok_zagrożenie < krok_A do PORZĄDEK(plan);  lub
                        Promocja: dodaj krok_B < krok_zagrożenie do PORZĄDEK(plan);
                   }
                if (ZGODNY(plan) == false) then return krok_zagrożenie ;
        }
        return ∅ ;
}
```

4.4. Przykład budowy planu

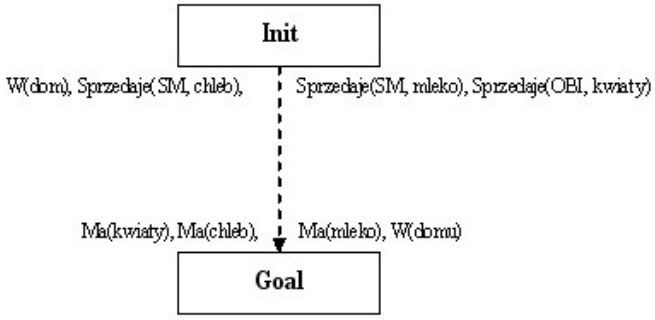
Podamy przykład budowy planu częściowo-uporządkowanego dla problemu agenta potrzebującego kilku produktów ze sklepów. Rozwiązaniem problemu (celem agenta) jest posiadanie: mleka, chleba i kwiatów oraz ponowne przebywanie w domu. Zdefiniujmy podstawowe (abstrakcyjne) akcje agenta:

1. Zakładamy, że akcja PÓJŚĆ pozwala przemieścić agenta pomiędzy dowolnymi dwoma miejscami.
2. Akcja KUPUJE oznacza zakup produktów i dla uproszczenia opisu nie wymaga ona precyzowania kwot potrzebnych dla dokonania zakupu. Niech OBI oznacza *market* przemysłowo-budowlany a SM – *supermarket*.
3. INIT to akcja inicjalizacji – wyznacza ona stan początkowy, w którym agent jest w domu a produkty są w sklepach.
4. GOAL to akcja zakończenia – wyznacza ona stan docelowy, w którym agent posiada produkty i jest w domu.

Przykład.
Zdefiniujmy operatory, których instancje będą krokami planu.

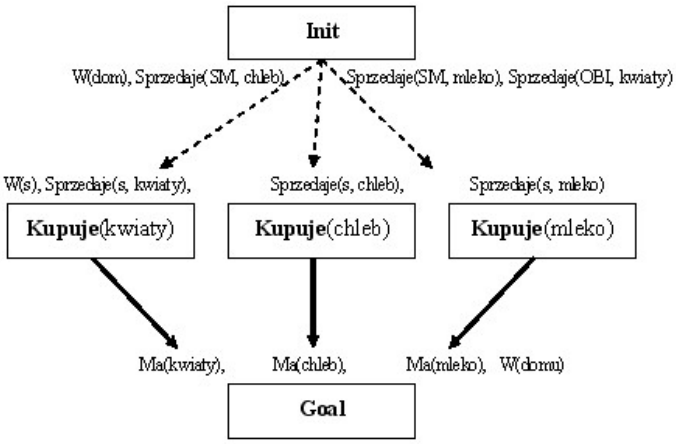
- Op(ACTION: Init,
  EFFECT: W(dom) $\wedge$ Sprzedaje(OBI, kwiaty) $\wedge$ Sprzedaje(SM, mleko) $\wedge$ Sprzedaje(SM, chleb)).
- Op(ACTION: Goal,
  PRECOND: Ma(kwiaty) $\wedge$ Ma(mleko) $\wedge$ Ma(chleb) $\wedge$ W(dom))
- Op(ACTION: Pójść(tam),
  PRECOND: W(tu),
  EFFECT: W(tam) $\wedge \neg$ W(tu)).
- Op(ACTION: Kupuje(x),
  PRECOND: W(sklep) $\wedge$ Sprzedaje(sklep,x),
  EFFECT: Ma(x)).

Plan początkowy (Rysunek 25) zawiera instancje dwóch domyślnych operatorów INIT i GOAL.



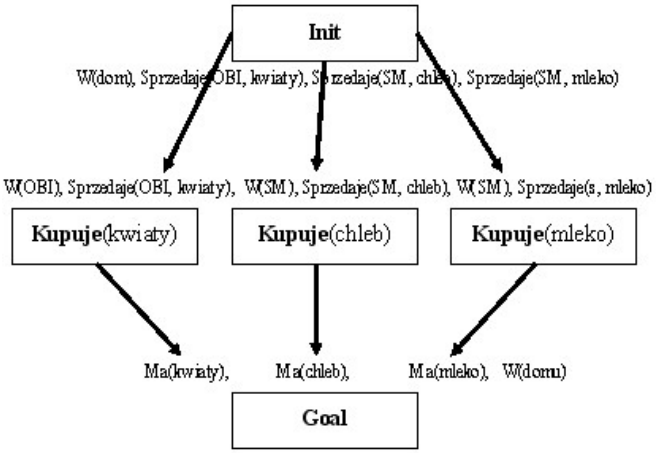
*Rysunek 25. Plan początkowy: PORZĄDEK(*plan_początkowy*) = {Init < Goal }*

Rozbudowa planu początkowego – dodane zostają trzy instancje (kroki) operatora KUPUJE, które zapewniają zachodzenie trzech predykatów MA – warunków operatora końcowego (Rysunek 26). Pogrubione strzałki reprezentują związki przyczynowo-skutkowe a cienkie strzałki wskazują „porządek” czyli kolejność wykonywania operacji.



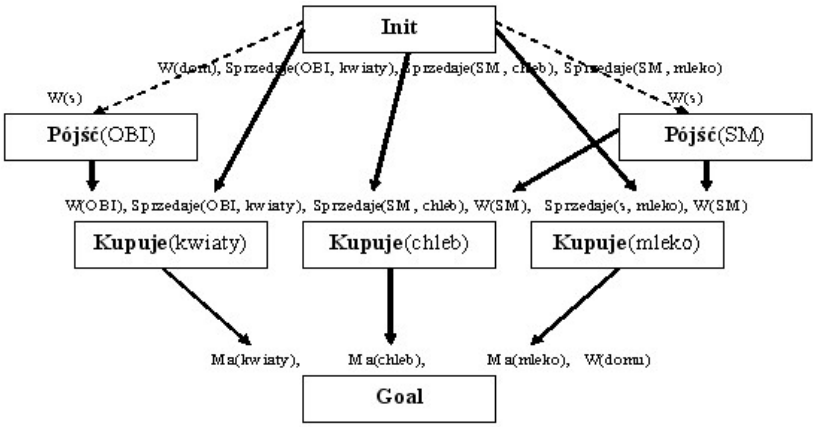
Rysunek 26. Plan po dodaniu trzech kroków KUPUJE.

Kolejna rozbudowa planu polega na stwierdzeniu faktu, że już istniejący w planie operator INIT wyznacza odpowiednie warunki dla trzech kroków KUPUJE (Rysunek 27).

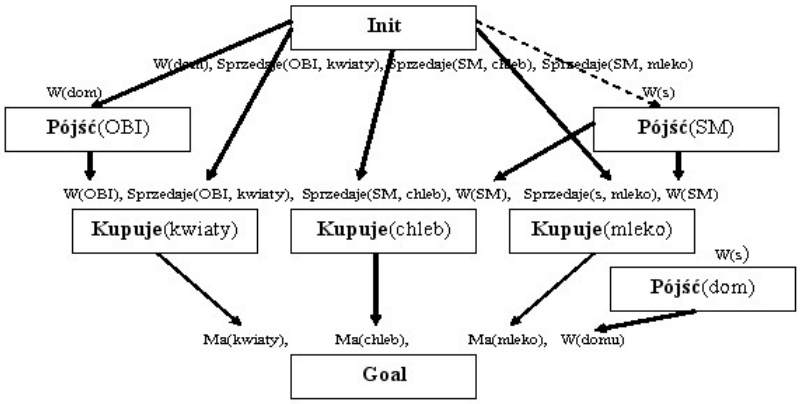


Rysunek 27. Plan po dodaniu związków przyczynowo-skutkowych INIT - KUPUJ.

Pozostały jeszcze do spełnienia warunki zadane predykatem W dla 4 kroków planu. W tym celu dodamy najpierw dwa kroki – instancje operatora PÓJŚĆ, tworzące warunki dla 3 kroków KUPUJE.
Warunki dla kroków PÓJŚĆ zostaną spełnione po dodaniu ich związków przyczynowych z krokiem INIT. Jednak wtedy powstanie kłopot (zagrożenie) spowodowany rozpoczynaniem z domu - agent nie może być następnie jednocześnie w 2 miejscach - OBI i SM. Zanim to jednak nastąpi dodanie już pierwszego związku INIT – PÓJŚĆ zmienia spełniony dotąd od początku warunek W(dom) dla operatora GOAL.
Oczywiście zarówno próba degradacji (przed INIT) jak i promocji (po GOAL) zawiodą, więc wykonywany jest nawrót i dodanie nowego kroku PÓJŚĆ(dom) dla spełnienia warunku W(dom) kroku GOAL (Rysunek 28).
Teraz ponownie można dodać związek przyczynowy dla INIT – PÓJŚĆ(OBI) (Rysunek 29) .



Rysunek 28. Plan po spełnieniu warunków dla kroków KUPUJ.

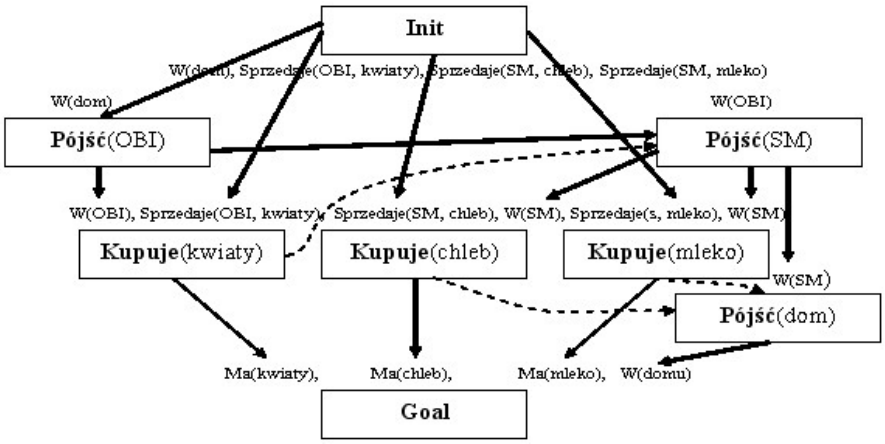


Rysunek 29. Plan po wprowadzeniu kroku PÓJŚĆ(dom).

Z kolei nastąpi próba spełnienia warunku dla operatora PÓJŚĆ(dom). Załóżmy, że dodany zostanie związek przyczynowy z operatorem PÓJŚĆ(SM). Należy jeszcze zapewnić, żeby akcja powrotu do domu wystąpiła po kupieniu mleka i chleba w supermarkecie (dodatkowe dwa kreskowane łuki dla relacji porządku), gdyż zmienia ona warunki dla obu kroków zakupu. Z tego samego powodu również akcja zakupu kwiatów musi odbyć się wcześniej. Ze względu na warunek W(OBI) kroku KUPUJE(kwiaty) niezgodny z warunkiem W(SM) kroku PÓJŚĆ(dom) ta operacja zakupu w OBI musi też odbyć się wcześniej niż akcja PÓJŚĆ(SM). Teraz pozostaje już tylko jeden możliwy związek przyczynowy dla kroku PÓJŚĆ(SM), konieczny po to, aby spełnić jego warunek W(s):

PÓJŚĆ(OBI) $\rightarrow^{W(OBI)}$ PÓJŚĆ(SM).

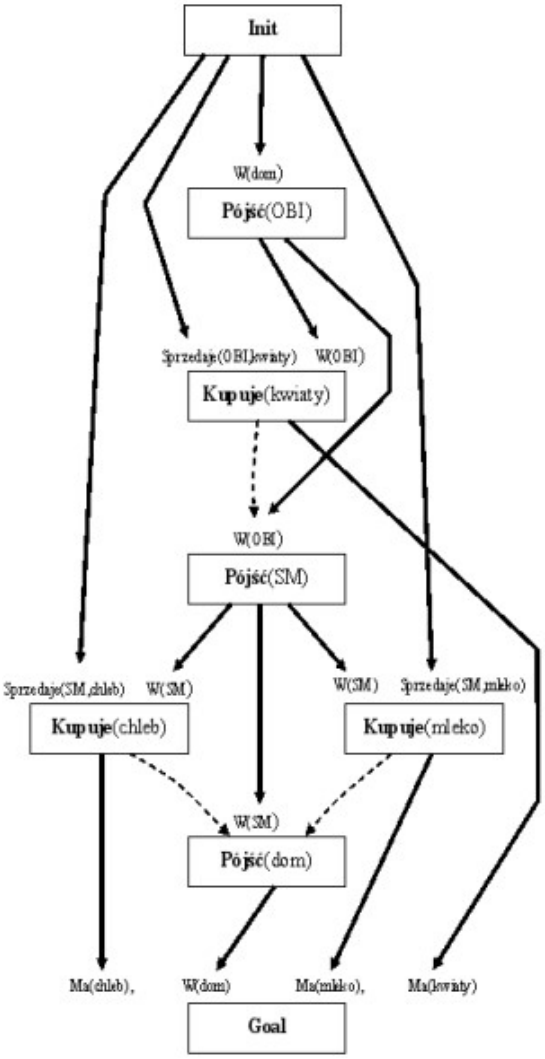
Ostateczny plan częściowo-uporządkowany dla tego problemu przedstawia Rysunek 30.



Rysunek 30. Końcowy plan częściowo-uporządkowany dla omawianego problemu.

Pozostaje nam jeszcze wykonać linearyzację powyższego planu. W tym celu uporządkujemy kroki tak, aby plan był wykonywany „od góry do dołu”. Pogrubione łuki wskazują zależności przyczynowo-skutkowe zapewniające spełnienie warunków dla poszczególnych akcji. Tym samym wyznaczają one także kolejność. Dodatkowo kolejność narzucana jest przez przerywane łuki porządku.

Po uwzględnieniu wszystkich wymagań co do kolejności kroków otrzymujemy postać planu łatwą do linearyzacji. Kroki wykonywane są od „góry do dołu”, a kroki znajdujące się na tym samym poziomie mogą być wykonane w dowolnej kolejności lub równolegle (Rysunek 31).



Rysunek 31. Uporządkowanie kroków w kolejności ich wykonywania w planie częściowo-uporządkowanym.

5. Metoda „Graphplan”

5.1. Graf planujący

Wprowadzimy specjalną strukturę danych o nazwie **graf planujący,** która pozwoli przedstawić zależności pomiędzy operatorami i predykatami warunków i efektów akcji wzdłuż „osi czasu”.
Graf planujący obejmuje „stany” charakteryzowane predykatami, „akcje” i „muteksy”:

- Sekwencja poziomów odpowiadająca przedziałom czasu – na przemian występują poziomy „stanów” (literały) (S_i) i „akcji” (A_i).
- Poziom S_0 zawiera literały spełnione w stanie początkowym.
- Akcja znajduje się w A_i wtedy, gdy jej warunek jest w S_i. Krawędzie prowadzą od warunków w S_i do efektów akcji w S_{i+1}.
- Krawędź pokazująca trwałość (niezmienność) literału, łączy literał w S_i z tym samym literałem w S_{i+1}.
- Wzajemna rozłączność - reprezentowana jest łukami typu „mutex” („mutual exclusion”):
    - Dwie akcje w A_i wzajemnie się wyłączają, jeśli jedna z nich (jej efekt) neguje warunek wykonania drugiej lub neguje efekt drugiej akcji.
    - Dwa literały w S_{i+1} połączone są łukiem „mutex” jeśli jeden jest negacją drugiego lub jeśli każda para generujących je akcji w A_i jest połączona łukiem „mutex”.

- Akcja nie może wystąpić w A_i jeśli dowolna para jej warunków jest w S_i uznana za wykluczającą się (jest połączona łukiem „mutex”).
- Dwie akcje w A_i wzajemnie się wyłączają, jeśli warunek jednej z nich wyklucza się z warunkiem drugiej.

Uwagi.

1. Liczba poziomów w grafie planującym, po których uzyskany zostaje określony literał jest dobrym oszacowaniem (optymistycznym) tego, jak trudno jest uzyskać ten literał wychodząc ze stanu początkowego.
2. Graf planujący wymaga literałów w postaci predykatowej – pozbawionych zmiennych. Ponieważ zarówno STRIPS jak i ADL dopuszczają jedynie literały podstawowe, tzn. występujące w nich termy nie zawierają symboli funkcji, obie reprezentacje mogą zostać przekształcone do postaci predykatowej.

Graf planujący może posłużyć do:

1. wyznaczenia oszacowania heurystyki dla „poinformowanej” strategii przeszukiwania przestrzeni planów, lub
2. do bezpośredniego wyznaczenia planu przez algorytm nazwany GRAPHPLAN-em.

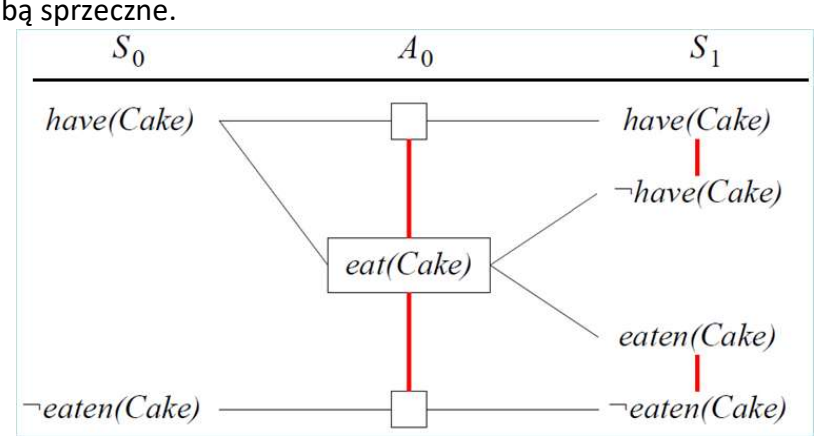
Przykład

Aby „mieć ciastko, zjeść ciastko i nadal je mieć”, zdefiniujemy operatory „jeść” (eat) i „piec” (bake) a warunek sytuacji końcowej utworzą iloczyn predykatów „mieć” (have) i „zjedzone” (eaten) :

OP(Init,
EFFECT: (*have(Cake)* $\wedge \neg$ *eaten(Cake)*))
OP(Goal,
PRECOND: (*have(Cake)* $\wedge$ *eaten(Cake)*))
OP(Action(*eat(Cake)*),
PRECOND: *have(Cake)*
EFFECT: *eaten(Cake)* $\wedge \neg$*have(Cake)*))
OP(Action(*bake(Cake)*),
PRECOND: $\neg$*have(Cake)*
EFFECT: *have(Cake)*))

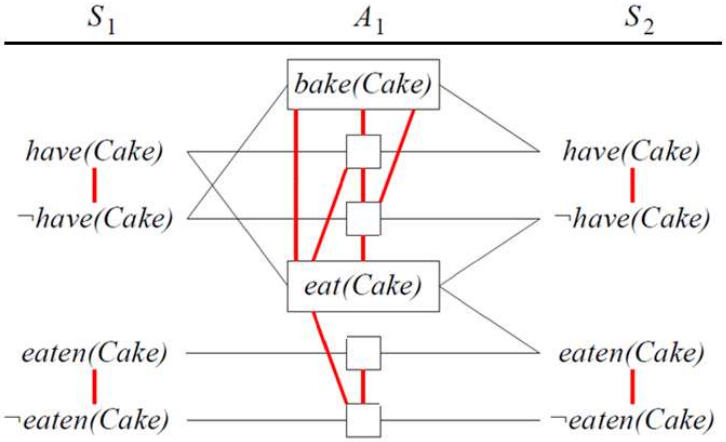
Początkowy poziom akcji (Rysunek 32):

- W S_1 literały *have*(*Cake*) i $\neg$*have*(*Cake*) są ze sobą sprzeczne i tworzą mutex; podobnie jest z *eaten*(*Cake*) i $\neg$*eaten*(*Cake*).
- W A_0 operacja *eat*(*Cake*) tworzy mutex-y z operacjami „przeroczystości”, gdyż ich efekty w S_1 są ze sobą sprzeczne.



Rysunek 32. Przykładowy problem “zjeść ciastko i mieć ciastko”. Pierwszy poziom akcji i stanu.

Drugi poziom akcji (Rysunek 33) obejmuje operacje „eat(Cake)” i „bake(Cake)”. Operacja *eat(Cake)* tworzy mutex z operacją *bake(Cake)* a obie operacje tworzą mutex-y z wiekszością operacji przezroczystości.



Rysunek 33. Drugi poziom akcji w omawianym przykładzie.

Osiągnięcie sytuacji końcowej zapewnia sekwencja klastrów operacji:
1: { *eat(Cake)* }
2: { *bake(Cake)*, przezroczytość(*eaten(Cake*))}.

Uwaga. Każda wielokrotność tej pary operacji również spełnia warunek sytuacji końcowej. Aby wprowadzić jednoznaczność planu należy rozszerzyć warunek operatora *eat(Cake)* o predykat *¬eaten(Cake).*

5.2. Algorytm „Graphplan”

Algortym **Graphplan** pobiera problem planowania zdefiniowany w STRIPS i wyznacza sekwencję akcji prowadzącą z początkowego stanu problemu do docelowego stanu problemu. Graphplan operuje na grafie planującym, w którym:

- węzły odpowiadają akcjom i literałom uporządkowanym w naprzemienne poziomy,
- łuki są 3 rodzajów: od literału (warunku) do akcji, od akcji do literału (efektu), od literału do literału (gdy nie ulega zmianie),

Pamiętane są listy wykluczających się literałów, czyli takich, które nie mogą być jednocześnie prawdziwe, a także listy wykluczających się akcji – nie mogą być wspólnie wykonywane na danym poziomie.

5.2.1. Algorytm

Algorytm “Graphplan” iteracyjnie rozszerza graf planujący (wstecz - „od tyłu do przodu), sprawdzając, czy nie ma rozwiązań o długości *N*-1 zanim zajmie się planami o długości N.

Graphplan wyznacza plan na podstawie grafu planującego analizując go od ostatniego poziomu wstecz. Iteracyjny sposób analizy odpowiada przeszukiwaniu wszerz pewnego drzewa decyzyjnego (Tabela 17):

- Węzłem drzewa jest podzbiór zgodnych ze sobą akcji na jednym poziomie grafu, których efektem są wszystkie te literały, które uznane zostały za cele w poprzedniku węzła.
- Węzeł początkowy jest zbiorem celów na ostatnim poziomie S_n grafu planującego.

- Istnieje krawędź w drzewie decyzyjnym od węzła (=podzbiór w S_i) do węzła(=podzbiór w S_{i-1}), jeśli istnieje podzbiór zgodnych akcji w A_{i-1} , które łączą je jako efekty i warunki.
- Celem przeszukiwania jest osiągnięcie S_0.

Graphplan na przemian:

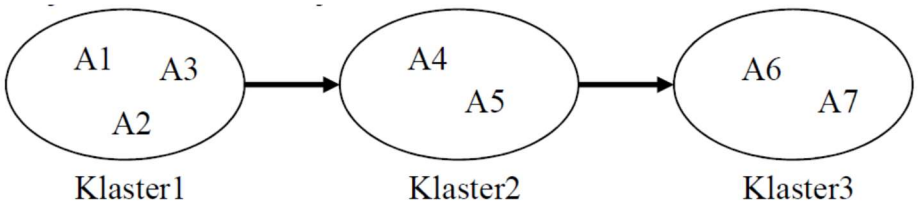
a) podejmuje wykrycie rozwiązania („czy osiągnięty został stan początkowy problemu”?) i
b) rozszerza graf o podzbiory akcji i warunki poprzedniego poziomu w grafie planującym (porusza się WSTECZ wzdłuż grafu planujacego).

Wynikiem pracy algorytmu GRAPHPLAN jest plan, w którym akcje nie są ani w pełni uporządkowane ani częściowo uporządkowane (PCzU). Powstaje pośrednie rozwiązanie (Rysunek 34):

- Plan składa się z sekwencji klastrów akcji,
- W ramach klastra kolejność akcji jest dowolna,
- Kolejność klastrów jest ustalona.

Tabela 17. Funkcja GRAPHPLAN.

```
function GRAPHPLAN(problem) returns solution lub failure
{  graph := GenerujPlanPoczątkowy(problem);
   goals := Cele[problem];
   loop do {
         if goals niepuste then do {
            solution := WyznaczRozwiązanie(graph, goals, Długość(graph));
             if solution ≠ failure then return solution;
         }
         else if  BrakRozwiązania(graph)
               then return failure ;
         graph := RozszerzGraf(graph, problem);
         goals := {wszystkie nie-mutex-y na ostatnim poziomie grafu};
   }
}
```



Rysunek 34. Struktura planu generowanego przez algorytm GRAPHPLAN.

Zalety GRAPHPLAN-u w porównaniu do planera PCzU:

- Znaczna poprawa efektywności w porównaniu z planerem PCzU, dzięki:
  - wzajemnemu wykluczaniu, równoległemu planowaniu, przycinaniu nieosiągalnych zbiorów celów, nie ma potrzeby podstawiania pod zmienne podczas przeszukiwania.
- Zaleta zakończenia pracy, gdy problem jest nierozwiązywalny:
  - wykrycie sytuacji, gdy dany poziom stanów w grafie planującym jest identyczny z kolejnym poziomem;
  - jeśli nie znaleziono planu przed powstaniem takiej sytuacji, to oznacza brak rozwiązania.

6. Planowanie hierarchiczne

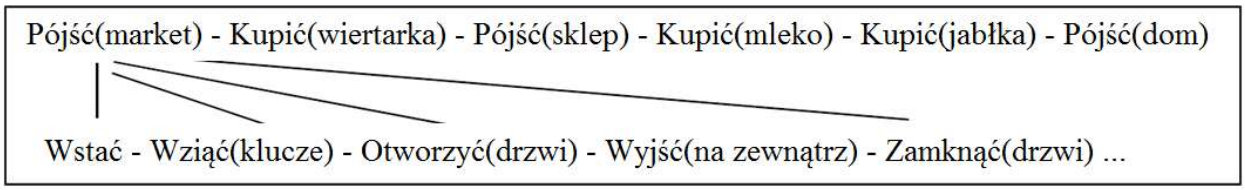
6.1. Zasada

Planowanie hierarchiczne odpowiada konstruowaniu programu metodą **zstępującą** i stosuje zasadę **hierarchicznej dekompozycji:**

- Rozpoczynamy od planu opisanego przy pomocy operatorów **abstrakcyjnych** (opis wysokiego poziomu);
- W kolejnych krokach każde wystąpienie abstrakcyjnego operatora zastępuje się jego **implementacją** – czyli **planem**, który go realizuje stosując przy tym operatory niższego poziomu;
- Proces dekompozycji jest kontynuowany tak długo, aż plan zawierać będzie jedynie operatory **podstawowe** (elementarne).

Przykład

Plan o postaci: [Pójść(sklep), Kupić(mleko), Pójść(dom)], jest dobry dla człowieka, ale nie dla robota (maszyny). Nawet pewna konkretyzacja tego planu nam nie wystarczy (Rysunek 35).



Rysunek 35. Plan wyrażony w terminach abstrakcyjnych akcji.

Potrzebny jest plan konkretny, posługujący się akcjami wykonywalnymi przez robota. Np. NaWprost(2 m), Obrót(+90 stopni). Wyróżnimy hierarchię warstw abstrakcji dla akcji:

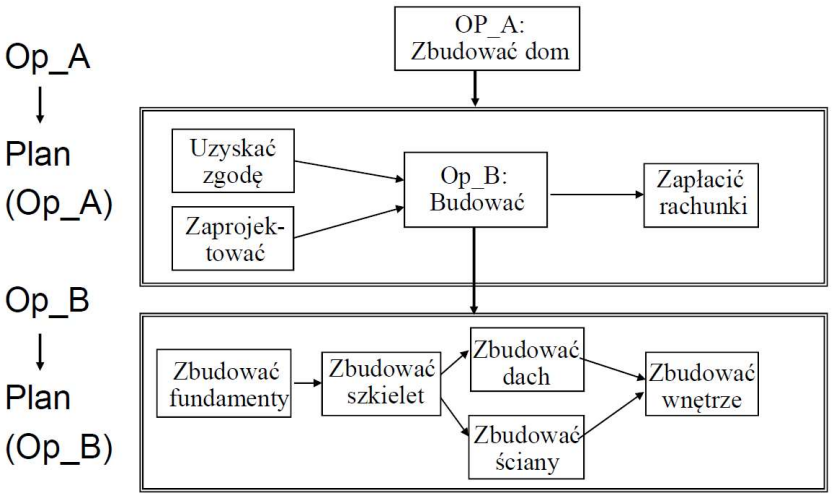
- Operatory najniższego poziomu to **operatory podstawowe**;
- **Operatory abstrakcyjne** występują na wyższych poziomach

Możliwość utworzenia planu dla rozwiązania danego zadania określimy mianem **rozwiązania**.

- **Rozwiązanie abstrakcyjne** to zupełny i poprawny plan zawierający przynajmniej jeden operator (krok) abstrakcyjny.
- **Rozwiązanie** to zupełny i poprawny plan zawierający wyłącznie operatory podstawowe (elementarne).
- Plan P nazywamy **abstrakcją** planu p, jeśli p można otrzymać z P stosując metodę dekompozycji.
- **Własność zstępujących rozwiązań**: każde rozwiązanie abstrakcyjne jest abstrakcją pewnego rozwiązania.
- **Własność wstępujących rozwiązań**: żaden sprzeczny plan abstrakcyjny nie jest abstrakcją rozwiązania.

6.2. Przykład dekompozycji zadania

Operator wyższego poziomu odpowiada planowi na niższym poziomie abstrakcji (Rysunek 36)



Rysunek 36. Przykład planu hierarchicznego.

Do notacji stosowanej w STRIPS dodajemy zbiór metod dekompozycyjnych. Każda metoda jest wyrażeniem postaci Decompose(o, p), gdzie o jest operatorem abstrakcyjnym, a p jest planem.

Przykład (Tabela 18)

Tabela 18. Przykład planu hierarchicznego

Decompose(Budować, Plan(KROKI: {S1:Zbudować(fundamenty), S2: Zbudować(szkielet), S3: Zbudować(dach), S4: Zbudować(ściany), S5:Zbudować(wnętrze)} PORZĄDEK: {S1<S2<S3<S5, S2<S4<S5}, PRZYPISANIA: {} ZWIĄZKI: {S1$\rightarrow^{fundamenty}$S2, S2$\rightarrow^{szkielet}$S3, S2$\rightarrow^{szkielet}$S4, S3$\rightarrow^{dach}$S5, S4 $\rightarrow^{ściany}$ S5 }))

Co należy uwzględnić podczas dekompozycji operatora?

- Efekty mogą zależeć od sytuacji; Np. Efektem akcji (Przekręć(kontakt)) może być zapalone światło, gdy nie było włączone, w przeciwnym razie gaśnie.
- Czas trwania: akcje mają czas trwania, podlegają ograniczeniom czasowym;
- Zasoby: akcje kosztują zasoby (np. zakupy wymagają pieniędzy), ale akcje mogą też generować nowe zasoby (np. pobrać pieniądze).

Przyjmijmy założenie: rozwiązanie wymaga 64 kroków, w każdym występuje 1 z 3 alternatywnych operatorów. Standardowe rozwiązanie: w najgorszym przypadku wymaga sprawdzenia 3^{64} operacji; złożoność jest wykładniczą funkcją długości rozwiązania. Planowanie hierarchiczne: jeśli każdy abstrakcyjny operator zostanie zdekomponowany na 4 kroki a liczba alternatywnych dekompozycji wynosi zawsze 3, z których dokładnie 1 jest elementem rozwiązania, to istnienie 64 kroków wymaga 3 warstw i w najgorszym przypadku należy sprawdzić: 3 * 4 + 3 * 4 * 4 + 3 * 4 * 4 *4 operacji; złożoność jest wykładnicza względem liczby warstw hierarchii, ale liczba warstw jest logarytmem z liczby kroków.

6.3. Dekompozycja operatora

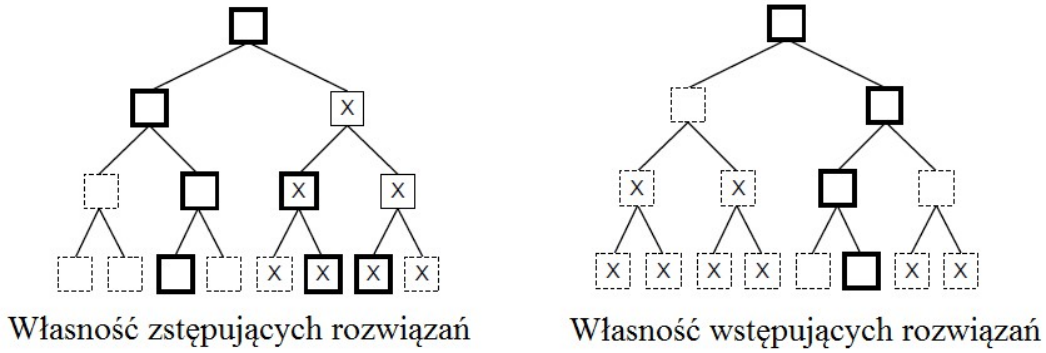
Plan P jest **poprawną** implementacją (dekompozycją) operatora O wtedy, gdy zachodzi:

1. Plan P jest poprawny, tzn. zbiory ograniczeń i wartościowań (przypisania) zmiennych są niesprzeczne;
2. Każdy efekt w O jest zapewniony przez jakiś krok w planie P i nie jest ponownie usuwany w innym kroku;
3. Każdy warunek kroku w planie P musi być uzyskany w poprzednim kroku lub być warunkiem wstępnym dla O i przed wykonaniem kroku nie został usunięty.

Dekompozycja operatora jest sensowna wtedy, gdy zachodzi ograniczona interakcja pomiędzy pojedynczymi warstwami. Daje nam to możliwość ponownego użycia operatora i definiowania biblioteki planów.

Dekompozycja musi prowadzić do akcji wykonywalnych przez agenta. Planowanie hierarchiczne jest możliwe wtedy, gdy zachodzi jeden z warunków (Rysunek 37):

1. dla każdego abstrakcyjnego planu istnieje odpowiedni plan wykonywalny (własność **zstępujących rozwiązań**);
2. żaden niezgodny plan abstrakcyjny nie ma odpowiedniego planu wykonywalnego (własność **wstępujących rozwiązań.**



Rysunek 37. Własność wstępujących rozwiązań nie zawsze zachodzi.. Należy dokonać modyfikacji niezgodnego planu abstrakcyjnego, aby uzyskać plan zgodny.

Niech, plan=M(o, plan_we), będzie metodą dekompozycyjną – modyfikującą plan „plan_we" do postaci „plan" w wyniku implementacji operatora „o". Operator „*a*" występujący w p nazywamy **operatorem głównym** metody M, jeśli każdy warunek operatora o jest warunkiem operatora *a* i każdy efekt operatora „o" jest efektem operatora *a*. Można pokazać, że jeśli każda metoda dekompozycyjna, którą stosuje planer, posiada dokładnie jeden operator główny, to zagwarantowana jest własność wstępujących rozwiązań.

6.4. Algorytm hierarchicznego planera

Ogólny pseudokod hierarchicznego planera przedstawiono w Tabela 19. Procedury WybierzPodcel, WybierzOperator i RozwiążZagrożenie są takie same jak w algorytmie planera PCzU Funkcja CzyRozwiązanie dodatkowo sprawdza, czy każdy operator w planie jest operatorem podstawowym. Funkcja WybierzAbstrakcyjny wybiera dowolny abstrakcyjny operator występujący w planie. Procedura WykonajDekompozycję wybiera metodę dekompozycyjną dla instancji operatora $krok_{abstr}$ i odpowiednio modyfikuje aktualny plan.

Tabela 19. Algorytm planowania przez hierarchiczną dekompozycję.

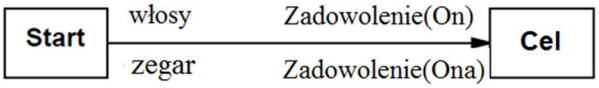
```
function HD_PCzU(plan, operatory, metody) : returns plan
// plan : abstrakcyjny plan zawierający kroki Init i Goal (i ewentualnie dalsze)
{ do {  // potencjanie nieskończona pętla
        if CzyRozwiązanie(plan) then return plan;
        [krok_wym, c] := WybierzPodcel(plan);
        WybierzOperator(plan, operator, krok_wym, c);
        krok_abstr := WybierzAbstrakcyjny(plan);
        WykonajDekompozycję(plan, metody, krok_abstr);
         krok_zagrożenie := RozwiążZagrożenia(plan);
        if (krok_zagrożenie ≠ ∅)  NawrótNapraw(krok_zagrożenie);
    }
}
```

Niech wybraną metodą dekompozycji jest: P = Dekompozycja($krok_{abstr}$, p). Wynikiem ma być modyfikacja P aktualnego planu p . Odbywa się to następująco:

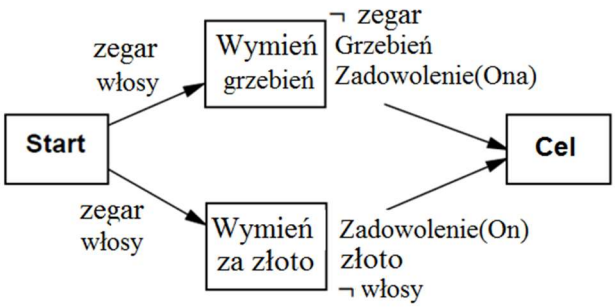
- Do zbioru kroków planu P dodajemy wszystkie instancje operatorów (kroki) planu p i usuwamy $krok_{abstr}$.
- Do zbioru wartościowań zmiennych planu P dodajemy wartościowania (przypisania) zmiennych planu p.
- Kopiujemy do P relacje porządku z p, zastępując każdy element porządku o postaci, $krok_a < krok_{abstr}$, zbiorem porządków { $krok_a < krok_{m1}$, …, $krok_a < krok_{mi}$}, gdzie $krok_{m1}$,…, $krok_{mi}$, to maksymalne (w relacji <) kroki w p.
- Każdy element porządku postaci, $krok_{abstr} < krok_a$, zastępujemy zbiorem { $krok_{m1} < krok_a$, …, $krok_{mj} < krok_a$}, gdzie $krok_{m1}$, …, $krok_{mj}$ są minimalnymi krokami planu p.
- Kopiujemy do P związki z p, zastępując każdy związek przyczynowy występujący w planie P o postaci, $krok_i \rightarrow^c krok_{abstr}$, zbiorem związków o postaci $krok_i \rightarrow^c krok_m$, gdzie $krok_m$ jest instancją operatora w p z warunkiem wstępnym c, i c nie jest warunkiem wstępnym żadnego wcześniejszego operatora w p.
- Podobnie, każdy związek postaci, $krok_{abstr} \rightarrow^c krok_j$, zastępujemy zbiorem związków postaci $krok_m \rightarrow^c krok_j$, gdzie $krok_m$ jest operatorem w p z efektem c i c nie jest efektem żadnego późniejszego operatora w p.

Na koniec wykonania głównej pętli planera, jeśli operacja dekompozycji doprowadziła plan do sprzeczności, wykonujemy nawrót.

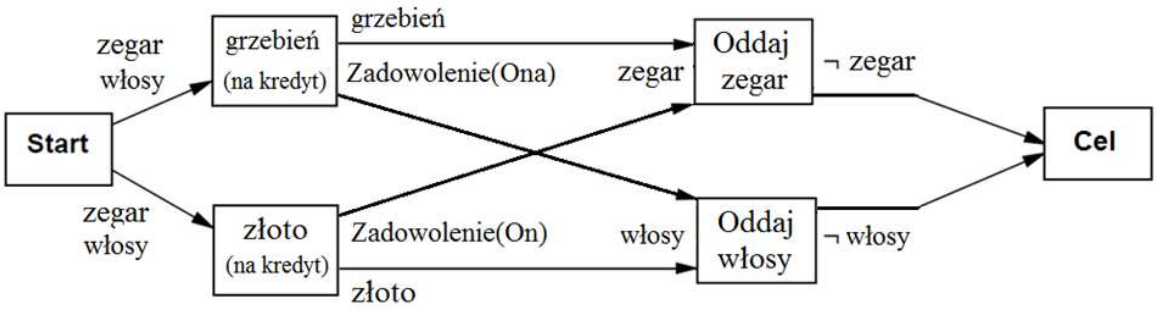
Przykład (Rysunek 38)



Plan początkowy



Niezgodny plan abstrakcyjny



Rysunek 38. Przykład modyfikacji niezgodnego planu w celu uzyskania poprawnego planu:

7. Pytania

7.1. Poszukiwanie celu

1. Przedstawić losowe, niepoinformowane algorytmy poszukiwania celu („próbkowanie" globalne i lokalne, „błądzenie przypadkowe").
2. Przedstawić poinformowane przeszukiwanie lokalne („przez wspinanie").
3. Przedstawić poinformowane losowe poszukiwanie („symulowane wyżarzanie").
4. Omówić algorytm genetyczny.

7.2. CSP

1. Jak reprezentujemy problem z ograniczeniami (CSP)?
2. Wyjaśnić zasadę działania algorytmu poszukiwania przyrostowego dla CSP ?
3. Na czym polegają usprawnienia algorytmu poszukiwania z nawrotami?
4. Na czym polega poszukiwanie ze stanem kompletnym dla CSP?

7.3. Mini-max

1. Wyjaśnić cel stosowania i zasady działania strategii „**przeszukiwanie Mini-max"**
2. **Omówić strategię „cięć α-β".** Czy jest ona optymalna?
3. 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?

7.4. Planowanie

1. Wyjaśnić problem planowania.
2. Czym są STRIPS i ADL?
3. Omówić podstawową strategię tworzenia planu częściowo-uporządkowanego.
4. Wyjaśnić istotę operacji „promocji" i „degradacji" oraz przyczyny ich stosowania podczas tworzenia planu.

8. Bibliografia

1. S. Russel, P. Norvig: *Artificial Intelligence. A modern approach.* Prentice Hall, 2002 (2nd ed.), 2013 (3d ed.). [Rozdziały 4, 5, 6, 11]
2. M. Flasiński: *Wstęp do sztucznej inteligencji*. Wydawnictwo Naukowe, PWN, 2011. [Rozdziały 4, 5]