Sztuczna Inteligencja i Systemy Doradcze

Przeszukiwanie przestrzeni stanów — problemy z więzami

Problemy z wiezami (CSP)

Ogólnie: stan jest "czarną skrzynką", dla której:

- można sprawdzić, czy jest celem
- dana jest wartość funkcji oceny użyteczności stanu
- można obliczyć sąsiednie stany

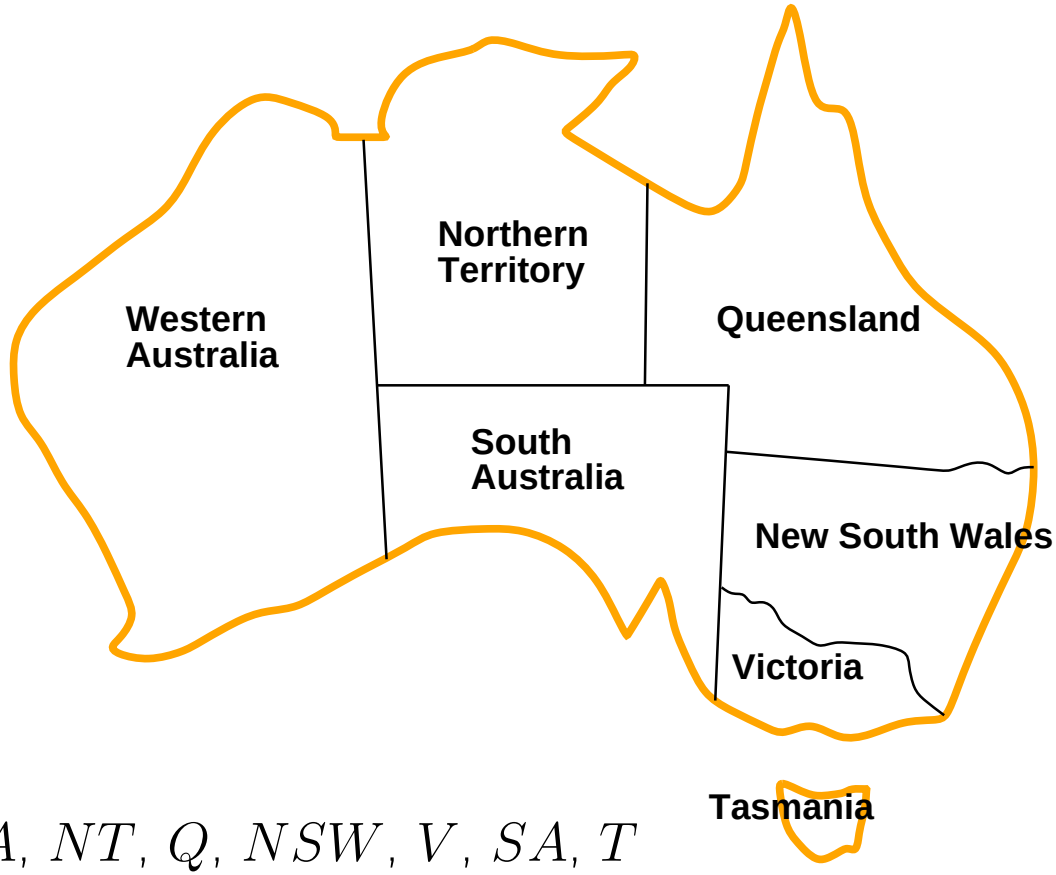
CSP:

- stan jest zdefiniowany przez *zmienne* X_i z *wartościami* z *dziedziny* D_i
- test celu jest zbiorem *więzów (ograniczeń)* specyfikujących
  - dopuszczalne kombinacje wartości dla podzbiorów zmiennych

CSP jest prostym przykładem *formalnego języka* do reprezentacji stanów

Reprezentacja CSP umożliwia skonstruowanie algorytmów bardziej efektywnych niż ogólne algorytmy przeszukiwania

Przyklad CSP: Kolorowanie mapy



Zmienne WA, NT, Q, NSW, V, SA, T

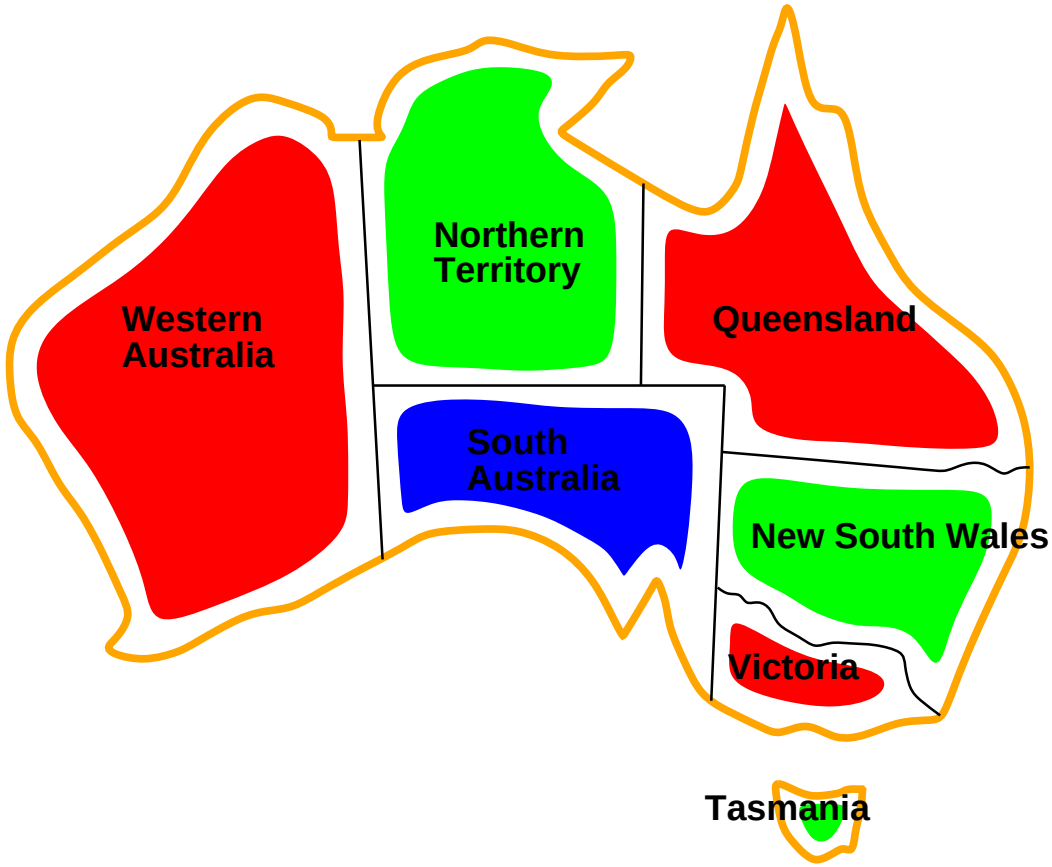
Dziedziny $D_i = \{red, green, blue\}$

Więzy: sąsiednie regiony mają mieć różne kolory

np. $WA \neq NT$ (jeśli symbol $\neq$ należy do języka), lub

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Przyklad CSP: Kolorowanie mapy

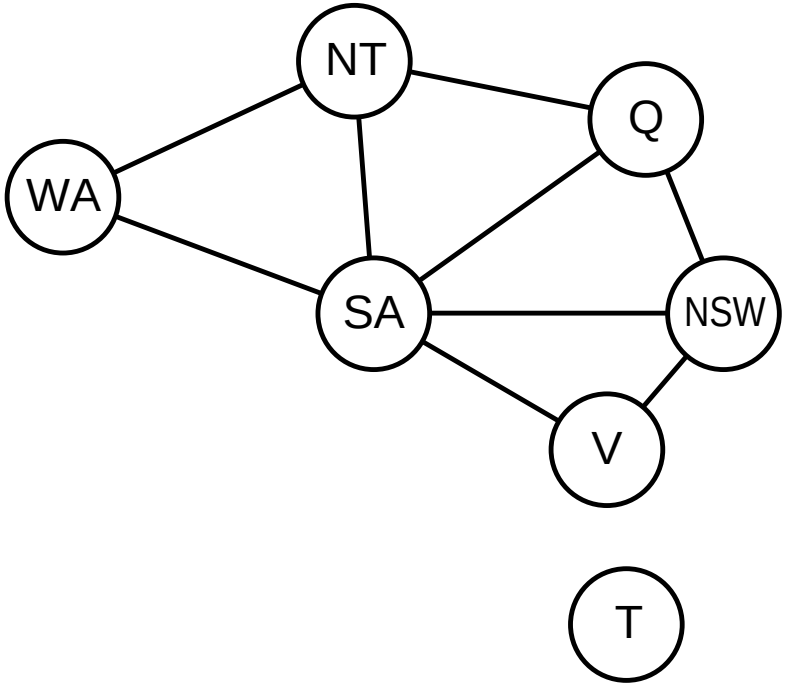


Rozwiązania są wartościowaniami spełniającymi wszystkie więzy, np.
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Graf zaleznosci

Binarny CSP: każde ograniczenie dotyczy co najwyżej dwóch zmiennych

Graf zależności: wierzchołki odpowiadają zmiennym, krawędzie — więzom



Algorytmy dla binarnego CSP używają struktury grafu do przyspieszenia szukania, np. Tasmania jest problem niezależnym od pozostałych regionów!

Rodzaje problemow CSP

Zmienne dyskretne

dziedziny skończone rozmiaru $d \Rightarrow O(d^n)$ pełnych wartościowań

- ♢ np. boolowskie CSP, w tym problem spełnialności (NP-zupełny)

dziedziny nieskończone (liczby całkowite, słowa, etc.)

- ♢ np. planowanie zadań, zmienne: daty początku i końca zadań
- ♢ wymaga języka więzów, np. $StartJob_1 + 5 \leq StartJob_3$
- ♢ problemy z więzami liniowymi są obliczalne
- ♢ w ogólności nierozstrzygalne

Zmienne ciągłe

- ♢ np. momenty początku i końca obserwacji przez Teleskop Hubble'a
- ♢ problemy z więzami liniowymi można rozwiązać
  w czasie wielomianowym (programowanie liniowe)

Rodzaje wiezow

Więzy unarne dotyczą pojedynczych zmiennych,
np. $SA \neq green$

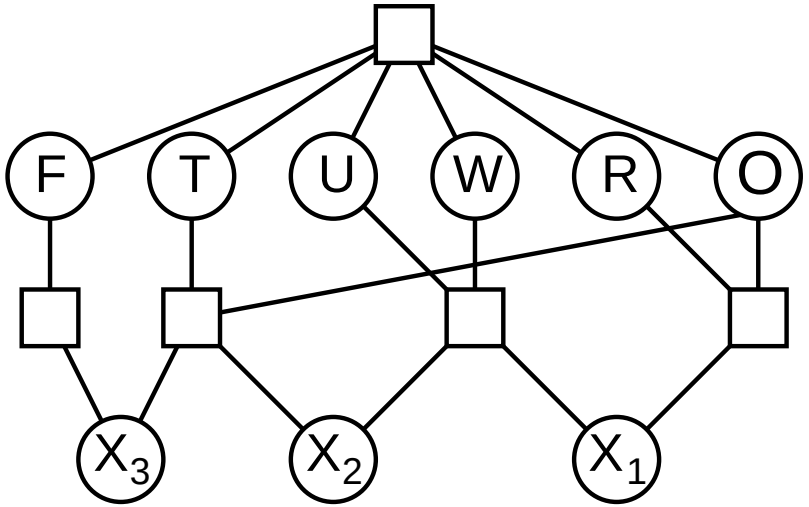
Więzy binarne dotyczą dwu zmiennych,
np. $SA \neq WA$

Więzy wyższego rzędu dotyczą 3 lub więcej zmiennych,
np. w kryptoarytmetyce

Preferencje (więzy nieostre), np. red jest lepszy niż $green$
często reprezentowane przez funkcję kosztu przypisania wartości do zmiennej
$\longrightarrow$ problemy optymalizacyjne z więzami

Przyklad: kryptoarytmetyka

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



Zmienne: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$

Dziedziny: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Więzy: F, T, U, W, R, O wszystkie różne

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

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

$X_2 + T + T = O + 10 \cdot X_3$

$X_3 = F$

Rzeczywiste problemy CSP

Problem przydziału
    np. kto którą klasę będzie uczyć?

Problem rozplanowania zadań
    np. gdzie i kiedy będą odbywać się poszczególne zajęcia?

Konfiguracja sprzętowa

Arkusze elektroniczne

Logistyka

Zaplanowanie produkcji

Planowanie kondygnacji

Wiele rzeczywistych problemów ma zmienne o wartościach rzeczywistych

Przeszukiwanie przyrostowe z powracaniem

Stany: zmienne częściowo przypisane (tzn. tylko niektóre)
więzy zawsze spełnione dla ustalonych zmiennych

Stan początkowy: pusty zbiór przypisań $\{\}$

Funkcja następnika: przypisuje wartość do nieprzypisanej zmiennej tak,
żeby nie powodować konfliktu więzów z dotychczasowym przypisaniem
$\Rightarrow$ porażka, gdy ustalenie kolejnej zmiennej jest niewykonalne

Cel: pełne przypisanie zmiennych

1) Dla problemu z n zmiennymi każde rozwiązanie jest na głębokości n
$\Rightarrow$ warto używać przeszukiwania wgłab
2) Ścieżka jest nieistotna, przypisanie zmiennych jest *przemienne*, np.
[najpierw $WA = red$ potem $NT = green$] to tak samo jak
[najpierw $NT = green$ potem $WA = red$]
$\Rightarrow$ wystarczy rozważyć jeden ustalony porządek przypisywania zmiennych

Przeszukiwanie przyrostowe z powracaniem

Przeszukiwanie przyrostowe z powracaniem (ang. backtracking)

przeszukiwanie wgłąb, każdy krok to ustalenie wartości jednej zmiennej

kolejność przypisywania zmiennych jest ustalona

jeśli ustalenie kolejnej zmiennej jest niewykonalne bez łamania więzów

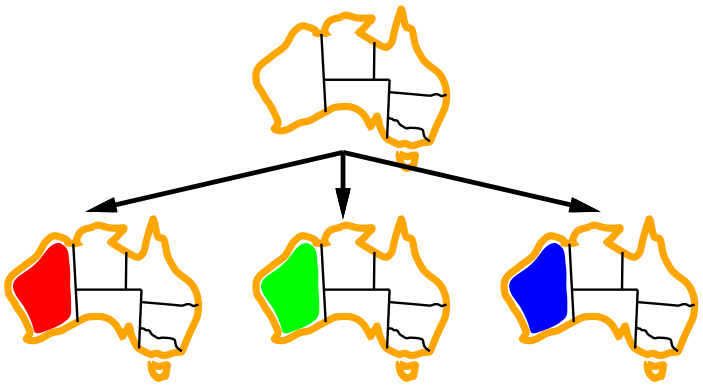
następuje *powrót, tzn. cofnięcie niektórych przypisań*

function BACKTRACKING-SEARCH(*csp*) **returns** solution/failure
  return RECURSIVE-BACKTRACKING([ ], *csp*)

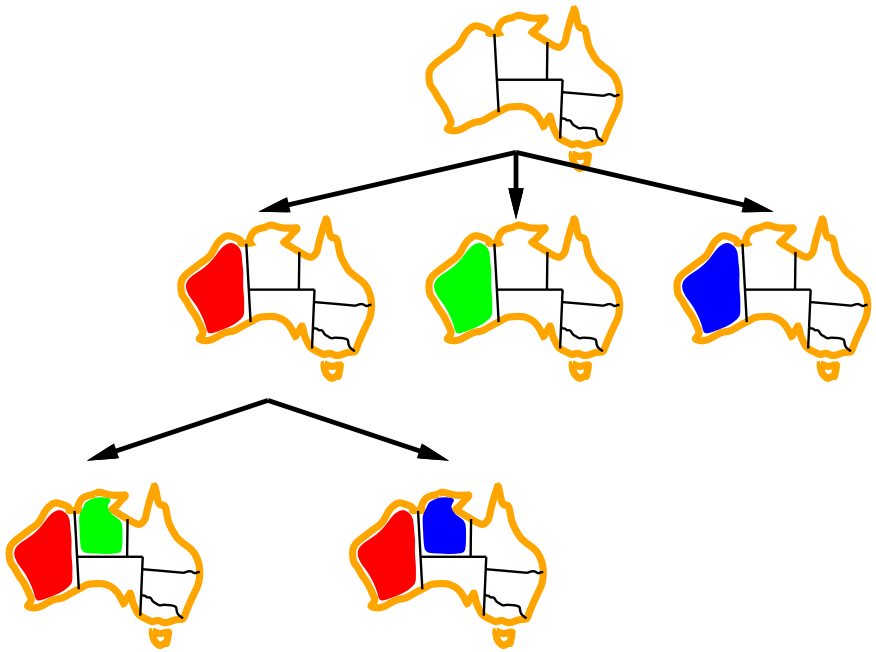
function RECURSIVE-BACKTRACKING(*assigned*, *csp*) **returns** solution/failure
  if *assigned* is complete **then return** *assigned*
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assigned*, *csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assigned*, *csp*) **do**
    if *value* is consistent with *assigned* according to CONSTRAINTS[*csp*] **then**
      result ← RECURSIVE-BACKTRACKING([*var* = *value*|*assigned*], *csp*)
      if *result* ≠ *failure* **then return** *result*
  end
  return *failure*

Przesz. przyrostowe z powracaniem: przyklad

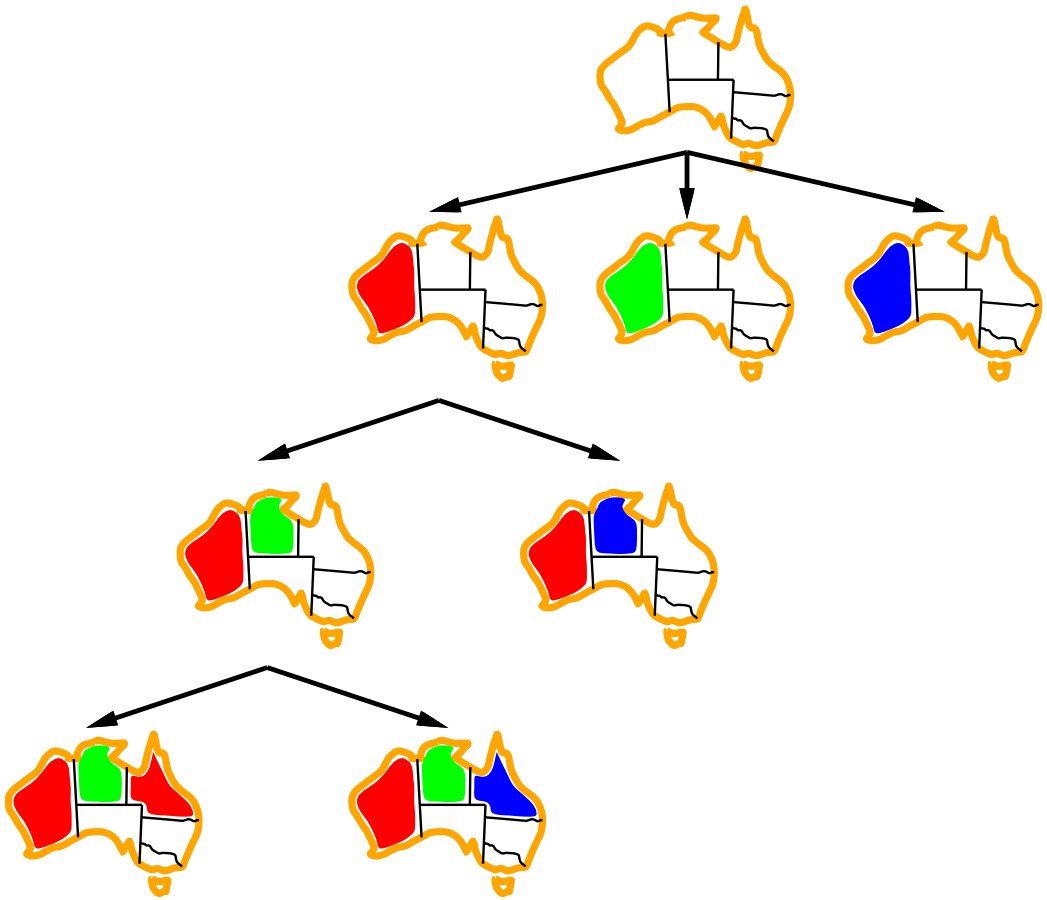
Przesz. przyrostowe z powracaniem: przyklad



Przesz. przyrostowe z powracaniem: przyklad



Przesz. przyrostowe z powracaniem: przyklad



Przesz. przyrostowe z powracaniem: wlasnosci

Uniwersalność??

Metoda jest uniwersalna dla wszystkich problemów z więzami

Złożoność pamięciowa??

Pamięta dotychczasowe przypisanie $\Rightarrow$ liniowa

Złożoność czasowa??

W każdym węźle sprawdza wszystkie możliwe przypisania do jednej zmiennej

$\Rightarrow$ rozgałezienie drzewa przesz. $b =$ rozmiar dziedziny zmiennych d

$\Rightarrow$ drzewo przeszukiwań ma d^n liści

Efektywność??

Rozwiązuje problem n-hetmanów dla $n \approx 25$

Heurystyki przyspieszajce

- ♢ Wybór zmiennej
  ⇒ najbardziej ograniczającej spośród najbardziej ograniczonych
- ♢ Wybór wartości zmiennej
  ⇒ najmniej ograniczającej
- ♢ Sprawdzenie wprzód

Binarny CSP:

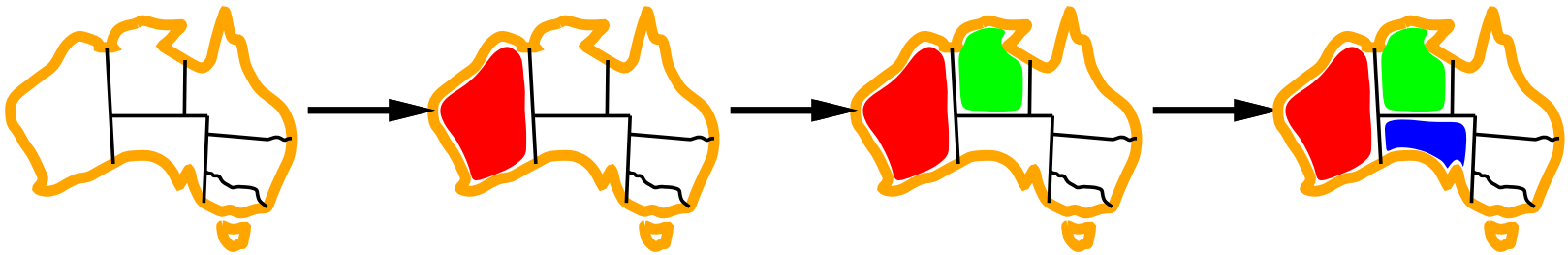
- ♢ Sprawdzenie spójności łukowej
- ♢ Analiza grafu zależności

Wybor zmiennej: najbardziej ograniczona

Wybór zmiennej w kolejnym kroku przeszukiwania:

nabardziej ograniczona zmienna, tzn.

zmienna z najmniejszą liczbą dopuszczalnych wartości

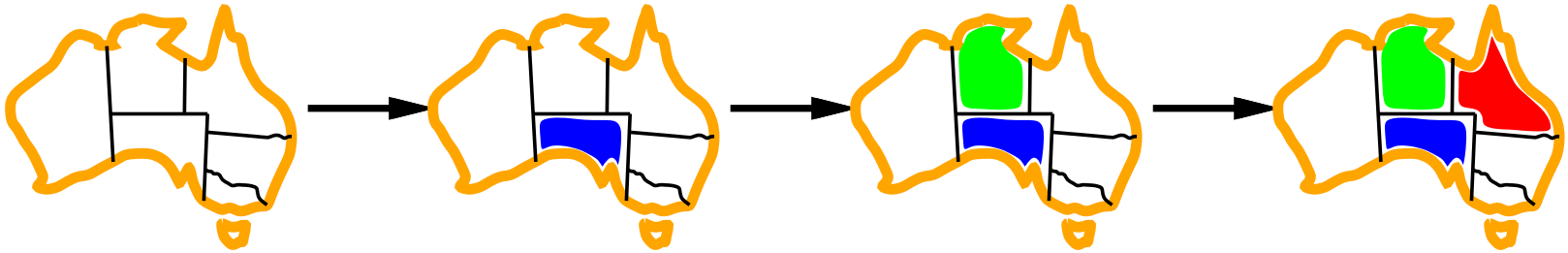


Wybor zmiennej: najbardziej ograniczajaca

Wybór zmiennej spośród najbardziej ograniczonych zmiennych:

nabardziej ograniczająca zmienna, tzn.

zmienna z największą liczbą więzów z pozostałymi zmiennymi

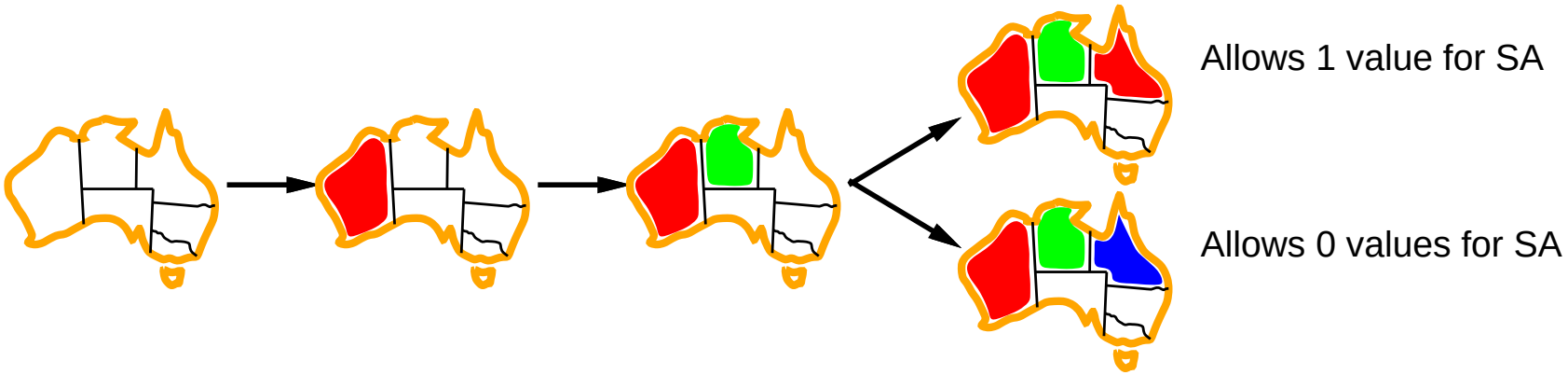


Wybor wartosci: najmniej ograniczajaca

Wybór wartości dla zmiennej wybranej w kolejnym kroku przeszukiwania:

najmniej ograniczająca wartość, tzn.

wartość, która eliminuje najmniej wartości dla pozostałych zmiennych

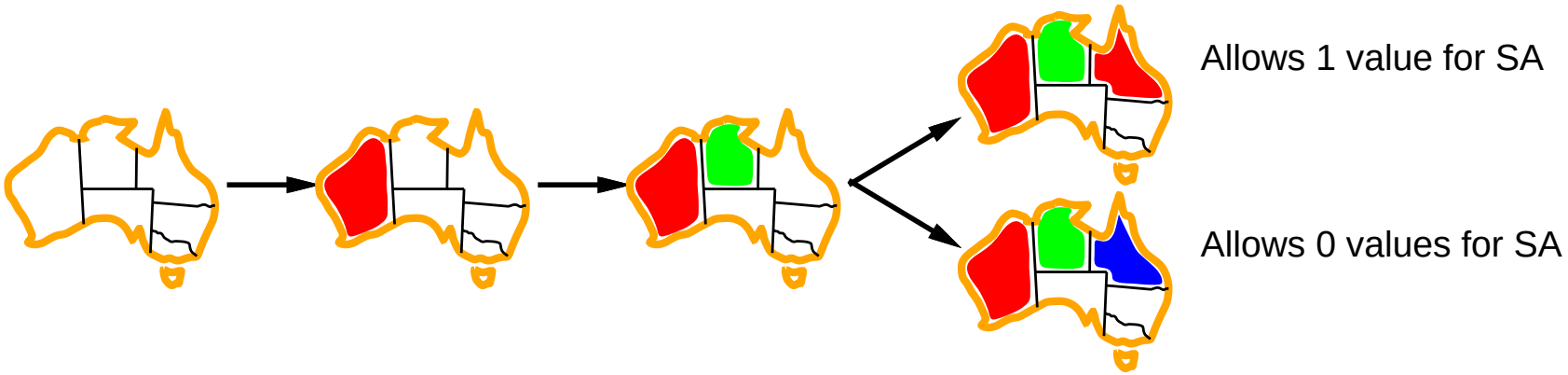


Wybor wartosci: najmniej ograniczajaca

Wybór wartości dla zmiennej wybranej w kolejnym kroku przeszukiwania:

najmniej ograniczająca wartość, tzn.

wartość, która eliminuje najmniej wartości dla pozostałych zmiennych

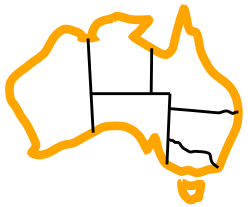


Połączenie heurystyk: wybór najbardziej ograniczającej spośród najbardziej ograniczonych zmiennych oraz wybór najmniej ograniczającej wartości zmiennej umożliwia znalezienie rozwiązania dla problemu 1000-hetmanów

Sprawdzenie wprzod

Pomysł:

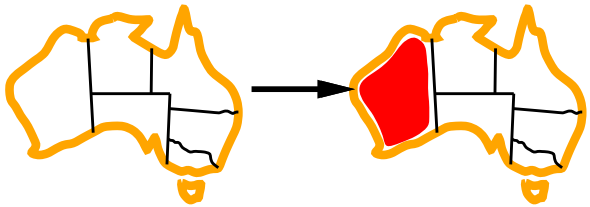
Przeszukiwanie sprawdza, czy dotychczas przypisane zmienne nie eliminują wszystkich wartości dla którejś z nieprzypisanych zmiennych
Cofa się, jeśli któraś zmienna nie ma już żadnej dopuszczalnej wartości



Sprawdzenie wprzod

Pomysł:

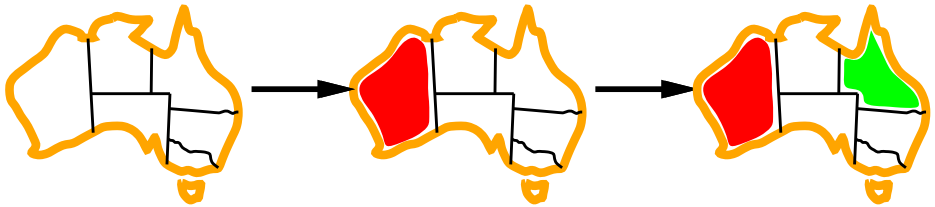
Przeszukiwanie sprawdza, czy dotychczas przypisane zmienne nie eliminują wszystkich wartości dla którejś z nieprzypisanych zmiennych
Cofa się, jeśli któraś zmienna nie ma już żadnej dopuszczalnej wartości



Sprawdzenie wprzod

Pomysł:

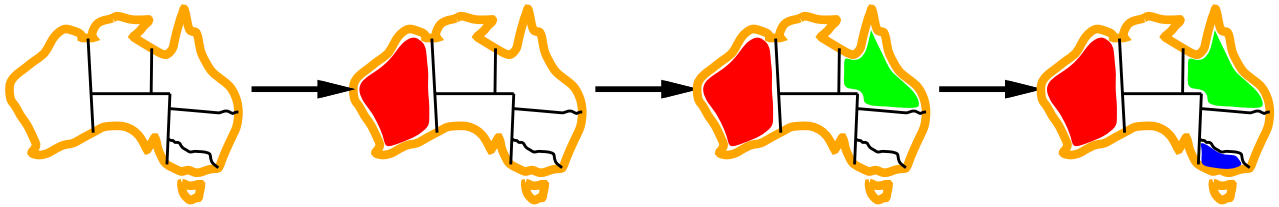
Przeszukiwanie sprawdza, czy dotychczas przypisane zmienne nie eliminują wszystkich wartości dla którejś z nieprzypisanych zmiennych
Cofa się, jeśli któraś zmienna nie ma już żadnej dopuszczalnej wartości

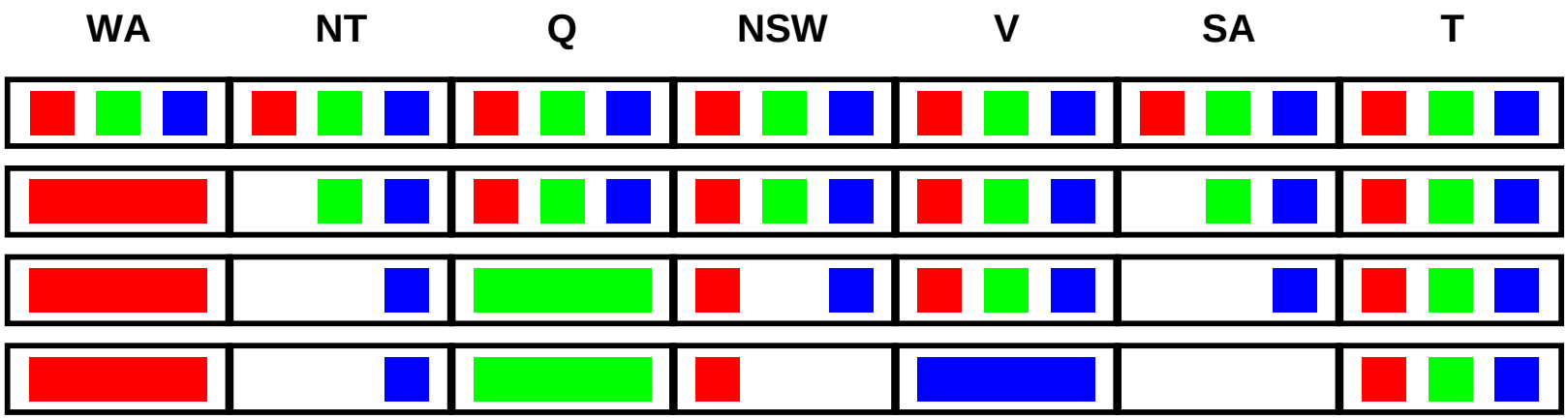


Sprawdzenie wprzod

Pomysł:

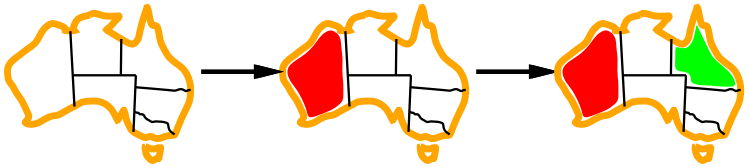
Przeszukiwanie sprawdza, czy dotychczas przypisane zmienne nie eliminują wszystkich wartości dla którejś z nieprzypisanych zmiennych
Cofa się, jeśli któraś zmienna nie ma już żadnej dopuszczalnej wartości





Spojnosc lukowa

Sprawdzenie wprzód propaguje informacje z przypisanych do nieprzypisanych więzów, ale nie od razu wykrywa wszystkie konflikty



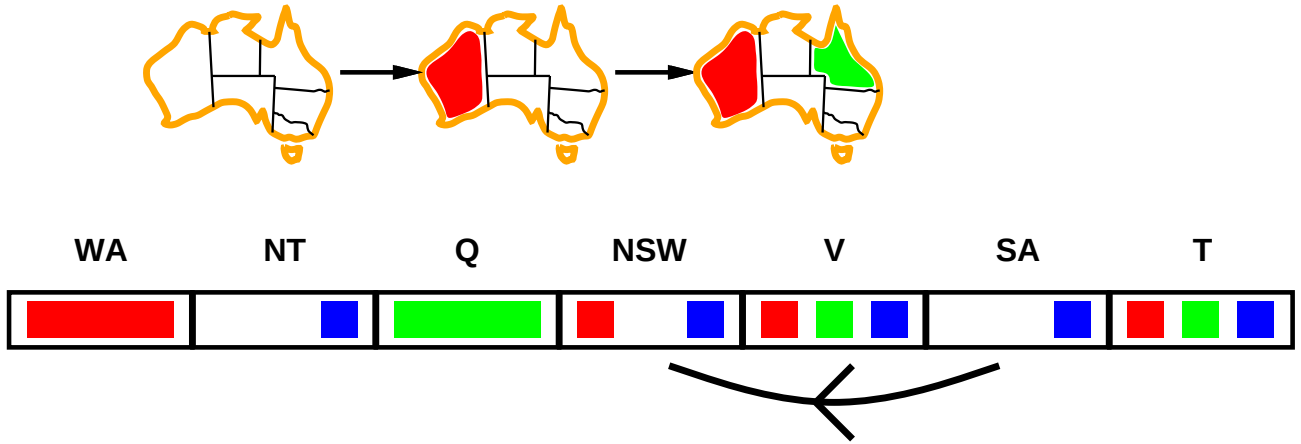
NT i SA nie mogą być oba niebieskie!

Spojność łukowa sprawdza spełnialność więzów lokalnie pomiędzy nieprzypisanymi zmiennymi

Spojnosc lukowa

Spójność łukowa jest określona dla binarnych CSP:
Łuk $X \rightarrow Y$ jest spójny wtw dla *każdej* wartości x na zmiennej X *istnieje* dopuszczalna wartość y na zmiennej Y

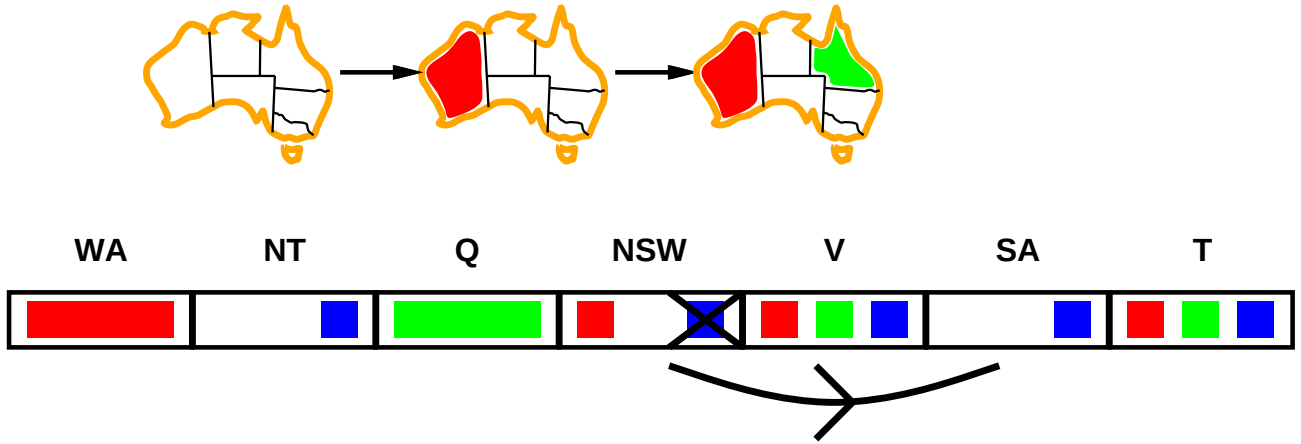
Najprostsza forma propagacji więzów sprawdza, czy każdy łuk jest spójny



Spojnosc lukowa

Spójność łukowa jest określona dla binarnych CSP:
Łuk $X \rightarrow Y$ jest spójny wtw dla *każdej* wartości x na zmiennej X *istnieje* dopuszczalna wartość y na zmiennej Y

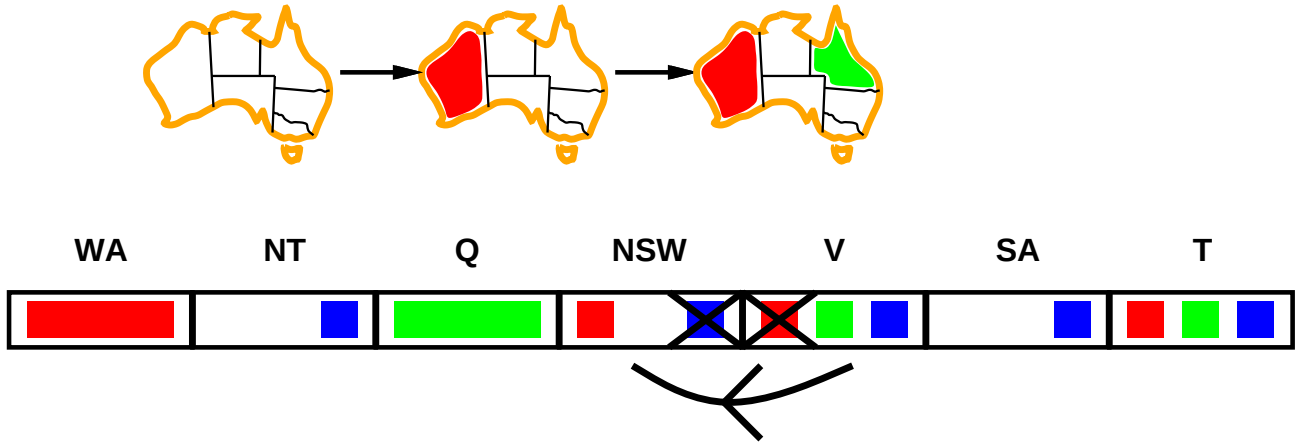
Najprostsza forma propagacji więzów sprawdza, czy każdy łuk jest spójny



Spojnosc lukowa

Spójność łukowa jest określona dla binarnych CSP:
Łuk $X \rightarrow Y$ jest spójny wtw dla *każdej* wartości x na zmiennej X *istnieje* dopuszczalna wartość y na zmiennej Y

Najprostsza forma propagacji więzów sprawdza, czy każdy łuk jest spójny

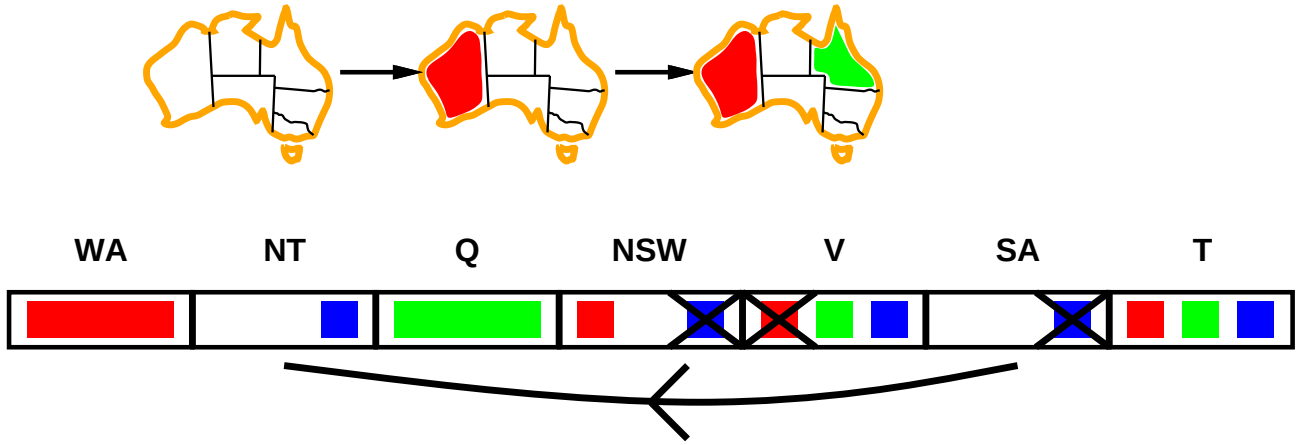


Jeśli X eliminuje pewną wartość, spójność łukowa dla zmiennych związanych z X musi być ponownie sprawdzona

Spojnosc lukowa

Spójność łukowa jest określona dla binarnych CSP:
Łuk $X \rightarrow Y$ jest spójny wtw dla *każdej* wartości x na zmiennej X *istnieje* dopuszczalna wartość y na zmiennej Y

Najprostsza forma propagacji więzów sprawdza, czy każdy łuk jest spójny



Jeśli X eliminuje pewną wartość, spójność łukowa dla zmiennych związanych z X musi być ponownie sprawdzona

Sprawdzenie spójności łukowej wykrywa konflikty wcześniej niż sprawdzanie wprzód

Spojnosc lukowa: algorytm

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains

inputs: *csp*, a binary CSP with variables $\{X_1,\ X_2,\ \ldots,\ X_n\}$

local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

while *queue* is not empty **do**
  $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST(*queue*)
  if REMOVE-INCONSISTENT-VALUES($X_i,\ X_j$) **then**
    for each X_k **in** NEIGHBORS[X_i] **do**
      add $(X_k,\ X_i)$ to *queue*

---

function REMOVE-INCONSISTENT-VALUES($X_i,\ X_j$) **returns** true iff we remove a value

removed $\leftarrow$ *false*
for each x **in** DOMAIN[X_i] **do**
  if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint between X_i and X_j
    then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
return *removed*

Spojnosc lukowa: wlasnosci

Złożoność czasowa??

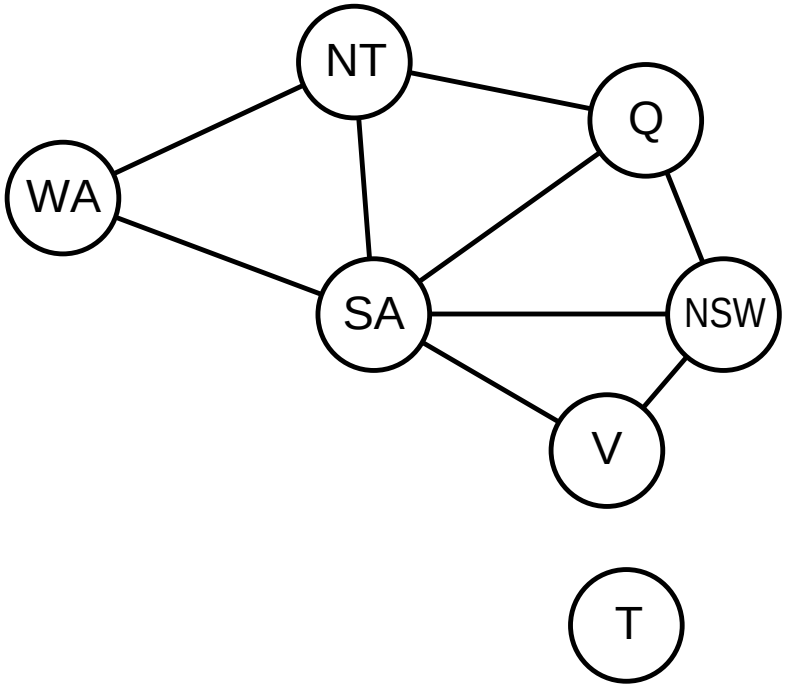
$O(n^2d^3)$, można poprawić do $O(n^2d^2)$

ale nie wykrywa wszystkich konfliktów w czasie wielomianowym!

Użyteczność??

Sprawdzenie może być wykonywane po każdym przypisaniu

Analiza grafu zaleznosci



Tasmania i kontynent są niezależnymi podproblemami

Identyfikowalne jako spójne składowe grafu zależności

Analiza grafu zaleznosci: efektywnosc

Niech każdy podproblem ma c zmiennych z n wszystkich

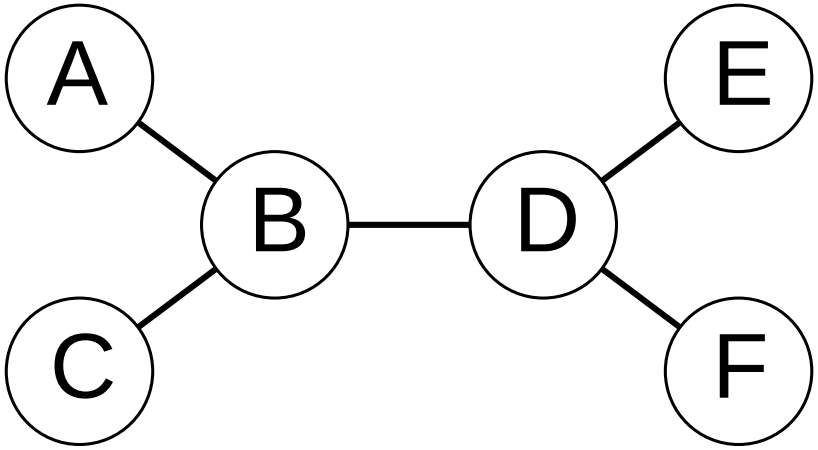
Złożoność czasowa: $n/c \cdot d^c$, *liniowa* od n

Przykład, $n = 80$, $d = 2$, $c = 20$

$2^{80} = 4$ miliardy lat przy szybkości 10 milionów węzłów/sek

$4 \cdot 2^{20} = 0.4$ sekundy przy szybkości 10 milionów węzłów/sek

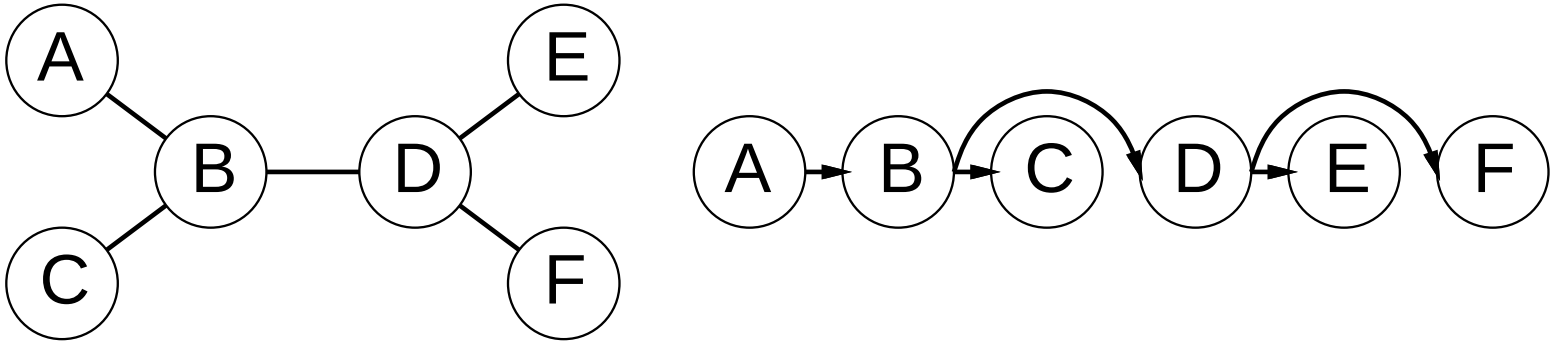
CSP z drzewiastym grafem zaleznosci



Twierdzenie: jeśli graf zależności dla binarnego CSP nie ma cyklu, problem można rozwiązać w czasie $O(n\,d^2)$

CSP z drzewiastym grafem zaleznosci: algorytm

1. Wybiera jedną zmienną jako korzeń drzewa, porządkuje liniowo zmienne tak, że węzły bliżej korzenia poprzedzają dalsze węzły

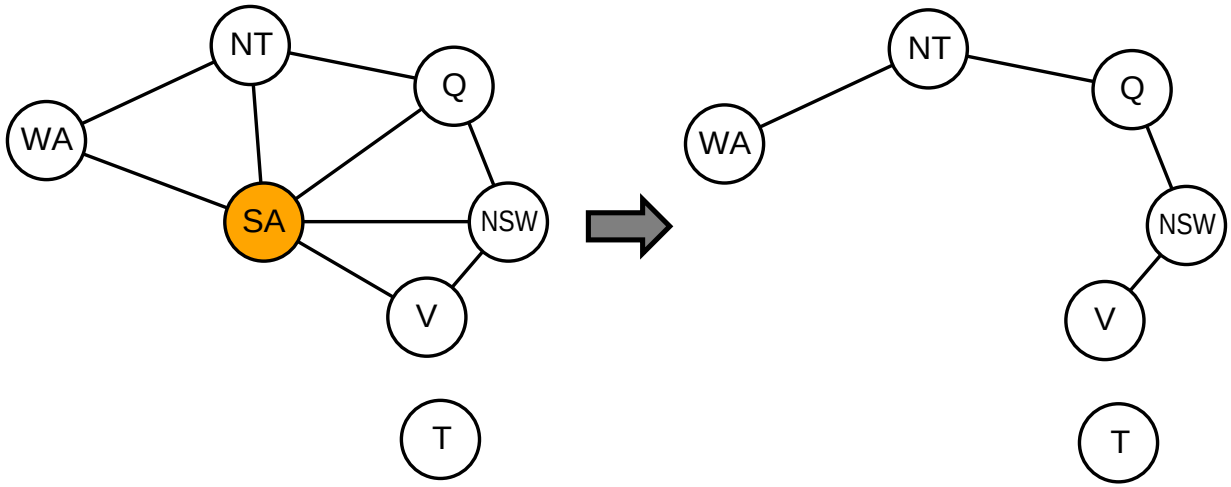


2. Dla j malejącego od n do 2: REMOVEINCONSISTENT$(Parent(X_j), X_j)$

3. Dla j od 1 do n, ustal X_j spójnie z $Parent(X_j)$

CSP z prawie drzewiastym grafem zaleznosci

Redukcja problemu z jedną zmienną "niedrzewiastą": ustala wartość zmiennej niedrzewiastej i ogranicza dziedziny pozostałych zmiennych



Redukcja problemu ze zbiorem zmiennych redukujących
ustala (na wszystkie możliwe sposoby) zbiór zmiennych
takich, że graf zależności dla pozostałych zmiennych jest drzewem

Rozmiar zbioru redukującego $= c$

$\Rightarrow$ złożoność $O(d^c \cdot (n-c)d^2)$, bardzo szybkie dla małych c

Iteracyjne poprawianie

Stany: wszystkie zmienne przypisane, więzy niekoniecznie spełnione

Stan początkowy: dowolny stan z pełnym przypisaniem zmiennych

Funkcja następnika: *zmienia* wartość zmiennej powodującej konflikt więzów w bieżącym stanie

Cel: wszystkie więzy spełnione

Algorytmy: hill-climbing, symulowane wyżarzanie, algorytm genetyczny

Minimalna liczba konfliktow: algorytm

Stan początkowy: losowy lub zachłannie minimalizujący liczbę konfliktów

CONFLICTS = liczba niespełnionych więzów po przypisaniu $var = v$

function MIN-CONFLICTS(csp, $max\text{-}steps$) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    $max\text{-}steps$, the number of steps allowed before giving up
  local variables: $current$, a complete assignment
    var, a variable
    $value$, a value for a variable

  $current \leftarrow$ an initial complete assignment for csp
  for $i = 1$ to $max\text{-}steps$ **do**
    $var \leftarrow$ a randomly chosen, conflicted variable from VARIABLES[csp]
    $value \leftarrow$ the value v for var that minimizes CONFLICTS(var, v, $current$, csp)
    set var=$value$ in $current$
    if $current$ is a solution for csp **then return** $current$
  end
  return $failure$

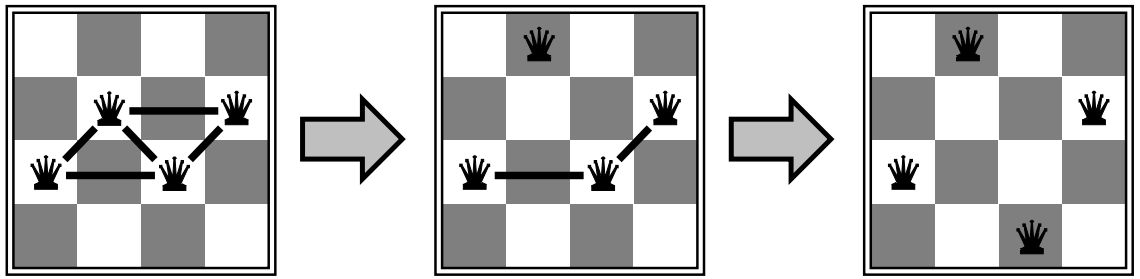
Minimalna liczba konfliktow: 4-hetmanow

Stany: 4 hetmanów w 4 kolumnach ($4^4 = 256$ stanów)

Operacje: przesunięcie hetmana w kolumnie

Cel: brak konfliktów

Ocena stanu: $h(n)$ = liczba konfliktów



Minimalna liczba konfliktow: efektywnosc

Z dużym prawdopodobieństwem rozwiązuje n-hetmanów z losowego stanu w prawie stałym czasie dla dowolnego n (np. $n = 10{,}000{,}000$)

To samo zachodzi dla dowolnego losowo wygenerowanego CSP
z wyjątkiem wąskiego zakresu wielkości

$$R = \frac{\text{liczba więzów}}{\text{liczba zmiennych}}$$