SID – Wykład 2
Przeszukiwanie

Dominik Ślęzak

Wydział Matematyki, Informatyki i Mechaniki UW
slezak@mimuw.edu.pl

Strategie heurystyczne

Strategie heurystyczne korzystają z dodatkowej, heurystycznej funkcji oceny stanu (np. szacującej koszt rozwiązania od bieżącego stanu do celu):

1. Przeszukiwanie pierwszy najlepszy
   - Przeszukiwanie zachłanne
   - Przeszukiwanie A*
2. Iteracyjne poprawianie
   - Przeszukiwanie lokalne zachłanne (hill-climbing)
   - Symulowane wyżarzanie
   - Algorytmy genetyczne

Przeszukiwanie pierwszy najlepszy

Używa funkcji użyteczności i wykonuje ekspansję najbardziej użytecznego stanu spośród wcześniej nieodwiedzonych stanów.
Implementacja: *fringe* jest kolejką porządkującą węzły rosnąco według wartości funkcji użyteczności.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop
        if fringe is empty then
            return failure
        end if
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then
            return node
        end if
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
    end loop
end function
```

Przypadki szczególne: przeszukiwanie zachłanne, przeszukiwanie A*.

Przeszukiwanie zachłanne

Funkcja użyteczności = heurystyczna funkcja oceny stanu $h(n)$
Szacuje koszt rozwiązania z bieżącego stanu n do najbliższego stanu docelowego.
Przeszukiwanie zachłanne wykonuje ekspansję tego węzła, który wydaje się być najbliżej celu.

Przeszukiwanie zachłanne: przykład

$h_{SLD}(n)$ = odległość w linii prostej z miasta n do Bukaresztu

Oradea, Zerind, Arad, Sibiu, Fagaras, Timisoara, Rimnicu Vilcea, Lugoj, Pitesti, Mehadia, Dobreta, Craiova, Bucharest, Giurgiu, Neamt, Iasi, Vaslui, Urziceni, Hirsova, Eforie

71, 75, 151, 140, 118, 99, 80, 211, 111, 97, 70, 146, 101, 75, 138, 120, 85, 90, 87, 92, 142, 98, 86

Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Przykład problemu: Maximum Satisfiability problem (MAX-SAT)

$$(p_{11} \vee p_{12} \vee \ldots \vee p_{1k_1}) \wedge (p_{21} \vee p_{22} \vee \ldots \vee p_{2k_2}) \wedge \ldots \wedge (p_{n1} \vee p_{n2} \vee \ldots \vee p_{nk_n})$$

gdzie p_{ij} są literałami.

- stany: wartościowania zmiennych zdaniowych
- cel: określenie czy (i dla jakiego) wartościowania formuła jest spełniona
- koszt rozwiązania: liczba klauzul niespełnionych

Przeszukiwanie A*

Pomysł: unikać ekspansji stanów, dla których dotychczasowa ścieżka jest już kosztowna.

Funkcja użyteczności: $f(n) = g(n) + h(n)$

$g(n)$ = dotychczasowy koszt dotarcia do stanu n
$h(n)$ = oszacowanie kosztu od stanu bieżącego n do stanu docelowego
$f(n)$ = oszacowanie pełnego kosztu ścieżki od stanu początkowego do celu prowadzącej przez stan n

Heurystyka dopuszczalna

Ogólnie o każdej funkcji heurystycznej $h(n)$ zakłada się, że $h(n) \geq 0$.

Funkcja heurystyczna $h(n)$ jest dopuszczalna, jeśli dla dowolnego stanu n spełniony jest warunek:

$$h(n) \leq h^*(n)$$

gdzie $h^*(n)$ jest rzeczywistym kosztem ścieżki od stanu n do celu.

Problem uproszczony — wersja oryginalnego problemu, dla której koszt rozwiązania jest zawsze nie większy niż koszt rozwiązania problemu oryginalnego.

Heurystyka dopuszczalna może być dokładnym kosztem rozwiązania uproszczonej wersji problemu.

Heurystyka dopuszczalna: najkrótsza droga

Funkcja odległości w linii prostej $h_{SLD}(n)$ jest dopuszczalna — nigdy nie przekracza rzeczywistej odległości drogowej.

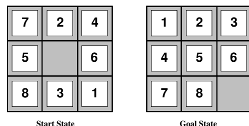
Straight−line distance to Bucharest

City	Distance
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Heurystyki dopuszczalne: 8-elementowe puzzle

Uproszczenie 1: klocek może być przesunięty na dowolne pole:
$h_1(n)$ = liczba klocków nie będących w docelowym położeniu

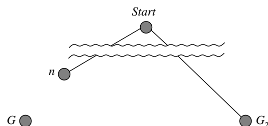
Uproszczenie 2: klocek może być przesunięty na dowolne sąsiednie pole:
$h_2(n)$ = suma odległości miejskiej (ilości ruchów) od docelowych miejsc dla poszczególnych klocków



$h_1(S) = 6$
$h_2(S) = 4+0+3+3+1+0+2+1 = 14$

Heurystyka dopuszczalna: optymalność A*

Twierdzenie: Przeszukiwanie A* bez eliminacji powtarzających się stanów przy użyciu heurystyki dopuszczalnej znajduje zawsze rozwiązanie optymalne.
Dowód: Załóżmy, że stan docelowy G_2 o nieoptymalnym koszcie rozwiązania został wstawiony do kolejki stanów. Niech n będzie dowolnym stanem na najkrótszej ścieżce do optymalnego celu G.



$$\begin{aligned} f(G_2) &= g(G_2) \text{ponieważ } h(G_2) = 0 \\ &> g(G) \text{ ponieważ } G_2 \text{jest nieoptymalny} \\ &\geq f(n) \text{ ponieważ } g(G) = g(n) + h^*(n) \geq g(n) + h(n) = f(n) \end{aligned}$$

$f(G_2) > f(n)$ dla wszystkich n z optymalnej ścieżki, A* wyjmie je przed G_2

Heurystyka spójna

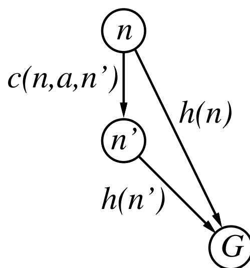
Heurystyka jest spójna jeśli dla każdego stanu n i każdej akcji a z tego stanu:

$$h(n) \leq c(n, a, n') + h(n')$$

gdzie $c(n, a, n')$ jest kosztem wykonania akcji a.

Fakt 1: Każda heurystyka spójna jest dopuszczalna.
Fakt 2: Jeśli heurystyka h jest spójna to ciąg wartości funkcji $f(n)$ wzdłuż dowolnej ścieżki w drzewie przeszukiwań stanów jest niemalejący.

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$



Heurystyka spójna: optymalność A*

Twierdzenie: Przeszukiwanie A* z eliminacją powtarzających się stanów przy użyciu heurystyki spójnej znajduje zawsze rozwiązanie optymalne.

Dowód:
$f(G) = g(G)$ dla każdego stanu docelowego $G \Longrightarrow f(G_{opt}) \leq f(G)$
Z lematu stan docelowy z optymalnym kosztem ścieżki G_{opt} będzie wyjęty z kolejki jako pierwszy spośród wszystkich stanów docelowych G.

Heurystyka dominująca

Niech h_1, h_2 - heurystyki dopuszczalne.

Heurystyka h_2 dominuje heurystykę h_1 jeśli $h_2(n) \geq h_1(n)$ dla wszystkich stanów n.

Twierdzenie: Wszystkie węzły odwiedzone przez algorytm A^* z heurystyką h_2 będą odwiedzone również przez algorytm A^* z heurystyką h_1.

Wniosek: jeśli h_2 dominuje h_1, to opłaca się użyć h_2.

Heurystyka dominująca: przykład

$h_1(n)$ = liczba klocków nie będących w docelowym położeniu
$h_2(n)$ = suma odległości miejskiej (ilości ruchów) od docelowych miejsc dla poszczególnych klocków

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

h_2 dominuje h_1, bo dla każdego stanu n zachodzi $h_2(n) \geq h_1(n)$, np.
$h_1(n) = 6$
$h_2(n) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$

Problemy z więzami (CSP - Constraint satisfaction problem)

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

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

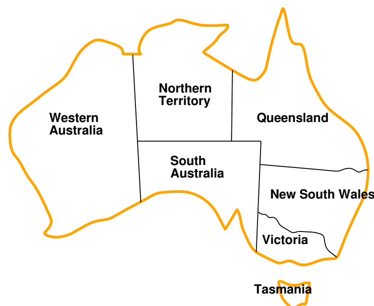
CSP:

- 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.

Przykład CSP: Kolorowanie mapy



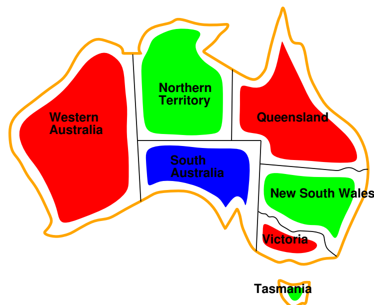
Zmienne: WA, NT, Q, NSW, V, SA, T

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

Więzy: sąsiednie regiony mają mieć różne kolory, np. $WA \neq NT$ (jeśli symbol $\neq$ należy do języka) lub

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

Przykład CSP: Kolorowanie mapy



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

Rodzaje więzów

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 Sudoku

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

Przykład: Sudoku

Zmienne: $X_{1,1}, X_{1,2}, \ldots, X_{9,9}$
Dziedziny: $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Więzy:

- $X_{1,1}, X_{1,2}, \ldots, X_{1,9}$ - wszystkie różne
- $X_{1,1}, X_{2,1}, \ldots, X_{9,1}$ - wszystkie różne
- $X_{1,1}, X_{1,2}, X_{1,3}, X_{2,1}, X_{2,2}, X_{2,3}, X_{3,1}, X_{3,2}, X_{3,3}$ - wszystkie różne
- $\ldots$

$X_{1,1}$	$X_{1,2}$	$X_{1,3}$		$\cdots$			$\cdots$	$X_{1,9}$
$X_{2,1}$	$X_{2,2}$	$X_{2,3}$		$\cdots$			$\cdots$	$X_{2,9}$
$X_{3,1}$	$X_{3,2}$	$X_{3,3}$		$\cdots$			$\cdots$	$X_{3,9}$
	$\cdots$			$\cdots$			$\cdots$	
	$\cdots$			$\cdots$			$\cdots$	
	$\cdots$			$\cdots$			$\cdots$	
$X_{7,1}$	$X_{7,2}$	$X_{7,3}$		$\cdots$			$\cdots$	$X_{7,9}$
$X_{8,1}$	$X_{8,2}$	$X_{8,3}$		$\cdots$			$\cdots$	$X_{8,9}$
$X_{9,1}$	$X_{9,2}$	$X_{9,3}$		$\cdots$			$\cdots$	$X_{9,9}$

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 $\emptyset$

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

Cel: pełne przypisanie zmiennych

Przeszukiwanie przyrostowe z powracaniem

1. Dla problemu z n zmiennymi każde rozwiązanie jest na głębokości n
   $\Longrightarrow$ warto używać przeszukiwania wgłąb
2. Ścieżka jest nieistotna, przypisanie zmiennych jest przemienne, np. [najpierw $WA = red$ potem $NT = green$] to tak samo jak [najpierw $NT = green$ potem $WA = red$]
   $\Longrightarrow$ 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

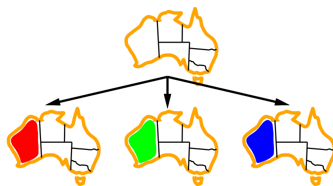
Przeszukiwanie przyrostowe z powracaniem

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING([], csp)
end function
```

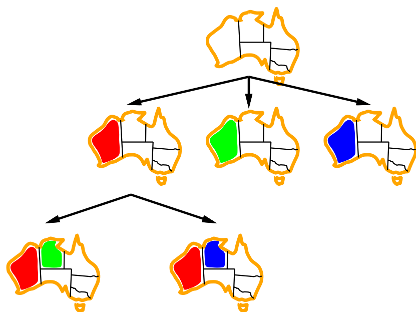
```
function RECURSIVE-BACKTRACKING(assigned, csp) returns a solution, or failure
    if assigned is complete then
        return assigned
    end if
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assigned, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assigned, csp) do
        if value is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

Przeszukiwanie przyrostowe z powracaniem: przykład

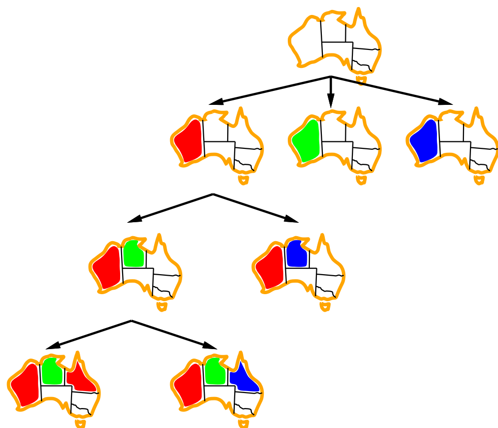
Przeszukiwanie przyrostowe z powracaniem: przykład



Przeszukiwanie przyrostowe z powracaniem: przykład



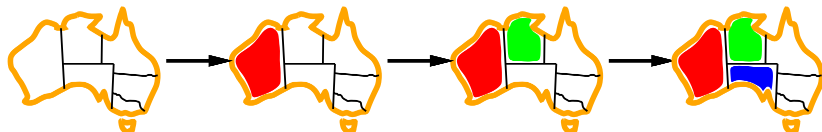
Przeszukiwanie przyrostowe z powracaniem: przykład



Heurystyki przyspieszające

Wybór zmiennej w kolejnym kroku przeszukiwania:

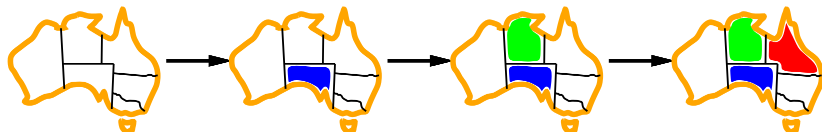
- najbardziej ograniczona zmienna, tzn. zmienna z najmniejszą liczbą dopuszczalnych wartości



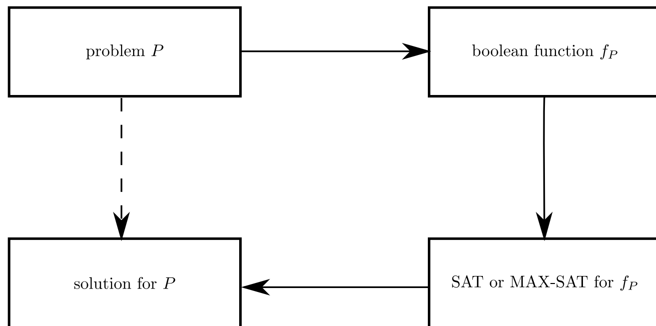
Heurystyki przyspieszające

Wybór zmiennej w kolejnym kroku przeszukiwania:

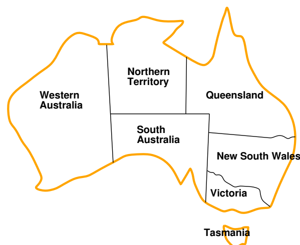
- najbardziej ograniczająca zmienna, tzn. zmienna z największą liczbą więzów z pozostałymi zmiennymi



Sprowadzanie problemu do rozwiązywania SAT



Przykład: Sprowadzanie problemu do rozwiązywania SAT



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

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

Więzy: sąsiednie regiony mają mieć różne kolory, np. $WA \neq NT$

Przykład: Sprowadzanie problemu do rozwiązywania SAT

$$(r_{WA} \wedge \neg g_{WA} \wedge \neg b_{WA} \wedge \neg r_{NT} \wedge \neg r_{SA}) \vee (g_{WA} \wedge \neg r_{WA} \wedge \neg b_{WA} \wedge \neg g_{NT} \wedge \neg g_{SA}) \vee$$
$$(b_{WA} \wedge \neg g_{WA} \wedge \neg r_{WA} \wedge \neg b_{NT} \wedge \neg b_{SA}) \vee \ldots$$

Przykład "SAT solver'a"

Algorytm DPLL (Davis-Putnam-Logemann-Loveland algorithm) – Efektywna metoda zaproponowana w 1962, bazująca na zasadzie algorytmu z powrotami.

- Algorytm wybiera literał i przypisuje mu wartość "*true*" upraszając formułę. astępnie, dla uproszczonej formuły, uruchamiane jest rekurencyjnie sprawdzenie jej spełnialności. Jeśli kończy się ono niepowodzeniem, literałowi przypisywana jest wartość "*false*" i wyliczenie jest powtarzane.
- Algorytm korzysta dodatkowo z dwóch prostych reguł:
  - Unit propagation - jeśli pewna klauzula jest złożona z jednego literału (ang. unit clause), to jej spełnialność poprzez przypisanie wartości zmiennym zdaniowym może być osiągnięta tylko w jeden sposób.
  - Pure literal elimination - jeśli zmienna zdaniowa występuje w literałach formuły tylko w jednej postaci (jako zmienna albo jako jej zaprzeczenie), to można wybrać takie jej wartościowanie, że wszystkie klauzule, które ją zawierają będą spełnione. Klauzule te mogą zostać usunięte z formuły.

Dziękuję za uwagę!