SID – Wykład 3
Przeszukiwanie iteracyjne

Dominik Ślęzak

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

Iteracyjne poprawianie

- Przy wielu problemach optymalizacyjnych ścieżka jest nieistotna: stan docelowy sam w sobie jest rozwiązaniem
- Przestrzeń stanów = zbiór konfiguracji “pełnych”; problem wymaga znalezienia konfiguracji optymalnej lub spełniającej pewne warunki, np. alokacja zasobów w czasie
- Dla tego typu problemów można użyć algorytmu iteracyjnego poprawiania: przechowuje tylko stan “bieżący” i próbuje go poprawić

Fakt: Algorytmy iteracyjnego poprawiania wykonywane są w stałej pamięci

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

Przeszukiwanie lokalne

Przeszukiwanie lokalne zastępuje stan bieżący jednym z jego bezpośrednich sąsiadów

```
function LOCAL-SEARCH(problem) returns a state
    inputs: problem - a problem
    local variables: current - a node
                     best - a node
    current ← MAKE-NODE(INITIAL-STATE[problem])
    best ← current
    repeat
        current ← any successor of current
        if VALUE[current] > VALUE[best] then
            best ← current
        end if
    until best is optimal, or VALUE[best] is high enough, or enough time has elapsed
end function
```

Przykład: przeszukiwanie lokalne - 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.

Ustalamy pewne wartościowanie zmiennych zdaniowych:
$x_1 = \mathit{true}, x_2 = \mathit{false}, x_3 = \mathit{false}, \ldots, x_M = \mathit{true}$
$\equiv$
$\mathbf{x} = (1, 0, 0, \ldots, 1)$

Bezpośredni sąsiedzi to takie wartościowania zmiennych, które różnią się wartoświowaniem dokładnie jednej zmiennej zdaniowej.

Hill-climbing

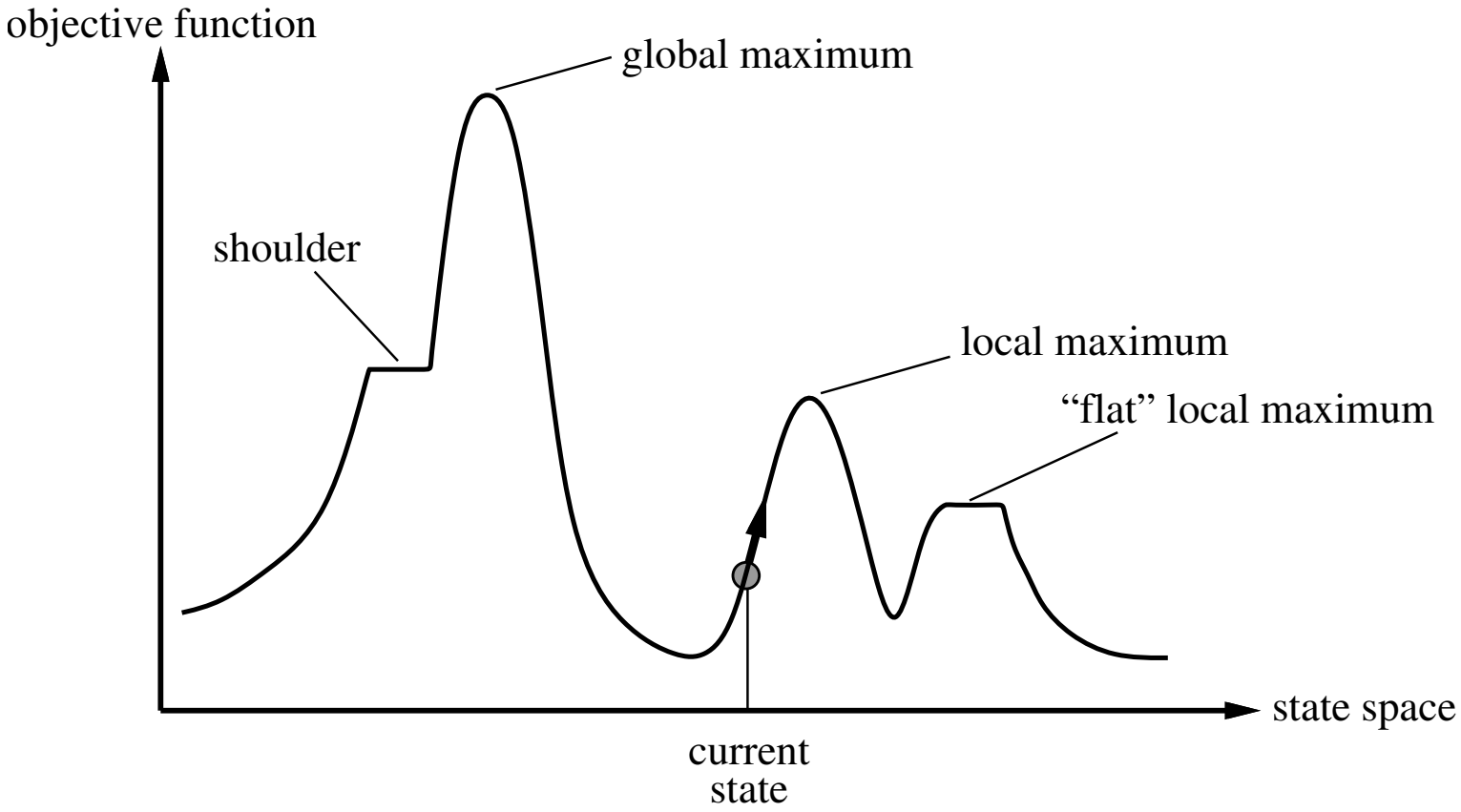
Inaczej przeszukiwanie lokalne zachłanne lub wspinanie wzdłuż gradientu.

Wybiera zawsze sąsiada z największą wartością funkcji oceny, tzn. wyznaczanego przez gradient funkcji.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem - a problem
    local variables: current - a node
                     neighbour - a node
    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop
        neighbour ← a highest-valued successor of current
        if VALUE[neighbour] < VALUE[current] then
            return STATE[current]
        end if
        current ← neighbour
    end loop
end function
```

Hill-climbing: lokalne maksima

Algorytm może “utknąć” w lokalnym maksimum funkcji oceny stanów.



Przykład: hill-climbing (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.
Zaczynamy od pewnego wartościowania zmiennych zdaniowych:
$x_1 = \mathit{true}, x_2 = \mathit{false}, x_3 = \mathit{false}, \ldots, x_M = \mathit{true}$
$\equiv$
$\mathbf{x} = (1, 0, 0, \ldots, 1)$

Spośród bezpośrednich sąsiadów wybieramy tego, dla którego ocena rozwiązania jest największa.

Start wielokrotny

Zaleta: Zwiększa szansę na znalezienie lokalnego maksimum bliskiego optymalnemu rozwiązaniu.

Cena: Wielokrotnie większy koszt czasowy.

```
function MULTISTART-HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem - a problem
    local variables: initial - an initial node in an iteration
                     localmax - the result of a single hill-climbing
                     best - a best node
    for a number of iterations do
        initial ← a random node
        localmax ← HILL-CLIMBING(problem, initial)
        if VALUE[localmax] > VALUE[best] then
            best ← localmax
        end if
    end forreturn best
end function
```

Symulowane wyżarzanie

Pomysł: dopuszcza “złe” posunięcia, ale stopniowo maleje ich częstość wraz z upływem czasu

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem* - a problem
    schedule - a mapping from time to temperature
  local variables: *current* - a node
    next - a node
    T - a “temperature" controlling probability of downward steps
  current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ to ∞ **do**
    $T \leftarrow$ *schedule*$[t]$
    if $T = 0$ **then**
      return *current*
    end if
    next $\leftarrow$ a randomly selected successor of *current*
    $\Delta E =$ VALUE[*next*] - VALUE[*current*]
    if $\Delta E > 0$ **then**
      current $\leftarrow$ *next*
    else
      current $\leftarrow$ *next* only with probability $e^{\Delta E/T}$
    end if
  end for
end function

Symulowane wyżarzanie: funkcja temperatury

Wybór temperatury początkowej: Na początku powinna umożliwiać akceptowanie wszystkich posunięć: $e^{\Delta E/T_0} \approx 1$ Wybór funkcji redukcji temperatury: Redukcja co d kroków ($d \approx$ stopień rozgałęzienia przestrzeni)

- czynnikiem geometrycznym $T := T * r$, $r \in [0.8; 0.99]$
- $T_k = \frac{T_0}{\log_2(k+2)}$, wartość po k-tej redukcji

Symulowane wyżarzanie: własności

Dla stałej wartości "temperatury" T, prawdopodobieństwo osięagnięcia stanów zbiega do rozkładu Boltzmana.

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

jeśli T maleje odpowiednio wolno $\Longrightarrow$ najlepszy stan będzie zawsze osiągnięty.

Opracowanie: Metropolis i inni, 1953, do modelowania procesów fizycznych

Szeroko stosowane m.in. w projektowaniu układów o dużym stopniu scalenia, w planowaniu rozkładu lotów pasażerskich.

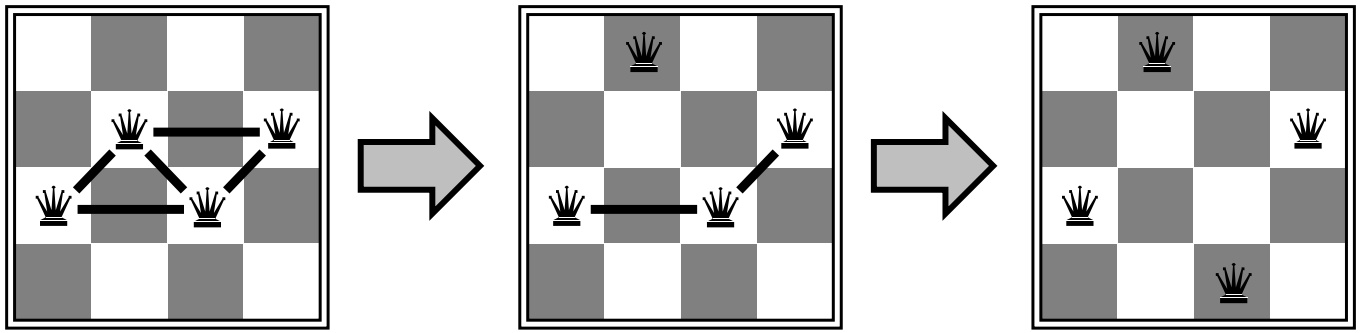
Minimalna liczba konfliktow: algorytm

Stan początkowy: losowy lub zachłannie minimalizujący liczbę konfliktów
CONFLICTS = liczba niespełnionych więzów po przypisaniu $var = v$

```
function MIN-CONFLICTS(csp, max-steps) returns a solution, or failure
    inputs: csp - a constraint satisfaction problem
                  max-steps - the number of steps allowed before giving up
    local variables: current - a complete assignment
                     var - an auxiliary variable
                     value - a value for a variable
    current ← an initial complete assignment for csp
    for i = 1 to max-steps do
        var ← a randomly chosen, conflicted variable from VARIABLES[csp]
        value ← 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 if
    end for
    return failure
end function
```

Minimalna liczba konfliktow: 4-hetmanów

Stany: 4 hetmanów w 4 kolumnach ($4^4 = 256$ stanów)
Operacje: przesunięcie hetmana w kolumnie
Cel: brak konfliktów
Ocena stanu: $h(n)$ = liczba konfliktów

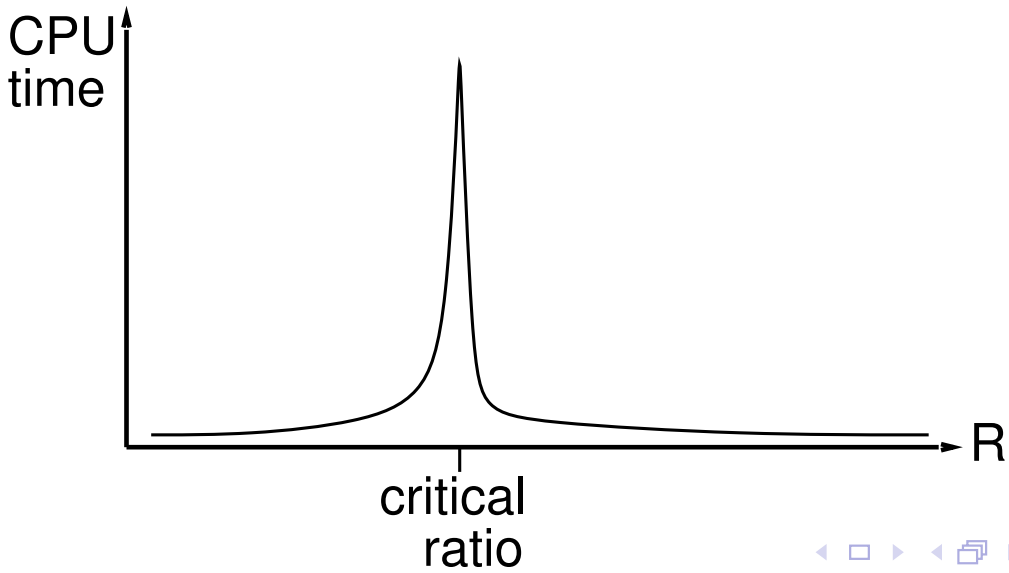


Minimalna liczba konfliktów: efektywność

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

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



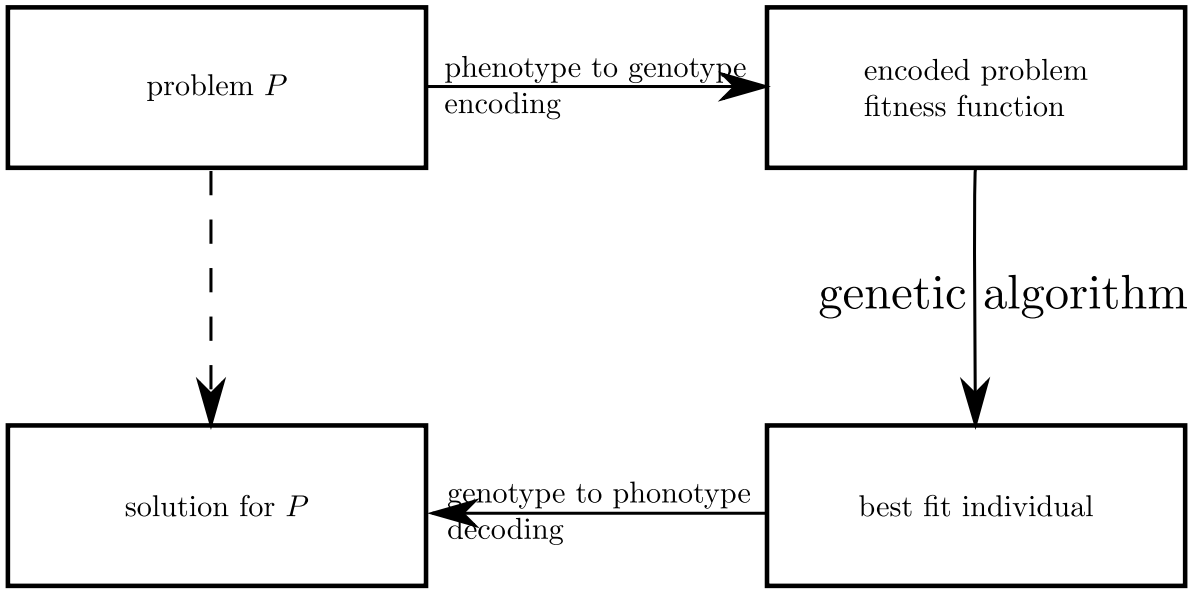
Algorytm genetyczny

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
    inputs: population - a set of individuals
    new_population ← empty set
    repeat
        for i = 1 to size(population) do
            x ← RANDOM-SELECTION(population, FITNESS-FN)
            y ← RANDOM-SELECTION(population, FITNESS-FN)
            child ← REPRODUCE(x, y)
            if (small random probability) then
                child ← MUTATE(child)
            end if
            add child to new_population
        end for
        population ← new_population
    until some individual is fit enough, or enough time has elapsed
    return the best individual in population, according to FITNESS-FN
end function
```

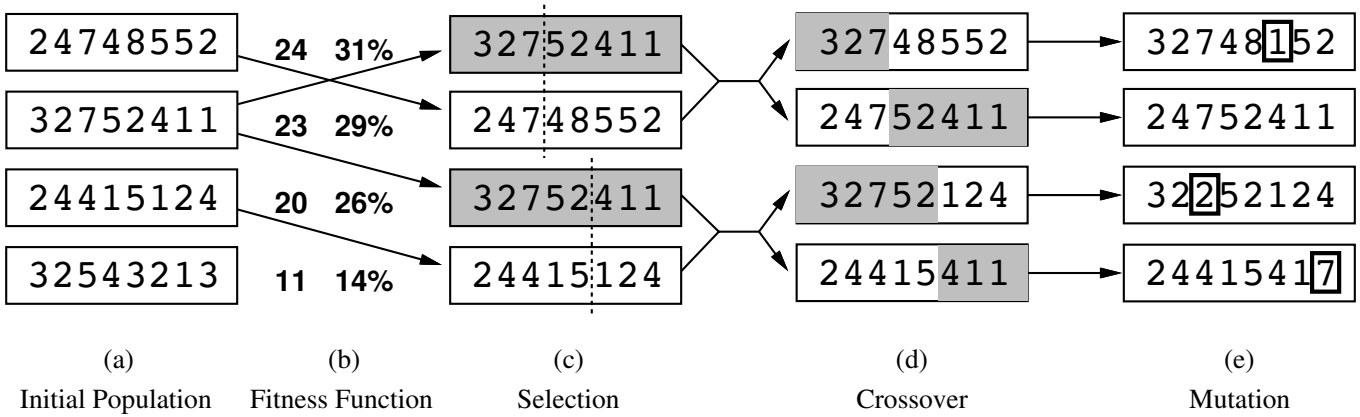
Funkcja REPRODUCE zwraca nowy stan będący losowym skrzyżowaniem (kombinacją) dwóch stanów-rodziców.

Funkcja MUTATE zmienia losowo pojedynczą informację w stanie.

Algorytm genetyczny

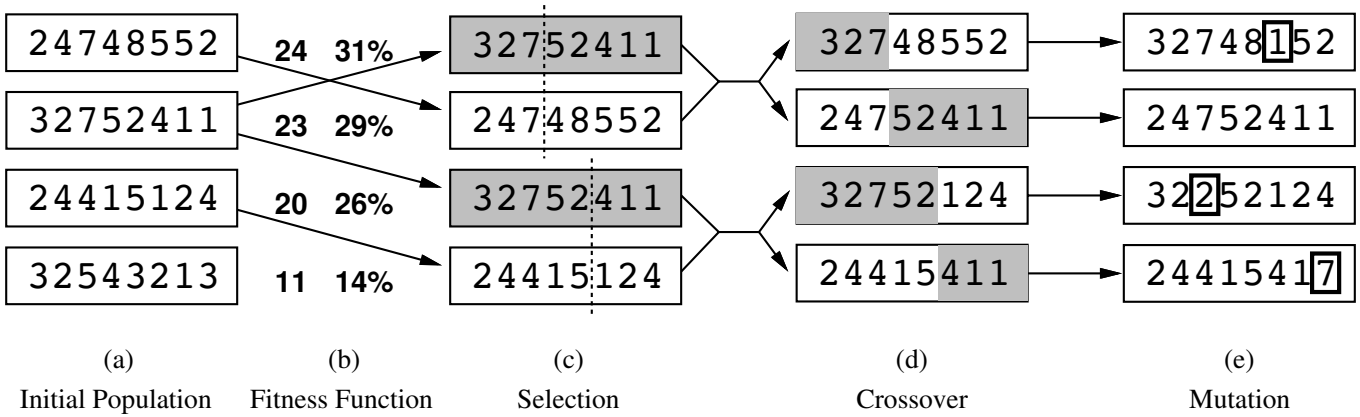


Algorytm genetyczny: kombinowanie i mutacja



function REPRODUCE(x, y) **returns** an individual
  inputs: x, y - parent individuals
  $n \leftarrow$ LENGTH(x)
  $c \leftarrow$ random number from 1 to n
  return SUBSTRING(x, 1, c) + SUBSTRING(y, $c+1$, n)
end function

Algorytm genetyczny: kombinowanie i mutacja



function MUTATE(x) **returns** an individual
  inputs: x - parent individuals
  $n \leftarrow$ LENGTH(x)
  $c \leftarrow$ random number from 1 to n
  mutated_x $\leftarrow x$
  randomly change element on the position c of *mutated_x*
  return *mutated_x*
end function

Przykład: Satisfiability problem (SAT)

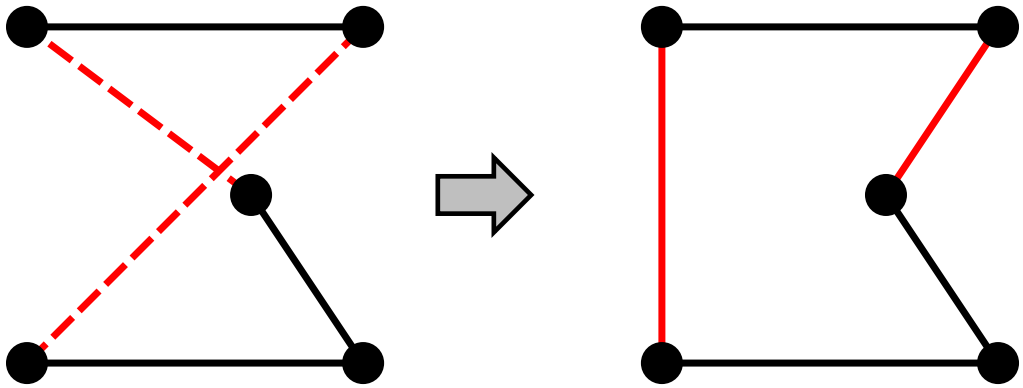
$$(p_{11} \vee p_{12} \vee \ldots \vee p_{1k_1}) \wedge (p_{21} \vee p_{22} \vee \ldots \vee p_{2k_2}) \wedge \ldots \wedge (p_{n1} \vee p_{n2} \vee \ldots \vee p_{nk_n})$$

gdzie p_{ij} są literałami.

Kodowanie problemu za pomocą wektora $0-1$.

Dobrze zdefiniowane operacje krzyżowania i mutacji.

Przykład: problem komiwojażera - TSP



Kodowanie rozwiązania za pomocą permutacji odwiedzonych wierzchołków grafu.

Krzyżowanie permutacji.

Mutacja - zamiana krawędzi.

Metody hybrydowe

Połączenie metod przeszukujących z metodami iteracyjnego poprawiania.

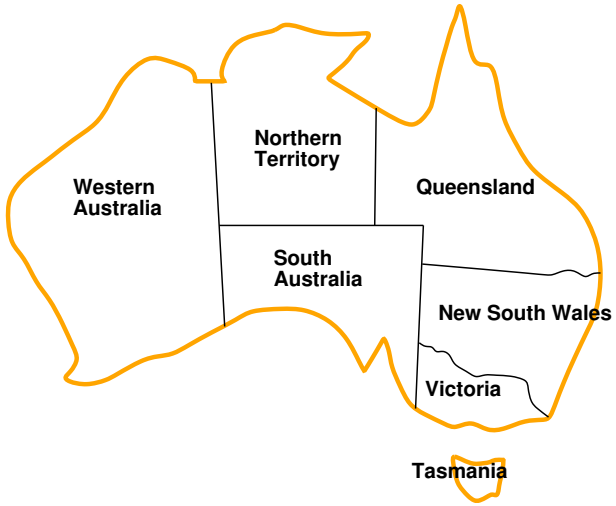
Działamy na pewnej, dobranej do problemu przestrzeni przeszukiwania. Przygotowujemy wejście dla algorytmów przeszukujących, które nie jest bezpośrednim zakodowaniem problemu, ale ułatwia wyliczanie rozwiązania. Za pomocą algortymów iteracyjnego poprawiania przeszukujemy przestrzeń zmodyfikowanego problemu.

Przykład: problem plecakowy (knapsack/rucksack problem)

Dysponujemy plecakiem o maksymalnej pojemności W. Mamy również do dyspozycji N przedmiotów $x_1, x_2, \ldots, x_N$, gdzie każdy element x_i ma określony ciężar w_i oraz wartość v_i. Chcemy wybrać taki podzbiór przedmiotów, aby ich sumaryczna wartość była jak największa ale tak, by mieściły się w plecaku.

Bardziej formalnie:
chcemy zmaksymalizować $\sum_{i=1}^{N} v_i x_i$
przy ograniczeniu $\sum_{i=1}^{N} w_i x_i \leq W$ gdzie $x_i \in \{0,1\}$ dla $i = 1,2,\ldots$

Przykład: Australia



Zmienne: WA, NT, Q, NSW, V, SA, T
Dziedziny: $D_i = \{red, green, blue\}$
Więzy: sąsiednie regiony mają mieć różne kolory, np. $WA \neq NT$

Standardowa metoda przeszukiwania nadaje kolory prowincji według zadanej przez permutację kolejności.

Przykład: 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.

Za pomocą permutacji ustalamy kolejność nadawania wartości zmiennym.

Teoria zbiorów przybliżonych

♢ Zbiory przybliżone (Pawlak, 1981)

♢ Redukty i reguły generowane z reduktów (Skowron, Rauszer, 1992)

$A = \{a_1, \ldots, a_n\}$ – zbiór cech (atrybutów) opisujących przykłady
U_{trn} – zbiór przykładów opisanych wektorami wartości cech $\langle x_1, \ldots, x_n \rangle$

Definicja
Zbiór atrybutów $R \subseteq A$ jest reduktem dla zbioru przykładów U_{trn}, jeśli

- dla każdej pary przykładów $x, y \in U_{trn}$ o różnych decyzjach $dec(x) \neq dec(y)$ istnieje $a_i \in R$ rozróżniający tę parę przykładów: $x_i \neq y_i$
- R jest minimalnym zbiorem mającym powyższą własność, tzn. dla dowolnego $R' \subset R$ istnieje para przykładów w U_{trn} o różnych decyzjach i takich samych wartościach na wszystkich atrybutach $a_i \in R'$

Redukty

Definicja

Redukt R jest minimalny, jeśli zawiera najmniejszą możliwą liczbę atrybutów, tzn. dla każdego reduktu R': $|R| \leq |R'|$

Fakt: Problem znalezienia minimalnego reduktu jest NP-trudny

Przykład:

	a	b	c	d	dec
x_1	0	2	1	0	0
x_2	1	2	2	1	0
x_3	2	0	2	1	1
x_4	0	2	1	1	2

Redukty:
$\{a, d\}, \{b, c, d\}$

Redukty minimalne:
$\{a, d\}$

Generowanie reguł z reduktu

$$Rules(R) := \{ \bigwedge_{a_i \in R} a_i = x_i \implies dec = dec(x) : x \in U_{trn} \}$$

Przykład:

	a	b	c	d	dec
x_1	0	2	1	0	0
x_2	1	2	2	1	0
x_3	2	0	2	1	1
x_4	0	2	1	1	2

Redukt:
$R = \{b, c, d\}$

Reguły:
$b = 2 \wedge c = 1 \wedge d = 0 \implies dec = 0$
$b = 2 \wedge c = 2 \wedge d = 1 \implies dec = 0$
$b = 0 \wedge c = 2 \wedge d = 1 \implies dec = 1$
$b = 2 \wedge c = 1 \wedge d = 1 \implies dec = 2$

Skracanie reguł z reduktu

Skracanie reguły polega na odrzuceniu niektórych selektorów z warunku reguły.

Metoda

Reguła $\alpha \wedge s \implies dec = d$ może zostać zastąpiona przez $\alpha \implies dec = d$, jeśli $\alpha \implies dec = d$ pozostaje spójna ze zbiorem treningowym.

Fakt

Może się zdarzyć, że różne reguły z tą samą decyzją zostaną skrócone do tej samej postaci

$\implies$ zbiór reguł po skróceniu może być mniejszy niż oryginalny.

Skracanie reguł z reduktu: przykład

	a	b	c	d	dec
x_1	0	2	1	0	0
x_2	1	2	2	1	0
x_3	2	0	2	1	1
x_4	0	2	1	1	2

$b = 2 \wedge c = 1 \wedge d = 0 \implies dec = 0$
$b = 2 \wedge c = 2 \wedge d = 1 \implies dec = 0$
$b = 0 \wedge c = 2 \wedge d = 1 \implies dec = 1$
$b = 2 \wedge c = 1 \wedge d = 1 \implies dec = 2$

Po skróceniu:
$b = 2 \wedge d = 0 \implies dec = 0$ lub $c = 1 \wedge d = 0 \implies dec = 0$
$b = 2 \wedge c = 2 \implies dec = 0$
$b = 0 \implies dec = 1$
$c = 1 \wedge d = 1 \implies dec = 2$

Klasyfikacja oparta na wsparciu

rules – zbiór reguł z jednoznaczną decyzją
U_{trn} – zbiór przykładów treningowych

x - obiekt do klasyfikacji

Klasyfikacja przez maksymalizację wsparcia:

$$rules(x) = \{\alpha \implies d \in rules : x \text{ spełnia } \alpha\}$$

$$\max \arg_d |\{y \in U_{trn} : \exists \alpha \implies d \in rules(x)\,(y \text{ spelnia } \alpha \wedge dec(y) = d)\}|$$

Redukty lokalne

Zbiór atrybutów $R \subseteq A$ jest reduktem lokalnym dla przykładu $x \in U_{trn}$ w zbiorze przykładów U_{trn}, jeśli

- dla każdego przykładu $y \in U_{trn}$ z inną decyzją $dec(y) \neq dec(x)$ istnieje $a_i \in R$ rozróżniający x od y: $x_i \neq y_i$
- R jest minimalnym zbiorem mającym powyższą własność, tzn. dla dowolnego $R' \subset R$ istnieje przykład w U_{trn} z inną decyzją i wartościami taki samymi jak x na wszystkich atrybutach $a_i \in R'$

Fakt 1:
Liczba reduktów lokalnych dla jednego przykładu może być wykładnicza względem liczby atrybutów i liczby przykładów treningowych

Fakt 2:
Problem znalezienia minimalnego reduktu lokalnego dla danego przykładu jest NP-trudny

Generowanie reguł z reduktów lokalnych

Reguła generowana z reduktu lokalnego R dla przykładu x:

$$\bigwedge_{a_i \in R} a_i = x_i \implies dec = dec(x)$$

Fakt 1: Reguła generowana z reduktu lokalnego jest reguła spójną minimalną (tzn. usunięcie któregokolwiek selektora powoduje utratę spójności)

Fakt 2: Zbiór reguł wygenerowanych ze wszystkich reduktów lokalnych = zbiór wszystkich minimalnych reguł spójnych = zbiór wszystkich reguł generowanych przez algorytm zupełny (z selektorami równościowymi)

Przypomnienie: Liczba wszystkich minimalnych reguł spójnych może być wykładnicza względem liczby atrybutów i przykładów treningowych

Fakt 3 (Bazan, 1998): Niech $rules_{all}$ – zbiór wszystkich minimalnych reguł spójnych. Istnieje algorytm symulujący klasyfikację z maksymalizacją wsparcia w zbiorze reguł $rules_{all}$ (bez jawnego liczenia reguł) wykonujący klasyfikację pojedynczego obiektu w czasie $O(|U_{trn}|^2|A|)$.

Dziękuję za uwagę!