Algorytmy i struktury danych

Metody konstrukcji algorytmów

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Algorytmy zachłanne

- zarówno algorytm Prima jak i Kruskala są przykładami algorytmów zachłannych
- jest to klasa algorytmów, które w każdym kroku wybierają lokalnie najlepsze rozwiązanie (np. krawędź o najmniejszej wadze)
- prawie dla każdego problemu jesteśmy w stanie skonstruować algorytm zachłanny, jednak nie zawsze jest to najlepsza strategia — algorytm zachłanny nie musi być algorytmem optymalnym
- np. szukanie najkrótszej ścieżki metodą zachłanną może nie dać najlepszego rozwiązania

Divide and Conquer

- technika Divide and Conquer ("dziel i rządź" / "dziel i zwyciężaj") polega na podziale problemu na mniejsze podproblemy (prostsze do rozwiązania), rozwiązaniu ich i scaleniu wyniku
- przykładami algorytmów stosujących tę technikę są algorytmy sortowania przez scalanie i Quicksort

Programowanie dynamiczne

- rozwiązujemy problem poprzez złożenie rozwiązań odpowiednich podproblemów
- każdy podproblem rozwiązujemy tylko raz, rozwiązanie zapamiętując w tabeli
- zazwyczaj stosowane do rozwiązywania problemów optymalizacyjnych: przy danych ograniczeniach mamy zmaksymalizować lub zminimalizować pewien koszt

Programowanie dynamiczne

- podczas rozwiązywania możemy wyszczególnić cztery etapy:
- scharakteryzowanie struktury optymalnego rozwiązania
- rekurencyjne rozwiązanie kosztu optymalnego rozwiązania
- obliczenie optymalnego kosztu metodą wstępującą – rozpoczynając od najmniejszych podproblemów, rozwiązujemy coraz większe, wykorzystując zapamiętane rozwiązania
- konstruowanie optymalnego rozwiązania na podstawie wyników wcześniejszych obliczeń

Programowanie dynamiczne

- jeżeli interesuje nas sam koszt rozwiązania optymalnego, krok czwarty można pominąć
- w przeciwnym przypadku, często opłaca się zapamiętywać dodatkowe informacje podczas wykonywania kroku trzeciego

Przykład — liczby Fibonacciego

- “rozwiążemy” liczby Fibonacciego

Przykład — liczby Fibonacciego

- “rozwiążemy” liczby Fibonacciego
- 1. scharakteryzowanie struktury optymalnego rozwiązania
- 2. rekurencyjne rozwiązanie kosztu optymalnego rozwiązania

Przykład — liczby Fibonacciego

- “rozwiążemy” liczby Fibonacciego
- 1. scharakteryzowanie struktury optymalnego rozwiązania
- 2. rekurencyjne rozwiązanie kosztu optymalnego rozwiązania
- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$

Przykład — liczby Fibonacciego

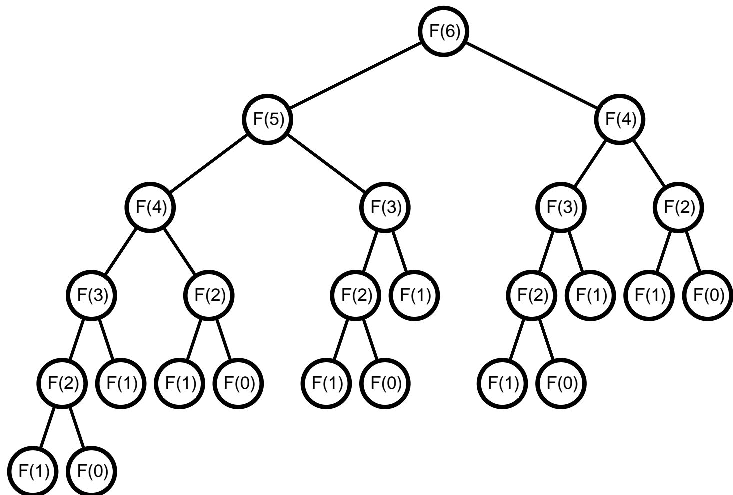
- 3. obliczenie optymalnego kosztu metodą wstępującą

Przykład — liczby Fibonacciego

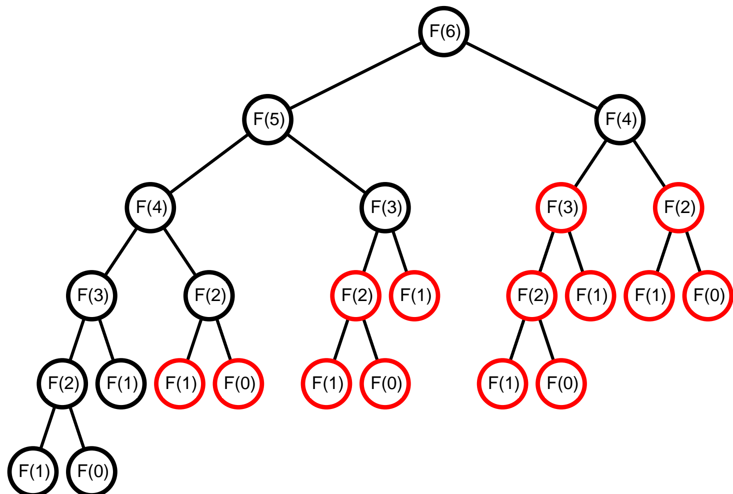
- 3. obliczenie optymalnego kosztu metodą wstępującą
- 3a. rekurencyjna implementacja

```
void FibRek(int n) {
     if(n < 2) return n;
     return FibRek(n - 1) + FibRek(n - 2);
     };
```

Przykład — liczby Fibonacciego



Przykład — liczby Fibonacciego



Przykład — liczby Fibonacciego

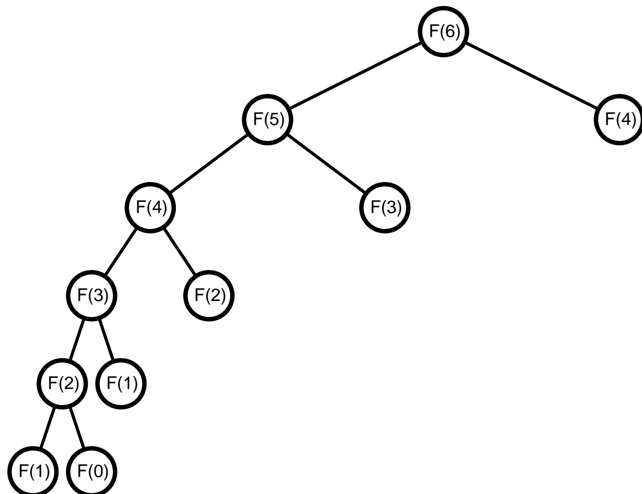
- 3. obliczenie optymalnego kosztu metodą wstępującą
- 3b. spamiętywanie rozwiązań cząstkowych

Przykład — liczby Fibonacciego

- 3. obliczenie optymalnego kosztu metodą wstępującą
- 3b. spamiętywanie rozwiązań cząstkowych

```
int bufor[...]  = { -1, -1, -1, ..., -1 };
...
void FibRek2(int n) {
     if(n < 2) return n;
     if(bufor[n] != -1) return bufor[n];
     return (bufor[n] = FibRek(n - 1) + FibRek(n - 2));
     };
```

Przykład — liczby Fibonacciego



Przykład — liczby Fibonacciego

- 3. obliczenie optymalnego kosztu metodą wstępującą
- 3d. obliczenie optymalnego kosztu metodą wstępującą

Przykład — liczby Fibonacciego

- 3. obliczenie optymalnego kosztu metodą wstępującą
- 3d. obliczenie optymalnego kosztu metodą wstępującą

```
int bufor[...]  = { -1, -1, -1, ..., -1 };
...
void Fib3(int n) {
     bufor[0] = 0;
     bufor[1] = 1;
     int i = 2;
     while(i <= n) {
          bufor[i] = bufor[i - 1] + bufor[i - 2];
          i++;
          };
     return bufor[n];
     };
```

Przykład — liczby Fibonacciego

- 3. obliczenie optymalnego kosztu metodą wstępującą
- 3e. (opcjonalne) redukcja zużytej pamięci

Przykład — liczby Fibonacciego

- 3. obliczenie optymalnego kosztu metodą wstępującą
- 3e. (opcjonalne) redukcja zużytej pamięci

```
void Fib4(int n) {
     int bprv = 0, bcur = 0, i = 0;
     while(i < n) {
          int t = bcur + bprv;
          bprv = bcur;
          bcur = t;
          i++;
          };
     return bcur;
     };
```

Problem plecakowy

- mamy plecak o określonej pojemności W oraz zestaw n przedmiotów, które możemy do tego plecaka włożyć
- każdy z przedmiotów ma określoną:
  - wagę w_i,
  - cenę c_i
- chcemy do plecaka zapakować przedmioty o jak największej łącznej wartości

Problem plecakowy

- gdy przedmioty są podzielne (np. są płynami) problem plecakowy można rozwiązać optymalnie metodą zachłanną
- wybieramy przedmioty o największym stosunku ceny do wagi ($\frac{c_i}{w_i}$) i wkładamy do plecaka

Problem plecakowy – podejście zachłanne

- wybieramy przedmioty w kolejności malejącego zysku na jednostkę masy
- kontrprzykład:
  $W = 4,\ w_1 = 3,\ c_1 = 5,\ w_2 = w_3 = 2,\ c_2 = c_3 = 3$
- aby znaleźć optymalne rozwiązanie zastosujemy tu programowanie dynamiczne

Problem plecakowy

- celem naszym jest maksymalizacja $\sum_{i=1}^{n} d_i c_i$ przy zachowaniu ograniczenia $\sum_{i=1}^{n} d_i w_i \leq W$, gdzie d_i to zmienna decydująca, czy przedmiot i wkładamy ($d_i = 1$) bądź nie ($d_i = 0$) do plecaka

Problem plecakowy

- załóżmy, że znamy optymalne rozwiązanie dla danej instancji problemu
- weźmy jeden z przedmiotów, który trafił do plecaka (niech będzie to przedmiot k)
- pozostała zawartość plecaka jest optymalnym rozwiązaniem problemu plecakowego dla pojemności plecaka $W - w_k$ i zestawu przedmiotów $\{1, 2, \ldots, k-1, k+1, k+2, \ldots n\}$
- gdyby było inaczej (nie byłoby to optymalne rozwiązanie), to zastępując przedmioty w pozostałej części plecaka tymi z optymalnego rozwiązania podproblemu, otrzymalibyśmy rozwiązanie lepsze niż optymalne

Problem plecakowy

- zdefiniujmy rekurencyjnie koszt optymalnego rozwiązania:
- koszt rozwiązania problemu dla wagi W i zestawu przedmiotów $\{1, 2, \ldots, j\}$ to:

$$K(j, W) = \begin{cases} 0 \text{ dla } j = 0 \\ 0 \text{ dla } W = 0 \\ max(K(j-1, W - w_j) + c_j, K(j-1, W)) \\ \qquad \text{o ile } w_j \leq W \\ K(j-1, W) \text{ w p.p.} \end{cases}$$

Obliczanie kosztu metodą wstępującą

- znamy koszty rozwiązań dla przypadków brzegowych ($W = 0$ lub pusty zbiór przedmiotów)
- dzięki przedstawionemu wzorowi możemy kolejno wyznaczać rozwiązania większych podproblemów, aż dojdziemy do rozwiązania całego problemu (wyznaczymy $K(n, W)$)

Problem plecakowy

K - tablica o wymiarach $n+1 \times W+1$

for $i = 0, \ldots, n$ **do** $K[i, 0] = 0$
for $i = 0, \ldots, W$ **do** $K[0, i] = 0$
for $i = 1, \ldots n$ **do**
  for $j = 1, \ldots W$ **do**
    if $w_j \leq W$ **then**
      $K[i,j] = max(K[i-1, j-w_i] + c_i, K[i-1, j])$
    else
      $K[i,j] = K[i-1, j]$
    end if
  end for
end for
return $K[n, W]$

Problem plecakowy

- rozmiar plecaka $W = 10$
- przedmioty:
  $w_1 = 5, c_1 = 1,$
  $w_2 = 3, c_2 = 2,$
  $w_3 = 3, c_3 = 1,$
  $w_4 = 4, c_4 = 4,$
  $w_5 = 1, c_5 = 1$

Pakowanie plecaka

W

i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										
5	0										

Pakowanie plecaka

W

i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0										
3	0										
4	0										
5	0										

Pakowanie plecaka

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0										
4	0										
5	0										

Pakowanie plecaka

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0										
5	0										

Pakowanie plecaka

W

i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0										

Pakowanie plecaka

W

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

Problem plecakowy

- złożoność algorytmu wynosi $O(nW)$ (nie jest wielomianowa!)
- wyznaczyliśmy wartość plecaka, jak poznać jego zawartość?

Problem plecakowy

- zaczynamy od $K[n, W]$
- porównujemy $K[n, W]$ z $K[n-1, W]$ oraz $K[n-1, W-w_n]+c_n$
- jeżeli $K[n, W] = K[n-1, W]$ to nie włożyliśmy przedmiotu n do plecaka, przechodzimy do $K[n-1, W]$ i powtarzamy operację
- jeżeli $K[n, W] = K[n-1, W-w_n]+c_n$ to włożyliśmy przedmiot n do plecaka, przechodzimy do $K[n-1, W-w_n]$ i powtarzamy operację

Pakowanie plecaka

W

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

Pakowanie plecaka

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

Pakowanie plecaka

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

Pakowanie plecaka

W

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

Pakowanie plecaka

W

i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

Pakowanie plecaka

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	2	2	2	2	2	3	3	3
3	0	0	0	2	2	2	3	3	3	3	3
4	0	0	0	2	4	4	4	6	6	6	7
5	0	1	1	2	4	5	5	5	7	7	7

Problem plecakowy

- jeżeli interesuje nas sama wartość rozwiązania, zamiast tablicy o wymiarach $n+1 \times W+1$ wystarczy nam tablica o wymiarach $2 \times W+1$
- w i-tym kroku wystarczy pamiętać tylko poprzedni wiersz tablicy

Problem plecakowy

K - tablica o wymiarach $2 \times W + 1$
for $i = 0, \ldots, W$ **do** $K[0, W] = 0$
for $i = 1, \ldots n$ **do**
  for $j = 1, \ldots W$ **do**
    $K[i\%2, j] = max(K[1-(i\%2), j-w_i] + c_i, K[1-(i\%2), j])$
    if $w_i \leq W$ **then**
      $K[i\%2, j] =$
      $max(K[1-(i\%2), j-w_i] + c_i, K[1-(i\%2), j])$
    else
      $K[i\%2, j] = K[1-(i\%2), j]$
    end if
  end for
end for
return $K[n\%2, W]$

Problem plecakowy

- dlaczego nie wystarczy tablica jednowierszowa?
- musimy znać wartość $K[i-1, j-w_i]$
- mając tylko jeden wiersz tablicy, obliczając $K[i,j]$ w $K[i, j-w_i]$ będzie optymalne rozwiązanie dla podproblemu o rozmiarze plecaka $j-w_i$ i przedmiotów $\{1, 2, \ldots, i\}$ (a nie $\{1, 2, \ldots, i-1\}$)
- jeżeli optymalne rozwiązanie podproblemu będzie wymagało włożenia przedmiotu i do plecaka, może się zdarzyć, że przedmiot ten włożymy do plecaka wielokrotnie
- chyba, że będziemy wypełniać tablicę w przeciwnym kierunku...

Alogrytm Floyda-Warshalla

- szukamy najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków
- opisujemy graf macierzą W:
  - $W[v, v] = 0$
  - $W[u, v] =$ długość krawędzi między u a v jeżeli sąsiadują ze sobą
  - $W[u, v] = \infty$ w przeciwnym wypadku
- wyznaczamy macierz odległości D; początkowo $D[v, v] = W[u, v]$
- graf nie może zawierać cykli o ujemnej wadze, ale może zawierać krawędzie o ujemnej długości

Alogrytm Floyda-Warshalla

- k-ty krok: rozważamy tylko ścieżki przechodządze przez wierzchołki $1 \dots k$
- dołączając wierzchołek k do zbioru wierzchołków dozwolonych w ścieżkach sprawdzamy, czy istnieje ścieżka z u do v przez k krótsza od znalezionej do tej pory, czyli czy $D[u,k] + D[k,v] < D[u,v]$
- jeżeli tak, aktualizujemy macierz D
- wykonujemy $O(n^3)$ kroków

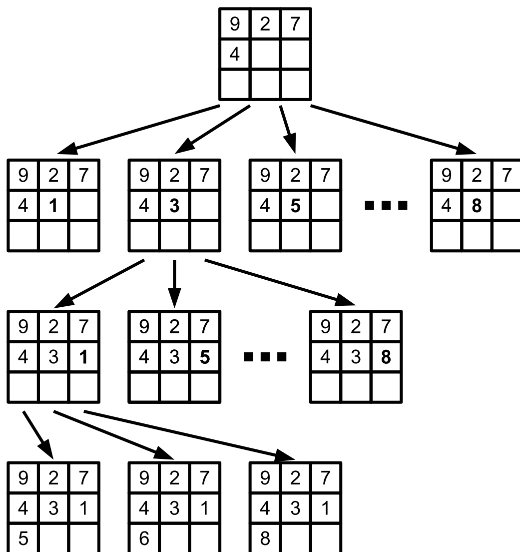
Alogrytm Floyda-Warshalla

```
1:  for i, j = 1 to n do
2:          D[i, j] = W[i, j]
3:  end for
4:  for k = 1 to n do
5:          for 1 ≤ i, j ≤ n, i, j ≠ k do
6:                  if D[i, j] > D[i, k] + D[k, j] then
7:                          D[i, j] = D[i, k] + D[k, j]
8:                  end if
9:          end for
10: end for
```

Backtracking

- budujemy rozwiązanie krok po kroku
- na każdym poziomie przeglądamy wszystkie (rozsądne) możliwości
- gdy dotrzemy do sprzeczności/niepoprawnego rozwiązania — przerywamy ("wycofujemy się")
- gdy dotrzemy do poprawnego rozwiązania przerywamy algorytm (gdy interesuje nas dowolne) lub aktualizujemy najlepsze (gdy szukamy ekstremum)
- zazwyczaj algorytm rekurencyjny

Backtracking



Backtracking

- sposób konstrukcji rozwiązania ma istotne znaczenie dla czasu działania algorytmu
- staramy się jak najszybciej (jak najwyżej w drzewie) wykrywać niepoprawne rozwiązania
- staramy się ograniczać rozgałęzianie drzewa

Branch-and-bound

- usprawnienie backtrackingu przez odrzucenie “nierokującej” części drzewa
- optymalizujemy wartość funkcji
- pamiętamy najlepsze dotychczasowe rozwiązanie
- potrafimy oszacować z dołu (przy minimalizacji) lub z góry (przy maksymalizacji) wartość całego rozwiązania mając jego fragment
- jeżeli w węźle oszacowanie jest gorsze od najlepszego rozwiązania pomijamy poddrzewo

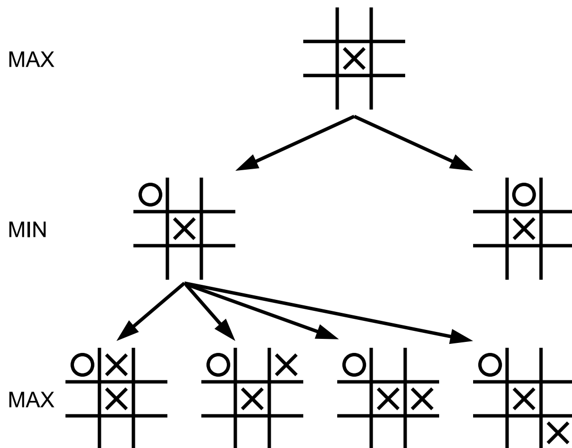
Min-Max

- dwóch graczy
- gra o sumie zerowej (zysk jednego gracza jest stratą drugiego)
- pierwszy gracz maksymalizuje wynik gry (np. $1 =$ wygrana gracza pierwszego, $0 =$ remis, $-1 =$ przegrana gracza pierwszego)
- drugi gracz minimalizuje wynik gry

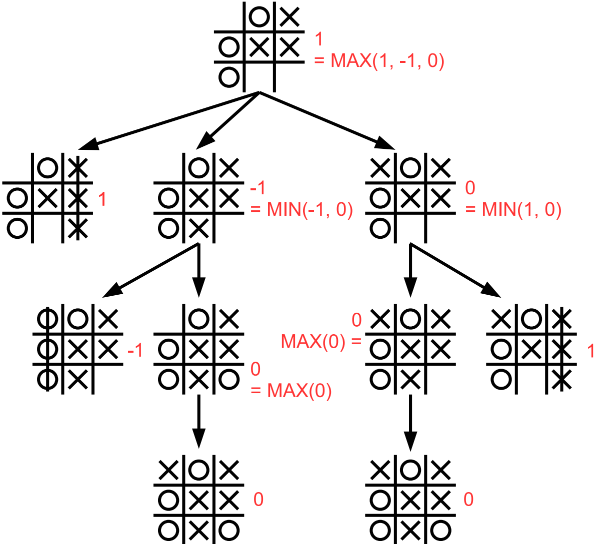
Min-Max

- na nieparzystych poziomach drzewa gry ruch wykonuje gracz pierwszy — wybiera maksimum z dzieci
- na parzystych poziomach drzewa gry ruch wykonuje gracz drugi — wybiera minimum z dzieci
- potrafimy obliczyć wartość węzła (stanu gry) w którym gra się kończy

Min-Max



Min-Max



Min-Max

- drzewo gry jest zazwyczaj za duże na pełen przegląd
- ograniczamy głębokość przeglądanego drzewa
- gdy dojdziemy do limitu głębokości i gra nie jest rozstrzygnięta, próbujemy oszacować stan gry — im lepsze oszacowanie tym lepiej gra nasz algorytm
- np.: 1000 = wygrana gracza pierwszego, -1000 = przegrana, 0 = remis, 800 = silna przewaga gracza pierwszego, -200 = lekka przewaga gracza drugiego itp.

Alfa-Beta

- staramy się ograniczyć rozmiar przeszukiwanego drzewa w algorytmie Min-Max
- pamiętamy dwa ograniczenia: α i β
- α to minimalny wynik gracza maksymalizującego — wiemy, że w przeanalizowanej już części drzewa gracz maksymalizujący może osiągnąć wynik co najmniej α
- β to maksymalny wynik gracza minimalizującego — wiemy, że w przeanalizowanej już części drzewa gracz minimalizujący może osiągnąć wynik co najwyżej β

Alfa-Beta

- w węźle odpowiadającym ruchowi gracza maksymalizującego, gdy otrzymamy wartość dziecka większą lub równą niż β — przerywamy
- w przeciwnym wypadku aktualizujemy α i kontynuujemy
- nie ma sensu analizowanie fragmentu drzewa, w którym przeciwnik traci więcej niż to konieczne, ponieważ ominie go
- analogicznie, w węźle odpowiadającym ruchowi gracza minimalizującego, gdy otrzymamy wartość dziecka mniejszą lub równą niż α — przerywamy
- w przeciwnym wypadku aktualizujemy β i kontynuujemy

Algorytmy metaheurystyczne

- algorytmy przybliżone na tyle ogólne, że można je zastosować do rozwiązania praktycznie dowolnego problemu
- często inspirowane mechanizmami i zjawiskami występującymi w przyrodzie
- najczęściej polegają na generowaniu kolejnych rozwiązań, aż zostanie spełniony warunek zakończenia

Algorytmy metaheurystyczne

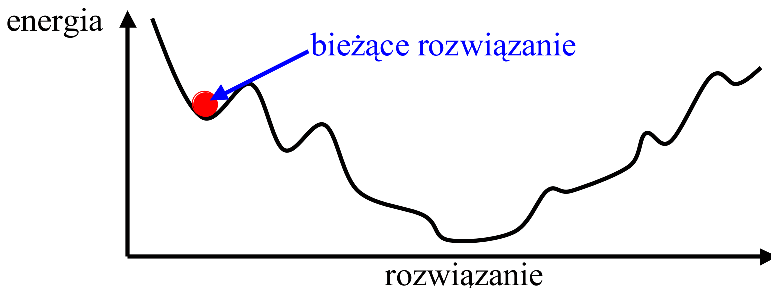
- typowe warunki zakończenia pracy:
  - wykonano K kroków
  - algorytm pracował przez T sekund
  - w K ostatnich krokach nie zanotowano poprawy najlepszego rozwiązania
  - przez K ostatnich kroków najlepsze rozwiązanie nie zmieniło się o więcej niż ϵ

Algorytmy metaheurystyczne

- nie są kompletne
  - należy wybrać odpowiednią reprezentację rozwiązania, która umożliwia wykonanie określonych operacji
  - posiadają zestaw parametrów, od których zależy efektywność ich działania
- są bardzo elastyczne, ale nie można ich zastosować "out of the box"

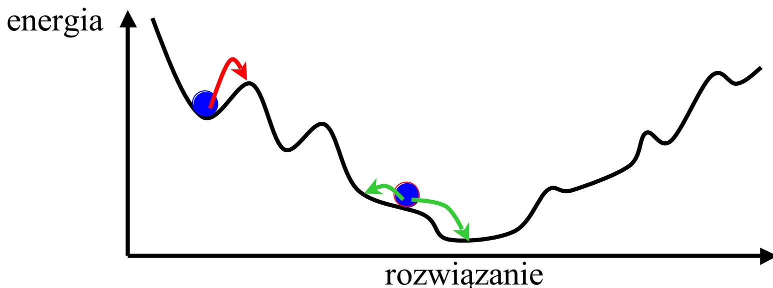
Symulowane wyżarzanie

- inspirowane procesami termodynamicznymi
- dążymy do minimalizacji energii układu (wartości naszego rozwiązania)
- dodatkowy parametr: temperatura



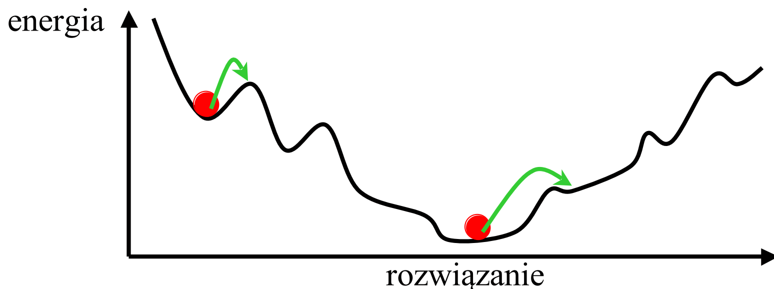
Symulowane wyżarzanie

- mała temperatura pozwala na skok do rozwiązania o mniejszej, lub niewiele większej energii



Symulowane wyżarzanie

- duża temperatura pozwala na skoki w miejsca o wyższej energii
- umożliwia opuszczenie lokalnego minimum
- grozi opuszczeniem minimum globalnego



Symulowane wyżarzanie

- zaczynamy od dużej temperatury, z czasem ją zmniejszając
- odpowiedni dobór tempa spadku gwarantuje znalezienie optimum (w nieskończoności)

Symulowane wyżarzanie

- wymagane operacje na reprezentacji:
- wylosowanie rozwiązania (początkowego)
- wylosowanie "sąsiedniego" rozwiązania

Symulowane wyżarzanie

```
wylosuj rozwiązanie początkowe r
ustaw temperaturę początkową T_0
ustaw k = 1
while nie jest spełniony warunek zakończenia do
    T_k = U(T_0)
    wylosuj sąsiada s
    if F(s) < F(r) then
        r = s
    else
        z prawdopodobieństwem 1 / exp((F(s) − F(r)) / T_k) ustaw r = s
    end if
end while
```

while nie jest spełniony warunek zakończenia **do**

$T_k = U(T_0)$

if $F(s) < F(r)$ **then**

$r = s$

else z prawdopodobieństwem $\frac{1}{\exp\left(\frac{F(s)-F(r)}{T_k}\right)}$ ustaw $r = s$

Symulowane wyżarzanie

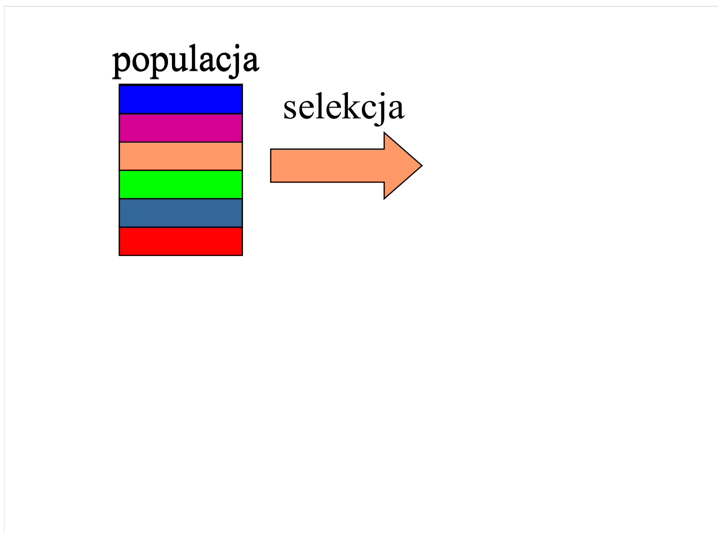
- funkcja U reguluje rozmiar dopuszczalnych skoków w zależności od czasu
- dowodzi się, że o ile temperatura obniżana jest nie szybciej niż $\frac{T_0}{\ln k}$, można znaleźć minimum globalne

Algorytm genetyczny

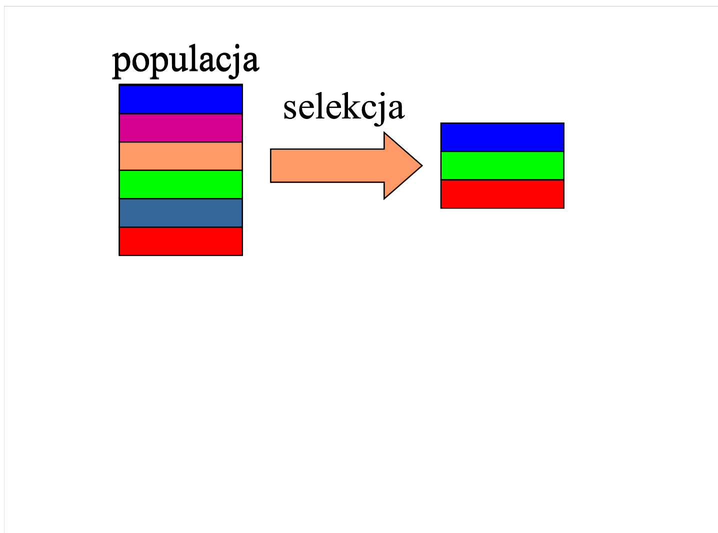
- inspirowany procesem ewolucji
- najlepiej dostosowane osobniki przeżywają i przekazują swoje geny największej liczbie potomków
- najgorzej dostosowani giną bezpotomnie
- każdy osobnik (rozwiązanie) jest sekwencją genów

Algorytm genetyczny

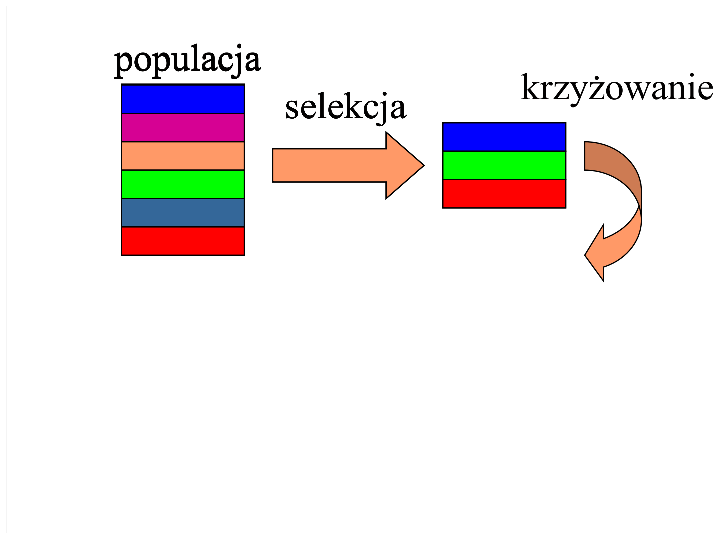
Algorytm genetyczny



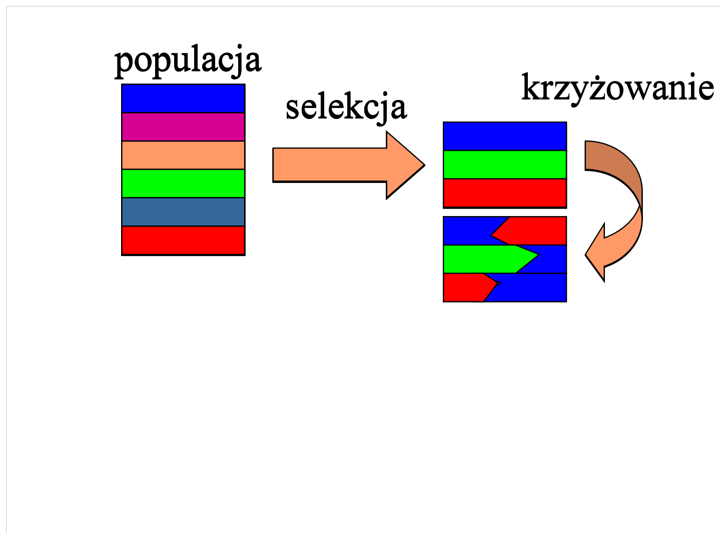
Algorytm genetyczny



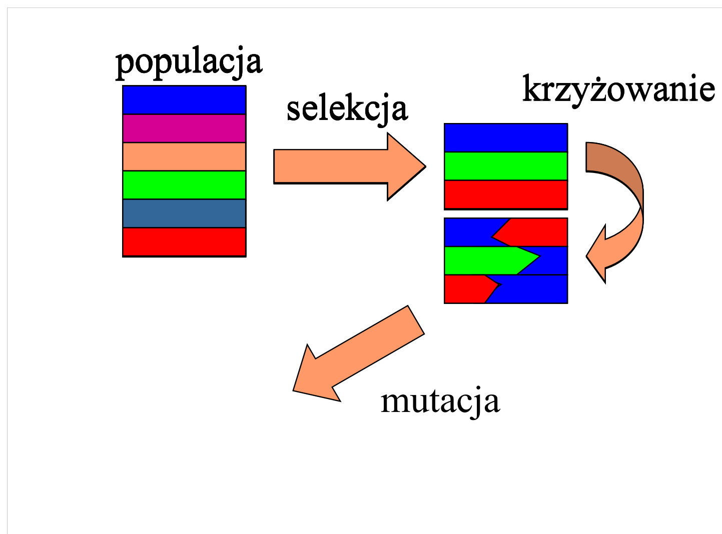
Algorytm genetyczny



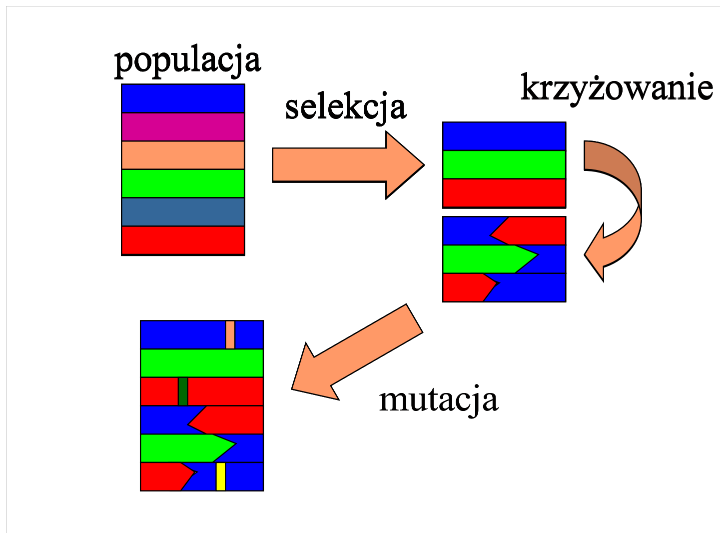
Algorytm genetyczny



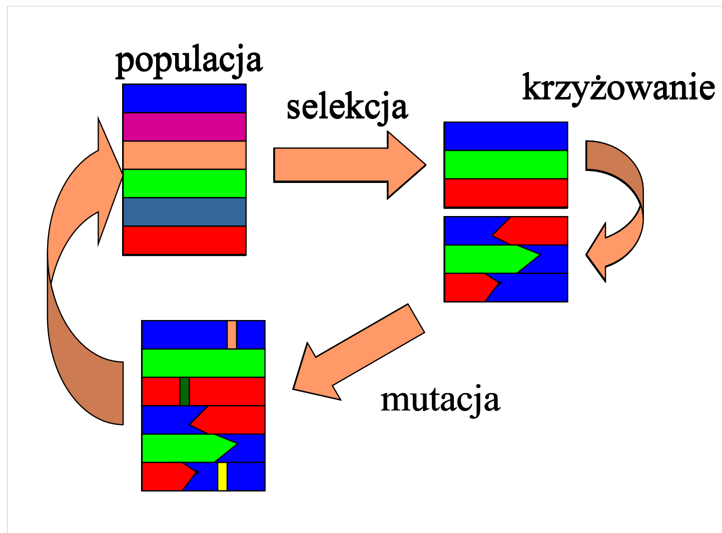
Algorytm genetyczny



Algorytm genetyczny



Algorytm genetyczny



Algorytm genetyczny

- operacje:
  - selekcja — wybór najlepiej dostosowanych osobników
  - krzyżowanie — utworzenie, na podstawie pary rodziców, pary potomstwa
  - mutacja — nieznaczna modyfikacja osobnika

Selekcja

- ruletka
  - w zależności od wartości dostosowania (im lepsza wartość rozwiązania tym większe dostosowanie) przydzielamy osobnikom prawdopodobieństwa wylosowania
  - losujemy osobniki z określonymi podobieństwami
- turniej
  - ustalamy rozmiar turnieju (np. 4 lub 8 osobników)
  - rozgrywamy między osobnikami turniej systemem pucharowym — wygrywa lepszy, przegrany odpada, aż otrzymamy jednego osobnika

Krzyżowanie

- jednopunktowe
  - losujemy punkt w genomie
  - rozcinamy oba osobniki w tym samym miejscu
  - łączymy je "na krzyż"
- dwupunktowe
  - losujemy dwa punkty w genomie
  - wymieniamy osobniki zawartością pomiędzy wylosowanymi punktami
- inne
  - ...

Krzyżowanie

- implementacja krzyżowania wymaga więcej uwagi niż selekcja
- musimy albo zapewnić, że krzyżowane osobniki będą poprawne, albo rozwiązać problem niepoprawnych osobników
- "lepiej" zapewnić poprawność osobników — algorytm będzie działał efektywniej
  - uważnie konstruując krzyżowanie (najlepsze rozwiązanie)
  - poprawiając niepoprawne osobniki

Krzyżowanie

- jeżeli nie jesteśmy w stanie zapewnić poprawności osobników, możemy wprowadzić “kary” za niepoprawność — im dalej od rozwiązania poprawnego jest osobnik, tym większa kara (zmniejszona wartość dostosowania)
- skrajnie niedostosowane osobniki możemy odrzucać (ale dopiero po pewnym czasie)

Mutacja

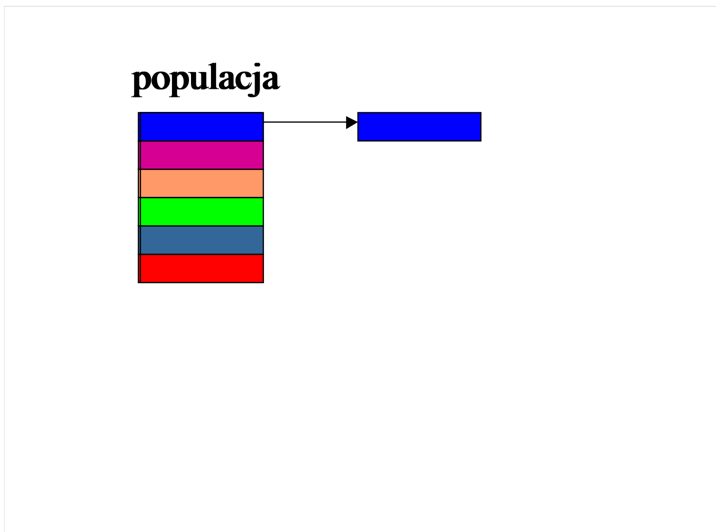
- drobna, losowa zmiana osobnika
- np. każdy bit osobnika zmienia się na przeciwny z prawdopodobieństwem p_m
- np. z prawdopodobieństwem p_m zmienia się losowy bit osobnika na przeciwny
- np. zamieniamy parę losowych elementów osobnika miejscami
- ...

Ewolucja różnicowa

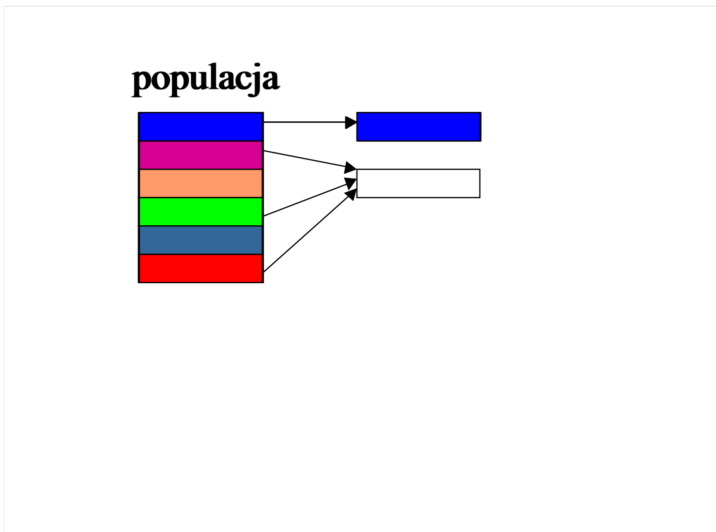
- algorytm nieco podobny do alg. genetycznego
  - populacja osobników
  - mutacja
  - zamiast krzyżowania — tworzenie na bazie kilku osobników nowego
  - zamiast selekcji — osobnik konkuruje z kandydatem na jego miejsce

Ewolucja różnicowa

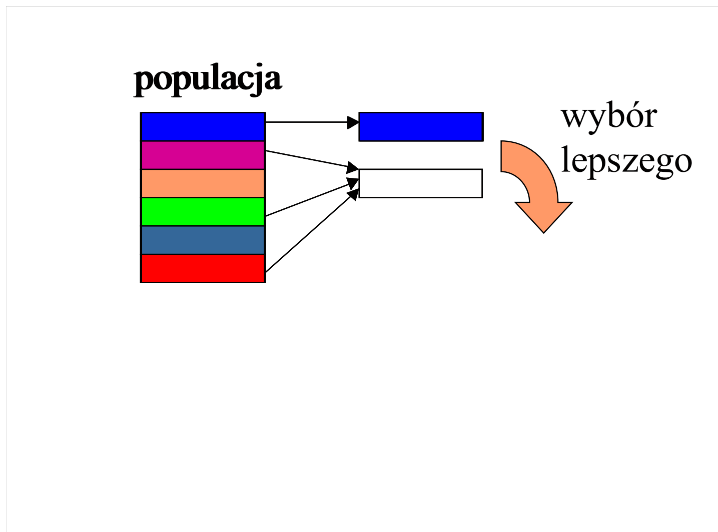
Ewolucja różnicowa



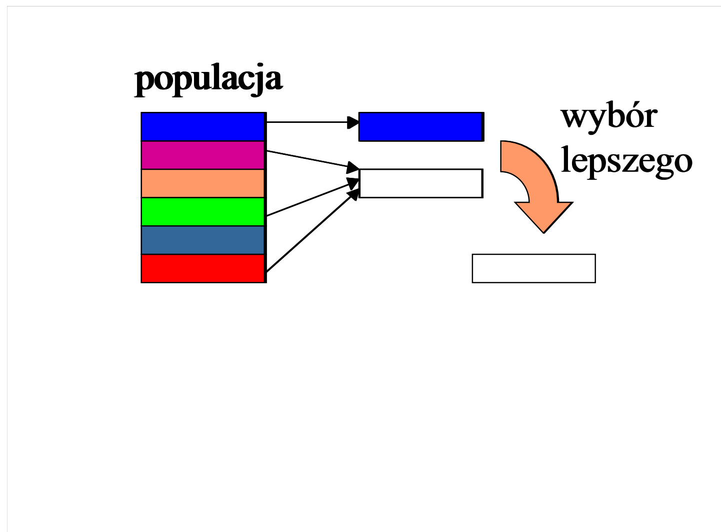
Ewolucja różnicowa



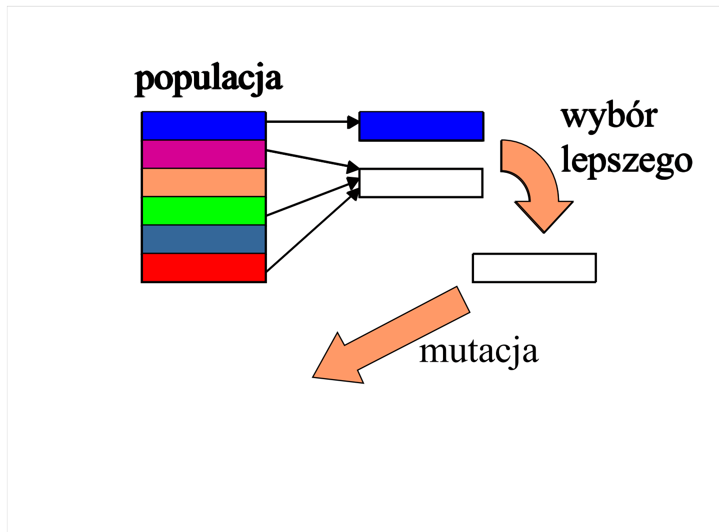
Ewolucja różnicowa



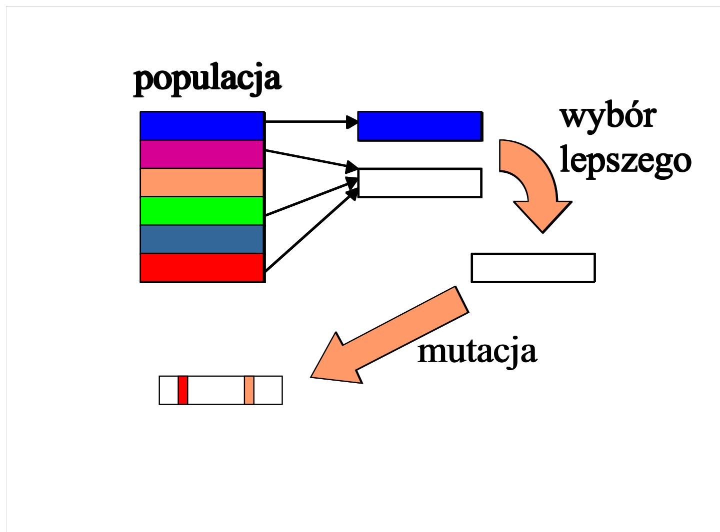
Ewolucja różnicowa

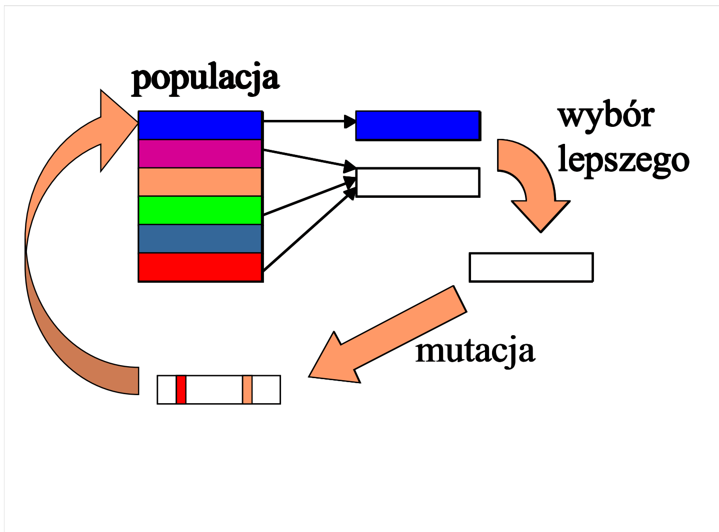


Ewolucja różnicowa

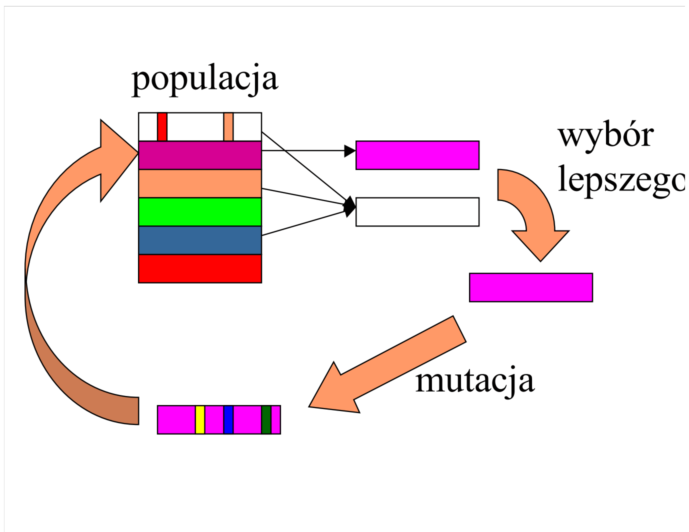


Ewolucja różnicowa




populacja
wybór
lepszego
mutacja

Ewolucja różnicowa



Ewolucja różnicowa

- osobnik musi być wektorem liczb
- nowy osobnik: $n_i = a_i + \phi(b_i - c_i)$ z prawdopodobieństwem p_c, w przeciwnym wypadku $n_i = a_i$, wykonujemy co najmniej jedną zmianę
- mutacja — z niewielkim prawdopodobieństwem zmieniamy element wektora o losową wartość

Optymalizacja rojem cząstek

- PSO (Particle Swarm Optimization)
- każdy osobnik:
  - ma pozycję (d liczb)
  - posiada prędkość (także d liczb)
  - pamięta najlepsze znalezione przez siebie rozwiązanie
- dodatkowo pamiętane jest globalne najlepsze rozwiązanie

Optymalizacja rojem cząstek

- w każdym kroku cząstka:
  - jest “przyciągana” w kierunku najlepszego swojego i najlepszego globalnego rozwiązania (modyfikacja prędkości)
  - przesuwa się zgodnie ze swoją prędkością
  - aktualizuje najlepsze znane sobie rozwiązanie

Optymalizacja rojem cząstek

1: wybierz ω, ϕ_p i ϕ_g z przedziału $\langle 0, 1 \rangle$
2: wylosuj początkowe pozycje (x_i), prędkości cząsteczek (v_i)
   i dotychczasowe najlepsze rozwiązania $(p_i = x_i)$
3: ustaw g jako najlepsze rozwiązanie spośród cząstek
4: **while** nie jest spełniony warunek zakończenia **do**
5:     **for** każda cząstka i **do**
6:         wylosuj r_p i r_g z przedziału $\langle 0, 1 \rangle$
7:         zmień prędkość $v_i = \omega v_i + \phi_p r_p (p_i - x_i) + \phi_g r_g (g - x_i)$
8:         zmień pozycję cząstki: $x_i = x_i + v_i$
9:         **if** $F(x_i) < F(p_i)$ **then**
10:             $p_i = x_i$
11:             zaktualizuj g (jeżeli $F(x_i) < F(g)$)
12:         **end if**
13:     **end for**
14: **end while**

Reprezentacja rozwiązania

- niech rozwiązaniem naszego problemu jest permutacja zbioru $\{1, 2, \ldots, n\}$
- jak zbudować z niej osobnika?

Reprezentacja rozwiązania

- najprostsze podejście — osobnik (rozwiązanie) to permutacja
- sąsiad/mutacja — np. zamiana dwóch pozycji miejscami
- krzyżowanie — problematyczne, może wymagać rekonstrukcji, np. powtarzające się zadania zastępowane są niewykorzystanymi w oryginalnej kolejności
- nie nadaje się do ewolucji różnicowej ani PSO

Reprezentacja rozwiązania

- reprezentacja przedziałowa
- osobnik to wektor liczb z przedziału $\langle 0, 1)$
- wartości z przedziału $\langle i - 1/n, i/n)$ odpowiadają liczbie i
- powtórzenia zastępowane są losowymi wartościami spośród niewykorzystanych
- sąsiad/mutacja — zmiana jednej wartości
- krzyżowanie — naturalne

Reprezentacja rozwiązania

- priorytety reguł
- osobnik to wektor liczb z przedziału $\langle 0, 1)$
- dodatkowo mamy zbiór k reguł wyboru liczby (np. wybierz największą, wybierz najmniejszą, itp.)
- wartości z przedziału $\langle i - 1/n, i/n)$ odpowiadają regule i
- przechodzimy po wektorze, przy każdej wartości wybierając liczbę spośród niewykorzystanych, zgodnie z bieżącą regułą
- sąsiad/mutacja — zmiana jednej reguły
- krzyżowanie — naturalne