(Meta)heurystyki
Notatki

Heurystyki — Optymalizacja Lokalna

Wstęp

Dlaczego optymalizacja jest trudna?

Trasa dostawcy przez **20 miast**: $20! = 2430902008176640000$ możliwych tras — brak możliwości sprawdzenia wszystkich. **Algorytm dokładny (brute force)** gwarantuje **optimum globalne**, jest niewykonalny dla $n > 20$ (NP-hard -> czas eksponencjalny)

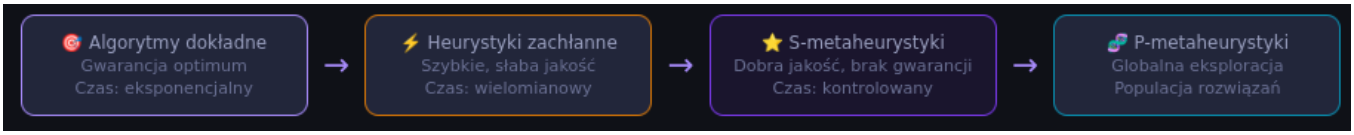
Ile to zajmuje komputerowi?

- 10^9 sprawdzeń/sekundę: ponad 77 lat
- 30 miast: miliardy razy dłużej niż wiek wszechświata
- 50 miast: całkowicie niewykonalne nawet kwantowo

Heurystyki jako rozwiązanie:

- "Wystarczająco dobra" trasa w sekundy
- 5-10% powyżej optimum -> akceptowalne przemysłowo
- UPS ORION: 400mln $/rok oszczędności
- Google Maps: heurystyki routingu w ułamkach sekundy

Spektrum algorytmów optymalizacyjnych



- **Algorytmy dokładne** — gwarancja optimum, czas: eksponencjalny
  Branch & Bound — optimum gwarantowane, czas: $O(2^n)$ w najgorszym wypadku, użycie: n < 30 dla TSP
- **Heurystyki zachłanne** — szybkie, słaba jakość, czas: wielomianowy
- **S-metaheurystyki** — dobra jakość, brak gwarancji, czas: kontrolowany
  SA / TS / VNS (optymalizacja lokalna) — jakość: 1-5% ponad optimum, czas: sekundy-minuty, praktycznie najlepszy wybór
- **P-metaheurystyki** — globalna eksploracja, populacja rozwiązań
  GA / PSO / ACO (optymalizacja globalna) — lepsza eksploracja globalna, wolniejsze per iterację, state of the art: hybrydowe

Taksonomia metaheurystyk

- **S-metaheurystyki (Single-solution)**
  - Operują na **jednym rozwiązaniu** (trajektoria)
  - Intensywna eksploatacja (intensification)
  - Przeszukiwanie lokalne — dobra jakość w sąsiedztwie
  - Przykłady: **HC, SA, TS, VNS, ILS, GRASP**
  - Szybkie, małe zużycie pamięci
- **P-metaheurystyki (Population)**
  - Operują na **populacji rozwiązań**
  - Intensywna eksploracja (diversification)
  - Przeszukiwanie globalne — unikanie lokalnych optimów
  - Przykłady: **GA, ES, PSO, ACO, DE**
  - Wolniejsze per iterację, naturalnie równoległe

Kluczowy dylemat każdej metaheurystyki: Intensification (eksploatacja dobrego regionu) vs Diversification (eksploracja nowych obszarów). Za dużo eksploracji = losowe przeszukiwanie. Za dużo eksploatacji = utknięcie w lokalnym optimum.

Co to jest heurystyka?

- **Definicja heurystyki** — regułą lub strategia ułatwiająca rozwiązanie problemu, **bez gwarancji optymalności**. Bazuje na doświadczeniu i intuicji.
- **Definicja metaheurystyki (Glover 1986)** — ogólna strategia wysokiego poziomu do projektowania heurystyk. “meta” = ponad - steruje heurystyką niższego poziomu.
- **Właściwości metaheurystyk (Talbi 2009)**:
  - Strategie **ogólne** — stosowalny do wielu problemów
  - **Niedeterministyczne** — element losowości
  - Brak gwarancji optymalności
  - Łatwe do implementacji, trudne do strojenia
  - Inspirowane naturą lub logiką

Praktyczna implikacja — Metaheurystyki są jak przepis kulinarny — ta sama metoda (np. SA) może rozwiązywać TSP, układanie planów lekcji i problem plecakowy. Zmienia się tylko kodowanie problemu.

Podstawy optymalizacji

Definicja formalna

$$\min f(x), \quad x \in S$$

gdzie:

- $f : S \to \mathbb{R}$ — funkcja celu (objective function),
- S — przestrzeń poszukiwań (search space),
- x^* — globalne rozwiązanie optymalne.

$$\max f(x) \equiv \min -f(x)$$

Typy przestrzeni S

- **Ciągła:** $x \in \mathbb{R}^n$ stosuje się pochodne i gradient.
- **Dyskretna:** $x \in \{0, 1\}^n$ obejmuje m.in. problemy binarne i permutacje.
- **Mieszana:** kombinacja zmiennych ciągłych i dyskretnych.
- **Kombinatoryczna:** liczba możliwości może rosnąć bardzo szybko, np. $n!, \quad 2$

Optimum lokalne

Punkt x^* jest optimum lokalnym, jeżeli jest lepszy od wszystkich rozwiązań w swoim sąsiedztwie $N(x^*)$:

$$f(x^*) \leq f(y) \quad \forall y \in N(x^*)$$

Optimum globalne

Punkt x^* jest optimum globalnym, jeżeli jest lepszy od wszystkich rozwiązań w całej przestrzeni poszukiwań S:

$$f(x^*) \leq f(y) \quad \forall y \in S$$

Kluczowy problem metaheurystyk

Algorytmy lokalne zazwyczaj prowadzą do znalezienia optimum lokalnego. Metaheurystyki wprowadzają mechanizmy umożliwiające ucieczkę z lokalnych minimów.
Każda metoda realizuje to w inny sposób:

- **SA (Simulated Annealing)** — wykorzystuje losowość,
- **TS (Tabu Search)** — wykorzystuje pamięć rozwiązań,
- **VNS (Variable Neighborhood Search)** — wykorzystuje zmianę sąsiedztwa.

Klasy Złożoności

- **Klasa P — Polynomial Time — rozwiązywalne wielomianowo**
  - Czas: $O(n^k)$ dla stałego k.
  - Przykłady: sortowanie $O(n \log n)$, MST $O(n \log n)$, najkrótsza ścieżka Dijkstry $O(n^2)$.
  - Rozmiar: $n = 10\,000$ jest wykonalny obliczeniowo.
- **Klasa NP — Nondeterministic Polynomial Time — weryfikowalne wielomianowo**
  - Certyfikat rozwiązania można sprawdzić w czasie: $O(n^k)$
  - Nie wiadomo, czy: $P = NP$ (problem związany z Nagrodą Milenijną).
  - Przykłady: SAT, problem ścieżki Hamiltona, Subset Sum.
- **NP-hard — trudniejsze niż NP**
  - Każdy problem z klasy NP redukuje się do problemu NP-hard.
  - Prawdopodobnie nie istnieje algorytm rozwiązujący je w czasie wielomianowym.
  - Przykłady: TSP, VRP, Scheduling, VLSI.

Rozmiar vs. czas dla TSP

n miast	Liczba stanów	Czas (10^9/s)
10	181 440	< 1 ms
20	6×10^{17}	19 lat
30	4×10^{30}	10^{21} lat
50	3×10^{63}	niemożliwe

Wniosek:
Dla problemów NP-hard w praktyce przemysłowej jedynym rozsądnym podejściem jest:

- algorytm przybliżony (heurystyki, metaheurystyki),
- albo algorytm dokładny z ograniczeniem czasowym (np. Branch & Bound z limitem czasu).

Definicja

Problem optymalizacji lokalnej można opisać trójką:

$$(S, N, f)$$

gdzie:

- S — przestrzeń poszukiwań,
- $N : S \to 2^S$ — funkcja sąsiedztwa,
- $f : S \to \mathbb{R}$ — funkcja celu.

Krajobraz optymalizacyjny oznacza „ukształtowanie terenu" w przestrzeni rozwiązań.

Typy krajobrazów (fitness landscapes / optimization landscapes)

- **Unimodalny:** jedno optimum globalne — metoda HC (Hill Climbing) wystarczy.
- **Multimodalny:** wiele optimów lokalnych — potrzebna jest metaheurystyka.
- **Deceptive:** lokalne optimum „wskazuje" błędny kierunek poszukiwań.
- **Rugged (chropowaty):** duże zmiany wartości funkcji celu f w sąsiedztwie.
- **Big Valley:** dobre rozwiązania są skupione w centralnym obszarze.

Problemy lokalnych optimów

- **Lokalne maksimum:** lepsze od wszystkich sąsiadów, ale gorsze od optimum globalnego.
- **Plateau:** sąsiednie rozwiązania mają identyczną wartość funkcji celu: $f(x_1) = f(x_2) = \cdots = f(x_n)$
- **Grzbiet (ridge):** sekwencja lokalnych optimów trudna do przekroczenia.
- **Basen przyciągania:** region przestrzeni rozwiązań prowadzący do tego samego optimum lokalnego.

Intuicja:

Wyobraź sobie wędrowca szukającego najwyższej góry w pokrytym mgłą terenie. Widzi tylko kilka metrów wokół siebie (swoje sąsiedztwo), dlatego musi wybrać odpowiednią strategię ucieczki, gdy znajdzie się na lokalnym szczycie.

Składniki każdej metaheurystyki S

1. **Reprezentacja (kodowanie)**: jak opisujemy rozwiązanie? Bity, permutacja, wektor rzeczywisty, graf
2. **Inicjalizacja**: punkt startowy — losowy lub heurystyczny (greedy)
3. **Funkcja celu f(x)**: jak oceniamy jakość rozwiązania? koszt, czas, odległość
4. **Sąsiedztwo N(x)**: zbiór rozwiązań osiągalnych jednym ruchem z x
5. **Strategia eksploracji**: first-improvement vs best-improvement
6. **Mechanizm ucieczki**: SA - losowość, TS - pamięć, VNS - zmiana N
7. **Kryterium zatrzymania**: max iteracji, brak poprawy, budżet NFE

Przykład z TSP

- **Reprezentacja**: permutacja [3,1,4,2,5] — kolejność miast
- **Inicjalizacja**: Nearest Neighbor greedy
- **f(x)**: suma długości krawędzi w cyklu (suma odległości przejechanej)
- **N(x)**: operator 2-opt (zamiana 2 krawędzi)
- **Kryterium**: max 10 000 iteracji lub 60 sekund

Zasada modularności: Wszystkie algorytmy (HC, SA, TS...) używają tych samych komponentów. Zmiana algorytmu to zmiana tylko mechanizmu ucieczki — reprezentacja i sąsiedztwo pozostają te same.

Typy reprezentacji rozwiązań

- **Binarna** $\{0,1\}^n$

$$x = [1,0,1,1,0,0,1,0]$$

  - Problem plecakowy: 1 oznacza wybór przedmiotu.
  - Selekcja cech w ML: 1 oznacza wybraną cechę.
  - Operatory: bit-flip mutation, 1-point crossover.

- **Permutacyjna** $\pi(n)$

$$x = [3,1,4,2,5]$$

  (permutacja oznaczająca kolejność)

  - TSP: kolejność odwiedzania miast.
  - Harmonogramowanie: kolejność wykonywania zadań.
  - Operatory: swap, inversion, insertion.

- **Rzeczywista** $\mathbb{R}^n$

$$x = [0.73, -1.24, 3.01, 0.55]$$

  - Optymalizacja ciągła: parametry inżynierskie.
  - Strojenie hiperparametrów ML.
  - Operator: Gaussian perturbation (zakłócenie Gaussowskie).

- **Grafowa / drzewiasta**

  Struktury: drzewo, graf, sieć, harmonogram.

  - Network design: topologia sieci.
  - Genetic Programming: drzewa wyrażeń.
  - Operatory: edge swap, subtree mutation.

Definicja sąsiedztwa N(x)

- **Formalna definicja**

$$N(x) \subseteq S$$

  Zbiór $N(x)$ to zbiór rozwiązań „blisko" x, definiowany przez operator ruchu (move operator).

- **Właściwości sąsiedztwa**

  - **Rozmiar**: $|N(x)|$ małe (szybsze przeszukiwanie) vs. duże (dokładniejsze). (Ile masz ruchów do wyboru)
  - **Symetria:** $y \in N(x) \Leftrightarrow x \in N(y)$ (ruch jest odwracalny)
  - **Spójność:** każde rozwiązanie jest osiągalne z każdego innego (przestrzeń jest spójna względem operatora). (każde rozwiązanie można osiągnąć z każdego innego, brak izolowanych komponentów)
  - **Inkrementalna ewaluacja:** zamiast pełnego obliczania $f(x)$ używa się: Δf (obliczanie zmiany, a nie pełnej funkcji kosztu)

- **Operatory sąsiedztwa dla TSP**

  - **2-opt:** usuń 2 krawędzie i odwróć segment. $|N| = O(n^2)$
  - **3-opt:** usuń 3 krawędzie, istnieje 8 możliwych rekonfiguracji. $|N| = O(n^3)$
  - **Or-opt:** przenieś 1–3 miasta. $|N| = O(n^2)$
  - **Swap:** zamień dwa miasta. $|N| = O(n^2)$
  - **Lin–Kernighan:** adaptacyjny k-opt (najbardziej efektywny w praktyce).

2-opt (koszt zmiany)

$$\Delta cost = d(i,j) + d(i+1,j+1) - d(i,i+1) - d(j,j+1)$$

Inicjalizacja i kryteria zatrzymania

- **Strategie inicjalizacji**
  - **Losowa:** prosta metoda zapewniająca dobrą dywersyfikację startową.
  - **Zachłanna (greedy):** np. nearest neighbor (NN) dla TSP — szybki, dobry punkt startowy.
  - **Wielokrotna:** random restart — wiele uruchomień, wybór najlepszego wyniku.
  - **Konstruktywna:** np. savings algorithm, Christofides.
  - **Heurystyczna:** wykorzystanie wiedzy domenowej.
- **Pułapka inicjalizacji**

  Start z rozwiązania greedy (NN) może dawać nawet ok. 25% od optimum, jednak po zastosowaniu Simulated Annealing (SA) wynik może być gorszy niż dla inicjalizacji losowej.

  Greedy start często „przyciąga" algorytm do lokalnego optimum w pobliżu rozwiązania NN.
- **Kryteria zatrzymania**
  - **Max iteracji:** prosty i przewidywalny czas wykonania.
  - **Max NFE:** liczba ewaluacji funkcji celu (standard benchmarkowy).
  - **Brak poprawy:** zatrzymanie po k iteracjach bez poprawy.
  - **Cel jakościowy:** zatrzymanie gdy $f(x) \leq target$ (gdy optimum jest znane).
  - **Limit czasu:** praktyczne w systemach czasu rzeczywistego.
- **Porównanie kryteriów**

Kryterium	Zalety	Wady
Max iteracji	Prostota	Nie uwzględnia złożoności ewaluacji
Max NFE	Uczciwe porównanie	Wymaga licznika
Brak poprawy	Adaptatywne	Ryzyko zbyt wczesnej zbieżności
Limit czasu	Praktyczne	Zależne od sprzętu

First Improvement (First Descent)

Metoda polega na zaakceptowaniu pierwszego sąsiada poprawiającego wartość funkcji celu:

```
for y in shuffle(N(x)):
    if f(y) < f(x):
        x = y
        break
```

Idea: zatrzymujemy się natychmiast po znalezieniu poprawy.

- **Zalety:**
  - średnio tylko $\frac{|N|}{2}$ ewaluacji na iterację,
  - bardzo szybka w praktyce dla dużych sąsiedztw (np. 2-opt dla $n = 1000$).
- **Wady:**
  - może pominąć lepsze rozwiązania w sąsiedztwie.

Best Improvement (Steepest Descent)

Metoda polega na przeglądzie całego sąsiedztwa i wyborze najlepszego sąsiada:

```
best_nbr = x
for y in N(x):
    if f(y) < f(best_nbr):
        best_nbr = y
x = best_nbr
```

Idea: wybieramy najlepszą możliwą poprawę w danym sąsiedztwie.

- **Zalety:**
  - dokładniejsza eksploracja przestrzeni,
  - każda iteracja daje maksymalną poprawę lokalną.
- **Wady:**
  - koszt obliczeniowy $O(|N|)$ w każdej iteracji.

Porównanie empiryczne

Dla problemu TSP z 2-opt ($|N| = O(n^2)$):

- First Improvement jest około $2\times$ szybszy,
- jakość końcowego rozwiązania jest podobna,
- Best Improvement jest korzystniejszy, gdy ewaluacja f jest tania.

Porównanie algorytmów lokalnego przeszukiwania

Algorytm	Kompleksność iteracji	Ewaluacje / iter.	Typowe max_iter
HC (First Improvement)	$O(\|N\|/2)$ (średnio)	$\|N\|/2$ (średnio)	100–1000
HC (Best Improvement)	$O(\|N\|)$	$\|N\|$	100–1000
SA	$O(1)$ na ruch	1 (losowy sąsiad)	10 000–100 000
TS	$O(\|N\|)$ + sprawdzenie tabu	$\|N\|$	1 000–10 000
VNS	$O(\|N_1\| + \|N_2\| + \dots)$	$\|N_k\|$ na krok	100–1 000
ILS	$O(LS)$ na restart	budżet LS	100–1 000 restartów

Inkrementalna ewaluacja (klucz do wydajności)

Zamiast pełnego obliczania: $f(x') = O(n)$ (użycie pełnej długości trasy), stosuje się różnicę: $\Delta f = f(x') - f(x) = O(1)$ (tylko zmienione elementy). Przyspieszenie może wynosić nawet $100{-}1000\times$ dla dużych instancji.

TSP 2-opt — ewaluacja inkrementalna

$$\Delta cost = d(i,j) + d(i+1,j+1) - d(i,i+1) - d(j,j+1)$$

Zamiast $O(n)$ wykonuje się tylko 4 operacje, niezależnie od rozmiaru problemu.

Twierdzenie No Free Lunch (NFL)

- **Wolpert & Macready, 1997**

  Twierdzenie (tzw. No Free Lunch Theorem) mówi, że uśrednione po wszystkich możliwych problemach, wszystkie algorytmy optymalizacyjne mają identyczną skuteczność:

$$\sum_f P(\text{cost} \mid A, f, n) = \sum_f P(\text{cost} \mid B, f, n)$$

- **Błędna interpretacja**

  „Wszystkie algorytmy są równie dobre w praktyce”

  FAŁSZ.

  W praktyce, dla konkretnych klas problemów (np. TSP, scheduling), niektóre algorytmy dominują inne nawet o rząd wielkości.

- **Właściwa interpretacja**

  - Nie istnieje algorytm najlepszy dla wszystkich możliwych problemów.
  - Dobry algorytm musi wykorzystywać strukturę konkretnej klasy problemów.
  - Wynik uzasadnia badania nad algorytmami specjalizowanymi.
  - Uzasadnia metaheurystyki jako podejście dobre dla klas problemów, a nie uniwersalne rozwiązanie.

- **Praktyczna implikacja**

  Dobieraj algorytm do klasy problemu.

  Przykładowo: Simulated Annealing (SA) dobrze sprawdza się dla TSP, ale może być gorszy niż Genetic Algorithm (GA) dla problemu plecakowego.

  W praktyce zawsze warto testować kilka metod na własnych danych.

Hill Climbing

Idea algorytmu

- Zacznij od rozwiązania startowego x_0.
- Generuj sąsiedztwo $N(x)$.
- Wybierz sąsiada poprawiającego wartość funkcji celu f.
- Akceptuj wyłącznie poprawy — nigdy gorszych ruchów.
- Powtarzaj aż do braku poprawy (osiągnięcie optimum lokalnego).

Metafora

Wędrowiec w gęstej mgle szuka szczytu góry. W każdym kroku porusza się tylko w górę. Gdy dociera do punktu, z którego każdy krok prowadzi w dół, zatrzymuje się.
Może to być jedynie lokalne wzgórze, a nie najwyższy szczyt.

Historia

- Jeden z najstarszych algorytmów optymalizacji lokalnej.
- Formalizacja: Arthur Samuel (1959) w kontekście gry w warcaby.
- Termin „Hill Climbing” został wprowadzony przez: Newell, Shaw i Simon (1959).
- Stanowi podstawę teoretyczną dla wielu metaheurystyk typu S.

Zastosowania Hill Climbing (HC)

- Inicjalizacja dla SA i TS — jako punkt startowy.
- Szybkie generowanie przybliżonych rozwiązań, gdy czas obliczeń jest krytyczny.
- Składnik lokalnego przeszukiwania w algorytmach memetycznych.
- Proste strojenie hiperparametrów.

Steepest Ascent (Best Improvement)

```
function HC_Best(x0, max_iter):
    x = x0
    for iter = 1 to max_iter:
        best_nbr = x
        for y in N(x):  # całe N(x)
            if f(y) < f(best_nbr):
                best_nbr = y

        if f(best_nbr) < f(x):
            x = best_nbr  # zaakceptuj
        else:
            break  # lokalne optimum

    return x
```

First Improvement

```
function HC_First(x0, max_iter):
    x = x0
    improved = True

    while improved and iter < max_iter:
        improved = False

        for y in shuffle(N(x)):
            if f(y) < f(x):
                x = y        # zaakceptuj
                improved = True
                break        # wyjdź z N

    return x
```

Porównanie wariantów

Cecha	Steepest	First Improvement
Ewaluacje	$\|N\|$ zawsze	$\approx \|N\|/2$ średnio
Jakość kroku	najlepszy możliwy	pierwszy poprawiający
Szybkość	wolniejszy	ok. $2\times$ szybszy
Zbieżność	mniej iteracji	więcej iteracji
Wynik końcowy	podobny	podobny

Reguła praktyczna

Jeśli $|N(x)| > 1000$, używaj First Improvement.
Dla małych sąsiedztw ($|N| < 100$) Steepest Ascent może dać nieco lepsze lokalne optimum.

Główne problemy Hill Climbing

1. **Lokalne optimum**

   Rozwiązanie, które jest lepsze od wszystkich swoich sąsiadów, ale niekoniecznie od wszystkich rozwiązań w przestrzeni S.

   - Hill Climbing (HC) nie ma jak z niego wyjść.
   - Każdy ruch pogarsza wartość funkcji celu f.
   - Algorytm zatrzymuje się.

   Rozwiązania: Simulated Annealing (SA) — losowość, Tabu Search (TS) — pamięć.

2. **Plateau**

   Obszar płaski (flat area), w którym wszyscy sąsiedzi mają identyczną wartość funkcji celu:

   - HC wykonuje ruchy boczne bez poprawy.
   - Istnieje ryzyko zapętlenia.
   - Trudne do wykrycia w praktyce.

   Rozwiązania: limit ruchów bocznych, SA.

3. **Grzbiet (Ridge)**

   Sekwencja lokalnych optimów prowadząca do optimum globalnego, ale nieosiągalna wprost przez standardowe sąsiedztwo N.

   - Optimum globalne znajduje się „za grzbietem”.
   - HC nie widzi ścieżki przejścia przez grzbiet.
   - Wymaga większego lub zmodyfikowanego sąsiedztwa (np. k-opt).

   Rozwiązania: Variable Neighborhood Search (VNS), 3-opt.

Kluczowa obserwacja

Każda kolejna S-metaheurystyka rozwiązuje te same problemy, ale innym mechanizmem:

- Simulated Annealing (SA): akceptuje gorsze rozwiązania losowo.
- Tabu Search (TS): blokuje powroty (pamięć).
- Variable Neighborhood Search (VNS): zmienia definicję sąsiedztwa.

Random Restart Hill Climbing (RRHC)

```
function RRHC(k_restarts, max_iter):
    best_overall = None

    for r = 1 to k_restarts:

        # Losowa inicjalizacja
        x0 = random_solution()

        # Lokalne przeszukiwanie
        x_local = HC_Best(x0, max_iter)

        # Aktualizacja najlepszego rozwiązania
        if best_overall is None or f(x_local) < f(best_overall):
            best_overall = x_local

    return best_overall
```

- **Teoria**

  Jeśli: $P(\text{HC finds global optimum}) = p$ dla jednej próby, to:

  $$P(\text{RRHC fails after } k \text{ tries}) = (1-p)^k \to 0$$

  Przykład: dla $p = 0.1$ i $k = 30$:

  $$P(\text{fail}) = 0.9^{30} \approx 4\%$$

- **Zalety RRHC**

  - Prosta implementacja — wystarczy pętla zewnętrzna.
  - Naturalnie równoległy (każdy restart niezależny).
  - Znacznie lepsze wyniki niż pojedynczy HC.
  - Dobry punkt odniesienia (baseline) dla metaheurystyk.

- **Wady RRHC**
  - Brak pamięci między restartami — każdy start od zera.
  - Nie wykorzystuje informacji z poprzednich iteracji.
  - Nieefektywny, gdy optima są skupione w małym obszarze.
  - Koszt: $k\times$ koszt(HC).
- **Kiedy używać RRHC**
  - Ewaluacja funkcji celu f jest tania (szybki HC).
  - Dostępna jest duża liczba rdzeni CPU.
  - Brak lepszych specjalizowanych algorytmów.
  - Jako baseline do porównania z SA i TS.

Stochastic Hill Climbing (SHC)
Zamiast wybierać najlepszego lub pierwszego sąsiada, losujemy sąsiada i akceptujemy go, jeśli poprawia wartość funkcji celu.

```
function SHC(x0):
    x = x0

    while not_terminated:
        y = random_neighbor(x)   # losowy sąsiad!

        if f(y) < f(x):
            x = y  # akceptuj

        # w przeciwnym razie odrzuć

    return x
```

Idea: wprowadzenie losowości w wyborze ruchu (zamiast deterministycznego HC).

HC z ruchami bocznymi (Plateau)

Algorytm pozwala na ruchy boczne (czyli $\Delta f = 0$), ale ogranicza ich liczbę.

```
function HC_Sideways(x0, max_side):
    x = x0
    side_moves = 0

    while side_moves < max_side:
        y = best_neighbor(x)

        Δ = f(y) - f(x)

        if Δ < 0:
            x = y
            side_moves = 0

        elif Δ == 0:
            x = y
            side_moves += 1

        else:
            break

    return x
```

Intuicja

Ruchy boczne pomagają przejść przez plateau (obszary płaskie), ale muszą być ograniczone (np. maksymalnie 100), aby uniknąć nieskończonego błądzenia po płaskim terenie.

Analiza wydajności HC — kiedy to wystarczy?

- **Wyniki HC na TSPLIB**

Instancja	Optimum	HC + 2-opt	Błąd
berlin52	7542	~ 8700	$\sim 15\%$
ch130	6110	~ 7300	$\sim 19\%$
pr1002	259045	~ 310000	$\sim 20\%$

  Wyniki dotyczą HC (Best Improvement) + 2-opt, pojedyncze uruchomienie.

- **Czas wykonania HC + 2-opt**

n **miast**	**Czas [ms]**	**Liczba iteracji**
50	< 1	~ 20
500	~ 50	~ 150
5000	~ 5000	~ 800

- **Kiedy HC wystarczy?**
  - Proof-of-concept lub szybki prototyp.
  - Błąd rzędu 10–20% jest akceptowalny.
  - Jako inicjalizacja dla SA lub TS (tzw. warm start).
  - Jako lokalne przeszukiwanie w algorytmach memetycznych.
  - Gdy budżet obliczeniowy jest bardzo mały (< 1s).

- **Kiedy HC nie wystarczy?**
  - Wymagana jakość rozwiązania < 5% od optimum.
  - Problem silnie multimodalny (wiele lokalnych optimów).
  - Duże instancje przemysłowe.
  - Gdy dostępny jest większy budżet czasowy (minuty/godziny).

Podsumowanie HC — co dalej?

- **Co wiemy o HC?**
  - Prosta implementacja: ok. 20 linii kodu.
  - Zawsze zbiega do lokalnego optimum.
  - Dwa podstawowe warianty: First Improvement i Best Improvement.
  - Random Restart znacząco poprawia wyniki.
  - Inkrementalna ewaluacja jest kluczową optymalizacją wydajności.
  - Błąd rzędu 15–20% od optimum dla problemu TSP.
- **Słabości HC**
  - Utyka w lokalnym optimum.
  - Brak pamięci — nie wie, gdzie już był.
  - Brak mechanizmu ucieczki z lokalnych minimów.
  - Wynik silnie zależy od inicjalizacji.
  - Problem plateau i ruchów bocznych.
- **Ewolucja S-metaheurystyk**
  - HC: tylko poprawy — punkt wyjścia.
  - SA (część 2): dodaje probabilistyczną ucieczkę przez akceptację gorszych rozwiązań.
  - TS (część 2): dodaje pamięć i zakazane ruchy.
  - VNS (część 2): zmienia strukturę sąsiedztwa.
  - ILS (część 3): perturbacja + restart.
  - GRASP (część 3): losowa konstrukcja rozwiązania.

SA — Simulated Annealing

Wyżarzanie metali — skąd pomysł?

- **Wyżarzanie metalu (metalurgia)**
  - **Etap 1 — Podgrzewanie:** metal jest podgrzewany do wysokiej temperatury. Atomy mają dużo energii i mogą swobodnie zmieniać położenie.
  - **Etap 2 — Wolne chłodzenie:** temperatura stopniowo spada, a atomy poszukują konfiguracji o minimalnej energii.
  - **Wynik:** powstaje kryształ o strukturze globalnie optymalnej (najniższej energii).
  - **Zbyt szybkie chłodzenie:** powstaje szkło — atomy zostają uwięzione w lokalnym minimum energii.
- **Kluczowe odkrycie Kirkpatricka (1983)**

  Rozkład Boltzmanna z fizyki statystycznej można zastosować bezpośrednio do optymalizacji kombinatorycznej.
- **Analogia: fizyka $\rightarrow$ optymalizacja**

Fizyka	**Optymalizacja**
Energia atomu E	Wartość funkcji celu $f(x)$
Konfiguracja atomów	Rozwiązanie x
Zmiana konfiguracji	Ruch do sąsiada x'
Temperatura T	Parametr kontrolny T
Stan równowagi	Lokalne optimum
Kryształ = minimalna energia	Globalne optimum

- **Historia**
  - Metropolis et al. (1953): algorytm symulacji Monte Carlo dla fizyki.
  - Kirkpatrick, Gelatt, Vecchi (1983): zastosowanie do optymalizacji — ponad 30 000 cytowań.
  - Černý (1985): niezależne odkrycie.
  - Jeden z najczęściej cytowanych artykułów w historii informatyki.

Kryterium akceptacji — serce SA

- **Kryterium Metropolisa (1953):**

$$P(\text{akceptuj}) = \exp\left(-\frac{\Delta E}{T}\right)$$

gdzie:

$$\Delta E = f(x') - f(x)$$

Dla minimalizacji: $\Delta E > 0$ oznacza pogorszenie rozwiązania.

- **Pełna reguła akceptacji SA**
  - Jeśli: $f(x') \leq f(x)$ zawsze akceptuj (ruch poprawiający lub neutralny).
  - Jeśli: $f(x') > f(x)$ zaakceptuj z prawdopodobieństwem:

$$P = \exp\left(-\frac{\Delta E}{T}\right)$$

  - Wygeneruj: $r \sim Uniform(0, 1)$ Jeśli: $r < P$ zaakceptuj ruch mimo pogorszenia.

- **Wpływ temperatury** T
  - $T \to \infty$: $P \to 1$ akceptujemy prawie wszystko (eksploracja).
  - $T \to 0$: $P \to 0$ akceptujemy tylko poprawy (zachowanie jak HC).
  - **Optymalne** T**:** kompromis pomiędzy eksploracją i eksploatacją.

- **Przykład numeryczny (**$\Delta E = +100$**)**

Temperatura T	$P = \exp(-100/T)$	**Interpretacja**
1000	0.905	$\sim 90\%$ szansa akceptacji
500	0.819	$\sim 82\%$ szansa
100	0.368	$\sim 37\%$ szansa
50	0.135	$\sim 14\%$ szansa
10	0.000045	prawie 0%
1	≈ 0	jak Hill Climbing

- **Intuicja**

  Na początku (wysoka temperatura T) SA wykonuje „pijacki krok” — swobodnie eksploruje przestrzeń rozwiązań.

  Z czasem, gdy temperatura maleje, algorytm staje się coraz bardziej konserwatywny.

  Kluczowym elementem jest odpowiedni harmonogram chłodzenia.

Pseudokod SA — pełna wersja

Algorytm Simulated Annealing (SA)

```
function SimulatedAnnealing(T0, alpha, T_min, L):
    x = init_solution()     # losowe lub konstrukcyjne
    x_best = x
    T = T0                  # temperatura początkowa

    while T > T_min:        # kryterium stopu
        for i = 1 to L:     # L kroków na temperaturę
            x' = random_neighbor(x)   # losowy sąsiad
            ΔE = f(x') - f(x)
            if ΔE <= 0:
                # poprawa -> zawsze akceptuj
                x = x'
            else:
                # pogorszenie -> akceptacja probabilistyczna
                r = random(0,1)
                if r < exp(-ΔE / T):
                    x = x'

            if f(x) < f(x_best):
                # zapamiętaj najlepsze rozwiązanie
                x_best = x

        T = T * alpha          # geometryczne chłodzenie
    return x_best
```

Parametry SA i ich znaczenie

- T_0 **(temperatura początkowa):** powinna być wystarczająco wysoka, aby większość ruchów była akceptowana (około 80%).
- α **(współczynnik chłodzenia):** zazwyczaj: $0.90 - 0.999$ Im bliżej 1, tym wolniejsze chłodzenie.
- T_{min} **(temperatura końcowa):** zazwyczaj: $0.001 - 1.0$ określa moment zakończenia algorytmu.
- L **(długość łańcucha):** liczba iteracji wykonywanych dla jednej temperatury.
  Typowo: $L = |N(x)|$ lub $L = n$

Typowe wartości dla TSP-100

- $T_0 = 1000 - 10000$ (zależnie od średniej wartości ΔE).
- $\alpha = 0.95 - 0.999$
- $T_{min} = 0.1 - 1.0$
- $L = n$ gdzie n oznacza liczbę miast.
- Łączna liczba iteracji: $\sim 10^5 - 10^7$

Harmonogramy chłodzenia w SA

- **Geometryczny (standard)**

$$T(k+1) = \alpha \cdot T(k)$$

  - $\alpha \in (0.9, 0.999)$
  - Prosta implementacja.
  - Najczęściej używany harmonogram w praktyce.
  - Problem: stała szybkość chłodzenia niezależnie od krajobrazu.
  - Wymagana liczba iteracji:

$$N = \frac{\log(T_{min}/T_0)}{\log(\alpha)}$$

- **Liniowy**

$$T(k+1) = T(k) - \Delta$$

  gdzie:

$$\Delta = \frac{T_0 - T_{min}}{N_{iter}}$$

  - Bardzo prosta implementacja.
  - Nierównomierne chłodzenie: na początku (wysokie T) chłodzi zbyt wolno, a na końcu zbyt szybko.
  - Zwykle daje słabszą jakość rozwiązań.
  - Rzadko stosowany w praktyce.

- **Adaptacyjny (zaawansowany)**

$$T \downarrow \quad \text{gdy} \quad acc\_rate > target$$

  - Monitorowanie współczynnika akceptacji co L iteracji.
  - Jeśli akceptowanych jest zbyt dużo ruchów: zmniejsz temperaturę szybciej.
  - Jeśli akceptowanych jest zbyt mało: spowolnij chłodzenie.
  - Utrzymuje:

$$acc\_rate \approx 20\% - 40\%$$

  - Najlepsza jakość wyników, ale trudniejsza implementacja.

- **Jak ustawić T_0?**

  Praktyczna reguła:

  1. Wykonaj 100 losowych ruchów.
  2. Oblicz średnią wartość pogorszenia:

$$\Delta E_{avg}$$

  3. Ustaw T_0 tak, aby początkowe prawdopodobieństwo akceptacji było:

$$P_{start} = \exp\left(-\frac{\Delta E_{avg}}{T_0}\right) \approx 0.8$$

  4. Stąd:

$$T_0 = -\frac{\Delta E_{avg}}{\ln(0.8)}$$

  Gwarantuje to około 80% akceptacji ruchów na początku działania SA.

- Im wolniejsze chłodzenie, tym lepsza jakość — ale czas rośnie liniowo z $1/(1-\alpha)$. Dla $\alpha = 0.999$ vs. $\alpha = 0.90$: 100x więcej iteracji dla $\approx 8\%$ poprawy jakości.
  Diminishing returns — wybierz alpha=0.99 jako dobry kompromis.

TS — Tabu Search

Czym jest Tabu Search?

- **Kluczowe idee Tabu Search (TS)**
  - **Nie zatrzymuj się w lokalnym optimum:** TS kontynuuje działanie nawet wtedy, gdy każdy dostępny ruch pogarsza wartość funkcji celu $f(x)$.
  - **Wybierz zawsze najlepszy dozwolony ruch:**

    $$x' = \arg\min\{f(x'') : x'' \in N(x),\ x'' \notin Tabu\}$$

    Nie ma losowania — wybór jest deterministyczny.
  - **Pamiętaj ostatnie ruchy:** lista Tabu przechowuje ostatnio wykonane ruchy.
  - **Zabraniaj powrotów:** ruchy znajdujące się na liście Tabu są zakazane przez T_{tabu} iteracji.

  Jak TS wychodzi z lokalnego optimum?

  W lokalnym optimum wszyscy sąsiedzi są gorsi, dlatego Hill Climbing zatrzymuje się.

  Tabu Search wybiera najlepszego dostępnego sąsiada (nawet jeśli jest gorszy od bieżącego rozwiązania), a następnie zabrania powrotu do poprzednich ruchów przez T_{tabu} iteracji.

  Wymusza to eksplorację nowych regionów przestrzeni rozwiązań.

- **Porównanie mechanizmów ucieczki**

Metoda	Mechanizm ucieczki	Pamięć
HC	brak — zatrzymuje się	brak
SA	losowa akceptacja: $P = \exp(-\Delta E/T)$	brak
TS	najlepszy dozwolony ruch	lista tabu
VNS	zmiana sąsiedztwa	brak

- **Historia i znaczenie**
  - Glover (1986): Future Paths for Integer Programming.
  - Pełna formalizacja: Glover (1989, 1990).
  - Szczególnie skuteczny dla problemów: QAP, scheduling, routing.
  - Często przewyższa SA w problemach kombinatorycznych.
  - Stanowi inspirację dla wielu współczesnych metaheurystyk.

Lista Tabu — implementacja i dylemat rozmiaru

- **Co przechowujemy na liście Tabu?**

- **Opcja A — Rozwiązania:** zapamiętywane jest całe rozwiązanie x.

  Proste, ale pamięciochłonne dla dużych problemów. Blokuje dokładne powtórzenia rozwiązań.
- **Opcja B — Ruchy (standard):** zapamiętywane są atrybuty ostatnio wykonanego ruchu.

  Dla TSP (2-opt): para (i, j).

  Efektywne, ale może blokować różne rozwiązania wynikające z tego samego ruchu.
- **Opcja C — Zmiany:** zapamiętywane są zmienione cechy.

  Przykład: dla problemu plecakowego — indeks dodanego/usuniętego przedmiotu.

- **Tenure — czas życia na liście**

  - T_{tabu}: liczba iteracji, przez którą ruch jest zabroniony.
  - Po T_{tabu} iteracjach ruch jest usuwany z listy (FIFO).
  - Typowe wartości:

    $$T_{tabu} = 5 - 20$$

  - Implementacja: kolejka FIFO lub tablica liczników iteracji.

- **Dylemat rozmiaru listy Tabu**

  - T_{tabu} **za małe (np. 1–3):** zbyt mało pamięci -> cykliczne powroty do lokalnych optimów.
  - T_{tabu} **za duże (np.** $n/2$**):** zbyt restrykcyjne -> pomijanie dobrych rozwiązań.
  - T_{tabu} **optymalne:** zazwyczaj: $\sqrt{n}$ do n dla problemu TSP.
  - **Dynamiczne** T_{tabu}**:** adaptacyjne zmiany podczas przeszukiwania.

- **Praktyczna zasada**

  Dla TSP-n:

  - zacznij od:

    $$T_{tabu} = \sqrt{n}$$

  - jeśli TS „kręci się w kółko" -> zwiększ T_{tabu},
  - jeśli prawie brak ruchów dopuszczalnych -> zmniejsz T_{tabu},
  - dynamiczna tenure:

    $$T_{tabu} \in [T_{min}, T_{max}]$$

    często działa lepiej niż stała wartość.

Kryterium Aspiracji — kiedy łamać regułę tabu?

- **Problem z listą tabu bez kryterium aspiracji**

  Może się zdarzyć, że ruch prowadzący do nowego globalnego optimum znajduje się na liście tabu — i Tabu Search go wtedy ignoruje.

  Kryterium aspiracji pozwala „nadpisać" regułę tabu w szczególnych przypadkach.

- **Kryterium aspiracji (A1–A3)**
  - **A1 — Poprawa globalnego optimum (standard):**

    Jeśli ruch tabu prowadzi do:

    $$f(x') < f(x_{best})$$

    to ruch jest wykonywany mimo tabu.

    Najczęściej stosowane kryterium.
  - **A2 — Poziom aspiracji:**

    Definiuje się próg:

    $$aspiration\_level$$

    Jeśli:

    $$f(x') < aspiration\_level$$

    ruch jest dopuszczony.

    Bardziej liberalne podejście.
  - **A3 — Intensywność poszukiwań:**

    Jeśli ruch jest tabu, ale dany obszar nie był długo eksplorowany, ruch zostaje zaakceptowany.

    Promuje dywersyfikację przeszukiwania.

- **Pseudokod z kryterium aspiracji A1**

```
for x2 in N(x):
    move = get_move(x, x2)

    if move not in tabu_list:
        candidates.append(x2)

    elif f(x2) < f(x_best):
        # KRYTERIUM ASPIRACJI A1
        # ruch tabu, ale poprawia globalne optimum
        candidates.append(x2)

x_next = argmin(f, candidates)
```

- **Co jeśli wszystkie ruchy są tabu?**
  - Sytuacja rzadka, ale możliwa przy dużym T_{tabu}.
  - **Rozwiązanie 1:** dynamiczne zmniejszenie T_{tabu}.
  - **Rozwiązanie 2:** wybór najlepszego ruchu spośród tabu (ignorowanie listy).
  - **Rozwiązanie 3:** restart z nowego punktu startowego.

Pseudokod Tabu Search

- **Pseudokod**

```
function TabuSearch(T_tabu, max_iter, max_no_improve):
    x = init_solution()
    x_best = x
    tabu = []   # FIFO kolejka długości T_tabu
    no_improve = 0

    for iter = 1 to max_iter:
        best_candidate = None
        best_f = ∞
        for x2 in N(x):   # PEŁNE sąsiedztwo!
            move = get_move(x, x2)
            if move not in tabu:
                if f(x2) < best_f:
                    best_candidate = x2
                    best_f = f(x2)
            elif f(x2) < f(x_best):
                # kryterium aspiracji A1
                best_candidate = x2
                best_f = f(x2)

        x_prev = x
        x = best_candidate
        tabu.append(get_move(x_prev, x))

        if len(tabu) > T_tabu:
            tabu.pop(0)

        if f(x) < f(x_best):
            x_best = x
            no_improve = 0
        else:
            no_improve += 1

        if no_improve > max_no_improve:
            break

    return x_best
```

- **Kluczowa różnica SA vs. TS w implementacji**
  - SA: losowy pojedynczy sąsiad $\rightarrow O(1)$ na iterację + ewaluacja.
  - TS: pełne sąsiedztwo $\rightarrow O(|N(x)|)$ na iterację + ewaluacje.
  - Dla TSP z 2-opt:
    $$|N(x)| = O(n^2)$$
  - TS ma mniej iteracji, ale każda jest znacznie droższa obliczeniowo.
  - Inkrementalna ewaluacja jest kluczowa dla wydajności TS.
- **Parametry TS**
  - T_{tabu}: rozmiar listy tabu (parametr kluczowy).
  - max_iter: maksymalna liczba iteracji.
  - $max\_no\_improve$: kryterium stopu bez poprawy.
  - Mniej parametrów niż SA -> łatwiejsze strojenie w praktyce.

Trzy poziomy pamięci w Tabu Search

- **Pamięć krótkoterminowa**
  - **Co:** lista ostatnich T_{tabu} ruchów.
  - **Cel:** zapobieganie natychmiastowym cyklom.
  - **Implementacja:** kolejka FIFO.
  - **Horyzont:** T_{tabu} iteracji.
  - **Zawartość:** atrybuty ruchów (np. swap (i, j)).
  - **Kryterium aspiracji:** może nadpisać zakaz.

  Standardowy element każdego Tabu Search.
- **Pamięć średnioterminowa**
  - **Co:** regiony przestrzeni z najlepszymi znalezionymi rozwiązaniami.
  - **Cel:** intensyfikacja — głębsza eksploracja obiecujących obszarów.
  - **Implementacja:** mapa częstotliwości + powrót do x_{best} + dodatkowe kryteria aspiracji.
  - **Horyzont:** dziesiątki–setki iteracji.
  - **Kiedy:** po braku poprawy przez K iteracji.

  Stosowana w bardziej zaawansowanych implementacjach.

- **Pamięć długookresowa**
  - **Co:** nieodwiedzone lub rzadko odwiedzane regiony przestrzeni.
  - **Cel:** dywersyfikacja — eksploracja nowych obszarów.
  - **Implementacja:** mapa częstotliwości ruchów oraz kara za często odwiedzane obszary.
  - **Horyzont:** całe przeszukiwanie.
  - **Kiedy:** gdy algorytm utyka w jednym regionie.

  Stosowana w zaawansowanych implementacjach.

- **Strategia Tabu Search w praktyce**
  - **(1) Krótkoterminowa pamięć:** zawsze aktywna.
  - **(2) Intensyfikacja:** po ok. 100 iteracjach bez poprawy: powrót do x_{best} i lokalne przeszukiwanie.
  - **(3) Dywersyfikacja:** po kolejnych ok. 200 iteracjach bez poprawy: restart z punktu odległego od x_{best}.

Simulated Annealing vs Tabu Search — kiedy co wybrać?

- **Simulated Annealing (SA)**

Cecha	**Opis**
Mechanizm ucieczki	Probabilistyczny: $P = \exp(-\Delta E/T)$
Pamięć	Brak (metoda bezstanowa)
Wybór sąsiada	Losowy pojedynczy sąsiad
Koszt iteracji	$O(1)$ + ewaluacja funkcji celu
Parametry	T_0, α, T_{min}, L
Implementacja	Bardzo prosta (ok. 30 linii kodu)
Zbieżność	Stochastyczna, teoretycznie gwarantowana (logarithmic cooling)
Najlepszy dla	Problemy ciągłe lub mieszane, mniej strukturalne

- **Tabu Search (TS)**

Cecha	Opis
Mechanizm ucieczki	Deterministyczny: wybór najlepszego dozwolonego ruchu
Pamięć	Lista tabu (krótkoterminowa + opcjonalnie średnio/długoterminowa)
Wybór sąsiada	Najlepszy dozwolony w $N(x)$
Koszt iteracji	$O(\|N(x)\|)$ + ewaluacje
Parametry	T_{tabu}, max_iter
Implementacja	Średnia (ok. 60 linii kodu)
Zbieżność	Brak formalnych gwarancji
Najlepszy dla	Problemy kombinatoryczne: QAP, scheduling, routing

VNS — Variable Neighborhood Search

Dlaczego zmiana sąsiedztwa pomaga uciec z lokalnego optimum?

- **Kluczowa obserwacja (Hansen & Mladenović, 1997):**

  1. Lokalne optimum dla N_1 może nie być lokalnym optimum dla N_2.
  2. Globalne optimum jest lokalnym optimum dla **każdego** sąsiedztwa.
  3. Dla wielu problemów kombinatorycznych lokalne optima dla różnych sąsiedztw są inaczej rozmieszczone w przestrzeni rozwiązań.

- **Formalna definicja**

  Niech:

  $$N_1, N_2, \ldots, N_{k_{max}}$$

  będzie rodziną sąsiedztw.

  Rozwiązanie x^* jest lokalnym optimum dla N_k, jeśli:

  $$\forall x' \in N_k(x^*): \quad f(x') \geq f(x^*)$$

  VNS wykorzystuje fakt:

  $$\text{lok.opt.}_{N_k} \neq \text{lok.opt.}_{N_{k+1}}$$

Typowa hierarchia sąsiedztw:

$$N_1 \subset N_2 \subset \cdots \subset N_{k_{max}}$$

(zagnieżdżone lub niezagnieżdżone).

- **Przykład: TSP z sąsiedztwami k-opt**
  - $N_1 = 2$**-opt:**

    Zamiana 2 krawędzi: $|N(x)| = O(n^2)$
  - $N_2 = 3$**-opt:**

    Zamiana 3 krawędzi: $|N(x)| = O(n^3)$
  - $N_3 = 4$**-opt:**

    Zamiana 4 krawędzi: $|N(x)| = O(n^4)$
  - **Double-bridge (4-opt):** specjalna perturbacja 4-opt, której nie można cofnąć jednym ruchem 2-opt.
  - Każde sąsiedztwo definiuje inny „krajobraz" przestrzeni rozwiązań, a więc inne lokalne optima.
- **Dwa rodzaje hierarchii sąsiedztw**
  - **Zagnieżdżone:**

    $$N_1 \subset N_2 \subset N_3$$

    Każde kolejne sąsiedztwo zawiera poprzednie.

    Przykład:

    $$1\text{-opt} \subset 2\text{-opt} \subset 3\text{-opt}$$
  - **Niezagnieżdżone:**

    $$N_1, N_2, N_3$$

    są rozłączne i reprezentują różne operatory.

    Przykład:

    $$2\text{-opt}, \quad or\text{-opt}, \quad 3\text{-opt}$$
  - W praktyce: niezagnieżdżone sąsiedztwa często zapewniają lepszą dywersyfikację.

Basic VNS — pseudokod i trzy fazy

- **Pseudokod**

```
function BasicVNS(N={N1,...,Nkmax}, t_max):
    x = init_solution()
    t = 0

    while t < t_max:          # kryterium stopu (czas/iter)
        k = 1
        while k <= k_max:

            # FAZA 1: Shaking
            x2 = random_point(N_k(x))

            # FAZA 2: Local Search od x2
            x3 = local_search(x2)
            # np. HC z N1

            # FAZA 3: Move or Not
            if f(x3) < f(x):
                x = x3
                k = 1          # reset - wróć do N1
            else:
                k = k + 1      # następne sąsiedztwo

        t = t + 1
    return x
```

- **Trzy fazy BVNS**

  - **Shaking (perturbacja):**

    Losowanie punktu:

$$x' = random\_point(N_k(x))$$

    Punkt jest oddalony od bieżącego rozwiązania x, ale znajduje się w sąsiedztwie N_k.

    Im większe k, tym silniejsza perturbacja i większy skok w przestrzeni rozwiązań.

- **Local Search:**

  Przeszukiwanie lokalne od x' aż do lokalnego optimum.

  Najczęściej używane jest sąsiedztwo N_1 z Hill Climbing.

  Wynik:

  $$x'' = \text{lokalne optimum w } N_1$$

- **Move or Not:**

  Akceptacja tylko poprawy: $f(x'') < f(x)$ wtedy: $x = x'', \quad k = 1$

  W przeciwnym przypadku: $k = k + 1$ i używane jest kolejne sąsiedztwo.

- **Dlaczego reset $k = 1$ po poprawie?**

  - Po znalezieniu lepszego rozwiązania x nowe sąsiedztwo $N_1(x)$ jest inne — zmienia się krajobraz i dostępne ruchy.
  - Należy zacząć od najmniejszego sąsiedztwa N_1: zapewnia to szybką eksplorację bliskiego otoczenia.
  - Większe sąsiedztwa N_k są używane dopiero wtedy, gdy mniejsze nie pozwalają znaleźć poprawy.

Rodzina algorytmów VNS

- **VND — Variable Neighborhood Descent**

  - Deterministyczna wersja VNS (bez fazy shaking).
  - Przeszukuje:

    $$N_1 \to N_2 \to N_3 \to \dots$$

    aż do lokalnego optimum.
  - Gdy pojawi się poprawa w N_k: wróć do N_1 (reset).
  - Gdy nie ma poprawy w żadnym sąsiedztwie: zakończ algorytm.
  - Gwarantuje:

    $$x^* \text{ jest lokalnym optimum dla wszystkich } N_k$$

  - Silniejsze kryterium zatrzymania niż Hill Climbing.

Pseudokod:

```
function VND(N={N1,...,Nkmax}):
    k = 1
    while k <= k_max:

        x2 = best_neighbor(x, N_k)

        if f(x2) < f(x):
            x = x2
            k = 1
        else:
            k = k + 1

    return x
```

- **GVNS — General VNS**
  - BVNS z procedurą Local Search zastąpioną przez VND.
  - Schemat:

$$Shaking(N_k) \rightarrow VND(x') \rightarrow Move\ or\ Not$$

  - Najsilniejsza wersja VNS.
  - Kosztowna obliczeniowo: VND wykonywane w każdej iteracji.
  - Najlepsze wyniki dla problemów: TSP oraz scheduling.
- **RVNS — Reduced VNS**
  - Shaking w N_k bez fazy Local Search.
  - Akceptacja wyłącznie poprawy:

$$f(x') < f(x)$$

  - Szybszy i prostszy niż BVNS.
  - Stosowany, gdy Local Search jest bardzo kosztowny.
  - Jakość wyników zwykle gorsza niż BVNS/GVNS.

- **Podsumowanie wariantów**

Wariant	Shaking	LS	Jakość
RVNS	✓	×	niska
BVNS	✓	HC/N_1	dobra
GVNS	✓	VND	wysoka
VND	×	VND	średnia/wysoka

Jak projektować hierarchię sąsiedztw do VNS?

- **Zasady projektowania hierarchii N_k**
  - **Rosnąca siła perturbacji:**

$$N_1 \text{ mała perturbacja (blisko } x), \quad N_{k_{max}} \text{ duża (daleko od } x)$$

  - **Różnorodność:** sąsiedztwa N_k powinny generować „różne" rozwiązania, a nie tylko skalować ten sam operator.
  - **Efektywność:** shaking w N_k musi być szybki:

$$O(1) \text{ lub } O(n)$$

  - **Dostosowanie do problemu:** sąsiedztwa powinny wynikać ze struktury problemu.
  - **Kluczowa różnica:**

$$\text{Shaking} \neq \text{Local Search}$$

    Shaking generuje punkt startowy, nie lokalne optimum.

- **Przykład dla TSP**
  - N_1 **— shaking 2-opt:** losowa zamiana 2 krawędzi. Prosta, ale słaba perturbacja.
  - N_2 **— or-opt(1):** przesunięcie 1 miasta w losowe miejsce.
  - N_3 **— or-opt(3):** przesunięcie 3 kolejnych miast.
  - N_4 **— double-bridge:** silna perturbacja 4-opt, której nie da się cofnąć jednym ruchem 2-opt.

- **Double-Bridge — dlaczego wyjątkowy?**
  - Dzieli trasę na 4 segmenty: $A - B - C - D$.
  - Łączy je w nowej kolejności:
    $$A - C - B - D$$
  - Wynik: trasa niemożliwa do uzyskania jednym 2-opt.
  - Gwarantuje przejście do nowego basenu przyciągania 2-opt.
  - Kluczowy składnik Iterated Local Search (część 3).
  - Połączenie:
    $$\text{Lin-Kernighan + double-bridge} = LKH$$
    (najlepszy solver TSP).
- **Praktyczna wskazówka**
  - Dla Local Search (VND) używaj małych sąsiedztw: 2-opt, or-opt.
  - Dla shaking używaj dużych i „nieodwracalnych": double-bridge.
  - Dzięki temu po Local Search nie wracamy natychmiast do poprzedniego lokalnego optimum.

Zestawienie trzech S-metaheurystyk — kiedy co wybrać?

- **Simulated Annealing**
  - **Mechanizm:** probabilistyczna akceptacja:
    $$\exp(-\Delta E/T)$$
  - **Pamięć:** brak (algorytm bezstanowy)
  - **Koszt iteracji:**
    $$O(1)$$
    — jeden losowy sąsiad
  - **Parametry:** T_0, α, T_{min}, L (4 parametry)
  - **Dla TSP-52:** ~ 7570 (0.4% od optimum)
  - **Zalety:** prosta implementacja, dowód zbieżności, metoda ogólna
  - **Wady:** trudne strojenie temperatury, wolna dla $n \gg 1000$

- **Tabu Search**
  - **Mechanizm:** deterministyczny — najlepszy dozwolony sąsiad
  - **Pamięć:** lista tabu (krótkoterminowa), opcjonalnie mapa częstotliwości
  - **Koszt iteracji:**

    $$O(|N(x)|)$$

    — pełne sąsiedztwo
  - **Parametry:** T_{tabu}, max_{iter} (2 kluczowe)
  - **Dla TSP-52:** ~ 7550 (0.1% od optimum)
  - **Zalety:** dobre dla problemów kombinatorycznych, niewiele parametrów
  - **Wady:** kosztowny per iterację, trudny wybór atrybutów tabu
- **VNS**
  - **Mechanizm:** zmiana sąsiedztwa + lokalne przeszukiwanie (LS)
  - **Pamięć:** brak (w BVNS)
  - **Koszt iteracji:** zależny od LS:

    $$O(LS)$$

  - **Parametry:** k_{max}, wybór sąsiedztw (najtrudniejszy element)
  - **Dla TSP-52:** ~ 7545 (0.04% od optimum)
  - **Zalety:** elegancki, skuteczny, dobrze współpracuje z double-bridge, brak pamięci
  - **Wady:** wymaga starannego projektowania hierarchii sąsiedztw

Drzewo decyzyjne: SA vs. TS vs. VNS

- **Użyj SA gdy...**
  - Problem jest ciągły lub mieszany (nie czysto kombinatoryczny)
  - Potrzebujesz szybkiego prototypu — SA jest łatwy w implementacji
  - Ewaluacja $f(x)$ jest tania (każda iteracja wymaga obliczenia f)
  - Nie masz dobrej definicji „atrybutów ruchów" dla Tabu Search
  - Potrzebujesz teoretycznej gwarancji zbieżności (logarytmiczne chłodzenie)
  - Problem ma wiele lokalnych optimów o podobnej strukturze basenów przyciągania

  Typowe zastosowania: scheduling, routing, VLSI placement, hyperparameter tuning
- **Użyj TS gdy...**
  - Problem jest czysto kombinatoryczny (QAP, permutacje)
  - Masz dobrze zdefiniowane ruchy z naturalnymi atrybutami
  - Inkrementalna ewaluacja Δf jest tania
  - Chcesz deterministycznego zachowania (reprodukowalność)
  - Masz ograniczony budżet iteracji — TS daje dobre wyniki szybko
  - Zależy Ci na intensywnej eksploracji jednego regionu

  Typowe zastosowania: QAP, Job Shop Scheduling, Vehicle Routing, Graph Coloring
- **Użyj VNS gdy...**
  - Masz kilka naturalnych operatorów sąsiedztwa dla problemu
  - Dysponujesz dobrą procedurą Local Search (np. HC z 2-opt dla TSP)
  - Chcesz prostoty implementacji bez pamięci
  - Problem ma hierarchię sąsiedztw:

    $$1\text{-opt} \subset 2\text{-opt} \subset 3\text{-opt}$$

  - Szukasz wysokiej jakości bez złożoności Tabu Search
  - Problem jest podobny do TSP / routing (VNS + double-bridge = state of the art)

  Typowe zastosowania: TSP, VRP, Bin Packing, Network Design

SA, TS i VNS — gdzie są używane w praktyce?

- **SA w praktyce**
  - VLSI Design (IBM, 1983): rozmieszczenie komponentów układów scalonych — pierwszy duży sukces przemysłowy SA
  - Strojenie hiperparametrów ML: AutoML, Neural Architecture Search (NAS)
  - Optymalizacja portfela: model Markowitza z ograniczeniami (finanse)
  - Modelowanie białek: struktura 3D białek (bioinformatyka)
  - Harmonogramowanie produkcji: job shop, flow shop scheduling
- **Tabu Search w praktyce**
  - Routing pojazdów (VRP): UPS, FedEx — optymalizacja tras dostaw (np. POPMUSIC i inne duże implementacje TS)
  - Przydział częstotliwości: sieci komórkowe — minimalizacja zakłóceń
  - Job shop scheduling: optymalizacja produkcji (Nowicki & Smutnicki, 1996 — rekordowe wyniki dla benchmarków)
  - Graph Coloring: planowanie zajęć, przydział zasobów
  - Bioinformatyka: wyrównywanie sekwencji genomowych
- **VNS w praktyce**
  - TSP — LKH (Lin-Kernighan-Helsgaun): VNS z double-bridge — jeden z najlepszych solverów TSP
  - VRP (Vehicle Routing): ALNS (Adaptive Large Neighborhood Search) — rozszerzenie idei VNS dla logistyki
  - Bin Packing: pakowanie kontenerów, cięcie materiałów
  - Network Design: projektowanie sieci telekomunikacyjnych
  - Data Mining: klasteryzacja p-median

Fundamentalny dylemat: intensyfikacja vs. dywersyfikacja

- **Fundamentalny dylemat optymalizacji**
  - **Intensyfikacja (exploitation):** głęboka eksploracja obiecującego regionu.
    Ryzyko: utknięcie w lokalnym optimum.
  - **Dywersyfikacja (exploration):** eksploracja nowych, niezbadanych regionów.
    Ryzyko: marnowanie czasu na słabe obszary.
- **Jak SA balansuje?**
  - Wysoka temperatura $T \rightarrow$ dywersyfikacja (akceptacja pogorszeń)
  - Niska temperatura $T \rightarrow$ intensyfikacja (zachowanie jak Hill Climbing)
  - Harmonogram chłodzenia kontroluje przejście między fazami
  - Wolne chłodzenie $\rightarrow$ dłuższa dywersyfikacja i zwykle lepsza jakość rozwiązań
- **Jak Tabu Search balansuje?**
  - Lista tabu (krótkoterminowa) $\rightarrow$ unikanie cykli i lokalna intensyfikacja w nowych kierunkach
  - Pamięć częstotliwości (długoterminowa) $\rightarrow$ kara za często odwiedzane regiony (dywersyfikacja)
  - Intensyfikacja: powrót do x_{best} + głębokie przeszukiwanie lokalne
  - Dywersyfikacja: restart z punktów dalekich od odwiedzonych regionów
- **Jak VNS balansuje?**
  - Local Search (z N_1) $\rightarrow$ intensyfikacja: zbieganie do lokalnego optimum
  - Shaking (z N_k) $\rightarrow$ dywersyfikacja: przejście do nowego regionu przestrzeni
  - Większe $k \rightarrow$ silniejsza dywersyfikacja
  - Akceptacja tylko poprawy $\rightarrow$ silna intensyfikacja

Iterated Local Search

ILS — Iterowane Przeszukiwanie Lokalne

- **Kluczowa idea ILS**
  - **Problem z Random Restart:** losowa inicjalizacja ignoruje informację o znanych dobrych rozwiązaniach i marnuje część budżetu obliczeniowego.
  - **Idea ILS:** zamiast losowo restartować algorytm, zaburz (perturb) obecne dobre rozwiązanie x^* i uruchom Local Search od nowego punktu x'.
  - **Efekt:** eksploracja pobliskich basenów przyciągania — nie całkowicie losowa, ale jednocześnie nie ograniczona do jednego optimum.
  - **Metafora:** jeśli zakopałeś skarb w pobliżu, nie szukasz ponownie na całym świecie, ale sprawdzasz obszar trochę dalej niż poprzednio.
- **Kluczowe spostrzeżenie**
  - Lokalne optima znajdujące się blisko globalnego optimum często są skupione w podobnych basenach przyciągania.
  - ILS eksploruje sąsiednie baseny poprzez perturbację, zamiast losowo szukać całkowicie nowego obszaru.
- **ILS vs. Random Restart — analogia**

Kryterium	**Random Restart**	**ILS**
Punkt startowy LS	Losowy	Zaburzony x^*
Pamięć	Brak	x^* (bieżące optimum)
Siła perturbacji	Maksymalna (cały obszar)	Kontrolowana
Eksploracja	Globalna, nieukierunkowana	Lokalna, ukierunkowana
Efektywność	Niska dla dużych n	Wyższa dzięki wykorzystaniu informacji

Pseudokod ILS — cztery komponenty

- **Pseudokod**

```
function ILS(max_iter):

    # (1) Inicjalizacja
    x0 = generate_initial()     # losowe lub zachłanne
    x* = LocalSearch(x0)        # pierwsze lokalne optimum

    for i = 1 to max_iter:

        # (2) Perturbacja - zaburz x*
        x' = Perturbation(x*, history)

        # (3) Local Search od x'
        x'' = LocalSearch(x')

        # (4) Kryterium akceptacji
        if Accept(x'', x*, history):
            x* = x''
            history.update(x*)

    return x*
```

- **4 decyzje projektowe w ILS**
  - Inicjalizacja — losowa, zachłanna lub znane dobre rozwiązanie x_{best}
  - Perturbacja — wybór operatora i jego siły
  - Local Search — np. HC, SA, TS lub VNS
  - Kryterium akceptacji — decyzja, czy x'' zastępuje x^*

- **Cztery kryteria akceptacji ILS**
  - **Always Accept:**

    $$x^* \leftarrow x''$$

    zawsze akceptuj.

    Silna dywersyfikacja, słaba intensyfikacja.
  - **Better Only:** akceptuj tylko jeśli:

    $$f(x'') < f(x^*)$$

    deterministyczne podejście, ale ryzyko stagnacji.
  - **SA-like:** akceptacja pogorszeń z prawdopodobieństwem:

    $$P = \exp(-\Delta/T)$$

    balans eksploracja–eksploatacja.
  - **Random Walk:** akceptuj losowo niezależnie od jakości.

    czysta dywersyfikacja.

- **Które kryterium wybrać?**
  - Dla TSP: Better Only + silna perturbacja (np. double-bridge) daje najlepsze wyniki.
  - Dla problemów z wieloma basenami przyciągania: SA-like działa lepiej.
  - Praktyczna reguła: zacznij od Better Only, jeśli brak popraw -> przejdź na SA-like.

Siła perturbacji — kluczowy dylemat ILS

- **Trzy scenariusze siły perturbacji**
  - **Za słaba perturbacja:** x' leży w tym samym basenie przyciągania co x^*.

    Local Search konwerguje do tego samego lokalnego optimum.

    ILS nie zmienia regionu — efekt podobny do braku perturbacji.
  - **Optymalna perturbacja:** x' trafia do sąsiedniego basenu przyciągania.

    Local Search konwerguje do innego lokalnego optimum.

    ILS efektywnie eksploruje nowe regiony przestrzeni.
  - **Za silna perturbacja:** x' jest praktycznie losowym rozwiązaniem.

    Strata informacji o x^* — ILS redukuje się do Random Restart.

    Niska efektywność w dużych przestrzeniach.

- **Idealna perturbacja**
  - Przenosi x^* do innego basenu przyciągania,
  - zachowuje część struktury rozwiązania.
  - Dla TSP: double-bridge (4-opt z przeplotem)
  - Nieodwracalna przez 2-opt, ale zachowuje część struktury trasy.
- **Perturbacje dla różnych problemów**
  - **TSP — double-bridge:** podział trasy na segmenty $A - B - C - D$ i rekombinacja $A - C - B - D$.
  - **Job Shop:** random reinsertion — losowe przeniesienie k operacji.
  - **Plecak:** bit-flip k elementów, gdzie k określa siłę perturbacji.
  - **Graph Coloring:** recolor block — zmiana kolorów całego spójnego podgrafu.
  - **Ogólna zasada:**
    - szybka: $O(n)$
    - zmienia 10–30% struktury
    - nieodwracalna przez Local Search
- **ILS dla TSP berlin52 — wpływ perturbacji**

Perturbacja	Wynik śr.	% GAP
1-opt (za słaba)	~ 7820	3.7%
2-opt random	~ 7640	1.3%
Double-bridge ✓	~ 7548	0.08%
5-opt random (za silna)	~ 7710	2.2%

ILS vs. VNS — podobieństwa i różnice

- **ILS — schemat działania**
  - Jedna perturbacja (stała lub adaptacyjna)
  - Po perturbacji zawsze wykonywane jest pełne Local Search aż do lokalnego optimum
  - Kryterium akceptacji decyduje, czy:

$$x^* \leftarrow x''$$

  - Historia opcjonalna (można zapamiętywać odwiedzone lokalne optima)
  - Intuicja: LS z inteligentnym restartowaniem
  - Schemat:

$$x^* \to \text{Perturb} \to x' \to LS \to x'' \to Accept \to x^*$$

- **VNS — schemat działania**
  - Hierarchia sąsiedztw:

$$N_1, N_2, \ldots, N_{k_{max}}$$

  - Shaking: losowy punkt w $N_k(x)$ — coraz dalej przy braku poprawy
  - Po shaking zawsze wykonywany jest Local Search w N_1
  - Po poprawie: reset $k = 1$
  - Intuicja: ILS z wieloma poziomami perturbacji
  - Schemat:

$$x \to Shake(N_k) \to x' \to LS(N_1) \to x'' \to \begin{cases} k = 1, x = x'' & \text{jeśli poprawa} \\ k = k + 1 & \text{w przeciwnym przypadku} \end{cases}$$

- **Porównanie kluczowych aspektów**

Aspekt	**ILS**	**VNS**
Perturbacja	Stała lub adaptacyjna	Hierarchia N_k
Eskalacja	Brak (lub ręczna)	Automatyczna ($k++$)
LS po perturbacji	Zawsze pełny LS	Zawsze LS z N_1
Kryterium akceptacji	Elastyczne (różne warianty)	Tylko poprawa
Prostota	Prostszy	Bardziej systematyczny
Wyniki TSP	Bardzo dobre (double-bridge)	Bardzo dobre (BVNS/GVNS)

- **Kiedy co wybrać**
  - **ILS:** gdy masz jedną dobrą perturbację (np. double-bridge)
  - **VNS:** gdy masz naturalną hierarchię operatorów (np. 2-opt, or-opt, 3-opt)
  - Dla TSP: oba podejścia dają bardzo podobne wyniki przy podobnym budżecie obliczeniowym.

GRASP

GRASP — Restricted Candidate List (RCL)

- **Dwie fazy GRASP**
  - **Faza 1 — Greedy Randomized Construction:** buduj rozwiązanie krok po kroku.

    Na każdym kroku:
    - oblicz koszt wszystkich kandydatów
    - zbuduj RCL (Restricted Candidate List)
    - losowo wybierz element z RCL
  - **Faza 2 — Local Search:** uruchom Local Search (np. HC) od zbudowanego rozwiązania aż do lokalnego optimum.
  - Powtarzaj całość przez max_{iter} i zapamiętuj najlepsze x_{best}.
- **RCL — klucz do GRASP**
  - RCL = Restricted Candidate List
  - Definicja:

$$RCL = \{c \in C : cost(c) \leq c_{min} + \alpha(c_{max} - c_{min})\}$$

  - $\alpha = 0$: tylko najlepszy kandydat (czysto zachłanne podejście)
  - $\alpha = 1$: wszyscy kandydaci (czysto losowe)
  - $\alpha \in (0.1, 0.3)$: wartości typowo stosowane w praktyce
- **Przykład: GRASP dla TSP (Nearest Neighbor)**
  - **Standardowy NN:** zawsze wybieraj najbliższe nieodwiedzone miasto — podejście deterministyczne.
  - **GRASP NN:**
    - oblicz odległości do nieodwiedzonych miast
    - zbuduj RCL: miasta o odległości $\leq d_{min} + \alpha(d_{max} - d_{min})$
    - losowo wybierz miasto z RCL
  - **Efekt:** każda iteracja daje inne rozwiązanie startowe, a Local Search prowadzi do różnych lokalnych optimów.
  - wybieramy najlepsze spośród wielu iteracji (np. 100+).
- **Dlaczego GRASP działa?**
  - Dobra heurystyka konstrukcyjna daje rozwiązania bliskie optimum (zachłanność).
  - Losowość zapewnia różnorodność eksploracji przestrzeni.
  - Local Search doprowadza każde rozwiązanie do lokalnego optimum.
  - Wielokrotne iteracje zwiększają pokrycie przestrzeni rozwiązań.

GRASP + Nearest Neighbor dla TSP — przykład

- **Faza konstrukcji dla TSP ($\alpha = 0.2$)**
  - Start: losowe miasto, np. miasto 1
  - Krok 1:

    Oblicz odległości do 4 nieodwiedzonych miast:

    $$d_2 = 10, \quad d_3 = 25, \quad d_4 = 18, \quad d_5 = 30$$

    $$c_{min} = 10, \quad c_{max} = 30$$

    Próg:

    $$threshold = 10 + 0.2 \times (30 - 10) = 14$$

    RCL:

    $$RCL = \{\text{miasto 2 } (d = 10)\}$$

    Tylko jeden kandydat $\rightarrow$ przejdź do 2.
  - Krok 2:

    Od miasta 2:

    $$d_3 = 12, \quad d_4 = 8, \quad d_5 = 20$$

    $$threshold = 8 + 0.2 \times (20 - 8) = 10.4$$

    $$RCL = \{\text{miasto 4 } (d = 8)\}$$

    Przejdź do 4.
  - Efekt $\alpha = 0.2$:

    Mała losowość — trasy są zbliżone do NN, ale nie identyczne.

- **Wpływ α na jakość (TSP berlin52, 100 iter.)**

α	Opis	Wynik śr.	Std
0.0	Czysto zachłanne	~ 8120	0
0.1	Mała losowość	~ 7820	± 85
0.2	Optimum	~ 7620	± 62
0.3	Średnia losowość	~ 7710	± 90
0.5	Duża losowość	~ 8050	± 140
1.0	Czysto losowe	~ 9100	± 320

- **Reaktywny GRASP**
  - Zamiast stałego $\alpha = 0.2$, algorytm śledzi skuteczność różnych wartości α.
  - Definiujemy:

$$P(\alpha) = \text{prawdopodobieństwo, że } \alpha \text{ dało najlepszy wynik w ostatnich 50 iteracjach}$$

  - W kolejnych iteracjach próbkujemy α zgodnie z rozkładem $P(\alpha)$.
  - Algorytm automatycznie dostosowuje wartość α do konkretnego problemu.

6 S-metaheurystyk — kompletne porównanie

Metoda	Mechanizm ucieczki	Pamięć	Koszt/iterację	Parametry	Impl. trudność	Najlepsze dla	TSP-52 %GAP
HC	Brak — zatrzymuje się	Brak	$O(\|N(x)\|)$	0	$\star$ Łatwa	Inicjalizacja, baseline	$\sim 20\%$
SA	Probabilistyczna (exp)	Brak	$O(1) + eval$	T_0, α, T_{min}, L	$\star\star$ Prosta	Ciągłe, szybki prototyp	$\sim 0.5\%$
TS	Deterministyczna (pamięć)	Lista tabu	$O(\|N(x)\|)$	T_{tabu}, max_iter	$\star\star\star$ Średnia	Kombinatoryczne, QAP	$\sim 0.1\%$
VNS	Zmiana sąsiedztwa	Brak	$O(LS)$ + shaking	k_{max}, wybór N_k	$\star\star\star$ Średnia	TSP, routing, gdy wiele N_k	$\sim 0.04\%$
ILS	Perturbacja + restart LS	x^* (best)	$O(LS)$ + perturb	siła perturbacji	$\star\star$ Prosta	TSP + double-bridge, scheduling	$\sim 0.08\%$
GRASP	Losowa konstrukcja	Brak (między iter.)	$O(konstr.) + O(LS)$	α, max_iter	$\star\star$ Prosta	Gdy dobra heurystyka konstr.	$\sim 0.6\%$

JAK PRAWIDŁOWO PORÓWNYWAĆ ALGORYTMY?

Poprawne porównywanie metaheurystyk

- **Procedura benchmarkingu krok po kroku**
  - Krok 1 — Ustal budżet:

    Zdefiniuj taki sam N_{eval} (liczba ewaluacji funkcji celu) dla każdego algorytmu — jest to jedyna uczciwa miara porównania.
  - Krok 2 — Uruchamiaj wielokrotnie:

    Minimum 30 niezależnych uruchomień z różnymi seed losowości:

    $$seed = 1, 2, \ldots, 30$$
  - Krok 3 — Zbierz metryki:
    - średnia,
    - mediana,
    - odchylenie standardowe,
    - minimum,
    - maksimum,
    - %GAP od optimum.
  - Krok 4 — Testuj:

    Użyj testu Wilcoxon Rank-Sum:

    $$\alpha = 0.05$$

    dla danych nieparametrycznych.
  - Krok 5 — Raportuj:

    Przygotuj tabelę ze wszystkimi metrykami, box-plot oraz czas obliczeń.

- **Wilcoxon Rank-Sum Test**
  - Hipoteza zerowa H_0:

    Wyniki algorytmów A i B pochodzą z tego samego rozkładu (brak różnicy).
  - Hipoteza alternatywna H_1:

    Algorytm A jest systematycznie lepszy od algorytmu B.
  - Test nie zakłada normalności danych — jest odpowiedni dla metaheurystyk.
  - Jeśli:

    $$p\text{-value} < 0.05$$

    odrzucamy H_0 — różnica jest statystycznie istotna.
  - Python:

    ```
    scipy.stats.wilcoxon(results_A, results_B)
    ```

- **Przykład poprawnego raportu benchmarku**

Algorytm	Śr.	Std	Min	Med.	%GAP	p-val vs. SA
SA	7652	48	7590	7648	1.46%	–
TS	7561	22	7543	7559	0.25%	<0.001 ✓
VNS	7548	12	7542	7546	0.08%	<0.001 ✓
ILS	7551	15	7542	7549	0.12%	<0.001 ✓
GRASP	7621	55	7560	7618	1.05%	0.042 ✓

30 uruchomień, budżet $= 10^6$ ewaluacji, $\quad berlin52, \quad optimum = 7542$

- **Częste błędy**
  - Porównanie tylko jednego uruchomienia.
  - Różne budżety obliczeniowe dla algorytmów.
  - Brak testu statystycznego.
  - Raportowanie wyłącznie średniej bez:

    $$std,\ min,\ max$$
  - Testowanie tylko jednej instancji.

    Należy testować minimum 5–10 różnych instancji o różnych rozmiarach.

No Free Lunch Theorem — co tak naprawdę mówi?

- **Twierdzenie No Free Lunch (Wolpert & Macready, 1997):**

  Dla dowolnych dwóch algorytmów A i B:

  jeśli A przewyższa B na pewnej klasie problemów, to B musi przewyższać A na innej klasie problemów o równym „rozmiarze".

  Po uśrednieniu po wszystkich możliwych problemach — każdy algorytm jest równie dobry (lub równie zły).

- **Co NFL oznacza dla praktyki?**

  - Nie istnieje „jeden najlepszy algorytm optymalizacyjny".
  - Dobór algorytmu musi uwzględniać strukturę problemu.
  - Benchmark na reprezentatywnych instancjach jest KONIECZNY.
  - Publikacje typu „nasz algorytm jest najlepszy" bez benchmarku są podejrzane.

- **Co NFL NIE oznacza?**

  - Nie: „Wszystkie algorytmy są tak samo dobre dla TSP" — dla konkretnej klasy problemów różnice są duże.
  - Nie: „Nie warto projektować nowych algorytmów" — warto, bo można je dopasować do konkretnych klas problemów.
  - Nie: „Nie można porównywać algorytmów" — można, ale tylko na konkretnej klasie instancji.
  - Tak: dobre algorytmy są dopasowane do struktury problemu (eksploatacja wiedzy a priori).

- **Praktyczna interpretacja inżynierska**

  - Masz problem TSP $\rightarrow$ TS lub VNS z 2-opt (sprawdzone dla TSP)
  - Masz problem Job Shop $\rightarrow$ TS z operatorem krytycznej ścieżki
  - Masz nowy problem $\rightarrow$ zacznij od SA (prosty), potem TS/VNS
  - Masz super komputer $\rightarrow$ GRASP (łatwo równoległy — embarrassingly parallel)

Zastosowania Heurystyk

Logistyka i Transport — największy obszar zastosowań

- **Problem komiwojażera (TSP) — klasyk**
  - Zastosowania:

    Planowanie tras kurierów, zbieranie odpadów, inspekcja sieci energetycznej, wiercenie PCB.
  - Najlepszy solver:

    LKH (Lin-Kernighan-Helsgaun) — VNS + double-bridge + 5-opt LS.

    Daje rozwiązania optymalne lub bardzo bliskie optimum dla prawie wszystkich znanych instancji TSP o rozmiarach $< 100\,000$ miast.
  - Rekord:

    TSP dla 85 900 miast (USA) — rozwiązany optymalnie przez Concorde (B&B), jednak LKH daje ok. 0.01% od optimum w ułamku czasu.
- **Vehicle Routing Problem (VRP)**
  - Uogólnienie TSP: wiele pojazdów, ograniczenia pojemności, okna czasowe.
  - UPS Project ORION (2012):
    - 55 000 tras dziennie w USA
    - heurystyki oparte na Tabu Search
    - oszczędności: 400 mln $ rocznie
    - redukcja emisji: 100 000 ton CO_2 rocznie
  - POPMUSIC:

    duży system TS dla VRP z milionami klientów — dekompozycja na podproblemy.
  - Przemysł:

    Amazon, FedEx, DHL — własne solvery VRP oparte na TS i SA.
- **Harmonogramowanie (Scheduling)**
  - Job Shop Scheduling:

    Najsłynniejszy wynik TS: Nowicki & Smutnicki (1996) — TS z operatorem krytycznej ścieżki. Przez ponad 20 lat był stanem sztuki.
  - Rozkłady lotów:

    KLM, Lufthansa — SA + TS do planowania załóg i maszyn.
  - Planowanie zajęć:

    Graph Coloring + TS — harmonogramy uczelni (np. University Timetabling Competition).
  - Produkcja przemysłowa:

    harmonogramowanie linii montażowych (SA / TS).

- **Rynek optymalizacji**

  Globalne wydatki na oprogramowanie optymalizacyjne: ponad 1 mld $ rocznie.

  Wiodące systemy:

  - Gurobi
  - CPLEX
  - LocalSolver

  Heurystyki stanowią często rdzeń ich komercyjnych solverów.

Bioinformatyka — optymalizacja w nauce i życiu

- Fałdowanie białek (protein folding)
- Sekwencjonowanie i analiza genomu
- Inne zastosowania biologiczne
  - Projekt eksperymentów: GRASP do optymalnego rozmieszczenia próbek
  - Analiza sieci biologicznych: Klasteryzacja genów — TS
  - Harmonogramowanie lab: Optimal batch scheduling — SA

Machine Learning — optymalizacja hiperparametrów i architektury

- Hyperparameter Optimization (HPO)
- Neural Architecture Search (NAS)
- Python: DEAP dla metaheurystyk

Jak zamodelować nowy problem — podejście inżynierskie

- **Kroki projektowania metaheurystyki dla nowego problemu**
  - Krok 1 — Sformułuj problem:

    Co optymalizujemy (f)? Jakie są ograniczenia? Minimalizacja czy maksymalizacja?
  - Krok 2 — Reprezentacja:

    Jak zakodować rozwiązanie: bity, permutacja, wektor rzeczywisty, drzewo?
  - Krok 3 — Dopuszczalność:

    Jak sprawdzić czy rozwiązanie jest dopuszczalne? Jak naprawić rozwiązania niedopuszczalne?
  - Krok 4 — Sąsiedztwo $N(x)$:

    Jakie ruchy definiują sąsiednie rozwiązania? Jak szybko obliczyć Δf?
  - Krok 5 — Inicjalizacja:

    Losowa czy zachłanna? Zachłanna inicjalizacja zwykle daje lepszy start.
  - Krok 6 — Wybór metody:

    SA / TS / VNS / ILS / GRASP — zależnie od charakterystyki problemu.

- **Typowe reprezentacje i sąsiedztwa**

Problem	Reprezentacja	Sąsiedztwo
TSP	permutacja $[1..n]$	2-opt, 3-opt, or-opt
Plecak	wektor bitowy $\{0,1\}^n$	flip 1 bitu
Job Shop	permutacja zadań	swap, reinsertion
Graph Coloring	wektor kolorów $[1..k]^n$	zmiana koloru węzła
Klasteryzacja	przypisanie $[1..k]^n$	przeniesienie punktu
Harmonogram	macierz przypisań	swap, shift

- **Złota zasada**
  - Sąsiedztwo powinno być **spójne**:
    każde dopuszczalne rozwiązanie osiągalne z każdego innego przez skończoną liczbę ruchów.
  - Sąsiedztwo powinno być **efektywne**:
    Δf obliczalne inkrementalnie w $O(1)$ lub $O(n)$.
  - Sąsiedztwo powinno być **wystarczająco bogate**:
    nie za małe (brak ekspresji), nie za duże (zbyt kosztowne).

Najczęstsze błędy na kolokwium — unikaj ich!

- **TOP 5 Błędów — Teoria i Definicje**
  - B1 — Zły wzór SA:

    $$P = \exp(\Delta E/T)$$

    zamiast:

    $$P = \exp(-\Delta E/T)$$

    Uwaga: $\Delta E = f(x') - f(x)$, więc dla pogorszeń $\Delta E > 0$, a $-\Delta E/T < 0 \Rightarrow P < 1$.
  - B2 — Zła definicja metaheurystyki:
    „Algorytm dający dobre wyniki" ≠ metaheurystyka.
    Poprawnie: „ogólna strategia wysokiego poziomu, niedeterministyczna, bez gwarancji optymalności".
  - B3 — Mylenie TS z SA:
    TS jest **deterministyczny** (najlepszy dozwolony ruch),
    SA jest **stochastyczny** (losowy sąsiad + probabilistyczna akceptacja).

- B4 — Lokalne vs globalne optimum:

  Lokalne optimum ≠ złe rozwiązanie — może być bardzo bliskie globalnemu.
- B5 — Brak uzasadnienia wyboru metody:

  Odpowiedź „wybieram SA” bez uzasadnienia = 0 punktów.

  Zawsze należy dodać: „wybieram SA, ponieważ... (mechanizm, koszt, struktura problemu)”.

- **TOP 5 Błędów — Ślady i Implementacja**

  - B6 — Zwracanie x zamiast x_{best} w SA:

    SA może skończyć w gorszym rozwiązaniu — zawsze zwracaj x_{best}.
  - B7 — Pełne $N(x)$ w SA:

    SA używa jednego losowego sąsiada ($O(1)$),

    TS używa pełnego sąsiedztwa ($O(|N(x)|)$).
  - B8 — Brak kryterium aspiracji w TS:

    TS może ominąć globalne optimum.

    A1: jeśli $f(x') < f(x_{best})$ -> wykonaj mimo tabu.
  - B9 — Brak resetu $k = 1$ w VNS:

    Po poprawie zawsze należy ustawić $k = 1$.
  - B10 — Błędne użycie α w GRASP:

    $\alpha = 0$ -> czysto zachłanne,

    $\alpha = 1$ -> czysto losowe,

    zalecane: $\alpha \in (0.1, 0.3)$.

Kluczowe wzory i schematy algorytmów — do nauki

- **SA — kluczowe wzory**

$$P(\text{akceptuj gorszy}) = \exp\left(-\frac{\Delta E}{T}\right)$$

$$T(k+1) = \alpha \cdot T(k)$$

$$T_0 = -\frac{\Delta E_{avg}}{\ln(P_{start})}$$

```
x = init()
T = T0

while T > T_min:
    x2 = random_neighbor(x)
    Δ = f(x2) - f(x)

    if Δ < 0 or rand() < exp(-Δ / T):
        x = x2

    if f(x) < f(best):
        best = x

    T *= alpha
```

- **HC — fundament**

```
x = init()

while improved:
    x = best_neighbor(x)
```

- **TS — szkielet**

```
x = init()
tabu = []

for iter in range(max_iter):
    x = best_admissible(N(x), tabu, best)

    tabu.append(move)
    if len(tabu) > T_tabu:
        tabu.pop(0)
```

- **VNS — BVNS**

```
x = init_LS()

while t < t_max:
    k = 1

    while k <= k_max:
        x2 = shaking(x, N_k)
        x3 = LS(x2)

        if f(x3) < f(x):
            x = x3
            k = 1
        else:
            k += 1
```

- **ILS**

```
x_best = LS(init())

for i in range(max_iter):
    x2 = perturb(x_best)
    x3 = LS(x2)

    if accept(x3, x_best):
        x_best = x3
```

- **GRASP**

```
best = None

for i in range(max_iter):
    x = greedy_random_construction(alpha)
    x = LS(x)

    if f(x) < f(best):
        best = x
```

- **RCL w GRASP**

$$threshold = c_{min} + \alpha \cdot (c_{max} - c_{min})$$

$$RCL = \{c \in C : \text{cost}(c) \leq \text{threshold}\}$$

Drzewo decyzyjne — szybki przewodnik wyboru algorytmu

- **Masz / chcesz: SA**
  - Typ problemu: ciągły / mieszany
  - Czas implementacji: szybka (ok. 30 linii)
  - Parametry: T_0, α, T_{min}, L (4)
  - Koszt iteracji: $O(1) + 1$ ewaluacja
  - Równoległość: tak (multiple runs)
  - Stan sztuki: dobre wyniki, ogólny algorytm
- **Masz / chcesz: TS**
  - Typ problemu: kombinatoryczny
  - Czas implementacji: średnia (ok. 60 linii)
  - Parametry: T_{tabu} (1 kluczowy)
  - Koszt iteracji: $O(|N|) + |N|$ ewaluacji
  - Równoległość: trudna
  - Stan sztuki: doskonałe dla problemów kombinatorycznych

- **Masz / chcesz: VNS**
  - Typ problemu: kombinatoryczny z wieloma N_k
  - Czas implementacji: średnia (ok. 50 linii)
  - Parametry: k_{max} (1 kluczowy)
  - Koszt iteracji: $O(LS) + shaking$
  - Równoległość: częściowa (multiple runs)
  - Stan sztuki: state-of-the-art dla TSP
- **Masz / chcesz: ILS**
  - Typ problemu: dowolny z dobrą perturbacją
  - Czas implementacji: szybka (ok. 40 linii)
  - Parametry: siła perturbacji (1)
  - Koszt iteracji: $O(LS) + perturb$
  - Równoległość: tak (multiple runs)
  - Stan sztuki: bardzo silny dla TSP (ILS + LK)
- **Masz / chcesz: GRASP**
  - Typ problemu: gdy istnieje dobra heurystyka konstrukcyjna
  - Czas implementacji: szybka (ok. 40 linii)
  - Parametry: α (1 kluczowy)
  - Koszt iteracji: $O(konstr.) + O(LS)$
  - Równoległość: łatwa (każda iteracja niezależna)
  - Stan sztuki: dobre wyniki, często z Path Relinking

Metaheurystyki — Optymalizacja Globalna

Wstęp

Dlaczego populacja rozwiązań?

- **Inspiracje z natury**
  - **Kolonia mrówek:** żadna mrówka nie zna całej mapy, ale razem znajdują najkrótszą drogę do jedzenia.
  - **Klucz ptaków:** każdy ptak widzi tylko swoich sąsiadów, jednak cały klucz porusza się optymalnie jako całość.
  - **Ewolucja:** miliardy iteracji selekcji naturalnej prowadzą do organizmów zoptymalizowanych pod kątem przetrwania.
- **Zalety populacji**
  - Równoległe przeszukiwanie wielu obszarów przestrzeni rozwiązań.
  - Wymiana informacji między rozwiązaniami.
  - Naturalna odporność na lokalne optima.
  - Możliwość efektywnej implementacji równoległej.
- **Wady populacji**
  - Wyższy koszt obliczeniowy pojedynczej iteracji.
  - Większa liczba parametrów wymagających strojenia.
  - Może działać wolniej lokalnie niż SA lub TS.
  - Ryzyko przedwczesnej zbieżności populacji.

Ogólny schemat P-metaheurystyki

- **Schemat działania**

1. Inicjalizacja populacji P — wygeneruj N rozwiązań (losowo lub heurystycznie).
2. Ewaluacja — oblicz wartość funkcji celu $f(x)$ dla każdego $x \in P$.
3. Sprawdzenie kryterium zatrzymania. Jeśli jest spełnione, zwróć najlepsze rozwiązanie.
4. Selekcja — wybierz najbardziej obiecujące rozwiązania z populacji P.
5. Generowanie nowych rozwiązań przez krzyżowanie, mutację lub inne operatory.
6. Aktualizacja populacji — zastąp stare rozwiązania nowymi.
7. Powrót do kroku 2.

- **Elementy wspólne wszystkich P-metaheurystyk**
  - Rozmiar populacji N — zwykle od 20 do 200 osobników.
  - Kryterium zatrzymania — maksymalna liczba iteracji, brak poprawy lub osiągnięcie zadanej wartości funkcji celu.
  - Mechanizm selekcji — preferuje lepsze rozwiązania, ale nie eliminuje całkowicie słabszych.
  - Operatory zmienności — zapewniają równowagę między eksploracją i eksploatacją przestrzeni rozwiązań.
- **Ogólny model aktualizacji populacji**

$$P(t+1) = \text{update}\big(P(t), \text{selection}(P(t)), \text{variation}(P(t))\big)$$

P-Metaheurystyki

Algorytm Genetyczny (GA) — idea

- **Inspiracja: Darwinowska ewolucja**
  - „Przeżycie najlepiej przystosowanych".
  - Lepsze osobniki częściej się rozmnażają.
  - Potomkowie dziedziczą cechy swoich rodziców.
  - Mutacja wprowadza nową zmienność do populacji.
  - Po wielu pokoleniach populacja osiąga coraz lepszą adaptację.
- **Holland (1975)**

  Adaptation in Natural and Artificial Systems — pierwsza formalna prezentacja algorytmów genetycznych.

  Schema Theorem: schematy o krótkim oddziaływaniu, niskim rzędzie i wysokiej wartości fitness rozmnażają się wykładniczo.
- **Kluczowe składniki GA**
  - Reprezentacja — jak kodujemy rozwiązanie?
  - Inicjalizacja populacji — N osobników.
  - Funkcja fitness — jak oceniamy jakość rozwiązania?
  - Selekcja — wybór rodziców.
  - Krzyżowanie — sposób łączenia informacji od rodziców.
  - Mutacja — losowe modyfikacje potomków.
  - Wymiana pokoleń — sposób zastępowania starej populacji nową.

GA: pętla główna

- **Pseudokod**

```
function GA(N, p_c, p_m, max_gen):
    P = initPopulation(N)
    evaluate(P)

    best = getBest(P)
    for t = 1 to max_gen:
        P_new = []
        P_new.add(elite(P, k = 2))
        while |P_new| < N:
            p1 = select(P)
            p2 = select(P)

            if random() < p_c:
                c1, c2 = crossover(p1, p2)
            else:
                c1, c2 = p1, p2
            if random() < p_m:
                mutate(c1)
            if random() < p_m:
                mutate(c2)

            P_new.add(c1, c2)

        P = P_new
        best = update(best, getBest(P))

    return best
```

- **Parametry GA**
  - Rozmiar populacji N: zwykle $50-200$.
  - Prawdopodobieństwo krzyżowania p_c: $0.6-0.95$.
  - Prawdopodobieństwo mutacji p_m: od $\frac{1}{n}$ do 0.1.
  - Maksymalna liczba pokoleń (`max_gen`): $100-10000$.
  - Elityzm k: około $1-5\%$ populacji.

- **Reguła praktyczna**

  Klasyczne ustawienie Hollanda:

  $$p_m = \frac{1}{n},$$

  gdzie n oznacza długość chromosomu.

  Daje to średnio jedną mutację na każdy chromosom.

Reprezentacje chromosomów

- **Binarna** $[0, 1]^n$

  Klasyczna reprezentacja Hollanda.

  ```
  Chromosom: 1 0 1 1 0 0 1
  ```

  - Zastosowanie: problem plecakowy, selekcja cech, układy logiczne.
  - Operatory: 1-point, 2-point, uniform crossover; bit-flip mutation.

- **Permutacyjna** $\pi(n)$

  Reprezentacja dla problemów kolejnościowych.

  ```
  Chromosom: [3, 1, 4, 2, 5]
  ```

  - Zastosowanie: TSP, harmonogramowanie, sequencing.
  - Operatory: OX, PMX, CX crossover; swap, inversion mutation.
  - **Uwaga:** standardowe operatory mogą tworzyć nielegalnych potomków.

- **Rzeczywista** $\mathbb{R}^n$

  Reprezentacja dla optymalizacji ciągłej.

  ```
  Chromosom: [0.73, -1.24, 3.01]
  ```

  - Zastosowanie: wagi sieci neuronowych, inżynieria, robotyka.
  - Operatory: BLX-α, SBX crossover; mutacja gaussowska.

- **Drzewowa (GP)**

  Genetic Programming (Koza, 1992) — reprezentacja w postaci drzew wyrażeń.

  - Zastosowanie: symboliczna regresja, automatyczne programowanie.
  - Problem: **bloat** — niekontrolowany wzrost rozmiaru drzewa.

Metody selekcji

- **Roulette Wheel (koło ruletki)**

  Prawdopodobieństwo selekcji proporcjonalne do wartości fitness:

  $$P(x_i) = \frac{f(x_i)}{\sum_j f(x_j)}$$

  - ✓ Naturalna presja selekcyjna.
  - × Wrażliwa na wartości odstające (super-fit osobniki).
  - × Problematyczna dla minimalizacji.

- **Tournament Selection**

  Wybierz k losowych osobników, wygrywa najlepszy:

  $$\text{winner} = \arg\max\{f(x) : x \in \text{tournament}\}$$

  - ✓ Odporny na skalowanie fitness.
  - ✓ Łatwa kontrola presji selekcji poprzez parametr k.
  - ✓ Najczęściej stosowana metoda w praktyce.

- **Rank Selection**

  Selekcja oparta na randze, a nie bezwzględnym fitness:

  $$P(x_i) = \frac{\text{rank}(x_i)}{\sum_j \text{rank}(x_j)}$$

  - ✓ Eliminuje problem skalowania fitness.
  - ✓ Stała presja selekcji w czasie.
  - × Wolniejsza konwergencja.

- **Praktyczna rada**

  Najczęściej stosowane podejście:

  Tournament selection z $k = 2$ lub $k = 3$.

  Zapewnia umiarkowaną presję selekcyjną i dobrą odporność na różne rozkłady fitness.

Operatory krzyżowania dla permutacji

- **OX — Order Crossover (dla TSP)**

  Zachowuje kolejność względną genów z rodzica P_2 spoza segmentu P_1.

```
P1: 1 2 3 4 5 6 7
P2: 3 7 2 1 6 5 4
```

  Krok 1: skopiuj segment

```
_ _ 3 4 5 _ _
```

  Krok 2: wypełnij z P_2 (kolejność: 2, 1, 6, 7)

```
C1: 2 1 3 4 5 6 7
```

- **PMX — Partially Mapped Crossover**

  Zachowuje pozycje bezwzględne poprzez mapowanie genów.

  Szczególnie użyteczny, gdy pozycja elementu ma znaczenie (np. harmonogramowanie).

- **CX — Cycle Crossover**

  Każdy gen dziedziczony jest z jednego z rodziców na tej samej pozycji.

  Zachowuje pozycje absolutne i cykle permutacji.

- **Porównanie dla TSP**

Operator	Zachowuje	Jakość
OX	kolejność względną	****
PMX	pozycje absolutne	***
CX	pozycje absolutne	***
ERX	krawędzie	*****

Operatory mutacji

- **Swap Mutation (permutacje)**

  Zamiana dwóch losowych genów miejscami.

  $$[1, 2, 3, 4, 5] \to [1, 5, 3, 4, 2]$$

  Prosty operator zapewniający niewielką, lokalną zmianę struktury permutacji.

- **Inversion Mutation (permutacje)**

  Odwrócenie wybranego segmentu permutacji.

  $$[1, 2, 3, 4, 5] \to [1, 4, 3, 2, 5]$$

  Operator równoważny ruchowi **2-opt** w problemie TSP.

- **Insertion Mutation (permutacje)**

  Usunięcie genu i wstawienie go w inne miejsce.

  $$[1, 3, 4, 2, 5] \to [1, 4, 2, 3, 5]$$

  Pozwala na bardziej elastyczne przestawienia niż swap.

- **Gaussian Mutation (ciągła)**

  Mutacja dla reprezentacji rzeczywistej — dodanie szumu gaussowskiego:

  $$x_i' = x_i + \mathcal{N}(0, \sigma^2)$$

  Parametr σ kontroluje siłę mutacji i może być adaptacyjny (np. w CMA-ES).

Strategie Ewolucyjne (ES) — wariant GA

- **Notacja ES**
  - (μ, λ)-ES: μ rodziców, λ potomków.

    Następna generacja: wybierane jest λ najlepszych osobników spośród potomków.
  - $(\mu + \lambda)$-ES: Następna generacja powstaje z wyboru μ najlepszych osobników spośród rodziców i potomków.
  - Zazwyczaj zachodzi relacja $\mu < \lambda$ (często $\lambda \approx 7\mu$).

- **Adaptacja parametrów ES**

  W strategiach ewolucyjnych parametr mutacji σ również podlega ewolucji:

  $$\sigma' = \sigma \cdot \exp\left(\tau \cdot \mathcal{N}(0,1)\right)$$

  Klasyczna **reguła 1/5 sukcesu**:

  - jeśli odsetek udanych mutacji $> 1/5$ -> zwiększ σ
  - jeśli $< 1/5$ -> zmniejsz σ

- **CMA-ES — State of the Art**

  Covariance Matrix Adaptation Evolution Strategy — jedna z najskuteczniejszych metod optymalizacji ciągłej typu black-box.

  - automatycznie adaptuje pełną macierz kowariancji mutacji
  - uczy się kierunków w przestrzeni rozwiązań
  - inwariantna względem rotacji układu współrzędnych
  - wykorzystuje historię udanych kroków do adaptacji rozkładu
  - implementacja: `pip install cma`

Strategie wymiany populacji

- **Generational (całkowita wymiana)**

  - Cała populacja $P(t)$ jest zastępowana przez nową populację $P(t+1)$.
  - Odpowiada klasycznemu modelowi „pokoleń” w biologii.
  - Ryzyko: utrata najlepszego rozwiązania, jeśli nie stosuje się elityzmu.
  - W praktyce stosuje się elityzm ($k \approx 1\% - 3\%$ najlepszych osobników).

- **Steady-State (wymiana częściowa)**

  - W każdej iteracji zastępowana jest tylko część populacji (np. λ najgorszych osobników).
  - Szybsza poprawa jakości lokalnej populacji.
  - Mniejsza różnorodność niż w podejściu generational.
  - Stosowane, gdy koszt ewaluacji funkcji celu jest niski.

- **Elityzm**

  - Najlepsze rozwiązania są zawsze przenoszone do kolejnej generacji bez zmian.
  - Bez elityzmu: możliwa utrata najlepszego rozwiązania przez mutację lub selekcję.
  - Z elityzmem: gwarantowane monotoniczne niepogarszanie najlepszego wyniku.

- **Porównanie strategii**
  - Generational: wysoka różnorodność, wolniejsza zbieżność.
  - Steady-State: niższa różnorodność, szybsza zbieżność.
  - Generational + elityzm: kompromis między eksploracją a eksploatacją.

Problemy GA i jak je rozwiązać

- **Przedwczesna zbieżność**
  - Populacja zostaje zdominowana przez jednego (lub kilku) bardzo dobrych osobników.
  - Spadek różnorodności prowadzi do utknięcia w lokalnym optimum.
- **Genetic Drift**
  - Losowe zmiany w małych populacjach powodują zanik dobrych alleli niezależnie od wartości fitness.
  - Efekt szczególnie silny przy małym rozmiarze populacji.
- **Techniki utrzymania różnorodności**
  - **Niching / Sharing**: wprowadzenie kary za podobieństwo między osobnikami.
  - **Crowding**: potomkowie zastępują najbardziej podobnych rodziców.
  - **Island Model**: podział populacji na subpopulacje z okresową migracją.
  - **Restarting**: ponowna inicjalizacja populacji przy braku postępu.
  - **Adaptive mutation (p_m)**: zwiększenie intensywności mutacji przy spadku różnorodności.
- **Island Model GA**
  - Populacja dzielona na K wysp, każda zawiera N/K osobników.
  - Co M generacji następuje migracja m najlepszych osobników między wyspami.
  - Model naturalnie równoległy — każda wyspa może być uruchamiana na osobnym rdzeniu lub wątku.

Zastosowania algorytmów genetycznych

- **Inżynieria i projektowanie**
  - Optymalizacja struktury anten (NASA, 2006)
  - Projektowanie aerodynamiczne skrzydeł samolotów
  - Topologia sieci elektrycznych (smart grid)
  - VLSI routing i placement
- **Uczenie maszynowe**
  - Optymalizacja hiperparametrów (alternatywa dla grid search)
  - Selekcja cech (feature selection)
  - Neural Architecture Search (NAS)
  - Kalibracja parametrów modeli symulacyjnych
- **Logistyka i harmonogramowanie**
  - Vehicle Routing Problem (VRP)
  - Job Shop Scheduling
  - Układanie planów zajęć (timetabling)
  - Optymalizacja sieci dostaw
- **Bioinformatyka**
  - Protein structure prediction
  - Multiple sequence alignment
  - Drug discovery — optymalizacja cząsteczek
  - Analiza sieci genów

Particle Swarm Optimization

Particle Swarm Optimization — idea

- **Inspiracja: stada i ławice**
  - Zachowanie ptaków szukających pożywienia: każdy osobnik eksploruje przestrzeń i jednocześnie korzysta z informacji zbiorowej.
  - Każda cząstka pamięta najlepsze miejsce, które sama odwiedziła ($pbest$).
  - Dodatkowo zna najlepsze rozwiązanie znalezione przez całe stado ($gbest$).
  - Ruch jest kompromisem między eksploracją a eksploatacją.

- **Kennedy & Eberhart, 1995**
  - N cząstek w D-wymiarowej przestrzeni.
  - Każda cząstka posiada pozycję x_i oraz prędkość v_i.
  - Każda cząstka pamięta swoje najlepsze dotychczasowe położenie $pbest$.
  - Globalnie znane jest najlepsze położenie całego roju $gbest$.
  - W każdej iteracji aktualizowana jest prędkość i pozycja cząstek.
- **PSO vs GA**
  - **Inspiracja:** GA — ewolucja biologiczna, PSO — zachowanie stada.
  - **Komunikacja:** GA — krzyżowanie, PSO — wymiana informacji przez prędkości.
  - **Pamięć:** GA — brak jawnej pamięci, PSO — $pbest$ i $gbest$.
  - **Parametry:** GA — p_c, p_m, N, PSO — ω, c_1, c_2, N.
  - **Implementacja:** PSO jest prostsze implementacyjnie.
  - **Reprezentacja:** PSO naturalnie działa w przestrzeniach ciągłych.

PSO: równania aktualizacji

- **Aktualizacja prędkości**

$$v_i(t+1) = \omega \cdot v_i(t) + c_1 \cdot r_1 \cdot (pbest_i - x_i(t)) + c_2 \cdot r_2 \cdot (gbest - x_i(t))$$

- **Aktualizacja pozycji**

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

- **Parametry PSO**
  - ω **(inertia — bezwładność)**: kontroluje wpływ poprzedniej prędkości. Typowo: malejące od $0.9 \to 0.4$.
  - c_1 **(cognitive component)**: przyciąganie do własnego najlepszego rozwiązania $pbest$. Typowo: $c_1 = 2.0$.
  - c_2 **(social component)**: przyciąganie do globalnego najlepszego rozwiązania $gbest$. Typowo: $c_2 = 2.0$.
  - $r_1, r_2 \in [0, 1]$: losowe współczynniki wprowadzające stochastyczność i wspierające eksplorację przestrzeni.

PSO dla problemów dyskretnych

- **Problem: PSO nie jest natywnie dyskretny**
  - Standardowe równania PSO zakładają przestrzeń ciągłą.
  - Problemy takie jak permutacje czy zmienne binarne wymagają modyfikacji reprezentacji.
- **Binary PSO (BPSO)**
  - Prędkość interpretowana jako prawdopodobieństwo poprzez funkcję sigmoidalną:

$$P(x_i = 1) = \sigma(v_i) = \frac{1}{1 + e^{-v_i}}$$

  - Zastosowania: selekcja cech, problem plecakowy.
  - Minimalna modyfikacja klasycznego PSO.
- **Permutation PSO (dla TSP)**
  - Pozycja reprezentowana jako permutacja miast.
  - Prędkość jako sekwencja operacji swap.
  - Aktualizacja:
    - $x_i + v_i$ = zastosowanie sekwencji swapów do permutacji.
    - $pbest - x_i$ = różnica wyrażona jako lista swapów prowadzących do $pbest$.
- **Porównanie dla TSP**
  - PSO (permutation): dobra jakość, niska prostota implementacji.
  - GA + OX: lepsza jakość, średnia złożoność.
  - ACO: najlepsza jakość, średnia złożoność.

Ant Colony Optimization

Ant Colony Optimization — idea

- **Jak mrówki szukają jedzenia?**
  - Mrówki wyruszają z kolonii i eksplorują przestrzeń rozwiązań w sposób losowy.
  - Na przemierzanych ścieżkach zostawiają ślad feromonowy.
  - Feromony z czasem parują (zanikają).
  - Krótsze ścieżki są częściej wybierane -> więcej feromonów.
  - Powstaje dodatnie sprzężenie zwrotne prowadzące do zbieżności.

- **Dorigo: ACO**
  - 1992 — Ant System (AS)
  - 1996 — Ant Colony System (ACS) — bardziej efektywny wariant
  - 1997 — Max-Min Ant System (MMAS) — kontrola zakresu feromonów, najlepsza stabilność
- **Elementy ACO**
  - τ_{ij} — intensywność feromonu na krawędzi (i, j)
  - η_{ij} — heurystyka (np. $1/d_{ij}$ w TSP)
  - α — wpływ feromonu (typowo $\alpha = 1$)
  - β — wpływ heurystyki (typowo $\beta \in [2, 5]$)
  - ρ — współczynnik parowania feromonu (typowo $\rho \approx 0.1$)
  - Q — ilość feromonu dodawanego przez mrówkę
  - m — liczba mrówek

ACO: równania probabilistyczne

- **Prawdopodobieństwo wyboru krawędzi (i -> j) przez mrówkę k:**

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum\limits_{l \in N_i^k} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta}$$

- **Parowanie feromonów (po każdej iteracji):**

$$\tau_{ij}(t+1) = (1-\rho)\,\tau_{ij}(t) + \Delta\tau_{ij}$$

$$\Delta\tau_{ij} = \sum_k \Delta\tau_{ij}^k$$

$$\Delta\tau_{ij}^k = \begin{cases} \frac{Q}{L_k}, & \text{jeśli mrówka } k \text{ przeszła przez } (i,j) \\ 0, & \text{w przeciwnym razie} \end{cases}$$

- **Parametry ACO (Talbi 2009):**
  - α — wpływ feromonów (typowo: 1)
  - β — wpływ heurystyki (typowo: 2–5)
  - ρ — współczynnik parowania (0.01–0.2)
  - m — liczba mrówek (10–50)

Warianty ACO: AS -> ACS -> MMAS

- **Ant System (AS) — Dorigo 1992**
  - Wszystkie mrówki aktualizują feromony
  - Globalna aktualizacja po każdej iteracji
  - Prosta implementacja
  - Wolna zbieżność
  - Jakość: referencyjna (baseline)
- **Ant Colony System (ACS) — Dorigo & Gambardella 1996**
  - Tylko najlepsza mrówka aktualizuje feromony globalnie
  - Lokalna aktualizacja feromonów w trakcie budowy rozwiązania
  - Pseudolosowy wybór (parametr q_0)
  - Szybsza zbieżność i lepsza jakość niż AS
  - **Polecany dla TSP**
- **MMAS — Stützle & Hoos 2000**
  - Ograniczenie feromonów: $\tau \in [\tau_{\min}, \tau_{\max}]$
  - Aktualizacja tylko przez najlepszą mrówkę (iteracyjną lub globalną)
  - Zapobiega stagnacji feromonów
  - Reinicjalizacja przy utknięciu
  - **Najlepsze wyniki dla TSPLIB**
- **Porównanie empiryczne (TSP, 500 miast)**
  - MMAS: 0.5% od optimum
  - ACS: 1%
  - AS: 3%

  Wszystkie warianty przewyższają SA przy tym samym budżecie obliczeniowym.

Differential Evolution

Differential Evolution (DE)

- **Idea DE — Storn & Price, 1997**

  Mutacja oparta na różnicy wektorów losowych osobników. Mechanizm automatycznie dopasowuje krok do geometrii przestrzeni przeszukiwania.

```
for każdy osobnik x_i:

    # Mutacja
    wybierz r1, r2, r3 != i
    mutant = x[r1] + F * (x[r2] - x[r3])

    # Krzyżowanie (binomial)
    for j = 1..D:
        if random() < CR or j == j_rand:
            trial[j] = mutant[j]
        else:
            trial[j] = x[i][j]

    # Selekcja (greedy)
    if f(trial) <= f(x[i]):
        x[i] = trial
```

- **Parametry DE**
  - N — rozmiar populacji: $5D$–$10D$
  - $F \in [0.4, 1.0]$ — współczynnik skalowania (siła mutacji)
  - $CR \in [0.1, 1.0]$ — prawdopodobieństwo krzyżowania
  - Warianty: DE/rand/1/bin, DE/best/1/bin, DE/rand/2/exp
- **Gdzie DE błyszczy?**
  - Optymalizacja ciągła wielowymiarowa
  - Funkcje multimodalne z licznymi dolinami
  - Kalibracja modeli numerycznych
  - Benchmarki CEC (real-parameter optimization)
  - **Często lepszy niż GA i PSO dla** $n > 20$

Porównanie P-metaheurystyk

- **Tabela porównawcza**

Algorytm	Inspiracja	Przestrzeń	Reprezentacja	Zastosowanie	Parametry
GA	Ewolucja	Obie	Dowolna	TSP, scheduling, ML	N, p_c, p_m
ES / CMA-ES	Ewolucja	Ciągła	Wektory $\mathbb{R}^n$	Inżynieria, black-box	μ, λ, σ
PSO	Stado	Ciągła*	Wektory $\mathbb{R}^n$	ML, inżynieria	N, ω, c_1, c_2
ACO	Mrówki	Dyskretna	Grafy	TSP, VRP, routing	m, α, β, ρ
DE	Różnicowanie	Ciągła	Wektory $\mathbb{R}^n$	Optymalizacja ciągła	N, F, CR
GA + LS	Hybrydowa	Obie	Dowolna	Wszystkie (najlepsza)	Wiele

- **Uwaga**

  PSO może być adaptowane do przestrzeni dyskretnych (BPSO), jednak w praktyce GA i ACO są zwykle bardziej naturalne i skuteczne dla problemów kombinatorycznych.

TSP — Problem Komiwojażera

Problem Komiwojażera (TSP)

- **Formalna definicja**

  Dane: graf pełny $G = (V, E)$ z n wierzchołkami (miastami) oraz macierzą odległości $D = [d_{ij}]$.

  Celem jest znalezienie cyklu Hamiltona (trasy odwiedzającej każde miasto dokładnie raz), o minimalnej długości:

$$\min \sum_{i} d_{i,\pi(i+1)}, \quad \text{gdzie } \pi(n+1) = \pi(1)$$

- **Złożoność**

  - Przestrzeń rozwiązań: $\frac{(n-1)!}{2} \approx 10^{157}$ dla $n = 100$
  - Problem NP-trudny (Garey & Johnson, 1979)
  - Najlepszy algorytm dokładny: $O(n^2 \cdot 2^n)$ — Held–Karp
  - Praktyczna granica dla metod dokładnych: $n \approx 20$

- **Warianty TSP**
  - Symmetric TSP (sTSP): $d_{ij} = d_{ji}$ — najczęstszy przypadek
  - Asymmetric TSP (ATSP): $d_{ij} \neq d_{ji}$ — np. drogi jednokierunkowe
  - TSP with Time Windows (TSPTW): ograniczenia czasowe
  - Multiple TSP (mTSP): wielu sprzedawców
  - VRP: uogólnienie z ograniczeniami pojazdów i pojemności
- **Zastosowania TSP**

  Logistyka (UPS, FedEx), wiercenie VLSI (PCB), sekwencjonowanie DNA, planowanie trajektorii ramienia robota, harmonogramowanie obserwacji teleskopów.

Heurystyki konstrukcyjne dla TSP

- **Nearest Neighbor (NN)**
  - Zacznij od losowego miasta
  - Przechodź do najbliższego nieodwiedzonego miasta
  - Powtarzaj aż wszystkie miasta zostaną odwiedzone
  - Na końcu wróć do miasta startowego

  Złożoność: $O(n^2)$ | Jakość: 25% ponad optimum
- **Greedy (zachłanne krawędzie)**
  - Posortuj wszystkie krawędzie rosnąco według d_{ij}
  - Dodawaj krawędź jeśli:
    - nie tworzy wierzchołka o stopniu > 2
    - nie tworzy cyklu (chyba że jest to ostatnia krawędź)

  Złożoność: $O(n^2 \log n)$ | Jakość: 15–20% ponad optimum
- **Insertion Heuristics**
  - **Nearest Insertion:** wstawiaj najbliższe miasto do istniejącej trasy
  - **Cheapest Insertion:** wstaw tam, gdzie wzrost kosztu jest minimalny
  - **Farthest Insertion:** zaczynaj od najdalszych miast (lepsza globalna struktura)

  Jakość: 8–15% ponad optimum (zależnie od wariantu)

- **Porównanie heurystyk dla TSP**

Metoda	Błąd (%)	Czas
Nearest Neighbor	20–25%	$O(n^2)$
Greedy edges	15–20%	$O(n^2 \log n)$
Cheapest Insertion	10–15%	$O(n^2)$
Farthest Insertion	8–12%	$O(n^2)$

Operatory lokalne dla TSP

- **2-opt (Lin 1965)**

  Usunięcie dwóch krawędzi i ponowne połączenie (odwrócenie segmentu).

$$\Delta cost = d(i,j) + d(i+1,j+1) - d(i,i+1) - d(j,j+1)$$

  - $|N(x)| = O(n^2)$ — wszystkie pary krawędzi
  - Akceptacja: jeśli $\Delta cost < 0$
  - Błąd: 3–5% ponad optimum

- **3-opt**

  Usunięcie trzech krawędzi i rozważenie 8 możliwych połączeń.

  - $|N(x)| = O(n^3)$
  - Lepsza jakość niż 2-opt
  - Znacznie większy koszt obliczeniowy
  - Błąd: 1–2% ponad optimum

- **Or-opt**

  Przeniesienie 1–3 kolejnych miast w inne miejsce trasy.

  - Szybszy niż 2-opt
  - Porównywalna jakość
  - Często używany jako operator mutacji w GA

- **Lin–Kernighan (LK)**

  Adaptacyjny k-opt wybierający k dynamicznie w trakcie działania.

  - Błąd: <1% ponad optimum
  - LKH (Helsgaun, 2000): <0.1% od optimum
  - State-of-the-art dla TSP (bez GA)
  - Implementacja: `lkh.dk`

GA dla TSP: kompletna konfiguracja

- **Konfiguracja GA dla TSP**

Element	Wybór	Uzasadnienie
Reprezentacja	Permutacja $[1..n]$	Naturalna dla TSP
Inicjalizacja	NN + perturbacja	Lepszy punkt startowy
Selekcja	Tournament $k = 3$	Odporny, elastyczny
Krzyżowanie	OX lub ERX	Zachowuje strukturę trasy
Mutacja	Inversion + swap	Odpowiada 2-opt i dywersyfikacji
Local search	2-opt każdy osobnik	Algorytm memetyczny
Wymiana	Generational + elityzm	Monotoniczna poprawa

- **Wyniki GA dla TSP**

Konfiguracja	Błąd
GA (bez LS)	5% ponad optimum
GA + 2-opt (memetyczny)	1%
GA + LK	<0.5%
ACO (MMAS)	0.5%
LKH (tylko LS)	<0.1%

- **Wniosek**

  Sama metaheurystyka populacyjna bez local search jest niewystarczająca dla TSP. Algorytmy memetyczne (GA + LS) osiągają wyniki konkurencyjne z najlepszymi metodami.

- **Konfiguracja GA dla TSP**

Element	Wybór	Uzasadnienie
Reprezentacja	Permutacja $[1..n]$	Naturalna dla TSP
Inicjalizacja	NN + perturbacja	Lepszy punkt startowy
Selekcja	Tournament $k = 3$	Odporny, elastyczny
Krzyżowanie	OX lub ERX	Zachowuje strukturę trasy
Mutacja	Inversion + swap	Odpowiada 2-opt i dywersyfikacji
Local search	2-opt każdy osobnik	Algorytm memetyczny
Wymiana	Generational + elityzm	Monotoniczna poprawa

- **Wyniki GA dla TSP**

Konfiguracja	Błąd
GA (bez LS)	5% ponad optimum
GA + 2-opt (memetyczny)	1%
GA + LK	<0.5%
ACO (MMAS)	0.5%
LKH (tylko LS)	<0.1%

- **Wniosek**

  Sama metaheurystyka populacyjna bez local search jest niewystarczająca dla TSP. Algorytmy memetyczne (GA + LS) osiągają wyniki konkurencyjne z najlepszymi metodami.

ACS dla TSP: pełna implementacja

- **Pseudokod**

```
function ACS_TSP(n, D, m, q0, beta, rho, tau0):
    tau[i][j] = tau0 for all i, j
    eta[i][j] = 1 / D[i][j]
    best = NN_solution()

    repeat:
        for k = 1..m:
            start[k] = random_city()
            tour[k] = [start[k]]
            while |tour[k]| < n:
                i = current_city(k)
                if random() < q0:
                    j = argmax (tau[i][l] * eta[i][l]^beta)
                else:
                    j = roulette(tau[i][:] * eta[i][:]^beta)
                tour[k].add(j)

                // lokalna aktualizacja feromonów
                tau[i][j] = (1 - rho) * tau[i][j] + rho * tau0

        best = update(best, tours)
        for (i, j) in best:
            tau[i][j] = (1 - rho) * tau[i][j] + rho / cost(best)
    return best
```

- **Parametry ACS dla TSP**
  - $m = n$ (jedna mrówka na miasto)
  - $q_0 = 0.9$ (silna eksploatacja)
  - $\alpha = 1, \beta = 2{-}5$
  - $\rho = 0.1$ (wolne parowanie feromonów)
  - $\tau_0 = \frac{1}{n \cdot C_{NN}}$, gdzie C_{NN} to koszt trasy NN
- **Kluczowa różnica ACS vs AS**

  Lokalna aktualizacja:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \rho\tau_0$$

  stosowana po każdym kroku mrówki powoduje, że kolejne mrówki są zniechęcane do używania tych samych krawędzi, co wprowadza naturalną dywersyfikację eksploracji.

Algorytmy memetyczne (MA) = GA + Local Search

- **Idea algorytmów memetycznych**

  Moscato (1989): połączenie ewolucji populacji z lokalnym ulepszaniem każdego osobnika.
  - GA zapewnia eksplorację globalną (różnorodność)
  - Local Search zapewnia eksploatację lokalną (intensyfikacja)
  - Wynik: znacznie lepsza jakość niż GA lub LS osobno
- **Pseudokod**

```
function Memetic_GA_TSP:
    P = initPopulation(N)
    for x in P:
        x = twoOpt(x)

    evaluate(P)
    while not terminated:
        p1, p2 = select(P)
        child = OX_crossover(p1, p2)
        child = inversionMutation(child)
        child = twoOpt(child)

        P = replace(P, child)

    return best(P)
```

- **Kiedy stosować local search?**
  - Zawsze po krzyżowaniu: najlepsza jakość, najwyższy koszt obliczeniowy
  - Tylko dla nowych osobników (dzieci): kompromis jakość/czas
  - Probabilistycznie (p_{LS}): LS tylko z pewnym prawdopodobieństwem
  - Tylko dla top-k: LS tylko dla najlepszych osobników w populacji
- **Wyniki memetycznych dla TSP**

Algorytm	berlin52	ch130
GA (bez LS)	$\sim 5\%$	$\sim 8\%$
GA + 2-opt	$\sim 1\%$	$\sim 2\%$
GA + LK	$< 0.5\%$	$< 1\%$
LKH (top)	$\approx 0\%$	$\approx 0\%$

- **Wniosek**

  Największy skok jakości w problemach takich jak TSP nie pochodzi z samej populacji ani samej mutacji, ale z hybrydy: globalna eksploracja + lokalna intensyfikacja.

TSPLIB — standardowe benchmarki

- **TSPLIB (Reinhardt, 1991)**

  Standardowy zbiór instancji testowych dla TSP i VRP. Używany przez całe środowisko badawcze do porównywania algorytmów.

  - berlin52 — 52 miasta, opt = 7542
  - ch130 — 130 miast, opt = 6110
  - pr1002 — 1002 miasta, opt znane
  - rl5915 — 5915 miast (trudna)
- **Porównanie algorytmów na TSPLIB**

Algorytm	Błąd śred.	Czas (n=1000)
Nearest Neighbor	$\sim 25\%$	$< 1s$
2-opt	$\sim 5\%$	$\sim 5s$
SA	$\sim 3\%$	$\sim 30s$
GA (memetyczny)	$\sim 1\%$	$\sim 2min$
ACO (MMAS)	$\sim 0.5\%$	$\sim 3min$
LKH	$< 0.1\%$	$\sim 2min$

- **Rekord absolutny**

  LKH (Lin-Kernighan-Helsgaun) + POPMUSIC rozwiązał instancję 1.904.711 miast (World TSP) z błędem $\sim 0.001\%$ po wielodniowych obliczeniach.

Jak wybrać algorytm dla TSP?

- **Zestawienie zależności: rozmiar problemu, czas i jakość**

Rozmiar n	Budżet czasu	Wymagana jakość	Zalecana metoda
$n < 20$	Dowolny	Optymalna	Held-Karp (dokładny)
$n < 100$	< 1 min	$> 5\%$ od opt.	Nearest Neighbor + 2-opt
$n < 100$	5–30 min	$< 1\%$ od opt.	SA lub TS z 2-opt
$n = 100$–1000	< 5 min	$< 2\%$ od opt.	GA memetyczny (GA + 2-opt)
$n = 100$–1000	30 min+	$< 0.5\%$	ACO (MMAS) lub LKH
$n > 1000$	godziny	$< 1\%$	LKH lub POPMUSIC
Online / dynamiczne	realtime	heurystyczna	NN + 2-opt iteracyjny

- **Reguła praktyczna**

  Zawsze zacznij od heurystyki Nearest Neighbor jako punktu startowego, a następnie zastosuj 2-opt. Dopiero jeśli wymagania jakościowe nie są spełnione, dodaj pełną P-metaheurystykę.

- **Implementacje**

  - Python: python-tsp, DEAP
  - Java: JCLEC
  - C++: OpenGA, LKH source code

TSP w praktyce przemysłowej

- **Logistyka: UPS ORION**

  Project ORION (2012): optymalizacja 55,000 tras kurierskich dziennie przy użyciu heurystyk TSP/VRP.

  Efekt: około 10 milionów galonów oszczędności paliwa rocznie oraz redukcja emisji CO_2 o około 100 000 ton.

- **Bioinformatyka: DNA Sequencing**

  Shortest Superstring Problem: składanie fragmentów DNA może być modelowane jako TSP na grafie nakładania.

  Sekwencjonowanie genomu: redukcja problemu do TSP o skali milionów „miast” (fragmentów).

- **VLSI Manufacturing**

  Optymalizacja ścieżki wiertarki w płytkach PCB.

  Każdy otwór = miasto, a czas przejścia między punktami = odległość.

  IBM od lat 90. rozwija i patentuje metody oparte na TSP dla problemów VLSI placement i routing.

- **Astronomia: telescope scheduling**

  Kolejność obserwacji obiektów przez teleskop modelowana jako TSP z ograniczeniami czasowymi (TSP with time windows).

  Przykład: Hubble Space Telescope — około 10% więcej obserwacji dzięki optymalizacji harmonogramu.

Krajobraz przestrzeni rozwiązań TSP

- **Struktura krajobrazu TSP**

  Euklidesowy TSP: hipoteza Big Valley — wiele lokalnych minimów skupionych wokół globalnego optimum. Im bliżej optimum globalnego, tym większa koncentracja dobrych rozwiązań.

- **Konsekwencje Big Valley**

  - Multi-start Local Search jest bardzo efektywny
  - GA z rekombinacją dobrych rozwiązań działa dobrze
  - Centrum masy dobrych rozwiązań stanowi przybliżenie globalnego optimum
  - Trudniejsze instancje mogą nie wykazywać struktury Big Valley

- **Trudne instancje TSP**

  - Clustered instances: grupy miast — heurystyki typu NN mogą dawać słabe wyniki
  - Random instances: brak struktury geometrycznej — trudniejsze dla Local Search
  - Structured with obstacles: ograniczenia geograficzne (np. rzeki, góry)
  - Very large n (> 10000): brak gwarancji jakości i skalowalności

- **Implikacja praktyczna**

  Zawsze należy testować algorytm na wielu instancjach o różnej strukturze. Algorytm skuteczny dla euklidesowego TSP może być nieskuteczny dla ATSP lub instancji klastrowanych.

Vehicle Routing Problem — rozszerzenie TSP

- **VRP: formalna definicja**

  Dane: depot (baza), n klientów, k pojazdów. Każdy klient ma zapotrzebowanie q_i. Każdy pojazd ma pojemność Q.

  Cel: przypisać klientów do pojazdów i zaplanować trasy minimalizując łączną odległość, przy spełnieniu ograniczeń:

  $$\sum q_i \leq Q \quad \text{dla każdego pojazdu}$$

- **Warianty VRP**
  - CVRP: ograniczenie pojemności pojazdu
  - VRPTW: okna czasowe
  - VRPB: backhauling (odbiór towarów w drodze powrotnej)
  - DVRP: wersja dynamiczna (nowe zamówienia w trakcie działania)
  - mTSP: wielu sprzedawców = VRP bez ograniczeń pojemności
- **Metaheurystyki dla VRP**
  - GA: chromosom reprezentuje przypisanie klientów do pojazdów
  - ACO: mrówki budują trasy jak w TSP, z dodatkowym etapem dekodowania tras pojazdów
  - SA + LNS: Large Neighborhood Search — jedna z najskuteczniejszych metod dla VRPTW
  - Clarke-Wright Savings: klasyczna heurystyka inicjalizacyjna
- **Praktyka: UPS ORION**

  System UPS ORION rozwiązuje CVRPTW dla około 55 000 tras dziennie.

  Każdy kierowca obsługuje średnio około 120 przystanków.

  Budżet obliczeniowy: poniżej 30 sekund na wygenerowanie trasy.

  W praktyce stosowane są hybrydowe heurystyki łączące konstrukcję i lokalne przeszukiwanie.

Optymalizacja zaawansowana

Multi-Objective Optimization Problem (MOP)

- **Formalna definicja MOP**

$$\min F(x) = (f_1(x), f_2(x), \ldots, f_n(x))$$

subject to:

$$g_i(x) \leq 0, \quad i = 1, \ldots, m$$

$$h_i(x) = 0, \quad i = 1, \ldots, p$$

$$x \in S \quad \text{(przestrzeń decyzyjna)}$$

- **Kluczowa różnica vs. optymalizacja jednokryterialna**
  - Zwracamy zbiór rozwiązań (front Pareto)
  - Nie istnieje jedno „najlepsze” rozwiązanie bez preferencji decydenta
  - Cele są sprzeczne (trade-off)
  - Decydent wybiera rozwiązanie po analizie frontu Pareto

- **Przykłady MOP w praktyce**
  - Portfel inwestycyjny: maksymalizacja zysku, minimalizacja ryzyka
  - Transport: minimalizacja czasu, kosztu i emisji
  - Inżynieria: minimalizacja wagi, maksymalizacja wytrzymałości
  - Machine Learning: maksymalizacja dokładności, minimalizacja złożoności modelu
  - Harmonogramowanie: minimalizacja makespan i opóźnień

Dominacja Pareto i front Pareto

- **Definicja dominacji**

  Rozwiązanie x dominuje y ($x \succ y$) gdy:

  $$f_i(x) \leq f_i(y) \quad \text{dla wszystkich } i = 1, \ldots, n$$

  $$\exists i: \ f_i(x) < f_i(y)$$

  Przykład (min f_1, min f_2):
  - $A = (2, 5)$
  - $B = (3, 4)$
  - $C = (2, 4)$

  C dominuje A (równe f_1, lepsze f_2) C dominuje B (lepsze f_1, równe f_2) A i B są wzajemnie niezdominowane

- **Front Pareto**

  Zbiór Pareto P^* to wszystkie niezdominowane rozwiązania w przestrzeni S.

  Front Pareto F^* to obraz P^* w przestrzeni celów.
  - może być nieciągły (zbiór punktów)
  - może być ciągły (krzywa kompromisów)
  - może być konweksowy lub niekonweksowy

- **Cel algorytmów MOP**

  Aproksymacja frontu Pareto tak, aby:
  - jak najlepiej pokryć przestrzeń celów
  - jak najbliżej rzeczywistego frontu Pareto

Trzy podejścia do optymalizacji wielokryterialnej

- **Agregacja (Scalarization)**

  Zamiana problemu MOP na problem jednokryterialny poprzez ważoną sumę:

  $$\min \sum_i w_i \cdot f_i(x)$$

  - prosta implementacja, możliwość użycia istniejących solverów
  - wymaga określenia wag a priori
  - nie znajduje rozwiązań na niekonweksowych fragmentach frontu Pareto

  Metoda ε-constraint stanowi alternatywne podejście.

- **Podejście Pareto (MOEA)**

  Aproksymacja całego frontu Pareto jednocześnie.

  - zwraca zbiór rozwiązań zamiast jednego wyniku
  - decyzja podejmowana post hoc przez decydenta
  - nie wymaga wcześniejszego określania wag
  - przykłady: NSGA-II, SPEA2, MOEA/D

  Podejście to jest obecnie dominującym standardem w MOP.

- **Podejścia interaktywne**

  Decydent uczestniczy w procesie optymalizacji.

  - iteracyjna wymiana informacji między algorytmem a decydentem
  - stopniowe doprecyzowanie preferencji
  - bardziej naturalne z punktu widzenia użytkownika
  - wymaga aktywnego udziału decydenta
  - przykład: metody reference point

NSGA-II — Non-dominated Sorting Genetic Algorithm II

- **Deb et al., 2002 — IEEE TEC**

  Najczęściej cytowany algorytm MOEA. Dwie kluczowe innowacje: **Non-dominated Sorting** oraz **Crowding Distance**.

```
function NSGA_II(N, max_gen):
    P = initPopulation(N)
    evaluate(P)

    for t = 1..max_gen:
        Q = generateOffspring(P)
        R = P u Q              // 2N osobników
        fronts = fastNonDominatedSort(R)
        P_new = []
        for Fi in fronts:
            if |P_new| + |Fi| <= N:
                P_new += Fi
            else:
                Fi.sort(crowdingDistance, descending)
                P_new += Fi[:N - |P_new|]
                break

        P = P_new

    return front[0]
```

- **Non-dominated Sorting**
  - Przydziela każdemu rozwiązaniu rangę Pareto.
  - Front F_1 – rozwiązania niezdominowane przez żadne inne.
  - Front F_2 – rozwiązania niezdominowane po usunięciu F_1.
  - Proces jest powtarzany aż do przypisania wszystkich rozwiązań do odpowiednich frontów.
  - Złożoność obliczeniowa: $O(m \cdot n^2)$, gdzie:
    - m – liczba funkcji celu,
    - n – liczba rozwiązań.

- **Crowding Distance**
  - Miara zagęszczenia rozwiązań na froncie Pareto.
  - Preferowane są rozwiązania znajdujące się w rzadziej zaludnionych obszarach, co zwiększa różnorodność populacji.
  - Wzór:

$$cd(x) = \sum_i \frac{|f_i(x_{\text{next}}) - f_i(x_{\text{prev}})|}{f_i^{\max} - f_i^{\min}}$$

Warianty algorytmów wielokryterialnych

- **SPEA2**
  - Zitzler (2001).
  - Zewnętrzne archiwum o rozmiarze $N + N_2$.
  - **Strength-based fitness**.
  - Gęstość estymowana metodą k**-th nearest neighbor**.
  - Lepsze pokrycie frontu Pareto niż NSGA-II.
  - Wolniejsze od NSGA-II ze względu na utrzymywanie archiwum.
- **NSGA-III**
  - Deb (2014).
  - Rozszerzenie NSGA-II o punkty odniesienia (reference points).
  - Lepszy dla problemów z wieloma celami (> 3).
  - Punkty odniesienia rozmieszczane są na hiperpłaszczyźnie.
  - Jeden z najlepszych algorytmów dla problemów many-objective.
  - Stosowany m.in. w AutoML oraz Neural Architecture Search (NAS).
- **MOEA/D**
  - Zhang i Li (2007).
  - Dekompozycja problemu wielokryterialnego na N podproblemów jednokryterialnych.
  - Każdy podproblem wykorzystuje inne wagi funkcji Tchebycheffa.
  - Sąsiednie podproblemy współpracują i wymieniają informacje.
  - Dobrze skaluje się do problemów z wieloma celami.
  - Często szybszy od NSGA-II dla trudnych instancji.
- **Praktyczny wybór algorytmu**
  - 2 cele: NSGA-II lub SPEA2.
  - 3–5 celów: NSGA-III.
  - Powyżej 5 celów: MOEA/D.
  - Jeśli znane są preferencje decydenta: MOEA/D z odpowiednio dobranymi wagami lub metoda ε-constraint.

Jak oceniać jakość frontu Pareto?

- **Hypervolume (HV)**
  - Objętość przestrzeni zdominowanej przez aproksymację frontu oraz punkt referencyjny r.

$$HV(A, r) = \lambda \left(\bigcup_{x \in A} [f_1(x), r_1] \times \cdots \times [f_n(x), r_n] \right)$$

  - Nie wymaga znajomości prawdziwego frontu Pareto.
  - Jednocześnie mierzy bliskość do frontu oraz stopień jego pokrycia.
  - Jest wrażliwy na wybór punktu referencyjnego r.
  - Koszt obliczeniowy wynosi około $O(n^{m/2})$ dla m funkcji celu.
- **Generational Distance (GD)**
  - Średnia odległość punktów aproksymacji od najbliższego punktu rzeczywistego frontu Pareto P^*.

$$GD(A, P^*) = \frac{\sqrt{\sum_{x \in A} \operatorname{dist}(x, P^*)^2}}{|A|}$$

  - Wymaga znajomości rzeczywistego frontu Pareto P^*.
  - Im mniejsza wartość GD, tym lepsza jakość aproksymacji.
- **Spread (Δ)**
  - Mierzy równomierność pokrycia frontu Pareto.
  - Dobre rozwiązanie oznacza równomierne rozmieszczenie punktów na całym froncie.

Zastosowania optymalizacji wielokryterialnej

- **Finanse: Portfel Markowitza**
  - $f_1 = \min$ ryzyko (wariancja portfela), $f_2 = \max$ oczekiwana stopa zwrotu.
  - Front Pareto odpowiada tzw. efficient frontier (granicy efektywnej), stanowiącej podstawę klasycznej teorii portfela inwestycyjnego (Nagroda Nobla, 1990).
- **Inżynieria mechaniczna**
  - Optymalizacja silnika:
    - maksymalizacja mocy,
    - minimalizacja zużycia paliwa,
    - minimalizacja emisji NO_x,
    - maksymalizacja trwałości.
  - Cztery sprzeczne kryteria optymalizowane m.in. za pomocą NSGA-III.
  - Airbus wykorzystuje algorytmy MOEA do projektowania skrzydeł samolotów.

- **Bioinformatyka**
  - **Drug discovery**:
    * maksymalizacja aktywności biologicznej,
    * minimalizacja toksyczności,
    * minimalizacja kosztu syntezy.
  - **Protein design**:
    * minimalizacja energii fałdowania,
    * maksymalizacja stabilności termicznej.
- **AutoML i NAS**
  - **Neural Architecture Search (NAS)**:
    * maksymalizacja dokładności,
    * minimalizacja liczby parametrów,
    * minimalizacja czasu inferencji.
  - Najczęściej stosowane algorytmy:
    * NSGA-II (np. Once-for-All, OFA),
    * MOEA/D (MO-NAS).

Optymalizacja w środowiskach dynamicznych

- **Czym jest DOP (Dynamic Optimization Problems)?**
  - Funkcja celu $f(x, t)$ lub przestrzeń decyzyjna $S(t)$ zmienia się w czasie.
  - Celem jest śledzenie oraz aproksymacja optimum w każdym momencie czasu t.
- **Przykłady DOP**
  - DVRP: dynamiczne zmiany zamówień w trakcie realizacji tras.
  - Scheduling dynamiczny: awarie maszyn i zmiany dostępności zasobów.
  - Portfolio rebalancing: zmieniające się ceny aktywów.
  - Traffic routing: wypadki, korki i zmienne warunki drogowe.
  - Game AI: zmieniający się stan gry w czasie rzeczywistym.
- **Techniki dla DOP**
  - Detekcja zmian: monitorowanie $f(x_{\text{best}})$; spadek jakości może wskazywać zmianę środowiska.
  - Reinicjalizacja: losowa część populacji po wykryciu zmiany.
  - Pamięć populacji: przechowywanie dobrych historycznych rozwiązań.
  - Utrzymanie różnorodności: stała kontrola zróżnicowania populacji.
  - Predykcja: modelowanie zmian środowiska i próba ich przewidywania.

- **Kluczowa obserwacja**
  - W problemach dynamicznych nie dąży się do szybkiej zbieżności do jednego optimum.
  - Kluczowe jest utrzymanie różnorodności populacji, aby umożliwić szybką adaptację do zmian.

Obsługa ograniczeń w metaheurystykach

- **Problem ograniczeń**
  - Wiele problemów praktycznych zawiera ograniczenia:

$$g_i(x) \leq 0 \quad \text{lub} \quad h_i(x) = 0$$

  - Operatory mutacji i krzyżowania mogą generować rozwiązania niedopuszczalne.
- **Death Penalty**
  - Odrzucenie niedopuszczalnych rozwiązań poprzez przypisanie: $f(x) = \infty$
  - Zalety:
    - prosta implementacja
  - Wady:
    - nieefektywne, gdy przestrzeń rozwiązań dopuszczalnych jest mała
- **Penalty Function**
  - Funkcja celu z karą:

$$f'(x) = f(x) + \lambda \cdot \sum \max(0, g_i(x))^2$$

  - Zalety:
    - łatwa implementacja,
    - działa w większości przypadków.
  - Wady:
    - trudne strojenie parametru λ.
  - Wariant: adaptacyjna kara — λ rośnie, gdy brak poprawy najlepszego rozwiązania.
- **Repair + Decoder**
  - **Repair**: naprawa niedopuszczalnych rozwiązań.
  - **Decoder**: mapowanie genotypu na zawsze dopuszczalny fenotyp.
  - Zalety:
    - gwarancja dopuszczalności rozwiązań,
    - zwykle najlepsza jakość wyników w praktyce.

Jak wybrać metodę optymalizacji?

- **Mały rozmiar (n < 20), 1 kryterium**
  - Rekomendacja: metody dokładne (Branch and Bound, Dynamic Programming)
  - Alternatywy: SA, TS
- **Ciągły, 1 kryterium, gładki**
  - Rekomendacja: metody gradientowe (L-BFGS, Newton)
  - Alternatywy: CMA-ES, DE
- **Ciągły, 1 kryterium, nieciągły lub wielomodalny**
  - Rekomendacja: CMA-ES lub Differential Evolution (DE)
  - Alternatywy: PSO, GA
- **Dyskretny problem kombinatoryczny (TSP, scheduling)**
  - Rekomendacja: SA lub TS (metaheurystyki pojedynczego rozwiązania) + dobra inicjalizacja
  - Alternatywy: GA memetyczny, ACO
- **Problemy grafowe (routing, network)**
  - Rekomendacja: ACO
  - Alternatywy: GA, SA
- **Duże problemy, dużo czasu, 1 kryterium**
  - Rekomendacja: GA memetyczny lub ACO
  - Alternatywy: SA z restartami
- **Wiele kryteriów (2–5)**
  - Rekomendacja: NSGA-II lub SPEA2
  - Alternatywy: MOEA/D, metoda ε-constraint
- **Wiele kryteriów (>5)**
  - Rekomendacja: MOEA/D lub NSGA-III
  - Alternatywy: agregacja funkcji celu
- **Środowisko dynamiczne**
  - Rekomendacja: GA z reinicjalizacją i utrzymaniem różnorodności
  - Alternatywy: SA z pamięcią
- **Problemy z ograniczeniami, mały obszar dopuszczalny**
  - Rekomendacja: decoder + GA lub TS
  - Alternatywy: penalty function + SA

Strojenie parametrów algorytmów

- **Problem strojenia**
  - Każda metaheurystyka posiada zestaw parametrów (np. N, p_c, p_m, T_0 w SA, tabu_size w TS).
  - Nieodpowiedni dobór parametrów prowadzi do znacznego pogorszenia wyników.
  - Parametry są często zależne od konkretnej instancji problemu.
- **Podejście 1: Grid Search**
  - Przetestowanie wszystkich kombinacji parametrów na zbiorze instancji.
  - Wysoki koszt obliczeniowy.
  - Dobre jako punkt wyjścia w badaniach eksperymentalnych.
  - Praktyczne jedynie dla małej liczby parametrów (2–3).
- **Podejście 2: Automatyczne strojenie**
  - **iRace** (R package): racing + testy statystyczne.
  - **SMAC**: optymalizacja bayesowska przestrzeni parametrów.
  - **Hyperopt**: TPE (Tree-structured Parzen Estimator), random search.
  - Strojenie wykonywane na zbiorze instancji treningowych.
  - Walidacja na oddzielnym zbiorze testowym.
- **Podejście 3: Parametry adaptacyjne**
  - Zmienne w czasie parametry algorytmów:
    - ω malejące w PSO ($0.9 \to 0.4$),
    - temperatura T malejąca w SA (harmonogram chłodzenia),
    - adaptacja p_m w GA przy spadku różnorodności,
    - CMA-ES: pełna adaptacja macierzy kowariancji.

Inne algorytmy bioinspirowane

- **ABC — Artificial Bee Colony**
  - Karaboga (2005).
  - **Employed bees**: intensywna eksploatacja źródeł.
  - **Onlooker bees**: selekcja proporcjonalna do jakości źródeł.
  - **Scout bees**: eksploracja nowych źródeł (gdy obecne zostają wyczerpane).
  - Dobry dla funkcji ciągłych.
  - Stosunkowo prosty w implementacji.

- **Firefly Algorithm**
  - Yang (2008).
  - Świetliki przyciągane są do jaśniejszych (lepszych wartości funkcji celu).
  - Jasność zależy od fitness oraz odległości.
  - Atrakcyjność maleje wraz z odległością:

$$\beta(r) = \beta_0 e^{-\gamma r^2}$$

  - Dobrze sprawdza się w problemach ciągłych, multimodalnych.
- **Krytyczne spojrzenie na nowe algorytmy**
  - Co roku proponowane są nowe „bioinspirowane”algorytmy (np. Whale Optimization, Grey Wolf, Bat Algorithm).
  - Często są to warianty GA lub PSO z nową metaforą biologiczną.
  - Zasada No Free Lunch nadal obowiązuje — brak uniwersalnie najlepszego algorytmu.
- **Zasada praktyczna**
  - Przed użyciem egzotycznego algorytmu należy sprawdzić:
    * dobrze nastrojony GA,
    * PSO,
    * SA.
  - W wielu przypadkach dają porównywalne wyniki przy mniejszej złożoności.

Algorytmy hybrydowe i portfolio

- **Strategie hybrydyzacji**
  - **Sekwencyjna**: GRASP (konstrukcja) $\rightarrow$ SA (optymalizacja lokalna).
  - **Zagnieżdżona**: GA jako algorytm zewnętrzny + TS dla każdego osobnika.
  - **Memetyczna**: GA + local search dla każdego potomka.
  - **Pipeline**: wynik GA jako punkt startowy dla SA.
  - **Kooperacyjna**: wiele metaheurystyk współdzielących informacje o rozwiązaniach.

- **Algorithm Portfolio**
  - Równoległe uruchomienie kilku algorytmów, wybór najlepszego wyniku po zadanym czasie.
  - Zalety:
    * brak konieczności wyboru algorytmu a priori,
    * naturalna równoległość,
    * wysoka skuteczność przy stałym budżecie czasu.
  - Przykład portfolio:
$$[SA, TS, GA, ACO] \rightarrow \text{best}$$
- **Hybrydyzacja GA + VNS (GVNS)**
  - GA zarządza populacją, VNS poprawia lokalnie każde rozwiązanie.
  - Każdy potomek jest ulepszany:
$$\text{child} = \text{VNS}(\text{child}, k_{\max} = 3)$$
  - VNS przeszukuje kolejne sąsiedztwa $N_1 \rightarrow N_2 \rightarrow N_3$.
  - Stan-of-the-art dla VRP z ograniczeniami.
  - Wyniki: poniżej 0.5% od najlepszych znanych rozwiązań.
- **Reguła hybrydyzacji**
  - Jeśli pojedynczy algorytm osiąga wyniki gorsze o więcej niż 5% od najlepszych znanych rozwiązań:
  - należy rozważyć hybrydyzację.
  - Zalecany punkt startowy: Memetic GA z 2-opt.

Równoległe metaheurystyki

- **Master-Slave (Replikacja)**
  - Jedna populacja, równoległa ewaluacja funkcji celu $f(x_i)$.
  - Master wykonuje: selekcję, krzyżowanie i mutację.
  - Slave wykonuje równoległe obliczenia fitness.
  - Zalety:
    * prosta implementacja (OpenMP, MPI),
    * niemal liniowe przyspieszenie wraz z liczbą rdzeni.

- **Island Model**
  - K niezależnych populacji (wysp).
  - Co M generacji następuje migracja najlepszych osobników.
  - Każda wyspa działa na osobnym rdzeniu lub węźle.
  - Zalety:
    * utrzymanie różnorodności,
    * często lepsze wyniki niż pojedyncza populacja o rozmiarze $K \times N$.
  - Parametry: liczba wysp K, częstotliwość migracji, liczba migrantów k.
- **Cellular GA**
  - Osobniki rozmieszczone na siatce 2D (topologia toroidalna).
  - Każdy osobnik komunikuje się tylko z sąsiadami (5- lub 9-sąsiedztwo).
  - Wolne propagowanie informacji sprzyja utrzymaniu różnorodności.
  - Zalety:
    * bardzo dobra jakość rozwiązań dla problemów multimodalnych.
  - Wady:
    * wolniejsza zbieżność.
- **Narzędzia implementacyjne**
  - Python: `multiprocessing`, DEAP (model island).
  - Java: JCLEC.
  - GPU: PyGAD, EvoTorch (backend PyTorch).
  - Klaster: Apache Spark, Databricks.

Estimation of Distribution Algorithms (EDA)

- **Idea EDA**
  - Zamiast operatorów genetycznych (krzyżowanie, mutacja) buduje się probabilistyczny model $P(x)$ rozkładu dobrych rozwiązań.
  - Nowe osobniki są generowane przez próbkowanie (sampling) z tego rozkładu.

- **Schemat EDA**

```
function EDA(N, max_gen):
    P = initPopulation(N)
    while not terminated:
        selected = select(P, N/2)
        model = estimateDistribution(selected)
        offspring = sample(model, N)
        P = offspring
    return best(P)
```

- **Warianty EDA**
  - PBIL: wektor prawdopodobieństw binarnych (najprostszy przypadek).
  - UMDA: univariate marginal distribution — niezależne zmienne.
  - BOA: Bayesian Optimization Algorithm — model zależności (sieć bayesowska).
  - ECGA: Extended Compact Genetic Algorithm.
- **EDA vs GA**
  - **GA**: krzyżowanie + mutacja.
  - **EDA**: modelowanie rozkładu + sampling.
  - GA nie modeluje zależności jawnie, EDA (np. BOA) może je reprezentować explicite.
  - GA ma parametry p_c, p_m; EDA wymaga określenia modelu.
  - Koszt GA jest zwykle niższy, EDA wymaga uczenia modelu.

Analiza zbieżności i diagnostyka

- **Co mierzyć?**
  - $best\ f(t)$: najlepsze znalezione rozwiązanie w generacji t.
  - $avg\ f(t)$: średnia wartość w populacji — informacja o eksploracji.
  - $std\ f(t)$: odchylenie standardowe — miara różnorodności.
  - $HV(t)$: dla problemów wielokryterialnych — jakość frontu Pareto w czasie.
  - Liczba ewaluacji funkcji celu — rzeczywisty budżet obliczeniowy.

- **Symptomy i diagnozy**
  - Plateau zbyt wcześnie → przedwczesna zbieżność:
    - zwiększyć mutację,
    - zmniejszyć presję selekcji.
  - Brak zbieżności → zbyt słaba selekcja:
    - zwiększyć tournament size k.
  - $best \approx avg$ przez cały czas → zbyt duża mutacja:
    - zmniejszyć p_m.
  - $std \to 0$ bardzo szybko → utrata różnorodności:
    - zbyt silny elityzm lub selekcja.
- **Metodologia eksperymentalna**
  - Minimum 30 niezależnych uruchomień (różne ziarna RNG).
  - Raportowanie: średnia, mediana, odchylenie standardowe, minimum, maksimum.
  - Testy statystyczne: Wilcoxon signed-rank (test nieparametryczny).
  - Stały budżet: liczba ewaluacji funkcji celu, nie liczba generacji.
  - Porównania na wielu instancjach o różnej strukturze.
- **Typowy błąd**
  - Raportowanie wyniku tylko jednego, najlepszego uruchomienia.
  - Poprawna praktyka: statystyki z wielu niezależnych uruchomień.
  - Pojedynczy wynik może być efektem losowości, a nie jakości algorytmu.

Black-Box Optimization i benchmarki

- **Black-Box scenario**
  - Znana jest jedynie funkcja celu $f : \mathbb{R}^n \to \mathbb{R}$, którą można ewaluować.
  - Znane są ograniczenia przestrzeni decyzyjnej (bounds).
  - Nieznane są: gradient, struktura funkcji, wypukłość, multimodalność.
  - Metaheurystyki są naturalnie przystosowane do problemów black-box — to jedna z ich głównych zalet.

- **Benchmarki BBOB (COCO)**
  - Zestaw 24 funkcji testowych o różnej charakterystyce.
  - Różne wymiary: $n = 2, 3, 5, 10, 20, 40$.
  - Metryka: ERT (Expected Running Time).
  - Standard w konkursach CEC.
  - Dostęp: `github.com/numbbo/coco`.
- **Ranking algorytmów (BBOB)**
  - CMA-ES: czołowe miejsca (top 1–3), metody ewolucyjne z adaptacją kowariancji.
  - L-BFGS-B: bardzo wysoka skuteczność dla funkcji gładkich (gradient-based).
  - DE: średnia skuteczność, metoda oparta na różnicach.
  - PSO: średnia skuteczność, podejście swarm-based.
  - GA (binarny): niższa skuteczność w problemach ciągłych black-box.
- **Wniosek praktyczny**
  - Dla ciągłej optymalizacji black-box standardem jest CMA-ES.
  - GA jest bardziej odpowiedni dla problemów kombinatorycznych z dobrze zdefiniowaną reprezentacją.

Metaheurystyki w uczeniu maszynowym

- **Neural Architecture Search (NAS)**
  - Problem: znalezienie optymalnej architektury sieci neuronowej.
  - Przestrzeń rozwiązań: typy warstw, połączenia, rozmiary warstw.
  - Evolutionary NAS: zastosowanie GA/ES do optymalizacji grafu architektury.
  - Multi-Objective NAS: jednoczesna optymalizacja accuracy, liczby parametrów oraz latencji.
  - Przykłady: Google AutoML, Once-for-All (MIT) wykorzystujące MOEA.
- **Hyperparameter Optimization**
  - Bayesian Optimization (BO): modele probabilistyczne (GP) + funkcje akwizycji.
  - Narzędzia: SMAC, Hyperopt, Optuna, Ray Tune.
  - CMA-ES: bardzo skuteczny dla 10–100 hiperparametrów.
  - GA: szczególnie efektywny dla przestrzeni mieszanych (ciągłe + dyskretne).

- **Evolutionary Reinforcement Learning**
  - Evolutionary Strategies (ES) jako alternatywa dla metod gradientowych w RL.
  - OpenAI ES (2017): konkurencyjny względem PPO w prostych środowiskach.
  - Zalety: łatwa równoległość, brak potrzeby backpropagation.
  - Neuroewolucja: NEAT (Stanley, 2002) — ewolucja struktury i wag sieci.
- **Trend**
  - Metaheurystyki + machine learning = AutoML.
  - Automatyczne projektowanie modeli ML przy użyciu metaheurystyk (Feurer, Hutter 2019).
  - Przykładowe biblioteki: auto-sklearn, TPOT, AutoKeras.