

# Inteligencja obliczeniowa - Inteligencja roju

Grzegorz Madejski

# Strategie roju

Szpaki:



Źródło obrazka: <https://www.smithsonianmag.com/>

# Strategie roju

Mrówki:



Źródło obrazka: <https://gardenerspath.com/>

# Strategie roju

- Strategie roju to dziedzina algorytmów, które inspirowane są naturą (podobnie jak algorytmy genetyczne czy sieci neuronowe).
- Jest to podejście, w którym istnieje wiele agentów lub cząstek (particles), zachowuje się jak zbiorowiska organizmów żywych w naturze (rój pszczół, ławica ryb, chmara ptaków, mrówki).
- Każda z cząstek ma swoją własną inteligencję, ale...
- Każda cząstka działa też tak, jak jej sąsiedzi (wpływ lokalny) czy cały rój (wpływ globalny).
- Istnieją zatem protokoły / zasady, które sterują całym rojem. Można mówić wtedy o inteligencji roju (swarm intelligence).

# Strategie roju

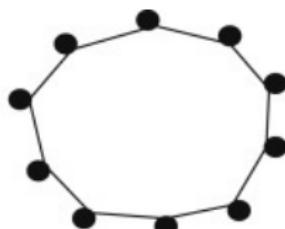
Inteligencję roju można wykorzystać do różnych zadań:

- Algorytm **PARTICLE-SWARM-OPTIMIZATION** (PSO): rój szpaków lata po okolicy szukając jedzenia. Gdy wypatrzą, pikują w stronę celu. Jedna jednostka potrafi ściągnąć całe stado. Matematycznie: szukamy optymalnego rozwiązania (max/min).
- Algorytm **ANT-COLONY-OPTIMIZATION** (ACO): rój mrówek szuka pożywienia. Gdy znajdą, tworzą feromonowe ścieżki do celu, by inne mrówki wiedziały gdzie iść. Matematycznie: szukanie ścieżek.

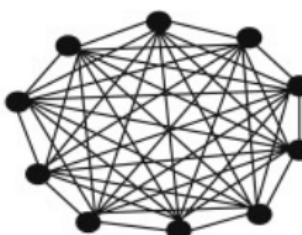
# Topologia sąsiedztwa

- Na zachowanie cząstek mają wpływ sąsiedzi.
- Podejście geograficzne: na moje zachowanie mają najbliżsi pod względem odległości sąsiedzi (podobnie jak kNN). Minus: trzeba obliczać odległości.
- Podejście sieci komunikacyjnej: każdej cząstce z góry i na stałe przydzielamy konkretnych sąsiadów.
- Jak przydzielać sąsiadów? Są różne topologie takich sieci.

# Topologia sąsiedztwa



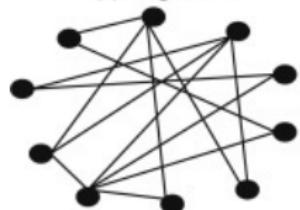
(a) ring/lbest



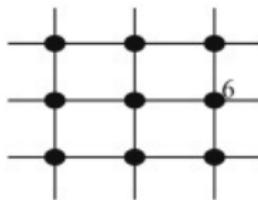
(b) gbest/all



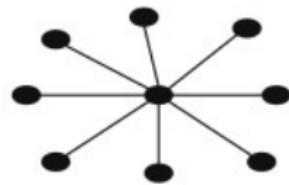
(c) wheels



(d) random



(e) von Neumann



(f) star

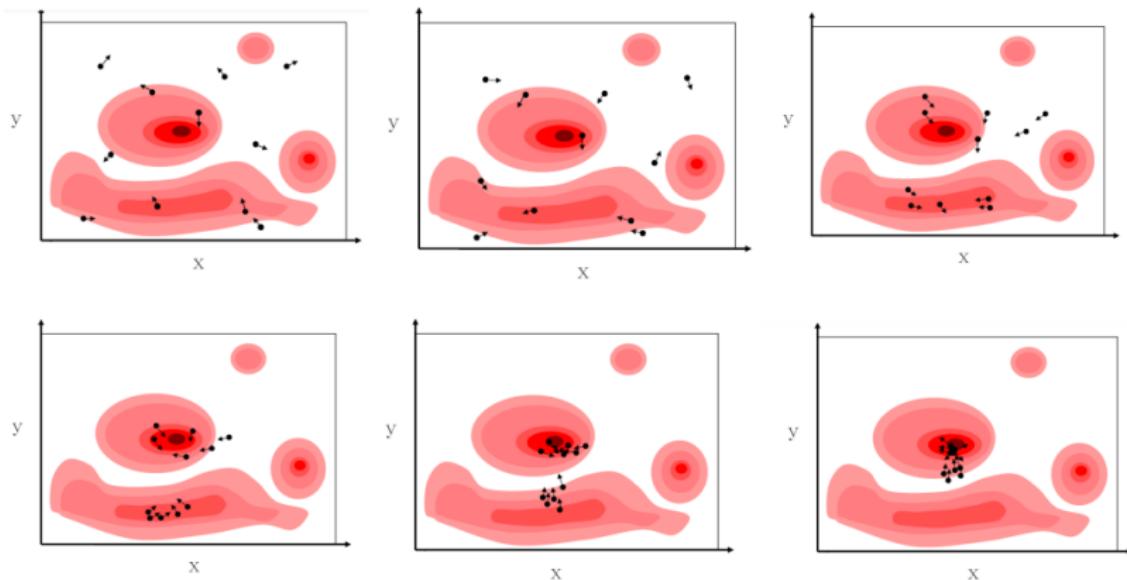
# Topologia sąsiedztwa

- Topologia gbest (maksymalne połączenia): najlepsze rozwiązanie lokalne jest jednocześnie globalnym. Szybko działa, ale wpada w pułapki (max/min lokalne)
- Topologia pierścienia: dwaj sąsiedzi. Typowa strategia lokalna. Działa wolniej, ale częściej znajduje dobre rozwiązania.
- Topologia koła: wszyscy połączeni z szefem/hubem.

# Algorytm PARTICLE-SWARM-OPTIMIZATION

- Algorytm PARTICLE-SWARM-OPTIMIZATION dostaje na wejścia funkcję i ma za zadanie znaleźć maximum (lub minimum) globalne tej funkcji w zadanym zakresie.
- Algorytm tworzy cząstki w danym obszarze. Cząstki wędrują, komunikują się ze sobą, by odnaleźć najlepsze rozwiązanie.
- Każda cząstka ma swoją pozycję w przestrzeni np.  $x = (0.2, 1.7)$  oraz prędkość np.  $v = (1.2, -0.2)$ .
- Nowe pozycje aktualizowane są przez dodanie do nich prędkości  $x_{new} = (1.4, 1.5)$ .
- Prędkość (szybkość, kierunek, zwrot) jest tak wybierana, żeby nakierować cząstkę na cel (najlepsze rozwiązanie).

# Algorytm PARTICLE-SWARM-OPTIMIZATION



# Algorytm PARTICLE-SWARM-OPTIMIZATION

funkcja PSO-MAX( $f$ ,  $\omega$ ,  $\phi_1$ ,  $\phi_2$ ):

$x_{glob.\text{best}} \leftarrow \text{WYBIERZ-LOSOWO}()$

dla każdej cząstki  $p \in Rój\text{Cząstek}$ :

$x, v \leftarrow \text{INICJUJ-LOSOWĄ-POZYCJĘ-I-PRĘDKOŚĆ}()$

$N(p) \leftarrow \text{WYBIERZ-SĄSIEDZTWO-CZĄSTKI}()$

$x_{ind.\text{best}} \leftarrow x$

dopóki ( $\neg$ warunek-końcowy):

dla każdej cząstki  $p \in Rój\text{Cząstek}$ :

$x_{ind.\text{best}} \leftarrow \text{ARG-MAX}(f(x), f(x_{ind.\text{best}}))$

$x_{glob.\text{best}} \leftarrow \text{ARG-MAX}(f(x), f(x_{glob.\text{best}}))$

$x_{loc.\text{best}} \leftarrow \text{ARG-MAX-WŚRÓD-SĄSIADÓW}(N(p))$

$r_1, r_2 \leftarrow \text{RANDOM}([0, 1])$

$v \leftarrow \omega v + \phi_1 r_1 (x_{ind.\text{best}} - x) + \phi_2 r_2 (x_{loc.\text{best}} - x)$

$x \leftarrow x + v$

zwrócić  $x_{glob.\text{best}}$

# Algorytm PARTICLE-SWARM-OPTIMIZATION

Wzór na wyznaczanie nowej prędkości  $v$ :

$$v = \omega v + \phi_1 r_1(x_{ind.\ best} - x) + \phi_2 r_2(x_{loc.\ best} - x)$$

zależy od kilku wybieranych przez użytkownika parametrów:

# Algorytm PARTICLE-SWARM-OPTIMIZATION

Wzór na wyznaczanie nowej prędkości  $v$ :

$$v = \omega v + \phi_1 r_1(x_{ind.\ best} - x) + \phi_2 r_2(x_{loc.\ best} - x)$$

zależy od kilku wybieranych przez użytkownika parametrów:

- $\omega$  - współczynnik bezwładności, określa wpływ prędkości z poprzedniego kroku (inertia)
- $\phi_1$  - współczynnik dążenia do najlepszego indywidualnego rozwiązania (cognitive parameter)
- $\phi_2$  - współczynnik dążenia do najlepszego lokalnego rozwiązania (social parameter)

# Algorytm PARTICLE-SWARM-OPTIMIZATION

Wzór na wyznaczanie nowej prędkości  $v$ :

$$v = \omega v + \phi_1 r_1(x_{ind.\ best} - x) + \phi_2 r_2(x_{loc.\ best} - x)$$

zależy od kilku wybieranych przez użytkownika parametrów:

- $\omega$  - współczynnik bezwładności, określa wpływ prędkości z poprzedniego kroku (inertia)
- $\phi_1$  - współczynnik dążenia do najlepszego indywidualnego rozwiązania (cognitive parameter)
- $\phi_2$  - współczynnik dążenia do najlepszego lokalnego rozwiązania (social parameter)

Parametry są liczbowe, z przedziału  $[0, 1]$ . Przykład:  $\omega = 0.9$ ,  $\phi_1 = 0.5$ ,  $\phi_2 = 0.3$ .

# Algorytm PARTICLE-SWARM-OPTIMIZATION

Uwaga! By prędkość nie rosła do nieskończoności, można wprowadzić ogranicznik prędkości. Dobry ogranicznik jest dany wzorem:

$$v_{max} = \frac{x_{max} - x_{min}}{N}$$

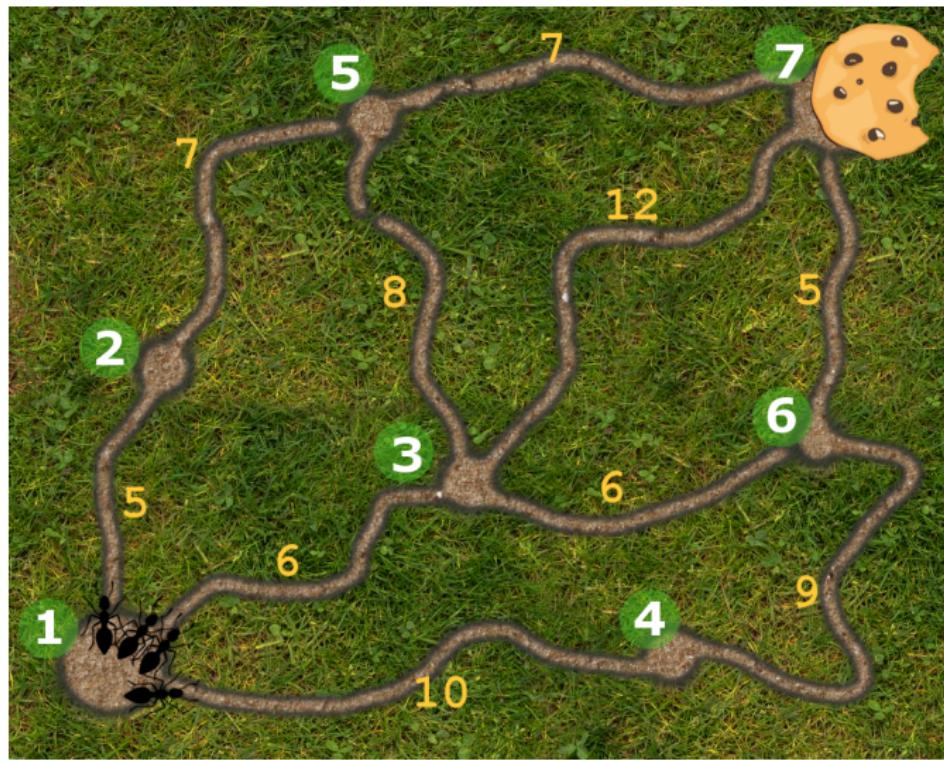
# Algorytm PARTICLE-SWARM-OPTIMIZATION

- W Pythonie, optymalizację PSO można zrealizować za pomocą paczki *pyswarms*
- Link: <https://pyswarms.readthedocs.io/en/latest/>.
- Popatrzmy na program *pso-sphere.py*, który szuka minimum funkcji sferycznej za pomocą PSO.
- Popatrzmy na program *pso-animation.py*, który pokazuje jak rój cząstek szuka minimum funkcji sferycznej.

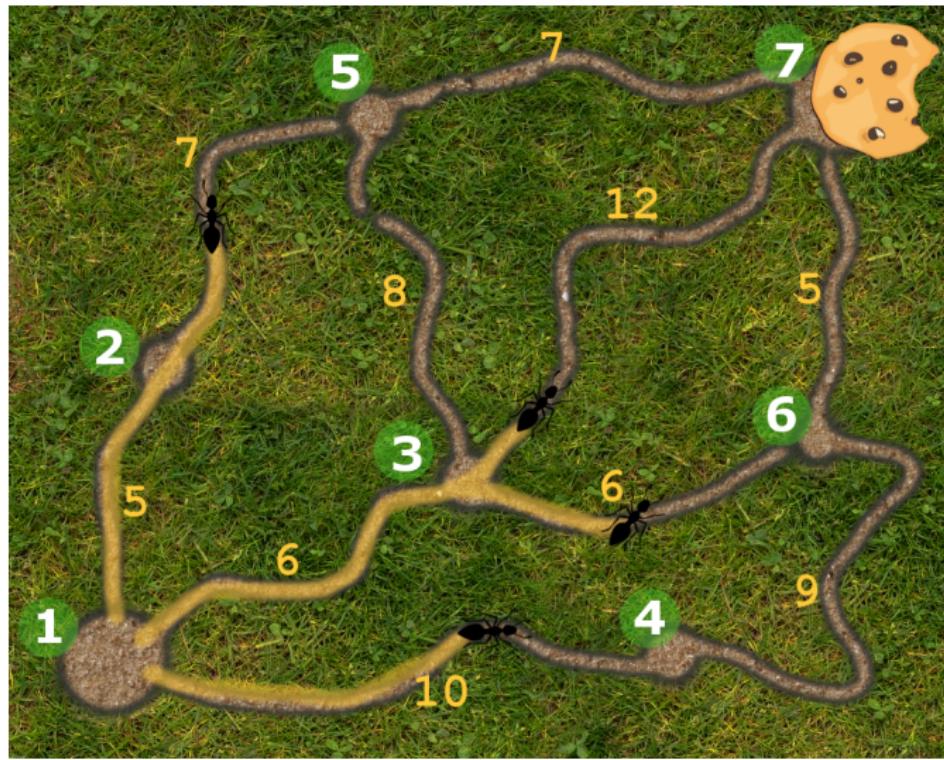
# Algorytm ANT-COLONY-OPTIMIZATION

- Algorytm optymalizujący kolonią mrówek (ACO) symuluje zachowanie mrówek, by rozwiązać problemy.
- Przeznaczony do znajdowania najbardziej optymalnych ścieżek (problem najkrótszej ścieżki, komiwojażer, labirynt, itp.)
- Mrówki szukają najlepszej ścieżki oznaczając ją feromonami.
- Ścieżki lepsze z czasem zostają mocniej oznaczone feromonami. Więcej mrowek je wybiera.
- Po wielu iteracjach algorytmu wybierana jest ścieżka najbardziej wybierana przez mrówki (najbardziej optymalna).

# Algorytm ANT-COLONY-OPTIMIZATION



# Algorytm ANT-COLONY-OPTIMIZATION



# Algorytm ANT-COLONY-OPTIMIZATION

- Fermonomy ulatniają się z czasem.
- Jeśli mrówka wybrała długą drogę, to zanim znajdzie pokarm to sporo feromonów się ułotni.
- Koncentracja feromonów na krtózzych drogach będzie silniejsza i będzie zwabiać więcej mrówek.

# ACO - Krok 1: Inicjalizacja

Pierwszy krok. Ustawiamy parametry ACO.

- Wielkość populacji (liczba mrówek):  $k$  (np. 20).
- Maksymalna liczba iteracji algorytmu:  $i_{max}$  (np. 400).
- Początkowe stężenie feromonu na wszystkich ścieżkach:  $\tau$  (np. 0.5)
- Współczynnik ulatniania fermonów:  $\rho$  (np. 0.05).
- Jaki wpływ na zachowanie mrówek będzie miał feromon: współczynnik  $\alpha$ , a jaki jakość/optymalność ścieżki  $\beta$ . Oba można ustawić na 1 (wolniejsze, bezpieczniejsze działanie), lub 2 (szybsze, niebezpieczniejsze działanie), lub z  $\beta > \alpha$  (szybsze zbieganie algorytmu do globalnego rozwiązania).

## ACO - Krok 2: Szukanie rozwiązań

Zaczyna się główna pętla programu: 1 do  $i_{max}$ . Drugi krok - mrówki szukają rozwiązań.

- Dla każdej z  $k$  mrówek: idź mrówką z węzła  $i$  do węzła  $j$  losując wybór drogi wzorem na prawdopodobieństwo:

$$P_{ij}^k = \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{m \in \text{węzły osiągalne z } i} \tau_{im}^\alpha \eta_{im}^\beta}$$

gdzie  $\tau_{ij}$  oznacza stężenie fermonu na ścieżce między węzłami  $i, j$ , a  $\eta_{ij}$  jakość/optymalność odcinka od  $i$  do  $j$ , u nas wynosi  $1/\text{odległość}(i,j)$ .

## ACO - Krok 2: Szukanie rozwiązań

- Oceń rozwiązania wszystkich mrówek (fitness) np. długość całkowitej drogi.
- Wybierz najlepsze rozwiązanie (rozwiązanie globalne).

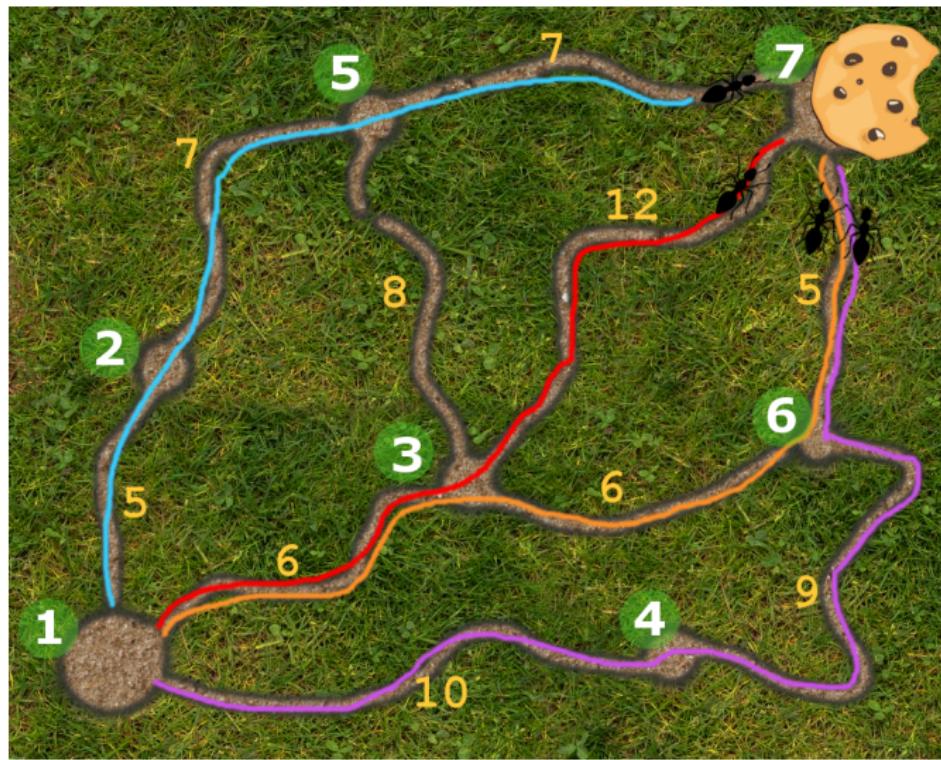
## ACO - Krok 3: Aktualizacja feromonów

Po przebiegu algorytmu zaktualizuj stężenie feromonów na ścieżkach wg wzoru:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{1 \leq l \leq k} \Delta\tau_{ij}^l$$

gdzie  $\tau_{ij}$  to stężenie feromonów pomiędzy węzłami  $i$  i  $j$ , a  $\Delta\tau_{ij}^l$  to liczba dodanych feromonów przez mrówkę  $l$  i wynosi zwykle  $\frac{1}{\text{długość drogi mrówki}}$ . Po aktualizacji feromonów wracamy do kroku 2, chyba że skończą się iteracje programu  $i_{max}$ .

# ACO: Zadanie



# ACO: Zadanie 1

## Zadanie 1

Korzystając z obrazka z poprzedniego slajdu i wzoru na dodawanie feromonów, oblicz ile fermonów będą miały odcinki między węzłami 1 i 2, 1 i 3, 1 i 4, 6 i 7 po jednym przebiegu mrówek.

Uwaga: początkowy poziom feromonów na ściezkach wynosi  $\tau = 0.5$ , a współczynnik jego ulatniania się wynosi  $\rho = 0.05$ .

# ACO: Zadanie 2

## Zadanie 2

Po wykonaniu zadania 1, oblicz jakie będą prawdopodobieństwa wyboru drogi z węzła 1, do węzłów 2, 3, 4. Przyjmujemy  $\alpha = 1$  i  $\beta = 1$ .

# ACO - Python

- W Pythonie, optymalizację ACO można zrealizować za pomocą prostego narzędzia *aco*
- Link: [https://pypi.org/project/aco/.](https://pypi.org/project/aco/)
- Popatrzmy na program *aco-tsp.py*, który optymalnej ścieżki dla problemu komiwojażera.