AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki

INSTYTUT INFORMATYKI

Sprawozdanie z Pracowni Problemowej

Równoległe i rozproszone metaheurystyki optymalizacyjne z desynchronizacją

Parallel and distributed optimization metaheuristics with desynchronization

Autorzy: Szymon Żychowicz
Kierunek studiów: Informatyka
Opiekun pracy: prof. dr hab. inż. Aleksander Byrski

Kraków, 2024

Cel przedmiotu

Pracownia problemowa jest przedmiotem mającym na celu zapoznanie się i zagłębienie się w tematykę związaną z wybranym tematem pracy magisterskiej. W przypadku mojego: *"Równoległe i rozproszone metaheurystyki optymalizacyjne z desynchronizacją"* są to zagadnienia związane z algorytmami metaheurystycznymi służącymi do optymalizacji.

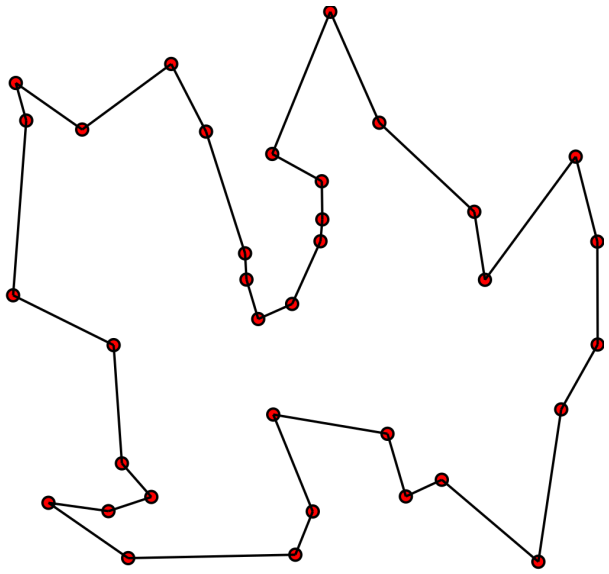
W sprawozdaniu po wprowadzeniu do algorytmów metaheurystycznych i ewolucyjnych, przedstawiona jest implementacja EMAS (wykorzystująca aplikację Pani Sylwii Bielaszek "HPC-Ray") oraz wyniki jej działania.

Aplikacja dostępna jest w repozytorium na GitHubie.
https://github.com/superiorshrimp/mgr-playground/

- main – bazowy kod "HPC-Ray", ze zmianami dla łatwiejszego uruchamiania i analizy
- add-emas – dodanie mojej implementacji EMAS do "HPC-Ray"

Algorytmy metaheurystyczne

Metaheurystyka jest ogólną strategią znalezienia rozwiązania problemu obliczeniowego. Algorytmy takie nie skupiają się na konkretnym problemie, a na sposobie efektywnego przeszukiwania przestrzeni rozwiązań. Przestrzeń ta, może być za duża, aby w rozsądnym czasie sprawdzić wszystkie możliwości. Potrzebny jest sposób na inteligentne sprawadzanie części rozwiązań, aby znaleźć satysfakcjonujące. Algorytmy metaheurystyczne muszą zachować pewien balans między eksploracją, a eksploatacją przestrzeni. Służą do znalezienia globalnego rozwiązania, do czego potrzebna jest eksploracja, ale aby było ono dobre wykorzystuje się eksploatację.



Rysunek 1: Travelling Salesperson Problem

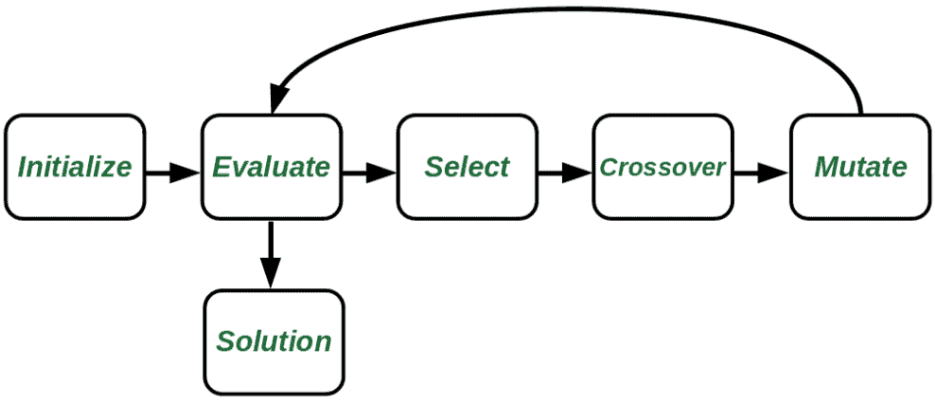
Problem komiwojażera (Travelling Salesperson Problem) polega na znalezieniu najkrótszej ścieżki po punktach w grafie, kończąc w początkowym, odwiedzając każdy maksymalnie jeden raz. Charakteryzuje się szybko rosnącą liczbą możliwych rozwiązań. Przez jego złożoność, dokładne algorytmy już dla niewielu punktów mają trudności. Wykorzystując algorytmy metaheurystyczne, można otrzymać suboptymalne (ale dobre) rozwiązanie, w dużo krótszym czasie. Jest to przykład z szerokiej gamy problemów, gdzie takie algorytmy mogą być najlepszym wyborem.

Algorytmy ewolucyjne

Algorytmy metaheurystyczne często bazują na naturze. Takim przykładem mogą być algorytmy ewolucyjne. Wykorzystują one darwinowską ideę selekcji naturalnej. Osobniki populacji (rozwiązania) na podstawie swojego przystosowania do środowiska (jakości rozwiązania problemu) mogą umrzeć, albo przez reprodukcję oddać swoje geny nowym jednostkom. Powoduje to, że słabsze rozwiązania są odrzucane, a lepsze promowane. Wprowadzając mutacje, można kontrolować różnorodność populacji.

Ogólny algorytm można opisać następująco:

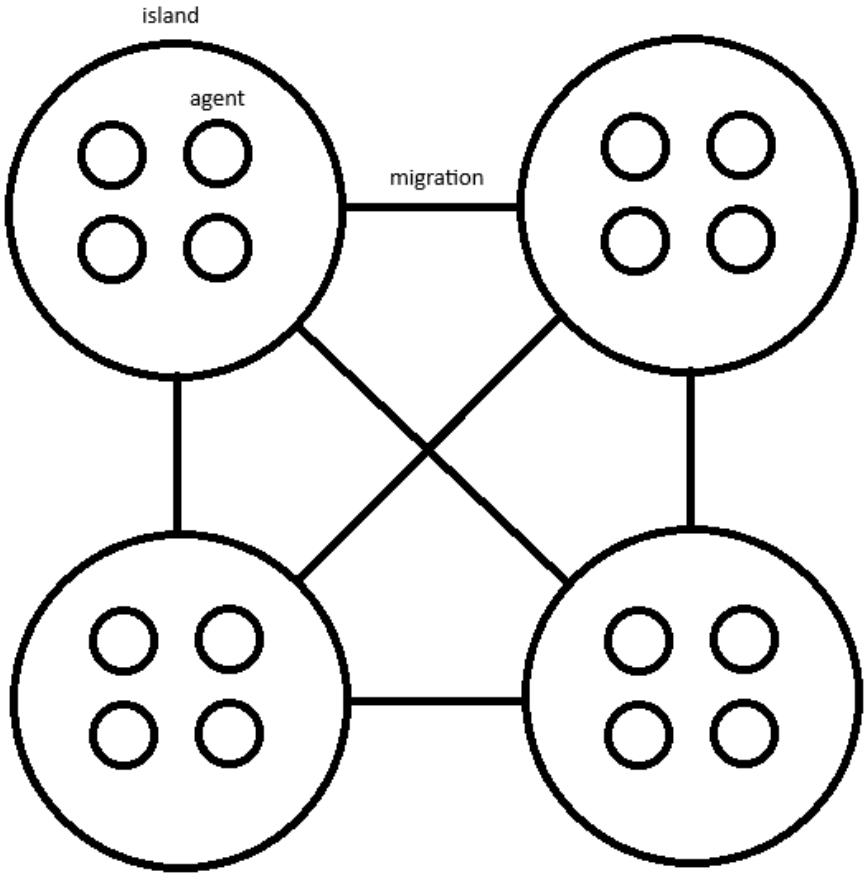
1. wygeneruj populację losowych osobników
2. powtarzaj do osiągnięcia wybranego warunku końca:
    - wyznacz jakość osobników (*evaluation*)
    - wybierz rodziców (*selection*)
    - wygeneruj dzieci przez skrzyżowanie rodziców i mutację potomka (*crossover*, *mutation*)
    - wyeliminuj słabe osobniki



Rysunek 2: Schemat

EMAS

Zachowania osobników w procesie ewolucji są naturalnie zdecentralizowane. Narzuca się przez to rozumienie populacji jako wieloagentowy system. EMAS (**E**volutionary **M**ulti-**a**gent **S**ystem) jest efektem wykorzystania tej abstrakcji na osobnikach i rozmieszczenia takich agentów na wyspach, wprowadzając operację migracji zgodnie z wybraną topologią. Taki podział jest przystosowany do realizacji równolegle, na wielu procesorach.



Rysunek 3: Wyspy w pełnej topologii

Implementacja

Klasa *Agent* reprezentuje agenta.

```python
class Agent:

    def __init__(
        self,
        x: list[float],
        energy: float,
        problem: Problem,
        lower_bound: float,
        upper_bound: float
    ):
        self.x = x
        self.energy = energy
        self.problem = problem
        self.fitness = problem.evaluate(x)
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound
```

Jej pole *x* reprezentuje genotyp agenta w postaci tablicy liczb zmiennoprzecinkowych. *energy* to wartość, która decyduje, czy agent umiera i czy może być potencjalnym rodzicem. *problem* jest instancją klasy, której metoda *evaluate* zwraca jakość rozwiązania na podstawie genotypu *x* osobnika. *lower_bound* oraz *upper_bound*, to zakres wartości, jakie mogą przyjmować geny.

W klasie zdefiniowane są operatory genetyczne.

```
@staticmethod
def crossover(p1, p2, split: float):
    c1, c2 = [], []

    for i in range(len(p1.x)):
        if random() < split:
            c1.append(p1.x[i])
            c2.append(p2.x[i])
        else:
            c1.append(p2.x[i])
            c2.append(p1.x[i])

    return c1, c2
```

Statyczna metoda *crossover* jako argumenty przyjmuje dwóch rodziców oraz parametr *split*. Zwraca parę genotypów dzieci powstałych przez skrzyżowanie genotypów rodziców, w taki sposób, że prawdopodobieństwo, że i-ty gen pierwszego dziecka, będzie pochodził od pierwszego rodzica (a drugiego dziecka od drugiego rodzica) wynosi $split \in (0, 1)$.

```
def mutate(self, mutation_coef: float):
    for i in range(len(self.x)):
        if random() < mutation_coef:
            self.x[i] = get_truncated_normal(
                mean = self.x[i],
                sd=1,
                low=self.lower_bound,
                upp=self.upper_bound
            )

    self.fitness = self.problem.evaluate(self.x)
```

Metoda *mutate* przyjmuje parametr $mutation\_coef \in (0, 1)$. Dla każdego genu osobnika przypisuje mu (z prawdopodobieństwem równym współczynnikowi mutacji) wartość zwracaną przez funkcję *get_truncated_normal* (zdefiniowaną poniżej), a następnie dokonuje ewaluacji nowego rozwiązania.

```
def get_truncated_normal(mean, sd, low, upp):
    return truncnorm.rvs(
        (low - mean) / sd,
        (upp - mean) / sd,
        loc=mean,
        scale=sd
    )
```

Funkcja korzysta z biblioteki *Scipy* do uzyskania wartości losowej z rozkładu normalnego o średniej *mean* i odchyleniu standardowemu *sd*, przyciętego w *low* i *upp*.

```python
@staticmethod
def reproduce(
    p1,
    p2,
    energy_reproduce_loss_coef: float,
    cross_coef: float,
    mutation_coef: float,
    alive_energy: float,
    fitness_average: float,
    n_agent: int,
    agents_count: int
):
    init_energy1 = p1.energy * energy_reproduce_loss_coef
    init_energy2 = p2.energy * energy_reproduce_loss_coef
    init_energy = init_energy1 + init_energy2

    p1.energy -= init_energy1
    p2.energy -= init_energy2

    c1_x, c2_x = Agent.crossover(p1, p2, cross_coef)
    c1 = Agent(
        c1_x, init_energy, p1.problem, p1.lower_bound, p1.upper_bound
    )
    c2 = Agent(
        c2_x, init_energy, p1.problem, p1.lower_bound, p1.upper_bound
    )

    if c1.fitness < fitness_average: c1.mutate(mutation_coef / 2)
    else: c1.mutate(mutation_coef * 2)
    if c2.fitness < fitness_average: c2.mutate(mutation_coef / 2)
    else: c2.mutate(mutation_coef * 2)

    if agents_count < n_agent / 2:
        c1.energy /= 2
        c2.energy /= 2
        return [c1, c2]
    return [c1] if c1.fitness < c2.fitness else [c2]
```

Statyczna metoda *reproduce* przyjmuje obu rodziców, średnią wartość fitness agentów *fitness_average*, początkową liczbę agentów *n_agent*, aktualną liczbę agentów *agents_count* oraz parametry: *energy_reproduce_loss_coef*, *cross_coef*, *mutation_coef* i *alive_energy*.

Rodzice przekazują część swojej energii ($energy\_reproduce\_loss\_coef \in (0,1)$) dzieciom. Te powstają przez wywołanie wyżej opisanej metody *crossover*. Następnie, jeśli wartość *fitness* osobnika jest lepsza niż średnia w populacji, to zmniejszana jest szansa mutacji genu dwukrotnie, a w przeciwnym wypadku jest ona podwajana. Po przeprowadzeniu mutacji na potomkach są one porównywane i jeśli liczebność populacji jest większa niż połowa początkowej, to zwracane jest lepiej przystosowane dziecko. Jeśli liczba osobników jest mniejsza, to zwracana jest para nowych agentów, aby zachować liczbę agentów blisko przewidywanego poziomu.

Klasa *EMAS* odpowiada za selekcję, walkę i usuwanie martwych agentów.

```
def reproduce(self):
    fitness_average = sum(
        [agent.fitness for agent in self.agents]
    ) / len(self.agents)
    c = []
    p = list(
            filter(
                lambda a: a.energy > self.config.reproduce_energy,
                self.agents
            )
        )
    shuffle(p)
    agents_count = len(self.agents)
    for i in range(len(p) // 2):
        offspring = Agent.reproduce(
            p[i],
            p[len(p) // 2 + i],
            self.config.energy_reproduce_loss_coef,
            self.config.cross_coef,
            self.config.mutation_coef,
            self.config.alive_energy,
            fitness_average,
            self.config.n_agent,
            agents_count
        )
        agents_count += len(offspring)
        c.extend(offspring)

    return c
```

Zdefiniowana jej metoda *reproduce* odpowiada za selekcję i repodukcję. Filtruje ona listę agentów, aby wybrać takich, którzy mają energię powyżej poziomu określonego przez parametr *reproduce_energy*, a następnie dobiera ich w losowe pary do reprodukcji. Zwraca ona listę nowych potomków.

```python
def fight(self):
    shuffle(self.agents)

    n = len(self.agents) // 2
    for i in range(n):
        a1, a2 = self.agents[i], self.agents[n+i]
        if a1.fitness > a2.fitness:
            a1, a2 = a2, a1

        energy_loss = a2.energy * self.config.energy_fight_loss_coef

        diff = np.sum(np.abs(np.array(a1.x) - np.array(a2.x)))
        diff /= (a1.upper_bound - a1.lower_bound)
        diff /= len(a1.x)

        energy_loss += a2.energy * (1-diff) * self.config.energy_diff_loss_coef
        if a2.energy - energy_loss < self.config.alive_energy:
            energy_loss = a2.energy - self.config.alive_energy

        a1.energy += (energy_loss + self.config.alive_energy)
        a2.energy -= (energy_loss + self.config.alive_energy)
```

Metoda *fight* dobiera losowo pary do walki. Jej efektem jest przekazanie przez przegranego aktora $energy\_fight\_loss\_coef \in \langle 0,1 \rangle$ część swojej energii wygranemu. Za pomocą parametru $energy\_diff\_coef \in \langle 0,1 \rangle$ można odebrać więcej energii, na podstawie podobieństwa walczących osobników.

```python
def remove_dead(self):
    self.agents = [
        agent for agent in self.agents
        if agent.energy > self.config.alive_energy
    ]
```

Metoda *remove_dead* usuwa agentów, których energia jest niewyższa niż parametr *alive_energy*.

Opisane mechanizmy zostały zintegrowane z programem Pani Sylwii Bielaszek "HPC-Ray". Zapewnia on otoczkę dla opisanych wyżej klas. Wprowadza operatory migracji między wyspami. Są one zarządzane przez framework *Ray*, który zapewnia równoległość działania programu.

Wyniki

Najpierw przedstawione zostaną wyniki algorytmu wewnątrz jednej wyspy (bez migracji). Optymalizacje przeprowadzone zostały w następujących konfiguracjach:

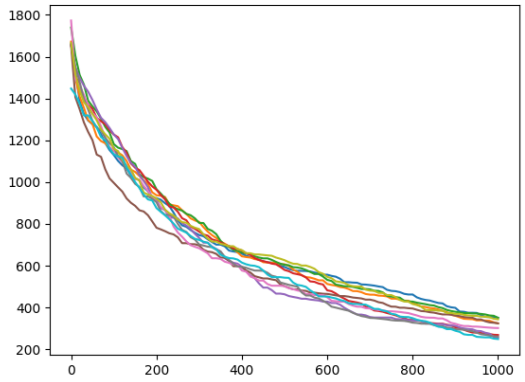
1. problem Rastrigin 100 wymiarów za pomocą 16 agentów
2. problem Rastrigin 100 wymiarów za pomocą 48 agentów
3. problem Rastrigin 200 wymiarów za pomocą 48 agentów

Pozostałe parametry są wspólne pomiędzy testami:

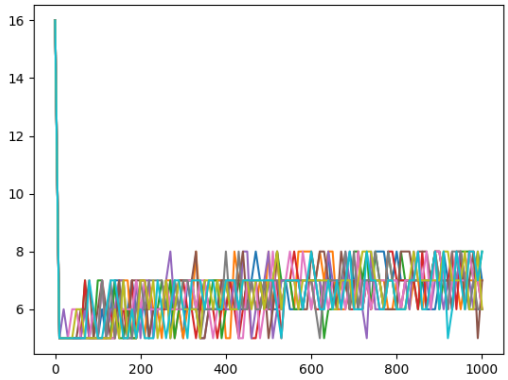
1. $n\_iter = 1000$
2. $lower\_bound = -5.12$
3. $upper\_bound = 5.12$
4. $start\_energy = 100$
5. $reproduce\_energy = 140$
6. $alive\_energy = 1$
7. $energy\_reproduce\_loss\_coef = 0.2$
8. $energy\_fight\_loss\_coef = 0.8$
9. $energy\_diff\_loss\_coef = 0.8$
10. $cross\_coef = 0.55$
11. $mutation\_coef = 0.02$

Na każdej z trzech konfiguracji wykonano po 10 optymalizacji. Wyniki pomiarów przedstawione są na kolejnej stronie.

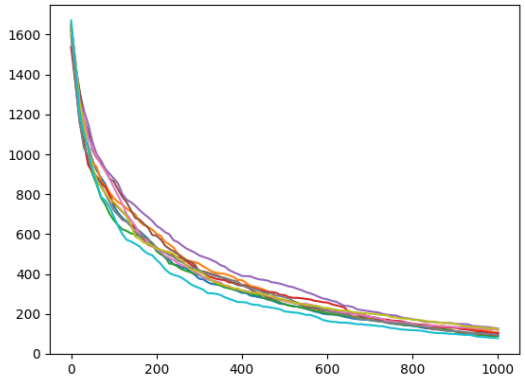
Rysunek 4: fitness; 100 wymiarów, 16 agentów



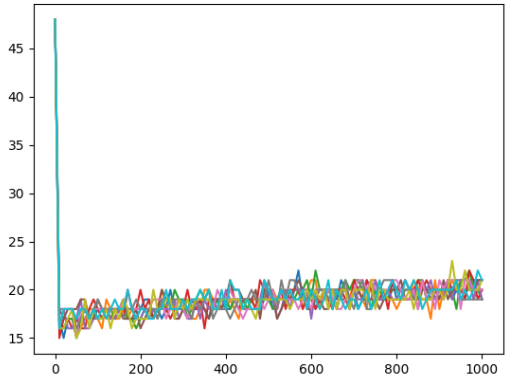
Rysunek 5: liczebność populacji; 100 wymiarów, 16 agentów



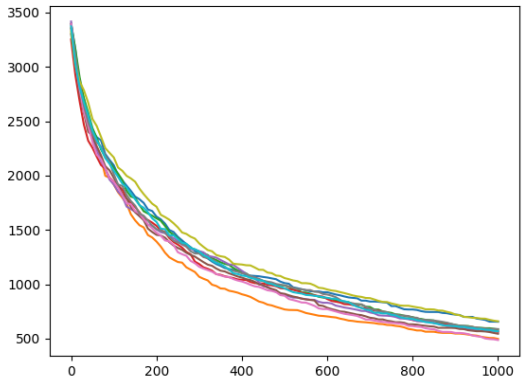
Rysunek 6: fitness; 100 wymiarów, 48 agentów



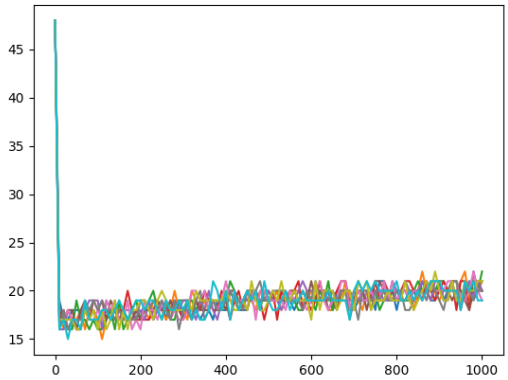
Rysunek 7: liczebność populacji; 100 wymiarów, 48 agentów



Rysunek 8: fitness; 200 wymiarów, 48 agentów



Rysunek 9: liczebność populacji; 200 wymiarów, 48 agentów



Na grafikach po lewej stronie strony przedstawiona jest najlepsza wartość fitness w populacji w kolejnych krokach. Po prawej natomiast widać ilość osobników w populacji w iteracjach.

Można zauważyć, że we wszystkich konfiguracjach fitness zbiega do coraz lepszych wartości. Przy 10 próbach dla danego rozmiaru problemu i ilości agentów wyniki dużo od siebie nie odbiegają.

W przypadku ilości osobników widoczny jest następujący trend: na samym początku jest tyle osobników ile podano. Liczba ta w pierszych iteracjach spada do około $\frac{1}{3}$ początkowej. Następnie dąży do prawie połowy wejściowej ilości. Dzieje się tak przez sposób implementacji metody agenta *reproduce*.

```
if agents_count < n_agent / 2:
        c1.energy /= 2
        c2.energy /= 2
        return [c1, c2]
    return [c1] if c1.fitness < c2.fitness else [c2]
```

Fragment ten zapewnia utrzymanie liczby agentów na takim poziomie, przez zwracanie dwóch potomków, gdy liczebność populacji jest mniejsza niż połowa początkowej.

W przypadku wykorzystania HPC-Ray (po wprowadzeniu wysp i migracji) wyniki zmieniają się w następujący sposób. Dla parametrów:

1. problem Rastrigin 100 wymiarów
2. 3 wyspy (pełna topologia)
3. 16 agentów na wyspę (w przypadku EMAS - początkowa ilość agentów)
4. 1000 ewaluacji (warunek końca)
5. 2 migrantów, co 20 ewaluacji
6. migracja najlepszego osobnika

rezultaty wartości fitness trzech pomiarów to:

EMAS:

1. 389, 443, 422 (średnia trzech wysp = 418)
2. 445, 494, 489 (średnia trzech wysp = 476)
3. 407, 460, 433 (średnia trzech wysp = 433)

DOMYŚLNY:

1. 452, 448, 443 (średnia trzech wysp = 447)
2. 501, 529, 519 (średnia trzech wysp = 516)
3. 486, 497, 496 (średnia trzech wysp = 493)

Zdaje się, że EMAS osiąga lepsze wyniki, jednak trzebaby wykonać znacznie więcej pomiarów, bo zaledwie 3 mogą nie oddawać rzeczywistego stanu.

Po zmianie wyboru osobnika do migracji z najlepszego do losowego, wyniki 3 pomiarów wyglądają następująco (dalej algorytm EMAS):

1. 395, 433, 424 (średnia trzech wysp = 417)
2. 409, 482, 477 (średnia trzech wysp = 456)
3. 464, 471, 471 (średnia trzech wysp = 469)

Znowu większa liczba pomiarów mogłaby zmienić wynik, ale te 3 pomiary dają podobne wyniki do poprzedniego eksperymentu. Rezultat kolejnej zmiany metody na wybór "najodleglejszego"osobnika w populacji, pokazany jest poniżej:

1. 455, 477, 468 (średnia trzech wysp = 467)
2. 447, 468, 459 (średnia trzech wysp = 458)
3. 495, 498, 516 (średnia trzech wysp = 503)

Dla tej metody pomiary dały najgorsze wyniki.