

Algorytmy i struktury danych

dokumentacja egzaminu praktycznego
Wydział Matematyki Stosowanej - Informatyka 3 semestr

Mikołaj Grzegorzek, grupa 1A

13 stycznia 2021

Spis treści

1	Opis problemu przedstawionego w zadaniu.	3
1.1	Treść zadania	3
2	Model Matematyczny	3
2.1	Wymiana informacji	3
2.2	Ruch	3
3	Algorytm	4
3.1	Pseudokod	4
3.2	Schemat blokowy	4
4	Implementacja	6
4.1	Klasa ABC	6
4.2	Inicjalizacje Koloni i pojedynczego osobnika	6
4.3	Obliczenia	7
4.3.1	Koszt	7
4.3.2	Dopasowanie	7
4.3.3	Prawdopodobieństwo	7
4.4	Nowe źródła pożywienia	8
4.4.1	Ruch	8
4.4.2	Wybieranie na zasadzie ruletki	9
4.4.3	Sprawdzanie granic	9
4.5	Fazy pszczół	9
4.5.1	Robotnice	9
4.5.2	Królowe	10
4.5.3	Skauci	10
4.6	Optymalizacja	10
4.6.1	Aktualizacja najlepszych rozwiązań	11
4.6.2	Dodawanie najlepszych rozwiązań do tablic z wynikami	11
4.7	Funkcje pomocnicze aby sformatować wyniki	11
4.7.1	Kilkukrotne wykonanie	11
4.7.2	Wykres i tabelka	12
4.7.3	Średnie	13
4.8	Funkcja do zminimalizowania	13
4.9	Wykonywanie algorytmu	13
5	Wyniki	14
5.1	Graficzne wyniki dla 5 pierwszych wykonania algorytmu	14
5.2	Statystyka z wyników	15
6	Wymagane biblioteki	15

1 Opis problemu przedstawionego w zadaniu.

1.1 Treść zadania

Zminimalizuj funkcję $f(x, y) = -(Floor[x * y]) + x^2 + y^2$ w zbiorze $[-50, 50]^2$ przy pomocy Algorytmu Pszczelego

2 Model Matematyczny

2.1 Wymiana informacji

Wymiana informacji między pszczołami jest modelowana równaniem (1)

$$P_i = \frac{fit_i}{\sum_{i=0}^n fit_i} \quad (1)$$

gdzie fit jest wartością dopasowania rozwiązań, do wyliczenia wartości fit używane jest równanie (2)

$$fit_i = \begin{cases} \frac{1}{1+F(x_i)} & \text{gdyn } F(x_i) \geq 0 \\ 1 + abs(F(x_i)) & \text{gdyn } F(x_i) < 0 \end{cases} \quad (2)$$

$F(x_i)$ jest wartością funkcji w punkcie, x_i jest wektorem zmiennych.
Służy do normalizacji wartości P_i

2.2 Ruch

Model ruchu opisywany jest równaniem (3)

$$x_i^{t+1} = x_i^t + \phi \cdot \beta \cdot \Delta x_{ik} \quad (3)$$

gdzie k jest losowym indeksem pszczoły, j jest losowym kierunkiem w którym pszczoła będzie zmierzać, β jest wektorem zer z wyjątkiem elementu o indeksie j , który jest 1, Δx_{ik} jest obliczana ze wzoru (4)

$$\Delta x_{ik} = x_{ij} - x_{kj} \quad (4)$$

3 Algorytm

3.1 Pseudokod

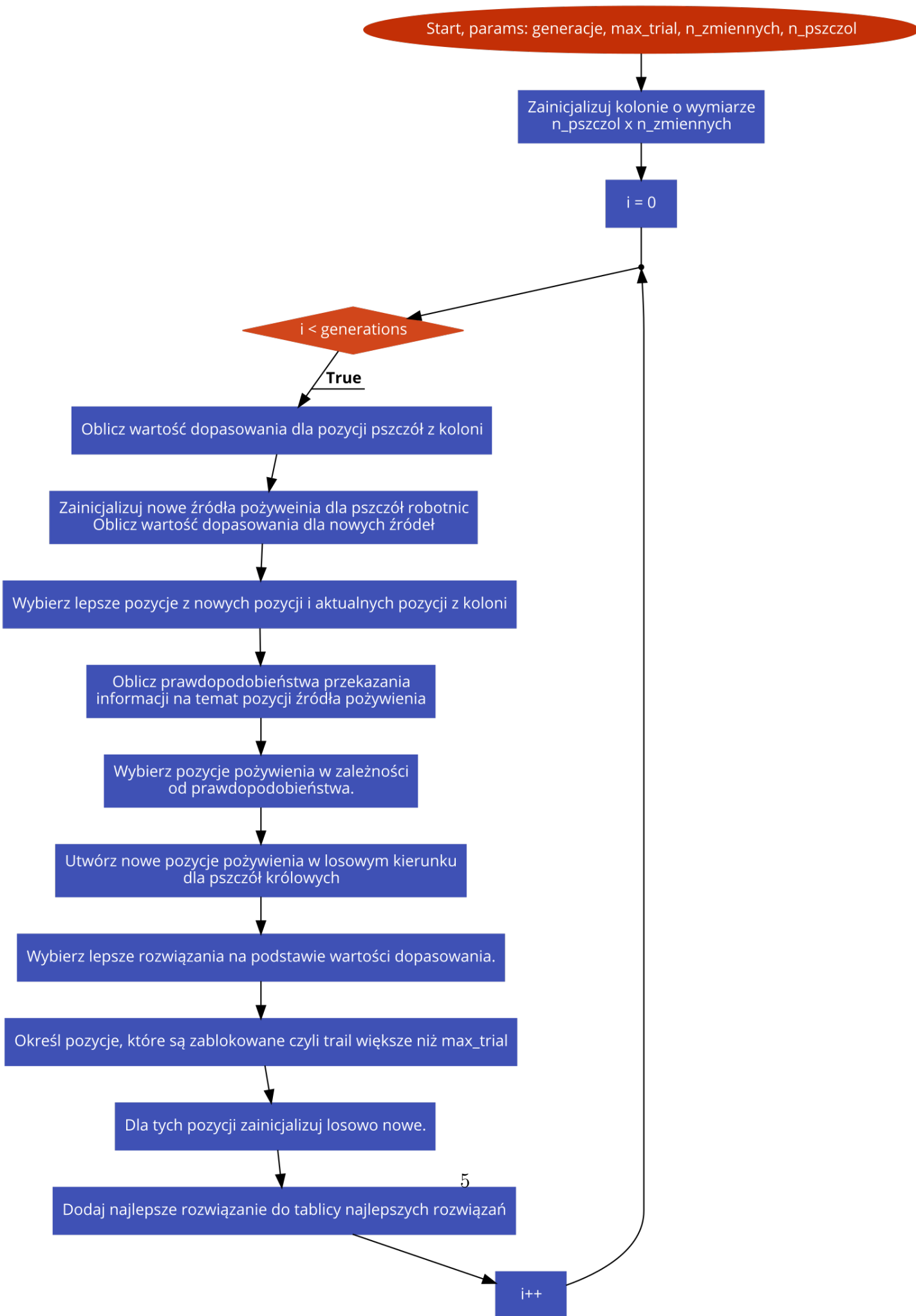
Algorytm Pszczeli

Data: Generacje, WielkośćKoloni, IlośćZmiennych, MaxPrób

Result: Tablica najlepszych wyników dla z każdej generacji ich kosztów

```
1 Zainicjalizować kolonie o wymiarach  $n\_pszczoł \times n\_zmiennych$ ;  
2  $i := 0$ ;  
3 while  $i < n\_generacji$  do  
4   Oblicz wartość dopasowania dla całej kolonii;  
5   Wyprodukuj nowe źródła pożywienia w sąsiedztwie pszczoł robotnic i wylicz  
   wartość dopasowania;  
6   Wybierz lepszą pozycję na podstawie wartości dopasowania źródeł pożywienia.;  
7   Oblicz prawdopodobieństwo wymiany informacji  $P_i$  za pomocą równania (1);  
8   Wyprodukuj nowe rozwiązania dla pszczoł królowych na bazie  
   prawdopodobieństw  $P_i$ ;  
9   Określ porzucone rozwiązania, jeśli jakieś istnieją i zastąp je nowymi losowo  
   wybranymi pozycjami pszczoł skautów;  
10  Zapamiętaj najlepsze rozwiązanie i dodaj je do tablicy najlepszych rozwiązań dla  
   danej generacji;  
11   $i++$ ;  
12 return Tablica najlepszych rozwiązań, Tablica kosztów dla danych rozwiązań
```

3.2 Schemat blokowy



4 Implementacja

4.1 Klasa ABC

Dany kod ukazuje metodę `__init__` klasy ABC, zawierają się tam inicjalizacje ważnych zmiennych.

```
1 class ABC:
2     def __init__(self, objective_func: Callable[[np.ndarray], np.ndarray],
3                   nvars: int, lb: float, ub: float,
4                   trial_max: int = 10, generations: int = 100, cs: int =
5                       50):
6         """
7         :param objective_func: Cost function
8         :param nvars: Dimensionality of problem
9         :param lb: Lower boundry
10        :param ub: Upper boundry
11        :param trial_max: Number of tries to update before scout phase
12        applies
13        :param generations: Number of iterations
14        :param cs: Colony Size
15        """
16        self.cs = cs
17        self.nvars = nvars
18        self.generations = generations
19        self.lb = lb
20        self.ub = ub
21        self.trial_max = trial_max
22        self.optimal_sol = np.array((1, self.nvars))
23        self.optimal_fitness = -1
24        self.optimality_tracking = np.zeros(shape=(generations, self.
25            nvars))
26        self.fitness_tracking = np.full(generations, -1)
27        self.cost_tracking = np.zeros(generations)
28        self.optimal_cost = None
29        self.obj_func = objective_func
30        self.trial = np.zeros(self.cs)
```

4.2 Inicjalizacje Koloni i pojedynczego osobnika

Inicjalizacje koloni nie tylko dla 2 zmiennych

```
1 def initialize_colony(self):
2     return np.random.uniform(self.lb, self.ub, (self.cs, self.nvars))
```

Inicjalizacje pojedynczego osobnika

```
1 def initialize_individual(self):
2     return np.random.uniform(self.lb, self.ub, (1, self.nvars))
```

4.3 Obliczenia

4.3.1 Koszt

Metoda `compute_cost` oblicza wartość funkcji w danym punkcie lub dla danej tablicy punktów.

```
1 def compute_cost(self, bees: np.ndarray) -> np.ndarray:
2     if bees.ndim == 1:
3         return self.obj_func(bees)
4     else:
5         return np.array(list(map(self.obj_func, bees)))
```

4.3.2 Dopasowanie

Metoda `compute_fitness` oblicza wartość dopasowania danego punktu lub tablicy punktów.

```
1 @staticmethod
2 def compute_fitness(cost):
3     '''
4     Static method that computes fitness value for given cost
5     :type cost: float or np.ndarray
6     :rtype: float or np.ndarray
7     '''
8     if type(cost) is np.ndarray:
9         fitness = np.zeros(cost.shape)
10        fitness[cost >= 0] = 1 / (1 + cost[cost >= 0])
11        fitness[cost < 0] = 1 + np.abs(cost[cost < 0])
12    else:
13        fitness = 1 / (1 + cost) if cost >= 0 else 1 + np.abs(cost)
14    return fitness
```

4.3.3 Prawdopodobieństwo

Metoda ta oblicza prawdopodobieństwo wybrania informacji podczas tańca robotnic przez królowe.

```
1 def compute_prob(self, col: np.ndarray) -> np.ndarray:
2     costs = self.compute_cost(col)
3     fitness = self.compute_fitness(costs)
4
5     prob = fitness / np.sum(fitness)
6     return prob
```

4.4 Nowe źródła pożywienia

4.4.1 Ruch

Metoda ta przeszukuje sąsiedztwo danej pszczoły w losowym kierunku, jeśli punkt wybrany okaże się być lepiej dopasowany to zostaje wstawiony do koloni na miejsce starego. Dla fazy królowych pozycje wybierane są na podstawie prawdopodobieństwa, przy pomocy metody wybierania na zasadzie ruletki.

```
1 def update(self, colony: np.ndarray, colony_fitness: np.ndarray, prob=
  None):
2     '''
3     Method that is used in employed bees phase and onlooker phase.
4     Generally it searches new location
5     Depends on prob param, it may be employed bee or onlooker.
6     :param colony: colony of bees
7     :param colony_fitness: fitness of each bee location
8     :param prob: probability that onlooker will go to place which is
        descibed by employeed bee which performs a dance
9     '''
10    n_new_bees = colony.shape[0]
11    newbees = np.copy(colony)
12    for i in range(n_new_bees):
13        idx = i
14        if prob is not None:
15            idx = self.roulette_wheel_selection(prob)
16        # Choose k randomly, not equal to i
17        k = np.random.choice(np.delete(np.arange(n_new_bees), i))
18
19        # Choose random variable
20        j = np.random.choice(self.nvars)
21
22        phi = np.random.uniform(-1, 1)
23
24        # Calculating new position
25        x = colony[idx, j] + phi * (colony[idx, j] - colony[k, j])
26
27        # Set bee's position to x or to the boundary if x exceeds it
28        newbees[idx, j] = self.check_boundries(x)
29
30    newbees_costs = self.compute_cost(newbees)
31    newbees_fitness = self.compute_fitness(newbees_costs)
32
33    # For the colony choose better positions
34    # Set to new bee where new bee fitness is greater than current bee
    fitness
35    colony[newbees_fitness > colony_fitness] = newbees[newbees_fitness >
        colony_fitness]
36
37    # Increment trial for bees that haven't changed position.
38    self.trial[newbees_fitness < colony_fitness] += 1
39
40    # Reset tial variable for new bees.
41    self.trial[newbees_fitness > colony_fitness] = 0
```

4.4.2 Wybieranie na zasadzie ruletki

Wybieranie punktu z zadanyam prawdopodobieństwem

```
1 def roulette_wheel_selection(self, p: np.ndarray) -> int:
2     '''
3     Method that chooses onlooker with given probability
4     :param p: probabilities
5     :return: index of chosen individual
6     '''
7
8     # Get random number between 0 and 1
9     rand = np.random.uniform()
10
11     subs = p - rand
12
13     # For each element in subs that is < 0, set to nan
14     subs[subs < 0] = np.nan
15
16     # if rand number is greater than each probability, each element in
17     # subs is nan
18     # so the procedure is repeated until there is an element with values
19     # != nan
20     while np.isnan(subs).all():
21         rand = np.random.uniform()
22         subs = p - rand
23         subs[subs < 0] = np.nan
24
25     # return index of minimum, which is the closest probability on the
26     # right side from rand
27     return np.nanargmin(subs)
```

4.4.3 Sprawdzanie granic

Metoda ta zwraca punkt jeśli mieści się w podanym zakresie do optymalizacji, a jeśli się nie mieści to zwraca daną granicę.

```
1 def check_boundries(self, x: float) -> float:
2     """
3     Method that checks if a new location is within boundries
4     :param x: new location
5     :return: float
6     """
7     if x < self.lb:
8         return self.lb
9     elif x > self.ub:
10        return self.ub
11    return x
```

4.5 Fazy pszczół

4.5.1 Robotnice

```
1 def employed_bee_phase(self, colony, colony_fitness):
2     self.update(colony, colony_fitness)
```

4.5.2 Królowe

```
1 def onlooker_be_phase(self, col, prob):
2     cost = self.compute_cost(col)
3     fitness = self.compute_fitness(cost)
4     self.update(col, fitness, prob)
```

4.5.3 Skauci

W tej fazie rozwiązania, które przekroczyły max_trial są zastępowane nowymi losowymi pozycjami.

```
1 def scout_phase(self, col):
2     for i in range(col.shape[0]):
3         # Create new scout for bees that trial exceeds its maximum value
4         if self.trial_max < self.trial[i]:
5             col[i] = self.initialize_individual()
6             self.trial[i] = 0
```

4.6 Optymalizacja

Główna metoda algorytmu

```
1 def optimize(self) -> [np.ndarray, np.ndarray]:
2
3     colony = self.initialize_colony()
4     for i in range(self.generations):
5         colony_cost = self.compute_cost(colony)
6         colony_fitness = self.compute_fitness(colony_cost)
7
8         self.employed_bee_phase(colony, colony_fitness)
9
10        # onlooker bee phase
11        prob = self.compute_prob(colony)
12
13        self.onlooker_be_phase(colony, prob)
14
15        # scout_phase
16        self.scout_phase(colony)
17
18        # Change optimal solution
19        self.get_best_sol(colony)
20
21        # Add best solution so far to the optimality tracking array.
22        self.add_solution_to_tracking_array(i)
23    return self.cost_tracking, self.optimality_tracking
```

4.6.1 Aktualizacja najlepszych rozwiązań

Metoda ta sprawdza czy najlepsze rozwiązanie z danej generacji jest lepsze niż dotychczasowe, jeśli jest lepsze to nadpisuje je

```
1 def get_best_sol(self, col):
2     cost = self.compute_cost(col)
3     fitness = self.compute_fitness(cost)
4
5     max_fitness = np.argmax(fitness)
6     if fitness[max_fitness] > self.optimal_fitness:
7         self.optimal_sol = np.copy(col[max_fitness])
8         self.optimal_fitness = np.copy(fitness[max_fitness])
9         self.optimal_cost = np.copy(cost[max_fitness])
```

4.6.2 Dodawanie najlepszych rozwiązań do tablic z wynikami

Metoda ta służy po to by móc odtworzyć rozwiązania po zakończeniu wykonywania algorytmu. Dodaje dotychczasowe najlepsze rozwiązanie na index aktualnej generacji.

```
1 def add_solution_to_tracking_array(self, i):
2     self.fitness_tracking[i] = self.optimal_fitness
3     self.optimal_tracking[i] = self.optimal_sol
4     self.cost_tracking[i] = self.optimal_cost
```

4.7 Funkcje pomocnicze aby sformatować wyniki

4.7.1 Kilukrotne wykonanie

Kilkukrotne wykonanie algorytmu aby sprawdzić ilość generacji potrzebnych, aby znaleźć minimum przy określonej maksymalnej ilości generacji.

```
1 def perform_n_runs(n: int, func, nvars: int, lb: float, ub: float,
2     generatios: int, verbose: bool=False) -> [list, list]:
3     costs = []
4     sols = np.zeros((n, generatios, nvars))
5     idx_of_final_sol = []
6
7     for i in range(n):
8         cost, sol = ABC(func, nvars, lb, ub, generations=generatios).
9             optimize()
10
11         tmp_sol = sol[-1]
12
13         # Get index of first occurrence of final solution
14         idx_of_final_sol.append(np.where(np.array(sol == tmp_sol).all(
15             axis=1))[0][0])
16
17         costs.append(cost[:idx_of_final_sol[i]])
18         sols[i] = sol
19
20     if verbose:
```

```

18         print('Execution: {nr}\nMinimum at: {sol}\nCost: {cost}\n
              nFinal solution at {idx} generation\n'.format(nr=i+1,
19                   sol=tmp_sol, cost=cost[-1], idx=idx_of_final_sol[i]))
20
21     return costs, sols, idx_of_final_sol

```

4.7.2 Wykres i tabela

Funkcja, która tworzy zestawienie wyników dla 5 pierwszych wywołań algorytmu. Wykonywana jest tylko gdy liczba zmiennych jest 2

```

1  def results_plt(costs: list, sols: np.ndarray, idx: list):
2      '''
3      Function that creates plot and table with performance of each
4      algorithm execution
5      :param costs: 2 dimensional list with costs tracking for each
6      algorithm execution
7      :param sols: 3 dimensional array with solutions for each algorithm
8      execution
9      :param idx: index of generation which attains a minimum
10     '''
11     if sols.shape[2] != 2:
12         return
13
14     costs = costs[:5]
15     sols = sols[:5]
16     idx = idx[:5]
17
18     sol = sols[:, -1, :]
19     best_costs = [cost[-1] for cost in costs]
20
21     fig, ax = plt.subplots(2, 1, gridspec_kw={'height_ratios': [3, 1]})
22
23     x, y = sol[:, 0], sol[:, 1]
24     cols = [['{:.2e}'.format(x[i]), '{:.2e}'.format(y[i]), idx[i], '{:.6
25             e}'.format(best_costs[i])] for i in
26             range(len(idx))]
27     colLabels = ['x', 'y', 'Generations needed', 'Cost']
28
29     row_idx = ['{i: ^10}'.format(i=i) for i in range(len(idx))]
30
31     tab0 = ax[1].table(cellText=cols, cellLoc='center', colLabels=
32         colLabels, rowLabels=row_idx,
33         rowColours=['orange'] * len(row_idx),
34         colColours=['palegreen'] * len(colLabels), bbox
35         =[0, -.2, 1, 1])
36
37     tab0.auto_set_font_size(False)
38     tab0.set_fontsize(9)
39     ax[1].axis('tight')
40     ax[1].axis('off')
41
42     line = [None] * len(idx)
43     for i, cost in enumerate(costs):
44         line[i], = ax[0].plot(np.arange(idx[i]), cost[:idx[i]])

```

```

38         line[i].set_label(str(i))
39
40     ax[0].legend()
41
42     ax[0].set_xticks([x for x in np.arange(0, max(idx), max(idx) // 10)
43                      ])
44     ax[0].set_xlabel('Generations')
45     ax[0].set_ylabel('Cost')
46     ax[0].grid()
47     ax[0].set_title('Cost plot after n generations for first 5
48                     executions')
49
50     file = 'ABCiterations_results.png'
51     plt.savefig(file)
52     print('Plot saved in file {}'.format(file))

```

4.7.3 Średnie

Funkcja oblicza średnią wartość obliczonych minimum i średnią ilość generacji potrzebnych by osiągnąć dane minimum.

```

1 def stat(idx, costs):
2     final_costs = np.array([cost[-1] for cost in costs])
3     final_cost_mean = np.mean(final_costs)
4
5     idx_mean = sum(idx) / len(idx)
6
7     print('Final cost mean: {cost_mean}\nMean of generations that
8           reached minimum: {idx_mean}'.format(cost_mean=final_cost_mean,
9           idx_mean=idx_mean))

```

4.8 Funkcja do zminimalizowania

```

1 def FunctionToMinimize(x):
2     return -np.floor(np.prod(x)) + x @ x.T

```

4.9 Wykonywanie algorytmu

Kod potrzebny aby zoptymalizować funkcje z przykładowymi parametrami, następnie wykonać tabelkę, wykres i policzyć średnie wszystkich wykonań

```

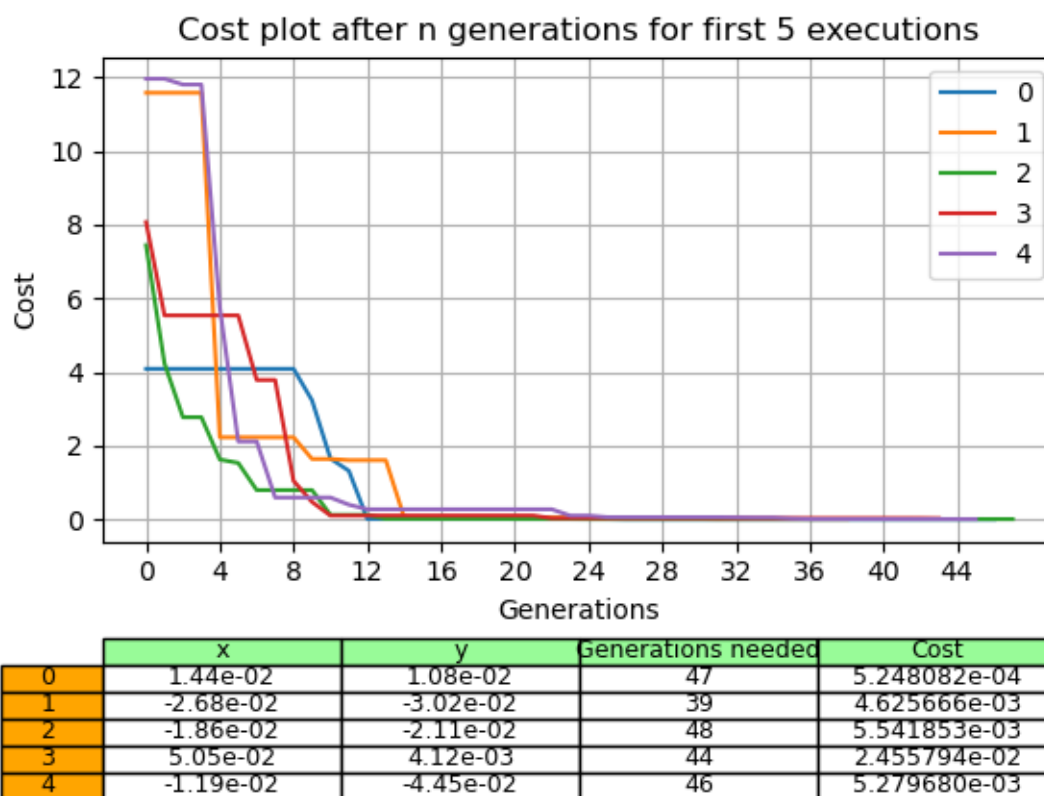
1 costs, sols, idx = perform_n_runs(20, FunctionToMinimize, 2, -50, 50,
2                                  50, True)
3 results_plt(costs, sols, idx)
4 stat(idx, costs)

```

5 Wyniki

5.1 Graficzne wyniki dla 5 pierwszych wykonania algorytmu

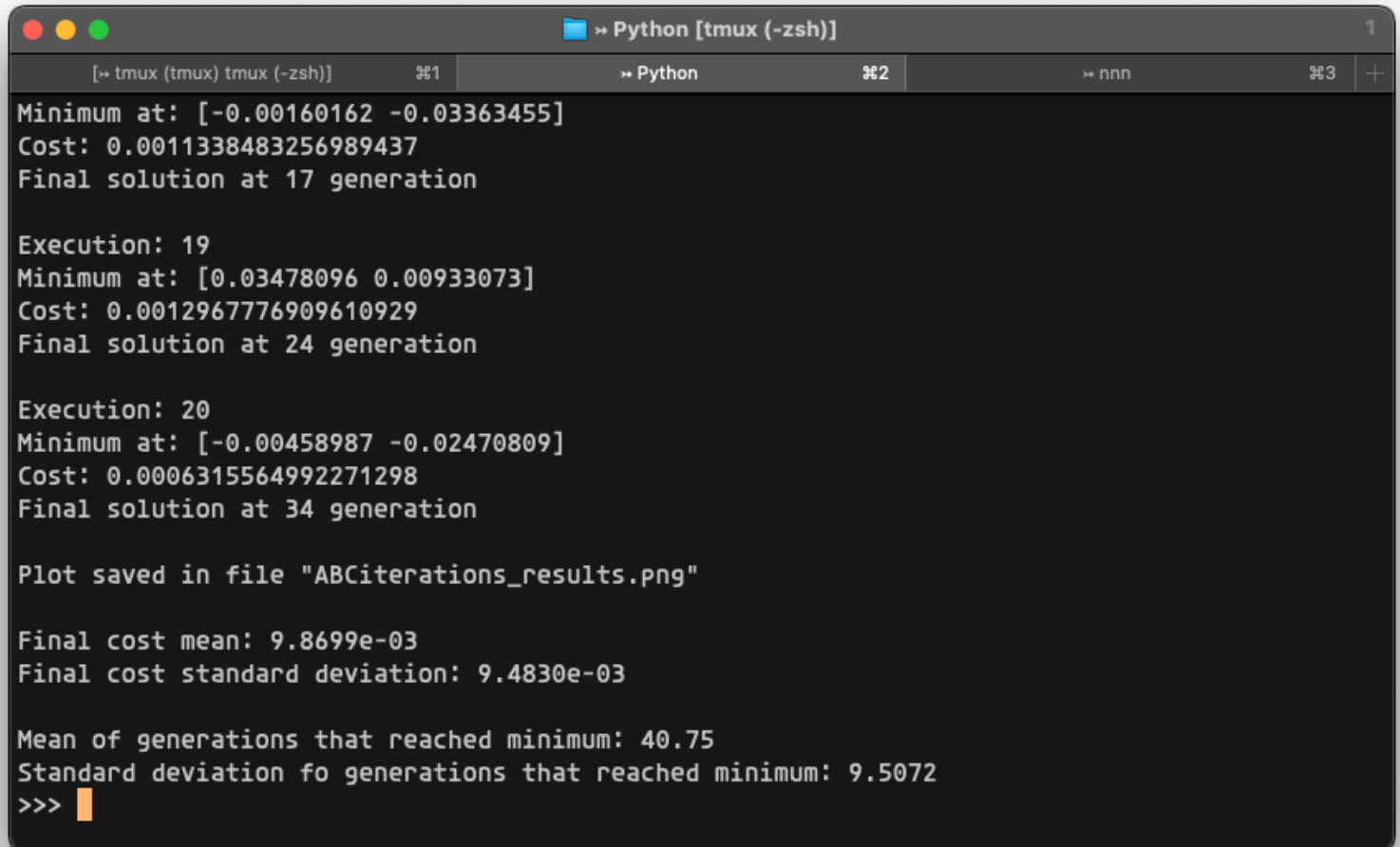
Wykres jaka była wartość funkcji dla dotychczasowego najlepszego rozwiązania w danej generacji. Tabelka Podsumowująca



Rysunek 2: Wyniki 5 pierwszych wykonania

5.2 Statystyka z wyników

Średnia, standardowe odchylenie obliczonych minimum i średnia, odchylenie standardowe ilości generacji potrzebnych by osiągnąć dane minimum.



```
Python [tmux (-zsh)]
[➤ tmux (tmux) tmux (-zsh)] %1      ➤ Python %2      ➤ nnn %3 +
Minimum at: [-0.00160162 -0.03363455]
Cost: 0.0011338483256989437
Final solution at 17 generation

Execution: 19
Minimum at: [0.03478096 0.00933073]
Cost: 0.0012967776909610929
Final solution at 24 generation

Execution: 20
Minimum at: [-0.00458987 -0.02470809]
Cost: 0.0006315564992271298
Final solution at 34 generation

Plot saved in file "ABCiterations_results.png"

Final cost mean: 9.8699e-03
Final cost standard deviation: 9.4830e-03

Mean of generations that reached minimum: 40.75
Standard deviation fo generations that reached minimum: 9.5072
>>> █
```

Rysunek 3: Wyniki Statystyczne

6 Wymagane biblioteki

Używane biblioteki zewnętrzne w moim kodzie to numpy i matplotlib. W razie potrzeby uruchomienia algorytmu pierw trzeba je zainstalować. Dołączyłem plik requirements.txt, za pomocą pip-a można doinstalować.

```
pip3 install -r requirements.txt
```