
# Optymalizacja procesu przydzielania pracowników do projektów

## Badania Operacyjne 2022

Anna Gut, Zuzanna Borek, Kamil Wrześniak

# Optymalizacja procesu przydzielania pracowników do projektów

Anna Gut, Zuzanna Borek, Kamil Wrześniak

Grupa poniedziałek 17:50 B

Czerwiec 2022

# 1. Wstęp

Celem tego projektu jest sprawdzenie skuteczności wykorzystania metod badań operacyjnych w rozwiązywaniu problemów trudnych obliczeniowo oraz takich, dla których nie istnieją algorytmy optymalne. Wymyślony przez nas problem odpowiedniego przypisania kandydatów do stanowisk, opisany dokładniej w kolejnym rozdziale tej pracy, okazał się dobrze dobrany do wspomnianego celu, ponieważ próba jego optymalnego rozwiązania metodami tradcyjnymi przy zadanych założeniach jest zadaniem niezwykle trudnym. W poniższej dokumentacji znajdują się dokładne opisy aspektów t. j. : postać zadanego problemu, implementacja jego rozwiązania, faza testowania oraz końcowe wnioski. Projekt był realizowany w ramach przedmiotu Badania Operacyjne na kierunku Informatyka na Akademii Górniczo-Hutniczej w Krakowie.

# 2. Opis problemu

Problemem, którym będziemy się w tej pracy zajmować, jest próba optymalnego przypisania kandydatów posiadających określony poziom zadanych umiejętności do dostępnych stanowisk w pewnej firmie. Każde ze stanowisk jest również opisywane sumarycznymi wartościami cech, które są na nim wymagane. Danymi wejściowymi, na których operować będzie stworzony przez nas algorytm w celu znalezienia najlepszego możliwego rozwiązania będą:

- lista rozważanych umiejętności
- lista pracowników wraz z tablicą wartości liczbowych określających natężenie każdej z umiejętności (w skali 0-10)
- lista stanowisk wraz z tablicą wartości liczbowych określających łączne zapotrzebowanie na każdą z umiejętności (w skali 0-100)

# 3. Model matematyczny

W tej sekcji przedstawiony jest wynik naszej pracy nad stworzeniem modelu matematycznego dla problemu opisanego w poprzednim punkcie.

## 3.1. Opis modelu

Poniżej znajdują się główne założenia dotyczące naszego modelu:

- Mamy M kandydatów na pracowników w firmie XYZ, gdzie M$\geq$0
- Firma XYZ ma dostępnych N stanowisk do obsadzenia, gdzie N$\geq$0
- Każde stanowisko może zostać obsadzone przez liczbę pracowników z przedziału [0, ... ,M]
- Każdy z kandydatów musi zostać przydzielony na dokładnie jedno stanowisko
- Istnieje zbiór K cech (umiejętności), każda z tych cech może przyjmować wartości nieujemne.
- Każdego pracownika opisuje zbiór K wartości naturalnych z przedziału [0,..,10], które odpowiadają rozwinięciu każdej z cech u pracownika (0 oznacza całkowity brak danej umiejętności, a 10 oznacza, że ta osoba jest ekspertem w danej dziedzinie).
- Każde stanowisko w firmie opisuje zbiór K wartości naturalnych z przedziału [0,..,100], które opisują wymagany łączny poziom każdej z cech na tym stanowisku (rozkład analogicznie jak przy pracownikach).
- Stanowisko przynosi firmie tym większy przychód, im bliżej jest do wypełnienia wymaganego poziomu każdej z cech przez pracowników

## 3.2. Struktury danych w modelu

Poniżej znajdują się założenia dotyczące danych, którymi będziemy się posługiwać przy definiowaniu rozwiązania oraz funkcji celu.

- M - ilość kandydatów
- N - ilość dostępnych stanowisk

- K - ilość cech
- $x_{i1}$, $x_{i2}$, ... , $x_{iK}$ - umiejętności pracownika i
- $y_{j1}$, $y_{j2}$, ... , $y_{jK}$ - łączne umiejętności wymagane na stanowisku j

## 3.3. Postać oczekiwanego rozwiązania

Rozwiązanie zwracane przez nasz algorytm będzie funkcją postaci: $F(m) = n$, która każdemu pracownikowi przypisuje stanowisko, na które zostanie zatrudniony

## 3.4. Funkcja celu

Funkcja celu w naszym modelu przyjmie postać:

$$\sum_{i=1}^{N} f_i(x_{i11}, ..., x_{i1K}, x_{i12}, ..., x_{i2K}, ..., x_{iP1}, ..., x_{iPK}, y_{i1}, ..., y_{iK}) -> min$$

Gdzie:

$$f_i(x_{i11}, ..., x_{i1K}, x_{i12}, ..., x_{i2K}, ..., x_{iP1}, ..., x_{iPK}, y_{i1}, ..., y_{iK}) = \sum_{j=1}^{K}(y_{ij} - \sum_{l=1}^{P} x_{lj})$$

$f_i$ - funkcja opisująca wypełnienie stanowiska umiejętnościami pracowników, gdzie $x_{abc}$ - wartość cechy c pracownika b, który został zatrudniony na stanowisku a

UWAGA! Jeśli wartość $\sum_{j=1}^{K}(y_{ij} - \sum_{l=1}^{P} x_{lj}) < 0$ to przyjmujemy wartość 0.

# 4. Opis algorytmów pomocniczych

## 4.1. Generowanie danych wejściowych

Dane wejściowe w naszej aplikacji są reprezentowane poprzez strukturę zawierającą następujące pola:

- skills - lista rozważanych umiejętnośc
- candidates - imiona kandydatów na stanowiska w firmie
- positions - lista ID dostępnych pozycji w firmie
- candidates_skills - listy zawierające wartości liczbowe określające poziom kolejnych umiejętności dla każdego z pracowników
- positions_skills - listy zawierające wartości liczbowe określający wymagany poziom kolejnych umiejętności na każdym ze stanowisk

Przykładowa struktura wejściowa dla przypadku z 5-cioma rozważanymi umiejętnościami, 6-cioma kandydatami i 4-ema stanowiskami może więc przyjąć następującą postać:

```
{
    'skills': ['Embedded Systems', 'Netsniff-ng', 'Novell Netmail',
               'NCBI Epigenomics', 'Eco-Investing'],
    'candidates': ['Will', 'James', 'Samuel', 'John', 'George', 'Sam'],
    'positions': ['b90fdaf3-eff5-41ea-9a7d-f2f9f100639d',
                  '14f146d1-71ec-4528-a021-4decb0f13918',
                  '55097c9b-5dcd-401d-8ca6-3d20214338fe',
                  '96545b0f-1c5f-4c87-aaa9-f6df2a4d62e3'],
    'candidates_skills': [[4, 4, 9, 6, 6], [3, 0, 3, 4, 5], [8, 3, 6, 7, 0],
                          [0, 9, 2, 1, 8], [6, 1, 10, 2, 7], [0, 3, 6, 6, 4]],
    'position_skills': [[90, 3, 41, 35, 21], [38, 22, 83, 2, 46],
                        [8, 99, 22, 92, 31], [38, 52, 46, 44, 77]]
}
```

Do generowania imion oraz nazw umiejętności wykorzystaliśmy dostępne w sieci darmowe API: *parseapi(1)* oraz *itsyourskills(2)*. Przykładowe ID pozycji oraz wartości liczbowe zostały wygenerowane przy pomocy bibliotek uuid oraz random.Poniżej znajdują się fragmenty napisanego przez nas skryptu do generowania danych:

Rysunek 1: Generowanie zbioru umiejętności

```
def get_sample_skillset(k: int):
    data = pd.read_excel(r'../resources/Skills.xlsx')
    skillset = []

    for row in data.values:
        skillset.append(row[0])
    random.shuffle(skillset)

    try:
        random_indexes = random.sample(range(0, len(skillset) - 1), k)
    except ValueError:
        print('Sample size exceeded population size.')
        return []

    solution = []
    for idx in random_indexes:
        solution.append(skillset[idx])
    return solution
```

Rysunek 2: Generowanie zbioru kandydatów

```
def get_sample_candidates_names(m: int):
    complete_url = names_url.replace("count", str(m))
    data = json.loads(requests.get(complete_url, headers=names_headers).content.decode('utf-8'))
    solution = []
    for result in data['results']:
        solution.append(result['Name'])
    return solution
```

Rysunek 3: Generowanie ID stanowisk

```python
def get_sample_positions_ids(n: int):
    solution = []
    for _ in range(n):
        solution.append(uuid.uuid4().__str__())
    return solution
```

Rysunek 4: Generowanie przykładowej struktury danych wejściowych

```python
def get_sample_case(k: int, m: int, n: int):
    solution = dict()
    skills = get_sample_skillset(k)
    candidates = get_sample_candidates_names(m)
    positions = get_sample_positions_ids(n)
    candidates_skills = [[] for _ in range(m)]
    for idx in range(m):
        for _ in range(k):
            candidates_skills[idx].append(random.randint(0, 10))

    positions_skills = [[] for _ in range(n)]
    for idx in range(n):
        for _ in range(k):
            positions_skills[idx].append(random.randint(0, 100))

    solution['skills'] = skills
    solution['candidates'] = candidates
    solution['positions'] = positions
    solution['candidates_skills'] = candidates_skills
    solution['position_skills'] = positions_skills
    return solution
```

## 4.2. Generowanie populacji początkowej rozwiązań

Jako rozwiązanie problemu dla zadanych wartości początkowych otrzymujemy strukturę, w której do każdego ID stanowiska jest przypisana lista kandydatów, którzy zostali do niego przypisani. Przykładowe rozwiązanie może więc wyglądać następująco:

```
{
  'e07cede2-eca9-4698-8659-93b2d40f4921': ['Samuel', 'John', 'Sam'],
  '6406658b-6333-43c1-9904-a07da08fd1ed': ['Will', 'James', 'George'],
  '00f242ec-48a7-4e9a-a847-c1dc18456427': []
}
```

Jeszcze innym sposobem na przedstawienie rozwiązania jest postać macierzowa, w której wartość 1 w komórce (x,y) oznacza, że do stanowiska o indeksie x został przypisany kandydat o indeksie y.
Przykład:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Poniżej znajdują się skrypty, które wykorzystaliśmy do generowania przykładowych rozwiązań. Pierwszy z nich operuje na strukturze wejściowej, której opis przedstawiliśmy w poprzedniej sekcji, a drugi przyjmuje dwie liczby całkowite M i N, gdzie M - liczba kandydatów, a N - liczba stanowisk i zwraca wynik w postaci macierzowej.

Rysunek 5: Generowanie rozwiązania na podstawie struktury wejściowej

```python
def gen_sample_solution(sample_case):
    n = len(sample_case['positions'])
    m = len(sample_case['candidates'])
    solution = dict()
    solution_matrix = [[0 for _ in range(m)] for _ in range(n)]
    for position in sample_case['positions']:
        solution[position] = []
    for candidate_idx, candidate in enumerate(sample_case['candidates']):
        idx = random.randint(0,n-1)
        solution[sample_case['positions'][idx]].append(candidate)
        solution_matrix[idx][candidate_idx] = 1
    return (solution, solution_matrix)
```

Rysunek 6: Generowanie rozwiązania na podstawie ilości kandydatów i stanowisk

```python
def gen_sample_sol_mn(m: int, n: int):
    solution_matrix = [[0 for _ in range(m)] for _ in range(n)]
    for idx in range(m):
        sol_idx = random.randint(0, n-1)
        solution_matrix[sol_idx][idx] = 1
    return solution_matrix
```

## 4.3. Kalkulator do obliczania kosztu

Zaimplementowaliśmy również kalkulator, który na podstawie otrzymanego rozwiązania wylicza wartość funkcji kosztu według wzoru z przedstawionego wyżej modelu matematycznego.

Rysunek 7: Kalkulator do obliczania wartości funkcji kosztu dla rozwiązania

```python
def calculate_cost(case, solution_matrix):
    k = len(case['skills'])
    cost = 0
    for position_idx, candidates in enumerate(solution_matrix):
        for skill_idx in range(k):
            curr_cost = case['position_skills'][position_idx][skill_idx]
            for candidate_idx, chosen in enumerate(candidates):
                if chosen == 1:
                    curr_cost -= case['candidates_skills'][candidate_idx][skill_idx]
            if curr_cost < 0:
                curr_cost = 0
            cost += curr_cost
    return cost
```

# 5. Algorytm pszczeli rozwiązujący powyższy problem

Do rozwiązania badanego problemu wykorzystaliśmy algorytm pszczeli, który jest jednym z popularniejszych algorytmów rojowych. Jego działanie oparte jest na symulacji roju owadów zbierających pożywienie. W modelu można wyróżnić dwie grupy pszczół - skautów i robotnice. Zadaniem pierwszej z nich jest przeszukiwanie terenu w poszukiwaniu pokarmu (eksploracja) natomiast robotnice zajmują się zbieraniem nektaru z wcześniej znalezionych obszarów (eksploatacja). Algorytm posiada następujące parametry sterujące:

- case - rozpatrywana instancja problemu
- ns - liczba pszczół skautów
- ne - liczba elitarnych sąsiedztw
- nre - liczba pszczół w każdym elitarnym sąsiedztwie
- nb - liczba zwykłych sąsiedztw
- nrb - liczba pszczół w każdym zwykłych sąsiedztwie
- ttl - czas życia niewydajnego sąsiedztwa
- radius - domyślny początkowy promień sąsiedztwa, radius $<$ m
- iterations - po ilu krokach algorytm się zatrzymuje

Inicjalizacja powyższych parametrów wykonywana jest w metodzie __init__ klasy Algorithm.

Rysunek 8: Inicjalizacja parametrów sterujących algorytmu

```
class Algorithm:
    def __init__(self,
                 case: dict,
                 ns: int = 2,
                 ne: int = 1,
                 nre: int = 2,
                 nb: int = 2,
                 nrb: int = 1,
                 ttl: int = 4,
                 radius: int = 10,
                 iterations: int = 100):
```

Na początku wszystkie pszczoły zostają umieszczone w losowych miejscach przestrzeni rozwiązania.

Rysunek 9: Inicjalizacja pszczół

```
# Inicjalizacja
self.hive: [Bee] = []
self.bee_scouts: [Bee] = []
self.elite_hoods: [Neighbourhood] = []
self.good_hoods: [Neighbourhood] = []
for i in range(ne):
    bees = []
    for j in range(nre):
        name = get_bee_name()
        bees.append(Bee(case, name, TYPES.elite_forager))
    self.elite_hoods.append(Neighbourhood(bees, self.radius, self.ttl))
    self.hive += bees
for i in range(nb):
    bees = []
    for j in range(nrb):
        name = get_bee_name()
        bees.append(Bee(case, name, TYPES.elite_forager))
    self.good_hoods.append(Neighbourhood(bees, self.radius, self.ttl))
    self.hive += bees
for i in range(ns):
    name = get_bee_name()
    new_bee = Bee(case, name, TYPES.scout)
    self.bee_scouts.append(new_bee)
    self.hive.append(new_bee)
```

Następnie w kolejnym etapie algorytmu (metoda step) aż do wykonania zadanej liczby iteracji następuje kolejno:

1. Powrót pszczół do ula. Najgorsze znalezione sąsiedztwa są zastępowane przez nowe znalezione przez skautów.

Rysunek 10: Powrót pszczół do ula

```
all_neighbourhoods = self.elite_hoods + self.good_hoods + [Neighbourhood([bee], self.radius, self.ttl)
                                                           for bee in self.bee_scouts]

best_solution, best_score = all_neighbourhoods[0].center.position, all_neighbourhoods[0].cost
if best_score < self.best_score:
    print_debug("Nowy najlepszy wynik", all_neighbourhoods[0].cost, 'Znaleziony przez pszczoły:',
                all_neighbourhoods[0].bees)
    self.best_solution = best_solution
    self.best_score = best_score

self.elite_hoods = all_neighbourhoods[:self.ne]
self.good_hoods = all_neighbourhoods[self.ne: (self.ne + self.nb)]
scout_hoods = [neighbourhood.bees for neighbourhood in all_neighbourhoods[(self.ne + self.nb):]]
self.bee_scouts = []
for bees in scout_hoods:
    self.bee_scouts += bees
```

2. Przypisanie pszczół do odpowiednich ról zgodnych z parametrami sterującymi.

Rysunek 11: Przypisanie pszczól do odpowiednich ról

```
self.elite_hoods = all_neighbourhoods[:self.ne]
self.good_hoods = all_neighbourhoods[self.ne: (self.ne + self.nb)]
scout_hoods = [neighbourhood.bees for neighbourhood in all_neighbourhoods[(self.ne + self.nb):]]
self.bee_scouts = []
for bees in scout_hoods:
    self.bee_scouts += bees

for hood in self.elite_hoods:
    if len(hood.bees) > self.nre:
        self.bee_scouts += hood.bees[self.nre:]
        hood.bees = hood.bees[:self.nre]

for hood in self.good_hoods:
    if len(hood.bees) > self.nrb:
        self.bee_scouts += hood.bees[self.nrb:]
        hood.bees = hood.bees[:self.nrb]

for hood in self.elite_hoods:
    index = self.nre - len(hood.bees)
    if len(hood.bees) < self.nre:
        hood.bees += self.bee_scouts[:index]
        self.bee_scouts = self.bee_scouts[index:]

for hood in self.good_hoods:
    index = self.nrb - len(hood.bees)
    if len(hood.bees) < self.nrb:
        hood.bees += self.bee_scouts[:index]
        self.bee_scouts = self.bee_scouts[index:]
```

3. Lokalne przeszukiwanie sąsiedztw polegające na wykonaniu radius zmian pozycji o 1. Jeśli nowa pozycja jest gorsza od poprzedniej, pszczoła wraca do poprzedniego miejsca.

Rysunek 12: Lokalne przeszukanie

```python
# 1. Lokalne poszukiwania
for index, hood in enumerate(self.elite_hoods):
    hood.search()
    if hood.ttl < 0:
        # porzuć
        deleted_neighbourhood = self.elite_hoods.pop(index)
        self.bee_scouts += deleted_neighbourhood.bees

for index, hood in enumerate(self.good_hoods):
    hood.search()
    if hood.ttl < 0:
        # porzuć
        deleted_neighbourhood = self.good_hoods.pop(index)
        self.bee_scouts += deleted_neighbourhood.bees
```

Rysunek 13: Metoda search

```python
def local_search(self, radius) -> bool:
    """
    Zmień pozycję pszczoły "r = neighbourhood radius" razy.
    Nie ma gwarancji, że odległość od poprzedniego rozwiązania wyniesie r, może wynieść mniej.
    Jeśli nowa pozycja jest gorsza, powróć do starej pozycji i zwróć False.
    W przeciwnym razie zostań przy nowej pozycji i zwróć True.

    Returns:
    bool: Czy pozycja się zmieniła?
    """
    old_position = self.position
    old_cost = self.cost
    for i in range(radius):
        self.position = self._gen_new_sol_m()
    self._evaluate()
    if self.cost > old_cost:
        self.position = old_position
        self.cost = old_cost
        return False
    return True
```

W przypadku gdy w obszarze nie uda się znaleźć lepszego miejsca jego rozmiar jest zmniejszany. Wykonywane jest to w metodzie _shrink.

Rysunek 14: Metoda _shrink

```python
def _shrink(self):
    self.ttl = max(int(0.8 * self.ttl), 1)
    if self.radius > 1:
        self.radius -= 1
```

4. Globalne przeszukiwanie polegające na wysłaniu skautów w losowe miejsca przestrzeni rozwiązania.

Rysunek 15: Globlne przeszukanie

```
# 2. Globalne poszukiwania
for bee in self.bee_scouts:
    bee.global_search()
```

Rysunek 16: Metoda global_search

```
def global_search(self):
    self.set_random_position()
    self._evaluate()
```

# 6. Opis uruchamiania aplikacji

Wszystkie stworzone przez nas skrypty zostały umieszczone w folderze scripts naszego projektu. W celu uruchomienia aplikacji dla przykładowego zestawu danych należy pobrać nasz projekt korzystając z linku:

`https://github.com/aniagut/BO-recruitment-process`

, a następnie z poziomu folderu *scripts* wywołać w konsoli polecenie:

python app_sample.py <l. umiejętności> <l. kandydatów> <l. pozycji>

Dla uprzednio zdefiniowanej struktury danych wejściowych w formacie opisanym powyżej i zapisanej w pliku JSON wywołujemy polecenie:

python app.py <ścieżka do pliku>

Algorytm dla zadanych (lub wylosowanych) danych wykona kolejno:

1. Wczytanie (lub wylosowanie) egzemplarza problemu
2. Inicjalizację solvera
3. Sześćdziesiąt sześć iteracji algorytmu
4. Wypisanie informacji w przypadku pojawienia się nowego, lepszego rozwiązania
5. Wypisanie otrzymanego wyniku

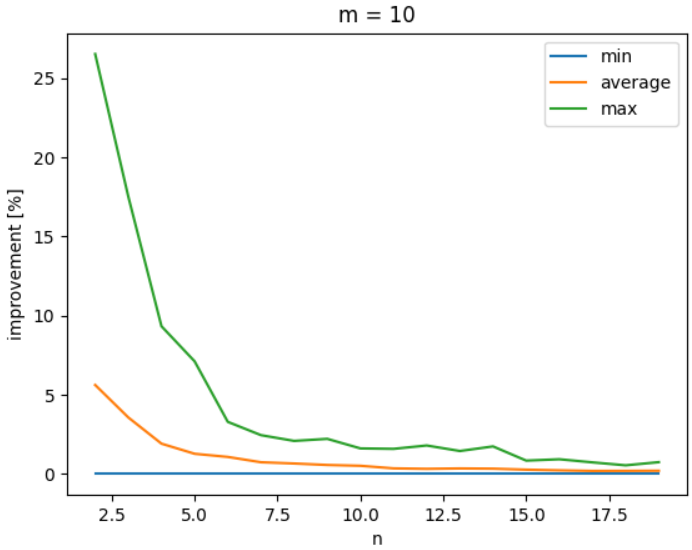
# 7. Opis fazy testowej

W fazie testowej sprawdzany był wpływ parametrów zadanego problemu i algorytmu na jakość uzyskanego rozwiązania końcowego. Ocena dokonywana była poprzez wyznaczenie stosunku uzyskanej wartości funkcji celu dla rozwiązania wyznaczonego przy pomocy algorytmu pszczelego, do kosztu rozwiązania losowego. W celu zmniejszenia wpływu parametrów losowych na uzyskane wyniki, każdy test został przeprowadzony 100 razy, a z uzyskanych wyników pozostawiono wartość minimalną, maksymalną oraz średnią.

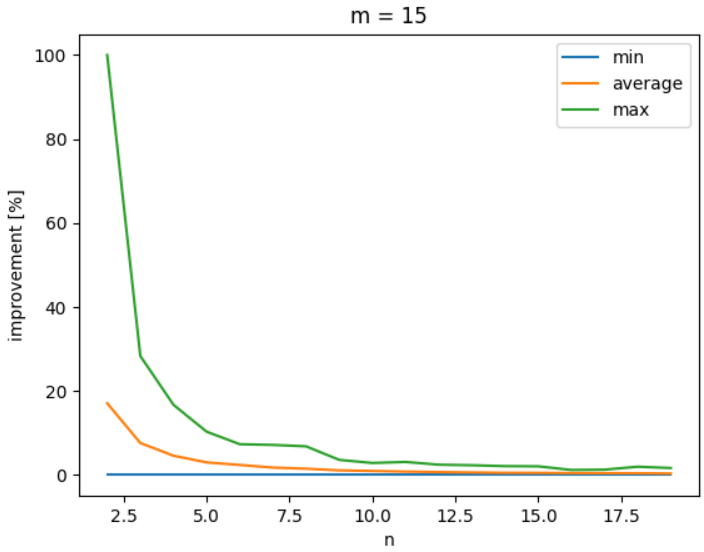
## 7.1. Badanie wpływu rozmiaru problemu na poprawę wyniku

W celu sprawdzenia wpływu rozmiaru problemu na uzyskany wynik, przeprowadzono trzy testy, w których zmianie podlegała liczba kandydatów m oraz liczba pozycji n. Zmienna k (ilość umiejętności) oraz liczba iteracji algorytmu były stałe i wynosiły odpowiednio 5 i 50.

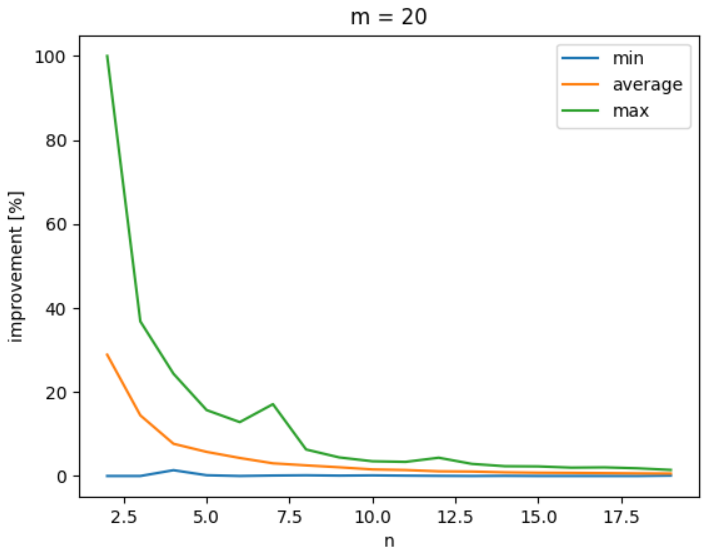
Rysunek 17: Test dla niewielkiej wartości m oraz zmiennego n



Rysunek 18: Test dla średniej wartości m oraz zmiennego n




Rysunek 19: Test dla dużej wartości m oraz zmiennego n

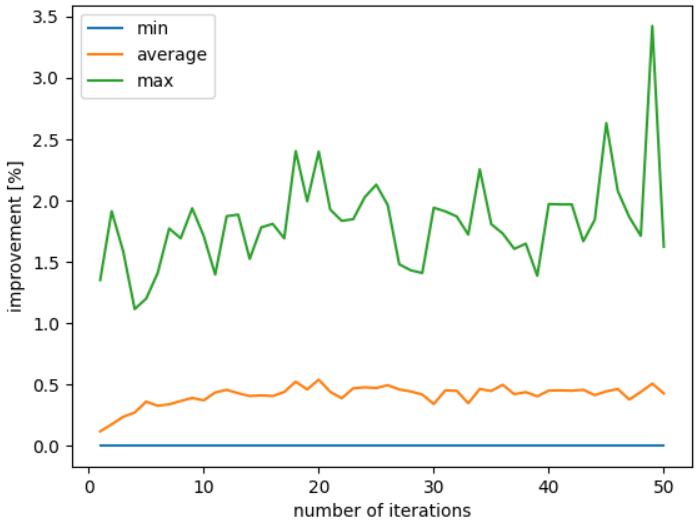


Z powyższych wykresów wynika, że dla małej ilości kandydatów (m=10) i małej ilości stanowisk (n=2) przy zastosowaniu naszego algorytmu jesteśmy w stanie poprawić uzyskane rozwiązanie o około 5%. Czym większa ilość kandydatów, tym większa jest uzyskana poprawa - dla 20 kandydatów i 2 stanowisk wynosi ona już około 25%. Poprawa ta spada wraz ze wzrostem ilości stanowisk.

## 7.2 Badanie wpływu liczby iteracji na poprawę wyniku

Ten pomiar został wykonany w celu sprawdzenia stopnia poprawy rozwiązania początkowego w zależności od ilości iteracji algorytmu. Przyjęto wartości parametrów problemu m = n = 10 oraz k = 5.

Rysunek 20: Test wpływu liczby iteracji

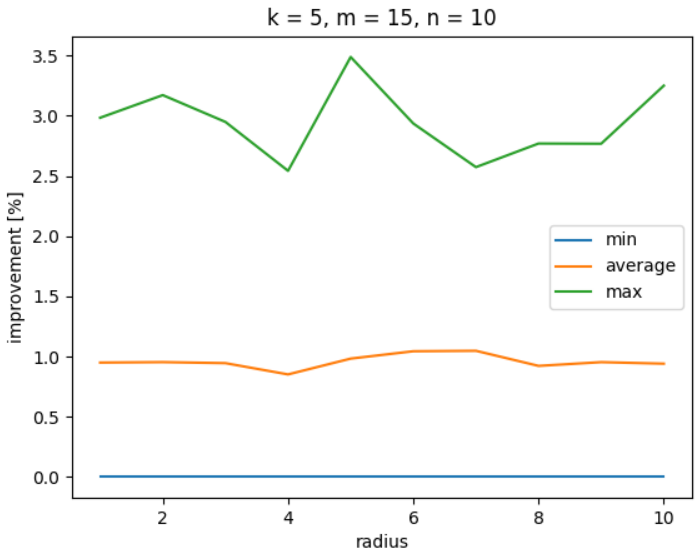



Analizując uzyskane wyniki, możemy stwierdzić, że uzyskana procentowa poprawa wyniku przestaje wzrastać dla liczby iteracji większej od 10. Pozwala to na wyciągnięcie wniosku, że wykonywanie zbyt wielu kroków algorytmu jest w przypadku naszego problemu nieopłacalne.

## 7.3 Badanie wpływu rozmiaru promienia otoczenia na poprawę wyniku

Następujący test bada wpływ rozmiaru promienia otoczenia na jakość uzyskanego rozwiązania. Pryzjęto m = 15, n = 10, k = 5, liczba iteracji = 50.

Rysunek 21: Test wpływu rozmiaru promienia otoczenia

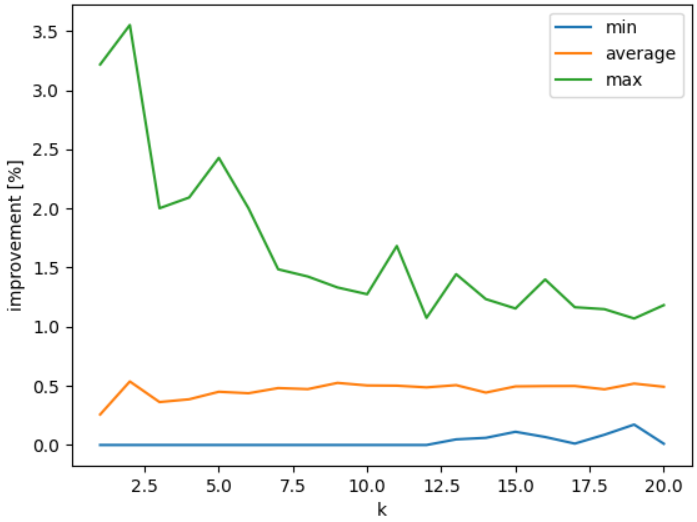



Na podstawie przedstawionego wykresu można stwierdzić, że rozmiar promienia otoczenia ma znikomy wpływ na uzyskane rozwiązanie.

## 7.4 Badanie wpływu ilości rozważanych umiejętności na poprawę wyniku

Ostatni test sprawdza, jaki wpływ na działanie algorytmu ma parametr zadania k - liczba rozpatrywanych umiejętności. Pozostałym zmiennym przypisano następujące wartości: n = m = 10, liczba iteracji = 50.

Rysunek 22: Test wpływu parametru k




Powyższy wykres wskazuje, że dla przyjętych pozostałych parametrów, ilość rozważanych umiejętności ma bardzo mały wpływ na poprawę wyniku.

# 8. Podsumowanie

Stworzony przez nas algorytm umożliwia znalezienie rozwiązania od kilku do kilkunastu procent lepszego od rozwiązania losowego (w zależności od wartości parametrów zadanego problemu oraz liczby iteracji), co biorąc pod uwage złożoność zadanego problemu jest zadowalającym rezultatem.
Parametrem zadania mającym największy wpływ na jakość wyniku końcowego jest n - liczba dostępnych stanowisk. Pozostale dwa prametry problemu po osiągnieciu dostatecznie dużej wartości przezstają mieć znaczenie.
Algorytm dość szybko przestaje poprawiać uzyskany w kolejnych krokach wynik, przez co ustawienie wysokiej liczby iteracji jest niopłacalne.

# 9. Bibliografia

[1] API with sample names. Available online at: `https://parseapi.back4app.com/classes/Listofnames_Complete_List_Names?limit=count&keys=Name`.

[2] API with sample skills. Available online at: `https://www.itsyourskills.com/purchase-skills-database`.