SPRAWOZDANIE

ALGORYTM GENETYCZNY – LABIRYNT

Maciej Chudziński 253748

Tematem mojego sprawozdania jest badanie algorytmu genetycznego oraz innych algorytmów rozwiązujących labirynt. Rozwiązanie zostało opracowane w języku Python oraz biblioteki Pyeasyga.

1. Dane wejściowe

Dane wejściowe czytane są z pliku tekstowego zawierającego labirynt w postaci kwadratu wypełnionym określonymi znakami oraz liczby kroków. Dane te odczytywane są za pomocą klasy MazeReader i przekształcane w tablicę stringów dla planszy labiryntu (board), oraz w zmienną steps dla liczby maksymalnych kroków.

Przykładowy input

2. Użyte algorytmy

- Algorytm genetyczny
- A*
- Przeszukiwanie wszerz (BFS)

3. Opis algorytmów

• Algorytm genetyczny:

Budowa chromosomu: każdy chromosom składa się z bitów o długości

2* [maksymalna ilość kroków]. Każda para bitów określa kierunek, w którym następuje przemieszczenie o jeden krok.

Działanie funkcji fitness_v1: Każdy chromosom zaczyna od pozycji początkowej oznaczonej jako „S” oraz otrzymuje wynik początkowy równy 0. Następnie rozpoczyna się iteracja po chromosomie, zmieniająca jego pozycję. Jeśli w danym kierunku występuje miejsce niedozwolone, oznaczone jako „#” to pozycja się nie zmienia. Jeśli chromosom dotarł do pozycji końcowej oznaczonej symbolem „E”, to zwraca wynik 0, jeśli nie, to 0- odległość od końca.

```python
def fitness_v1(individual, maze: MazeGA):

    score = 0
    maze.current_position = maze.start_position.copy()
    for i in range(int(len(individual)/2) - 1):
        maze.move(individual[2*i], individual[2*i+1], maze.current_position)
        if maze.get_distance_to_end(maze.current_position[0], maze.current_position[1]) == 0:
            return 0
    score -= maze.get_distance_to_end(maze.current_position[0], maze.current_position[1])

    return score
```

Działanie funkcji fitness_v2: Funkcja f2 różni się od f1 tym, że dodatkowo zapamiętywane są poprzednie kroki. Gdy chromosom ma zamiar poruszyć się w kierunku pozycji, na której już wcześniej był, nie zmienia on pozycji oraz otrzymuje ujemny punkt.

```python
def fitness_v2(individual, maze: MazeGA):

    score = 0
    prev_pos = []
    maze.current_position = maze.start_position.copy()
    prev_pos.append(maze.current_position.copy())

    for i in range(int(len(individual)/2) - 1):
        inc_pos = maze.current_position.copy()
        maze.move(individual[2*i], individual[2*i+1], inc_pos)
        is_in = False
        for pos in prev_pos:
            if pos == inc_pos:
                score -= PENALTY
                is_in = True
                break
        if is_in:
            continue
        prev_pos.append(inc_pos.copy())
        maze.current_position = inc_pos

        if maze.get_distance_to_end(maze.current_position[0], maze.current_position[1]) == 0:
            return 0

    score -= maze.get_distance_to_end(maze.current_position[0], maze.current_position[1])

    return score
```

Maksymalnym wynikiem funkcji fitness w obu przypadkach jest 0, równoważne z dotarciem do celu.

Ruch w obu funkcji opiera się o funkcję move, która czyta z każdej pary bitów chromosomu kierunek, w którym następuje przemieszczenie.

Działanie własnej funkcji krzyżującej: w funkcji została dodana większa szansa (70%) na krzyżowanie w pierwszej połowie chromosomu.

```
def my_crossover(parent_1, parent_2):

    rand_cross_at_start = random.randrange(1, 10)
    if rand_cross_at_start < 7:
        index = random.randrange(1, int(len(parent_1)/2))
    else:
        index = random.randrange(int(len(parent_1)/2), len(parent_1))
    child_1 = parent_1[:index] + parent_2[index:]
    child_2 = parent_2[:index] + parent_1[index:]
    return child_1, child_2
```

- A*

  Algorytm A* przyjmuje jako input tablicę stringów, z której wyciąga wszystkie wolne pola (spacje), oraz pozycje startową oraz końcową. Z pozyskanych danych tworzy obiekty klasy ANode, która przetrzymuje pozycję pola, flagę czy zostało ono odwiedzone, referencję do rodzica (wstępnie nieustaloną), listę obiektów sąsiednich pozycji oraz wartości funkcji f, g, h, które ustalane są podczas działania głównej funkcji algorytmu.

  Funkcja poszukiwania ścieżki działa na zasadzie wrzucania kolejnych ANode'ów do listy, które są nieodwiedzonymi sąsiadami pozycji z najmniejszym wynikiem funkcji f, zaczynając od początkowej pozycji. Gdy sąsiedzi danej pozycji zostaną dodani do kolejki, ona sama zostaje z niej usunięta, oraz oflagowana jako odwiedzona. Funkcja kończy działanie, gdy lista jest pusta, lub gdy w niej znalazła się pozycja końcowa.

```
def run(self):

    while len(self.queue) > 0 and self.end_node.visited is not True:
        best_choice = min(self.queue, key=attrgetter('f'))
        best_choice.visited = True
        self.queue.remove(best_choice)
        for n in best_choice.neighbors:
            if not n.visited:
                self.queue.append(n)
                n.parent = best_choice
                n.count_functions(self.end_node)

    self.print_visited()
```

- BFS

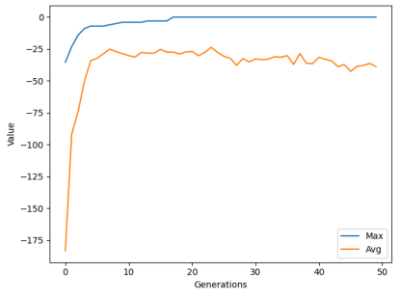
Algorytm przeszukiwania wszerz przyjmuje takie same dane jak A*. Każda dostępna pozycja posiada obiekt klasy Node.

Sam algorytm działa na zasadzie wrzucania kolejnych nieodwiedzonych sąsiadów do listy. Na początku wrzucony do niej zostaje obiekt pozycji początkowej. Funkcja zdejmuje ostatni obiekt, oraz flaguje go jako odwiedzony, następnie dokłada jego sąsiadów. Algorytm kończy działanie, gdy lista jest pusta, lub gdy w niej znalazła się pozycja końcowa.

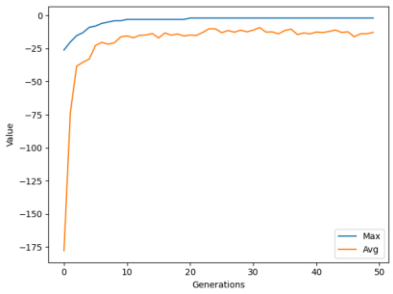
```
def run(self):

    start = time_ns()
    while len(self.queue) != 0:
        if self.end_node in self.queue:
            self.end_node.visited = True
            break

        else:
            popped = self.queue.pop(0)
            popped.visited = True
            for p in popped.neighbors:
                if not p.visited:
                    self.queue.append(p)
    print(f'\nin {(time_ns() - start) / 1000000}ms')
    self.print_visited()
```

4. Optymalizacja algorytmów

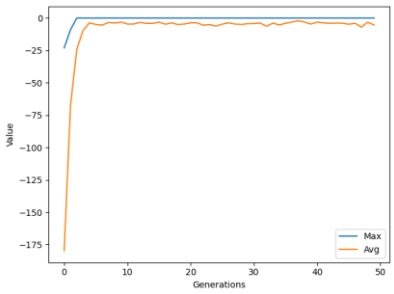
4.1. Wykresy dla przypadku z zajęć, gdzie wszystkie dotarły do celu, dla różnych mutacji:



Mutation probability= 1



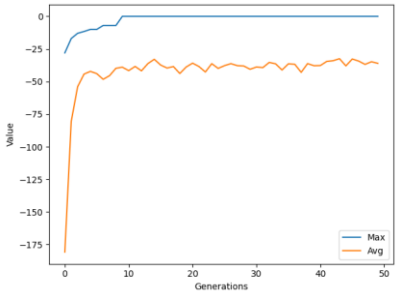
Mutation probability= 0.5



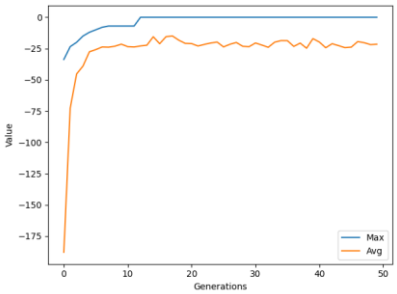
Mutation probability= 0.1

Dla wszystkich przypadków populacja wynosiła 500, z elityzmem oraz własnym krzyżowaniem i przy użyciu fitness_v2.

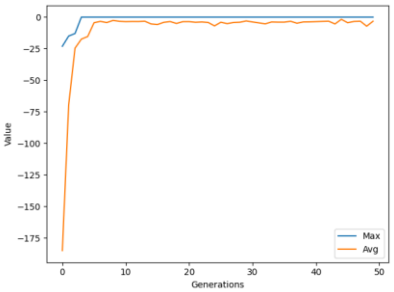
4.2. Ten sam przypadek dla domyślnej funkcji krzyżowania:



Mutation probability= 1

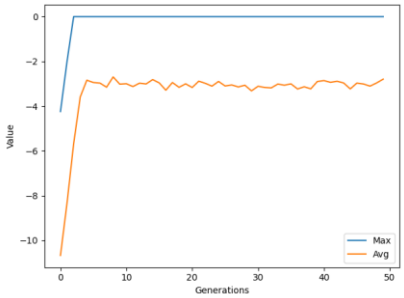


Mutation probability= 0.5

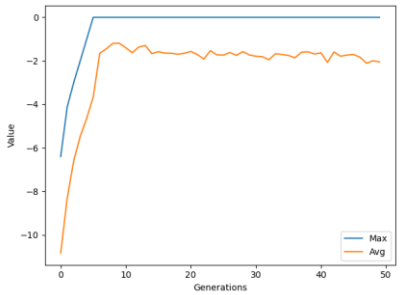


Mutation probability= 0.1

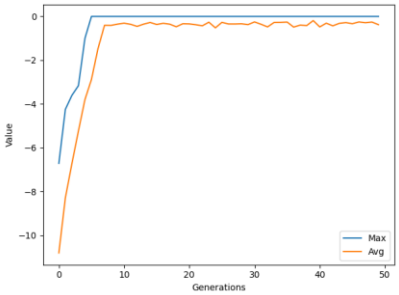
4.3. Dla fitness_v1 z własnym krzyżowaniem, elityzmem oraz populacją 500:



Mutation probability= 1

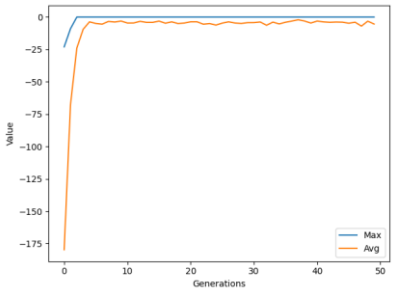


Mutation probability= 0.5

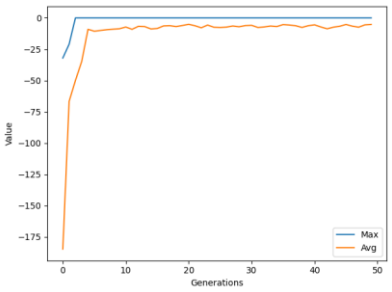


Mutation probability= 0.1

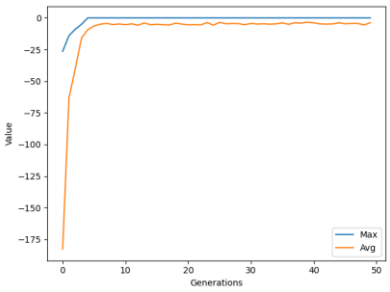
4.4. Dla fitness_v2 z własnym krzyżowaniem, elityzmem, stałą mutacją = 0.1 i zmienną populacją:



Population = 500



Population = 1000



Population = 2000

4.5. Pomiar czasów:

Populacja: 5000, elityzm: tak, mutacja: 10%, fitness_v2				
Input	m2: 8x8 22 kroki	m1: 9x9 20 kroków	m3: 12x12 40 kroków	m4: 22x22 100 kroków
Czas	4784.2799ms	212.5488ms	2080.6816ms	5798.8702ms
Generacje	4	1	2	3
Dla A* oraz BFS czas dla wszystkich przypadków wynosił ~1ms				

5. Wnioski z badań

- Fitness_v1 posiada niską skuteczność w przypadku, gdy pozycje startu oraz końca znajdują się w małej odległości, rozdzielone przeszkodą znacznie wydłużającą ścieżkę pomiędzy nimi, natomiast w przypadku dużej odległości i w miarę prostej ścieżki algorytm nie ma problemu z dotarciem.
- Fitness_v2 radzi sobie najlepiej w każdym przypadku labiryntu, jednak nie daje 100% szansy na dotarcie do celu.
- Algorytmy genetyczne radzą sobie dużo lepiej, jeśli mają większą ilość maksymalnych kroków, oraz większą populację.
- BFS oraz A* są lepsze do rozwiązywania labiryntów. Algorytmy zapewniają zdecydowanie krótszy czas oraz 100% skuteczność.