Generowanie labiryntów przy użyciu algorytmu genetycznego

Bartłomiej Wujec

26 kwietnia 2022

1 Wstęp

Labirynt jest łamigłówką logiczną, w której celem jest przejście od punktu A do punktu B, nie przechodząc tym samym przez ściany labiryntu. Z pomocą algorytmu genetycznego, oraz innych poznanych technik postaram się generować poprawne labirynty, ocenić jakość labiryntów, a także ocenić wydajność i efekty pracy algorytmów .

2 Algorytm genetyczny

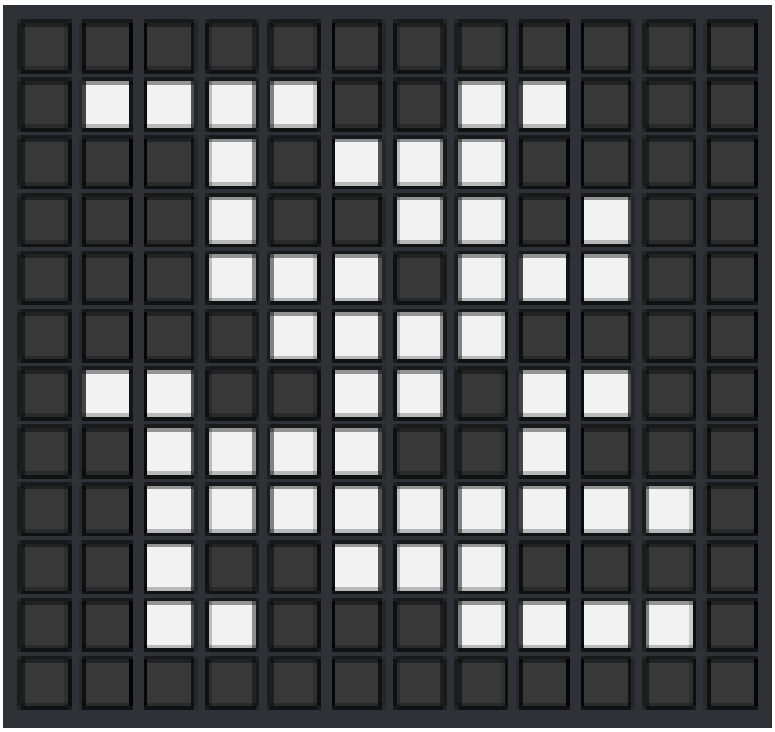
2.1 Funkcja fitness

```
def fitness(solution, solution_idx):
    score = 0
    if solution[0] == 0 and solution[maze_side**2-1] == 0:
        score += (maze_side**2)/2

    solvable, traversed = maze_solver(solution)

    if solvable:
        score += (maze_side**2)/2

    for i in range(0, maze_side**2):
        if solution[i] == 0:
            if sum_neighbours(i, solution) in (0, 1, 8):
                score += -1
            if possible_moves(i, solution) == 0:
                score += -2
        if solution[i] == 1:
            if sum_neighbours(i, solution) == 8:
                score += -1

    empty_cells = int(maze_side**2 - sum(solution.tolist())
  )
    not_accessible = traversed-empty_cells
    score += not_accessible

    return score
```

Propozycja funkcji fitness. Funkcja nagradza za „wolne" rogi wyznaczające start i koniec labiryntu. Kolejnym elementem który nagradza jest możliwość przejścia labiryntu. Ponad to karze za komórki otoczone z wszystkich stron ścianami oraz komórki które nie mają żadnej lub tylko jedną ściane w swoim sąsiedztwie. Funkcja negatywnie punktuje za każdą komórkę do której nie można się dostać. Ma to na celu ograniczenie ilości komórek i ich systemów które nie są połączone z resztą labiryntu.

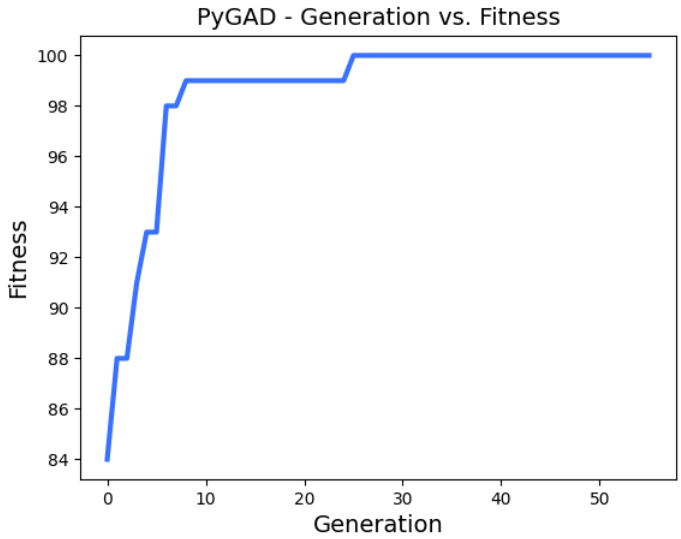
2.2 Wyniki generowania labiryntów

2.2.1 Labirynt 10x10



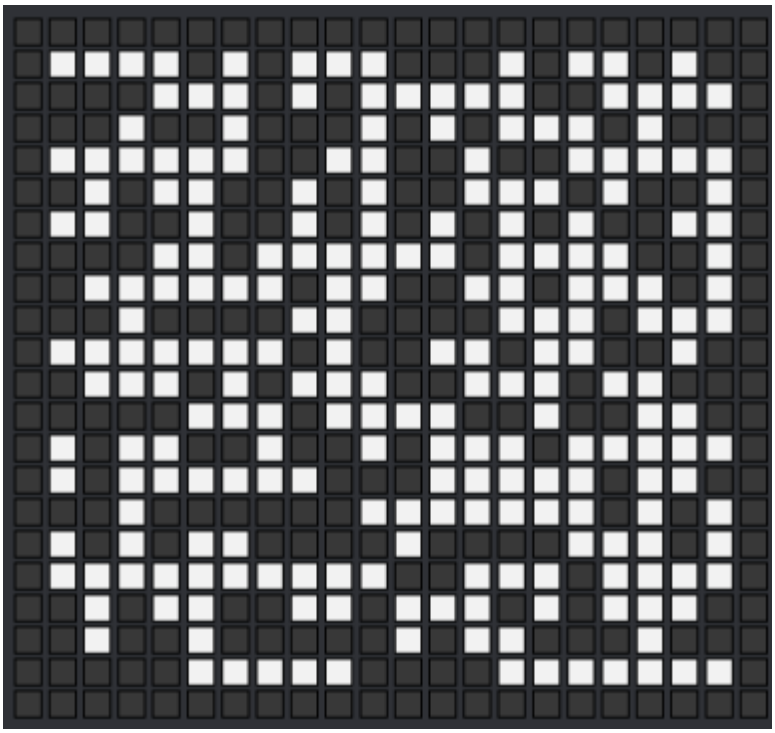
Algorytm genetyczny generuje labirynt o maksymalnej ocenie w średnio 61 pokoleniach, przy polulacji wielkości 100, oraz w czasie średnio 6 sekund.

Pokoleń(średnia)	61
Populacja	100
Rozmnażających się rodziców	10
Pozostających rodziców	5
Procent mutacji	1
Czas	5,5 sek



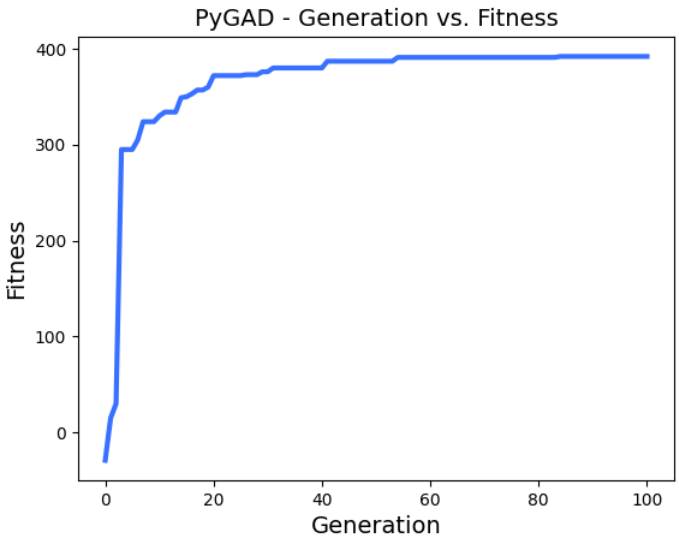
Rysunek 1: Wykres fitness

2.2.2 Labirynt 20x20



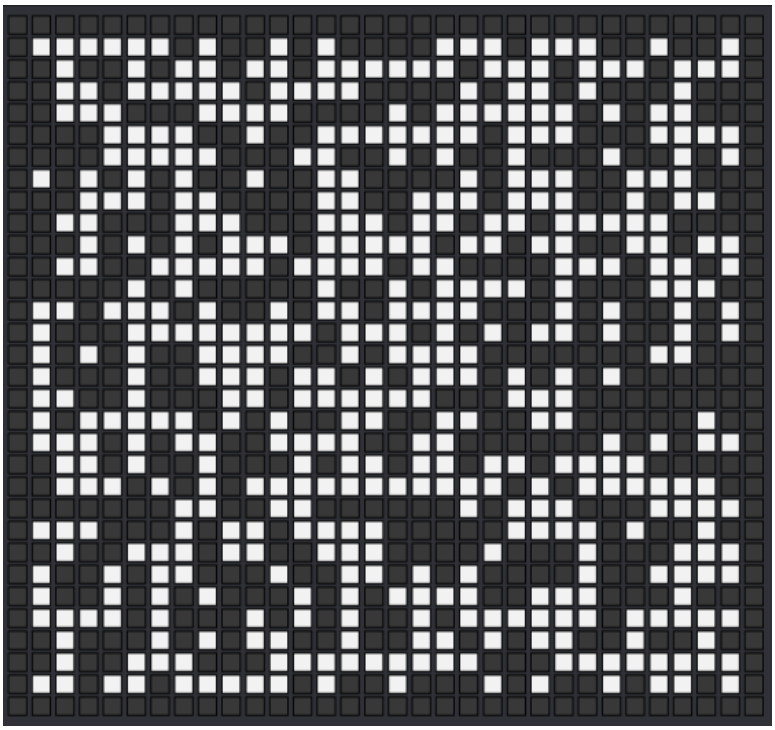
Algorytm genetyczny generuje labirynt w średnio 99 pokoleniach, przy polulacji wielkości 100, oraz w czasie 49 sekund. Stagnacja wyniku nie wyspępuje, lub występuje przez mniej niż 50 pokoleń. Możemy zauważyć, że labirynt zawiera obszary do których nie można się dostać nie łamiąc zasad, ale sam labirynt jest poprawny i posiada rozwiązanie.

Pokoleń(średnia)	99
Populacja	100
Rozmnażających się rodziców	10
Pozostających rodziców	5
Procent mutacji	1
Czas(średnia)	28,5 sek



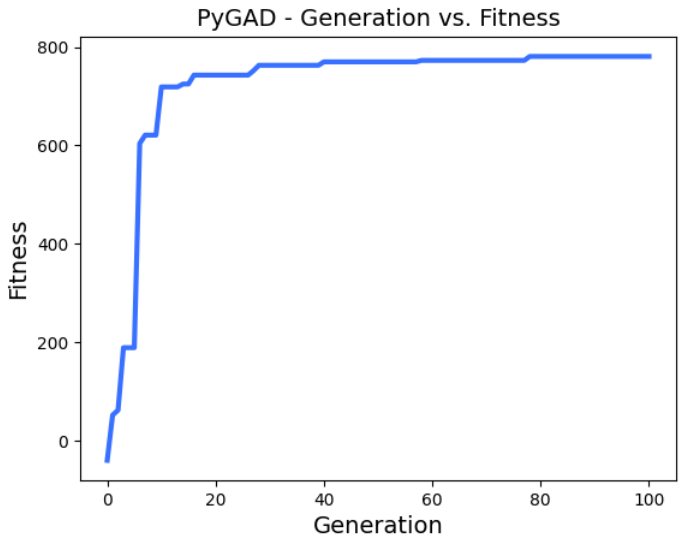
Rysunek 2: Wykres fitness

2.2.3 Labirynt 30x30



Algorytm genetyczny generuje labirynt w 100 pokoleniach, przy polulacji wielkości 100, oraz w średnim czasie 63 sekund. Stagnacja nie występuje, lub występuje przez mniej niż 50 pokoleń. Możemy zauważyć, że labirynt posiada wiele obszarów do których nie można się dostać nie łamiąc zasad, ale sam labirynt jest poprawny i posiada nietrywialne rozwiązanie.

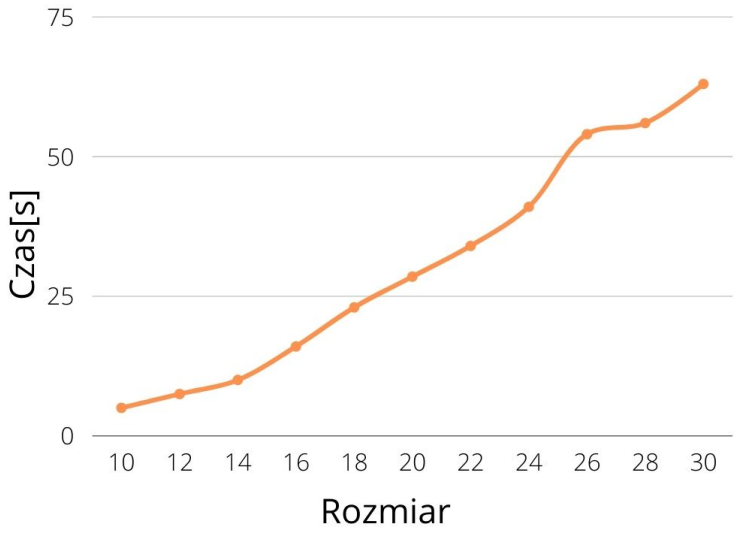
Pokoleń	100
Populacja	100
Rozmnażających się rodziców	10
Pozostających rodziców	5
Procent mutacji	1
Czas	63 sek



Rysunek 3: Wykres fitness

Jak możemy zobaczyć na wykresie, czas potrzebny na wygenerowanie labiryntu rośnie w sposób bliski liniowemu.

2.3 Rozmiar a czas generowania

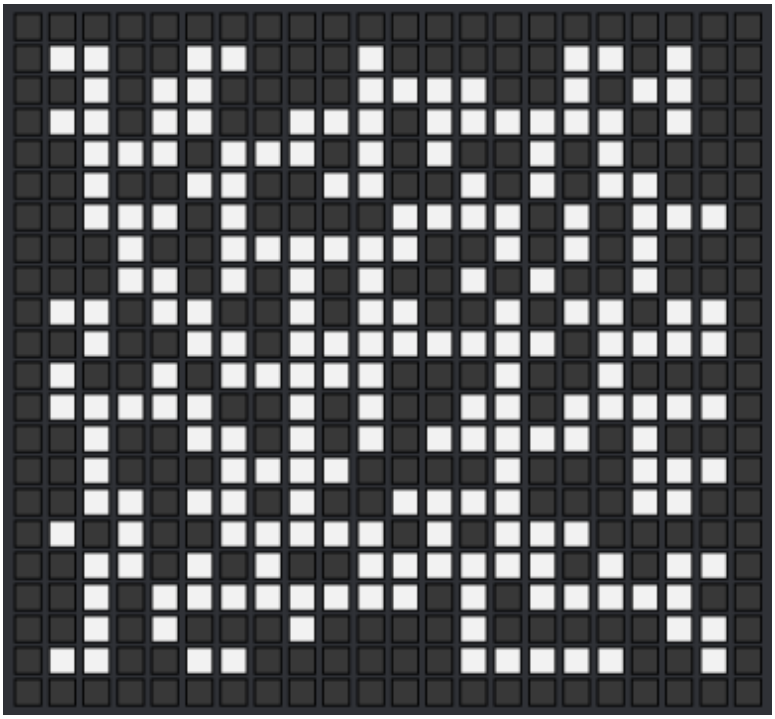


2.4 Alternatywne fukcje fitness i ich wyniki

2.4.1 Kary za mała i duża ilość ścian w są siedztwie

```
def fitness(solution, solution_idx):
    score = 0
    if solution[0] == 0 and solution[maze_side**2-1] == 0:
        score += (maze_side**2)/2

    solvable, traversed = maze_solver(solution.tolist())

    if solvable:
        score += (maze_side**2)/2

    for i in range(0, maze_side**2):
        if solution[i] == 0:
            if sum_neighbours(i, solution) in (0, 1, 2, 7,
  8):
                score += -2
            if possible_moves(i, solution) == 0:
                score += -2
        if solution[i] == 1:
            if sum_neighbours(i, solution) > 4:
                score += -1

    empty_cells = int(maze_side**2 - sum(solution.tolist())
  )
```

```
    not_accessible = traversed-empty_cells
    score += not_accessible
    return score
```



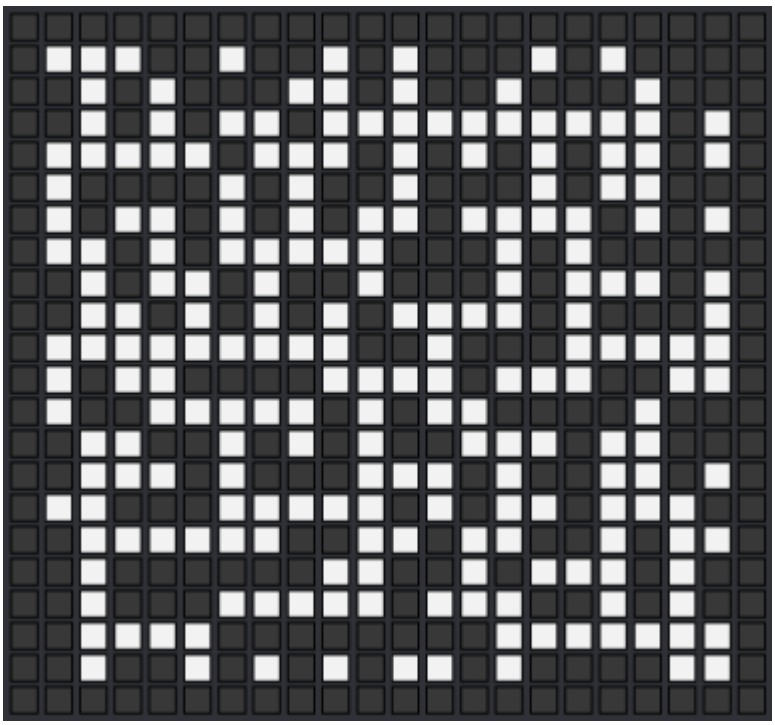
Rysunek 4: Przykładowy labirynt generowany przez alternatywny GA

Zmniejszenie progu karania za za mała i za dużą ilość ścian w są siedztwie powoduje, że „sale" praktycznie nie występują, a labirynt składa się głównie z samych korytarzy.

2.4.2 Kara za mało poziomych ścian

```
def fitness(solution, solution_idx):
    score = 0
    if solution[0] == 0 and solution[maze_side**2-1] == 0:
        score += (maze_side**2)/2

    solvable, traversed = maze_solver(solution.tolist())

    if solvable:
        score += (maze_side**2)/2

    for i in range(0, maze_side ** 2):
        if solution[i] == 0:
            if sum_neighbours(i, solution) in (0, 1, 2, 7,
  8):
                score += -2
            if possible_moves(i, solution) == 0:
                score += -2
        if solution[i] == 1:
```

```
            if sum_neighbours(i, solution) > 4:
                score += -1
    row_sum = 0

    for i in range(0, maze_side**2):
        row_sum += solution[i]
        if (i + 1) % maze_side == 0:
            if row_sum > maze_side/1.5:
                score += -8

    empty_cells = int(maze_side**2 - sum(solution.tolist())
)
    not_accessible = traversed-empty_cells
    score += not_accessible
    return score
```



Rysunek 5: Przykładowy labirynt generowany przez alternatywny GA

Kary za mało poziomych ścian powinny produkować labirynty z wieloma poziomymi scianami. Funkcja słabo radzi sobie z generowaniem poziomych tuneli

3 Generowanie labiryntu za pomocą PSO

3.1 Technika i parametry

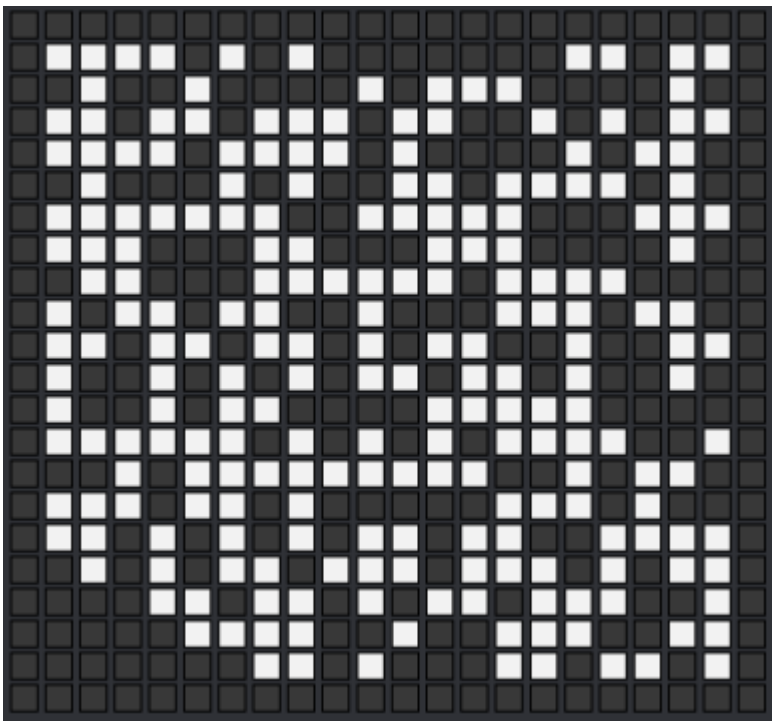
Labirynt jest genereowany z pomocą binary PSO z wartościami w zakresie systemu binarnego.

Rysunek 6: Parametry optymalizacyje PSO

c1	c2	w	k	p	n_particles	iters
0.5	0.3	0.9	2	1	30	1000

```
def f(x):
    num_of_particles = x.shape[0]
    j = [fitness(x[i]) for i in range(0, num_of_particles)]
    return numpy.array(j)


optimiser = BinaryPSO(30,
                      maze_side**2,
                      options=options,
                      velocity_clamp=(-6, 6))

best, solution = optimiser.optimize(f, iters=1000, verbose=
   True)
```

3.2 Wyniki



Rysunek 7: Labirynt wygenerowany przez binary PSO. 20x20

Labirynty generowane przez PSO wypadają gorzej niż te generowane przez GA. Często generują labirynty które mają więcej niedostępnych pół niż ich odpowiednicy generowani przez GA. Średni czas generowania to 82 sekundy dla labiryntu o ścianie wielkości 20, tym samym czasowo wypada gorzej niż algorytm genetyczny.

4 Podsumowanie

Generowanie labiryntów z pomocą algorytmu genetycznego i optymalizacji cząsteczkowej jest możliwa i oferuje akceptowalne skutki. Czas jaki jest wymagany aby wygenerować labirynt wypada gorzej niż typowe metody(np. backtracking) które generują poprawe labirynty szybciej i bez błedów lub niedostępych obszarów.