Zadanie: Rozwiązywanie labiryntu przy użyciu języka R i algorytmów genetycznych
(pakiet „genalg”).

Problem 2: Labirynt
Mamy labirynt 12x12 pól, przedstawiony na rysunku. Przez labirynt poruszamy się przesuwając z pola na pole (ruch: w lewo, prawo, do góry lub na dół). Nie możemy wchodzić na ściany (czarne pola). Naszym celem jest dojście z pola S, do pola E w maksymalnie 40 krokach. Czy istnieje taka droga?

I. Generowanie labiryntów

Na potrzeby projektu sformułowałem skrypt generaujący losowe labirynty o podanych wymiarach. Ponieważ skrypt nie weryfikuje poprawności, część labiryntów może być nierozwiązywalna i wymaga ręcznej korekty.

Skrypt pozwala także na wizualizację labiryntu przy pomocy pakietu „pheatmap” oraz zapisanie go automatycznie do pliku .pdf.

Skrypt do generowania znajduje się w pliku **„labyrinth_generator.R”**

II. Rozwiązywanie labiryntu przy pomocy algorytmów genetycznych.

1. Przedstawienie labiryntu jako matrycy

W celu znalezienia rozwiązania labirynt należy przedstawić w postaci kwadratowej matrycy zawierającej dane typu boolean (0,1). Labirynt otoczony jest granicami składającymi się wyłącznie z 1.

W skrypcie „labyrinth_genalg.R” wykorzystane są 3 sposoby wczytywania matrycy na podstawie zadanych danych
- uzupełnienie ramowego labiryntu wektorami przedstawiającymi poszczególne wiersze
- stworzenie matrycy dla zadanego wektora
- zmiana wymiarów wektora przy użyciu metody dim()

2. Algorytm genetyczny z pakietem „genalg”

2.1. Metoda rbga.bin

Na potrzeby projektu pierwotnie algorytm miał bazować na binarnych wektorach o długości równej podwojonej ilości kroków, zaś każdy ruch miał być wyrażony w postaci dwóch bitów (00,01,10,11).

Po zbadaniu problemu zdecydowałem się na odejście od tej metody i zastosowanie rbga przyjmującego wektor float.

2.2. Metoda rbga przyjmująca wektor float

W celu znalezienia rozwiązania przyjęto, że każdy przypadek przedstawiony jest jako wektor float o długości równej ilości kroków, przyjmujący wartości z przedziału <0.51 , 4.49>.

Dzięki temu rozwiązaniu każdy gen chromosomu koduje pełen krok, jednak funkcja fitness musi zaokrąglać każdy gen do liczy całkowitej, czyli {1,2,3,4}, co oznacza odpowiednio:

#step up - „1”
#step down - int „2”
#step right - int „3”
#step left - int „4”

Na potrzeby zadania przyjęto, że ilość kroków = rozmiar labiryntu * 4

Funkcja rbga pakietu genalg wymaga zdefiniowania wektorów zawierających minimalne i maksymalne wartości dla każdego chromosomu.
vMin <- rep(0.51, lab_size * 4); vMax <- rep(4.49, lab_size * 4)

Funkcja fitness ma postać:

```
evaluate3 <- function(string=c()) {

  x <- 2;
  y <- 2;
  exit_coordinates_sum <- lab_size * 2 + 2;
  returnVal3 <- exit_coordinates_sum;

  for(step in 1:length(string)) { if (x==lab_size + 1 & y==lab_size + 1) { returnVal2 = 0; break
  } else if (round(string[step],0)==1 & labyrinth[x-1,y]==0) { x <- (x - 1)
  } else if (round(string[step],0)==2 & labyrinth[x+1,y]==0) { x <- (x + 1)
  } else if (round(string[step],0)==3 & labyrinth[x,y+1]==0) { y <- (y + 1)
  } else if (round(string[step],0)==4 & labyrinth[x,y-1]==0) { y <- (y - 1)
  } else next
  }
  returnVal3 <- (exit_coordinates_sum - x - y);
  returnVal3
}
```

Funkcja dla każdego chromosomu przyjmuje początkowe koordynaty (pozycja „Tezeusza”) x,y i dla każdego kroku zaokrągla dany gen przyjętego chromosomu weryfikuje dany (pętla o ilości iteracji (1 , ilość kroków)

- czy osiągnięto cel (exit) czyli koordynaty x = y = (rozmiar labiryntu -1)
  => funkcja zwraca wartość 0 i przerywa pętlę
- czy dany krok jest możliwy do wykonania
  => zmienia daną koordynatę (x,y) zgodnie z wykonanym krokiem

Po wykonaniu pętli funkcja zwraca wartość z przedziału (0, 18), czyli sumę różnicy koordynatów x,y pozycji „Tezeusza” i wyjścia.

Metoda rbga ma postać:

```
genalg_labyrinth <- rbga(stringMin=vMin, stringMax=vMax,
```

```
suggestions=NULL,
popSize=200, iters=200,
mutationChance=0.1,
elitism=T,
monitorFunc=NULL, evalFunc=evaluate3,
showSettings=FALSE, verbose=FALSE)
```

Przyjęto szansę mutacji jako 10%, z uwagi na losowość labiryntów jest zasadne uwzględnienie tego aspektu w propozycjach rozwiązań.

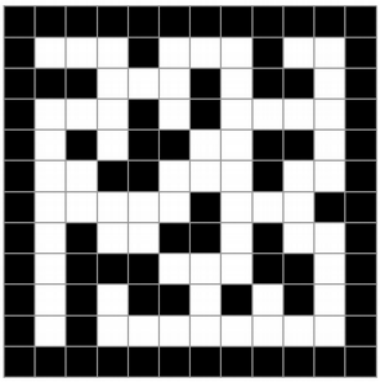
Całość kodu wraz z opisem kroków zawarta jest w pliku **„labyrinth_genalg.R”**

III. Labirynty

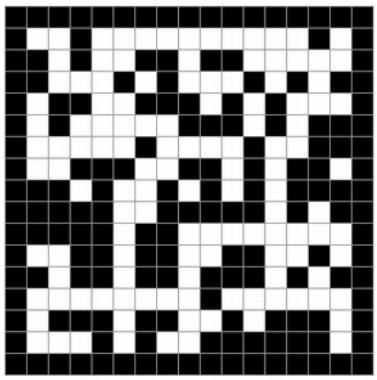
Algorytm zastosowano dla labiryntów o 3 rozmiarach: 10x10, 15x15 (całkowity rozmiar labiryntu jest większy o 2 ze względu na dodanie granic, tj 12x12 i 17x17).

Labirynt 10x10 przyjęto z zadania, pozostałe 2 zostały wygenerowane losowo i skorygowane tak, aby posiadały rozwiązanie.

Rys. 1 Labirynt 10x10

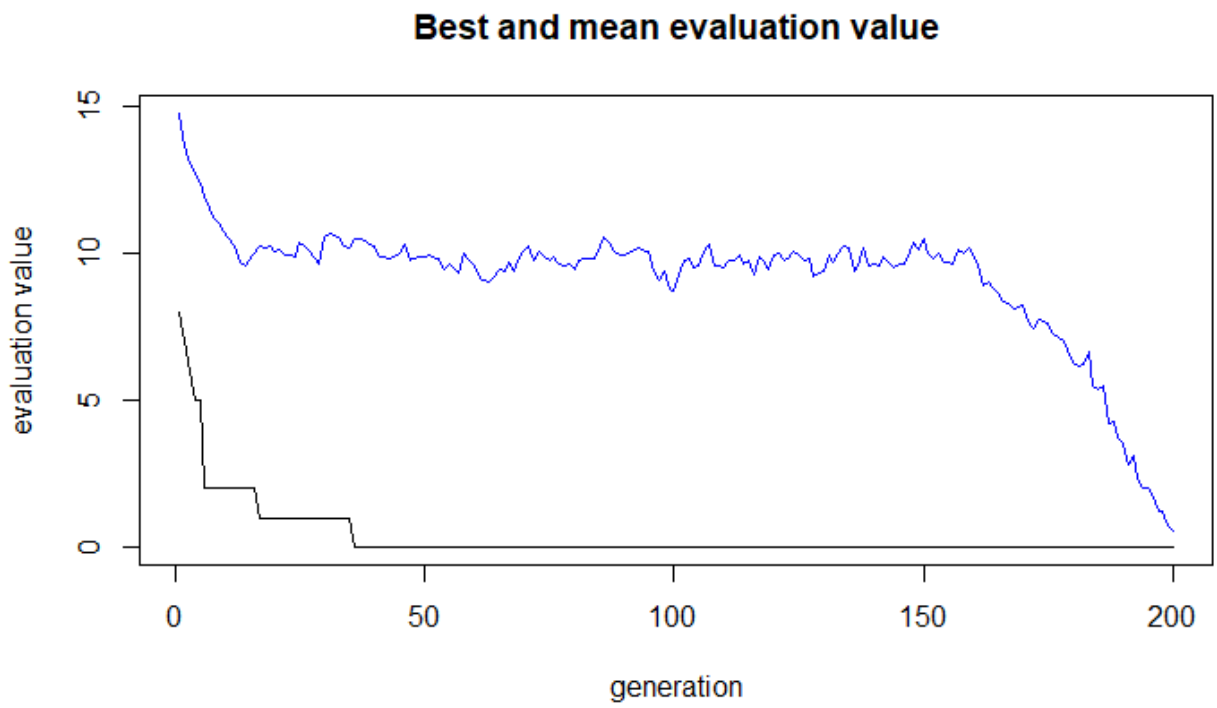


Rys.2 Labirynt 15x15



IV. Wyniki zastosowania algorytmu

1. Labirynt 10x10

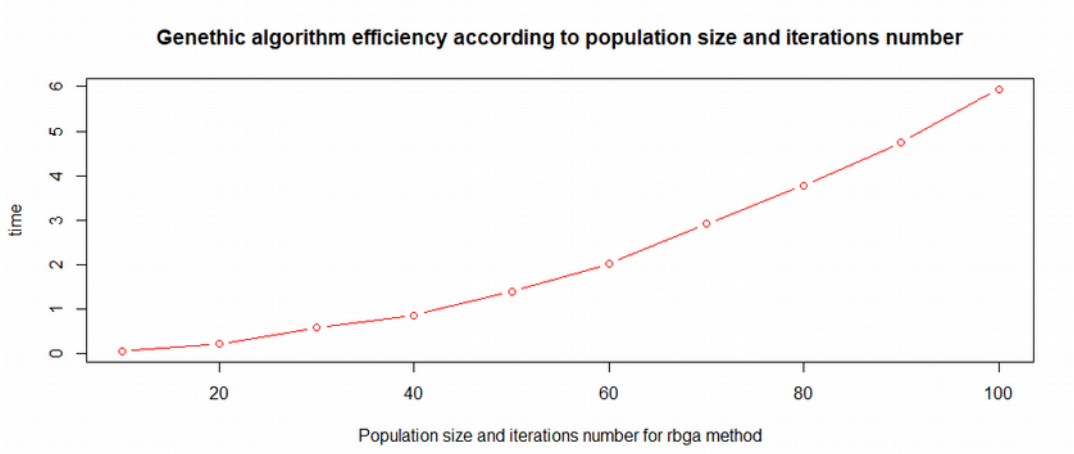


Funkcja system.time pokazuje, że przy założonej populacji 200 równej ilości iteracji czas na wykonanie algorytmu wynosi 24.68 sekundy. Najlepsze rozwiązanie zostało znalezione już w 35 pokoleniu

Analizując wykorzystanie zasobów w zależności od wielkości populacji uzyskujemy wyniki:

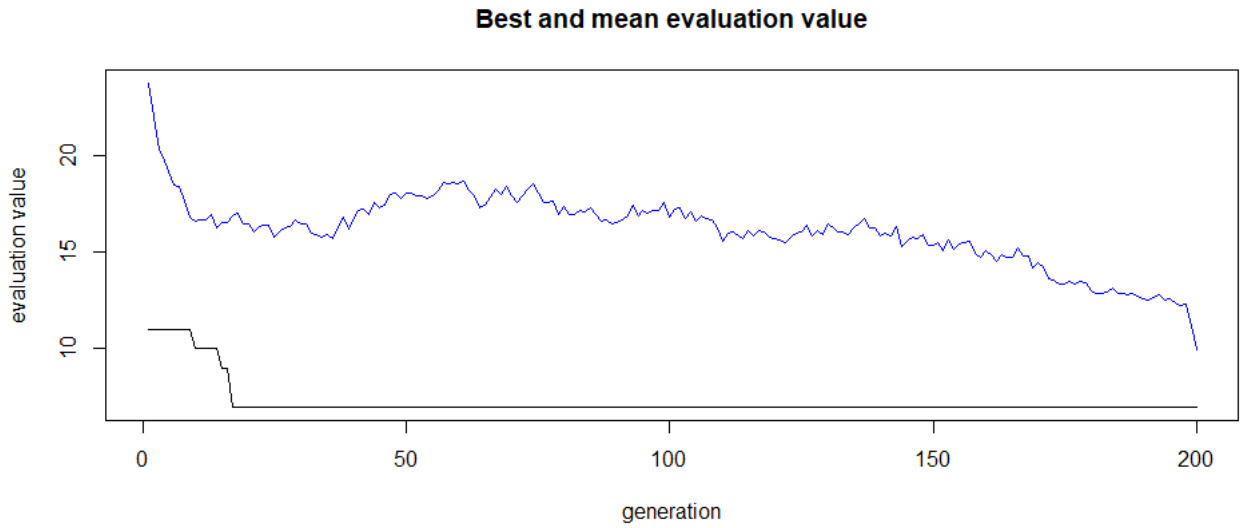
```
      pop&iter   user    system time
 [1,]       10   0.06        0.06
 [2,]       20   0.22        0.22
 [3,]       30   0.56        0.58
 [4,]       40   0.87        0.87
 [5,]       50   1.39        1.39
 [6,]       60   2.04        2.03
 [7,]       70   2.92        2.92
 [8,]       80   3.76        3.78
 [9,]       90   4.75        4.75
[10,]      100   5.92        5.94
```

Wykres:



Jak wynika z wykresu wykorzystanie zasobów w zależności od wielkości populacji i ilości iteracji wzrasta prawie liniowo.

2. Labirynt 15x15



Matryca z wynikami system.time w zależności od wielkości populacji i ilości iteracji:

	pop&iter	user	system	time
[1,]	15	0.19	0.00	0.19
[2,]	30	0.81	0.00	0.84
[3,]	45	1.78	0.00	1.78
[4,]	60	3.07	0.00	3.08
[5,]	75	4.94	0.00	4.94
[6,]	90	7.15	0.00	7.16
[7,]	105	10.17	0.00	10.18
[8,]	120	13.94	0.02	14.04
[9,]	135	17.07	0.00	17.12
[10,]	150	25.72	0.20	83.92
[11,]	165	25.25	0.00	25.61
[12,]	180	28.48	0.00	28.57
[13,]	195	33.00	0.02	33.04
[14,]	210	38.50	0.00	38.53
[15,]	225	45.25	0.03	45.54

Wykres tej zależności: