[ALHE 21L] Wykorzystanie algorytmów ewolucyjnych / genetycznych do pomocy królikowi wielkanocnemu w optymalnym znajdowaniu jajek wielkanocnych

Dokumentacja końcowa

Projekt wykonywany w ramach przedmiotu ALHE
Autorzy: Iwo Sokal (300255), Paweł Martyniuk (300220)

Treść

Zadanie polega na znalezieniu optymalnej strategii dla królika, aby efektywnie znajdował jajka wielkanocne oraz nie uderzył w ścianę. Zakładamy siatkę 2D (N x N wymiarową), każde pole może mieć jeden z trzech stanów: być puste, może znajdować się na nim jajko, może być ścianą. Królik może poruszyć się góra / lewo / dół / prawo, poruszyć się losowo, podnieść jajko, stać w miejscu (7 akcji).

Założenia

- Pole może być puste, zawierać jajko lub być ścianą.
- Królik może poruszyć się góra / lewo / dół / prawo, poruszyć się losowo, podnieść jajko, stać w miejscu.
- Osobnikiem będzie sekwencja ruchów, składająca się z ciągu akcji o stałej, określonej długości. Dzięki temu będzie można zrobić łatwiej operacje krzyżowania.
- Szukamy optymalnej szybkości znajdowania jajek, nie szukamy rozwiązania dla znajdowania wszystkich jajek.
- Komenda stój będzie używana w celu wyrównania długości wszystkich osobników.

Środowisko

Aplikacja została napisana w systemie operacyjnym Windows w środowisku QT Creator z wykorzystaniem C++. Używamy QT 5 zamiast interfejsu konsolowego, ponieważ w ten sposób lepiej jest zobrazowany wynik działania algorytmu ewolucyjnego, a sama aplikacja jest bardziej przyjazna w użytkowaniu oraz ładniejsza.

Sposób rozwiązania

Chcemy zastosować algorytm ewolucyjny. Osobnikiem będzie sekwencja ruchów (np. ← → ? ↑ ↓ O X). Staramy się znaleźć taką sekwencję ruchów, która pozwoli królikowi jak najszybciej znajdować jajka. Program będzie znajdował rozwiązania dla zadanej liczby ruchów.
Dostępne ruchy:

- → - idź w prawo (RIGHT)
- ← - idź w lewo (LEFT)
- ↑ - idź w górę (UP)
- ↓ - idź w dół (DOWN)
- ? - idź w prawo/lewo/górę/dół (RANDOM)
- O - podnieś jajko (TAKE)
- X - stój (STOP)

Nasz algorytm działa następująco:

- Inicjalizacja - losowa
- Ocena przystosowania - *ilość zebranych jajek / ilość ruchów do momentu ich zgromadzenia*, im jej wartość większa tym lepiej. Każda wykonana akcja zwiększa licznik ruchów o 1, chyba że liczba zebranych jaj będzie równa łącznej liczbie jaj na planszy to wtedy akcja STOP nie będzie zwiększała tego licznika, dzięki czemu nie jakość osobnika nie będzie wtedy osłabiana.
- Selekcja - turniejowa (jest zaimplementowana również selekcja losowa, jednak daje zdecydowanie gorsze wyniki więc nie ma sensu jej rozważać w naszym algorytmie).
- Krzyżowanie - jednopunktowe z jednakowym punktem krzyżowania dla wszystkich osobników.
- Mutacja - zmiana losowego genu na jeden z pozostałych 6.
- Sukcesja - generacyjna.

W celu badania jakości zaimplementowanego rozwiązania będziemy symulować ruch królika po planszy, aby ustalić ile jajek zdobył i w ilu ruchach.

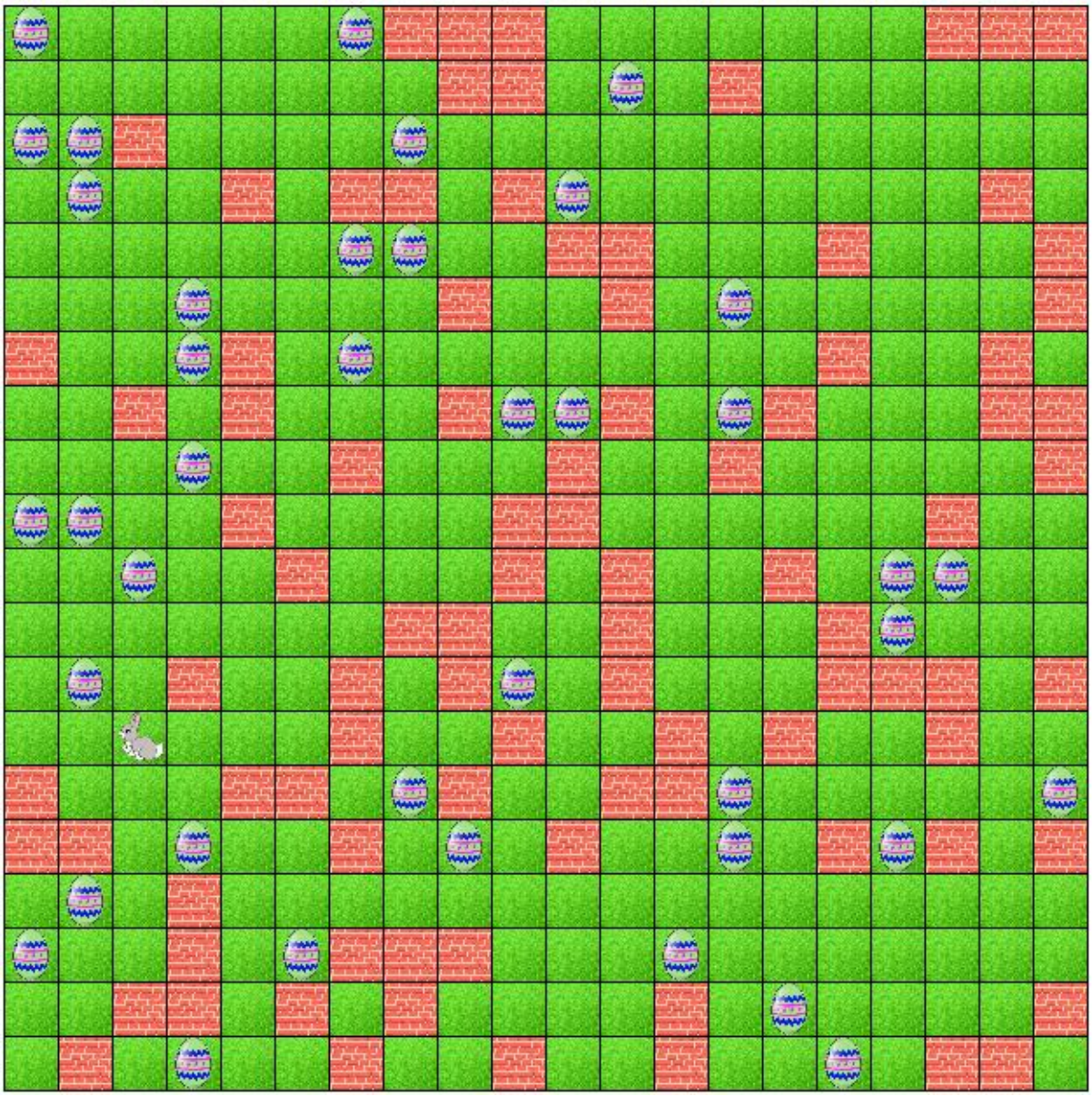
Szczegóły implementacyjne

- Board - przechowuje planszę po której porusza się królik, zrealizowany jako singleton.
- Field - przechowuje pojedyncze pole. Każde pole wchodzi w skład całej planszy.
- FieldType - określa typ pola. Pole może być typu:
    - WALL - królik nie może stanąć na tym polu.
    - EMPTY - na tym polu nic nie ma, królik może się po nim poruszać.
    - EGG - w tym polu znajduje się jajo, po zebraniu jaja przez królika pole staje się polem typu EMPTY.
- Move - przechowuje pojedynczy ruch królika. Maksymalna liczba ruchów każdego osobnika jest określana przy inicjalizacji populacji.
- MoveType - określa typ ruchu królika. Ruch może być typu:
    - UP - królik próbuje przesunąć się o 1 pole w górę.
    - RIGHT - królik próbuje przesunąć się o 1 pole w prawo.
    - DOWN - królik próbuje przesunąć się o 1 pole w dół.
    - LEFT - królik próbuje przesunąć się o 1 pole w lewo.
    - TAKE - królik próbuje podnieść jajo.
    - STOP - królik nic nie robi.
    - RANDOM - królik wykonuje 1 z losowych ruchów wymienionych powyżej. Po wykonaniu ruchu zapisuje się wylosowany ruch do zmiennej randomSelectedMove w celu późniejszego odtworzenia trasy królika w czasie wizualizacji. UWAGA! Jeżeli pewien osobnik będzie miał w swojej liście ruchów ruch typu RANDOM to jeżeli przeżyje do następnego pokolenia i nawet nie zostanie skrzyżowany z innym osobnikiem ani też żaden z jego genów nie zostanie poddany mutacji to jest szansa że zostanie wylosowany inny ruch i jakość dopasowania tego osobnika spadnie.
      Przy wyświetlaniu w wizualizacji aktualnie wykonywanego ruchu przez królika jeżeli ruch jest typu RANDOM to przy nazwie wyświetlanego ruchu zostaje dopisany znak *. Na przykład zwykły ruch w górę będzie wyświetlany: UP, natomiast ruch w górę w ramach ruchu losowego: *UP.
- Specimen - odpowiada za pojedynczego osobnika populacji. Przechowuje kolekcje wykonywanych ruchów.
- Population - przechowuje kolekcję osobników oraz jednego najlepszego osobnika.
- QtBoard -odpowiada za wizualizacje. Prezentuje ocenę i ilość zebranych jaj przez najlepszego osobnika, a także pokazuje krok po kroku poruszanie się królika po planszy wraz z pokazywaniem aktualnie wykonywanego ruchu.

W ramach programu możemy zmieniać następujące parametry:

- ilość generacji
- rozmiar populacji
- maksymalna liczba ruchów wykonywana przez królika
- prawdopodobieństwo krzyżowania 0-100%
- prawdopodobieństwo mutacji 0-100%
- rozmiar planszy(np. dla wartości 10 zostanie wygenerowana plansza 10x10 o łącznej liczbie 100 pól).
- ilość jaj na planszy

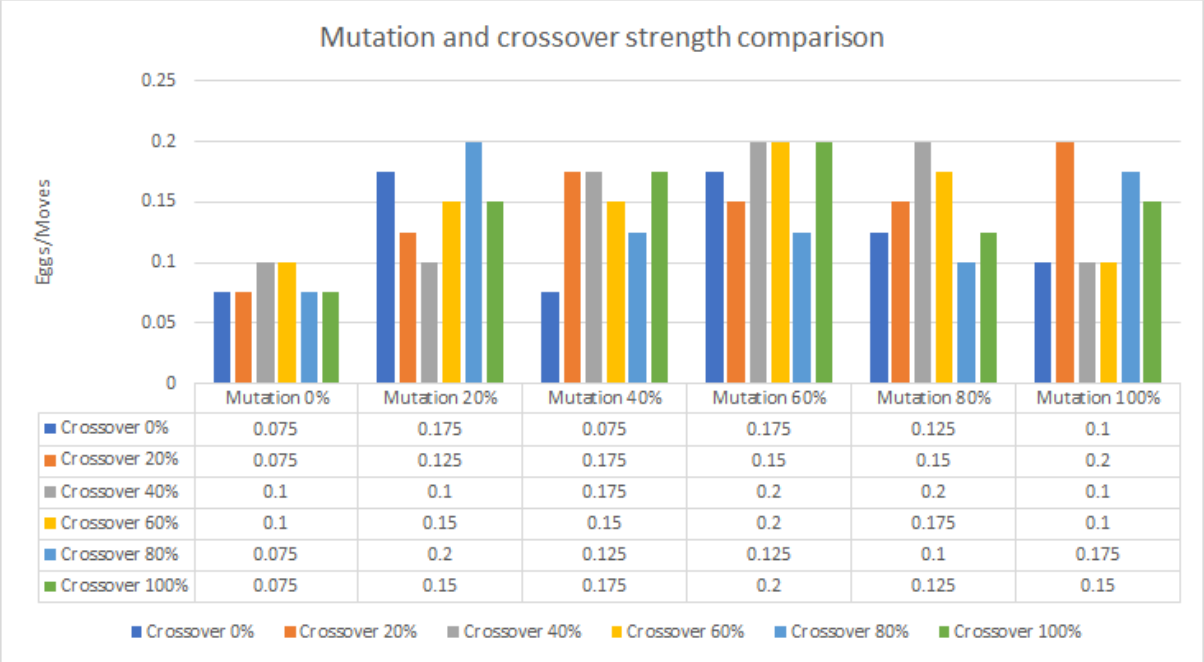
Testy



Plansza 20x20 wygenerowana dla ziarna równego 100.
Estymowane optymalne rozwiązanie dla maksymalnej liczby ruchów równej 40: 12/40 jajek zdobytych w 40 ruchach.

Porównanie wpływu mutacji i krzyżowania

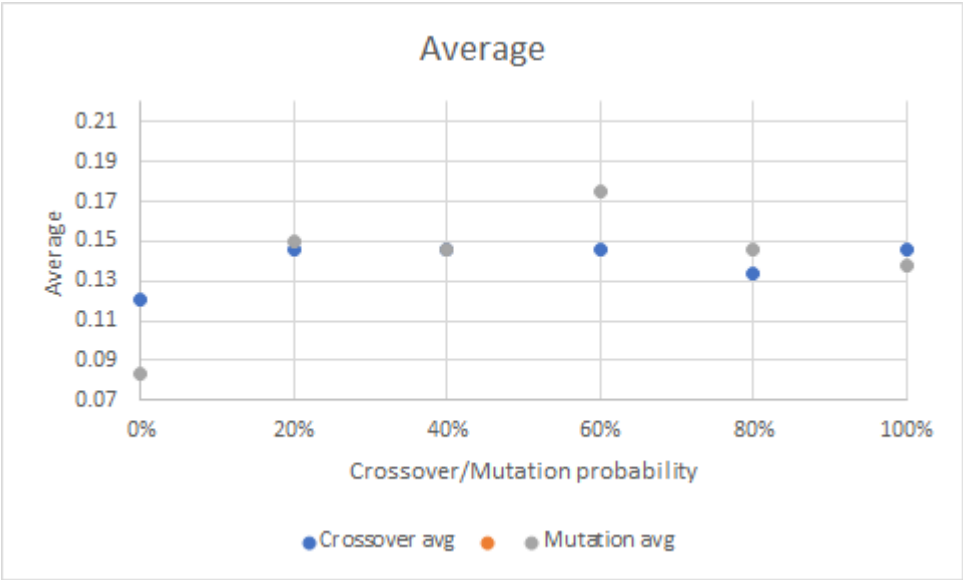
Liczba generacji: 1000
Rozmiar populacji: 20
Rozmiar osobnika: 40
Rozmiar planszy: 20x20
Ilość jajek na planszy: 40
Średni czas wykonania jednego pomiaru: 30s



	Mutation 0%	Mutation 20%	Mutation 40%	Mutation 60%	Mutation 80%	Mutation 100%
Crossover 0%	0.075	0.175	0.075	0.175	0.125	0.1
Crossover 20%	0.075	0.125	0.175	0.15	0.15	0.2
Crossover 40%	0.1	0.1	0.175	0.2	0.2	0.1
Crossover 60%	0.1	0.15	0.15	0.2	0.175	0.1
Crossover 80%	0.075	0.2	0.125	0.125	0.1	0.175
Crossover 100%	0.075	0.15	0.175	0.2	0.125	0.15

Wnioski

- Najlepsze wyniki wystąpiły dla następujących prawdopodobieństw:
  - mutacja 20% krzyżowanie 80%
  - mutacja 60% krzyżowanie 40%
  - mutacja 60% krzyżowanie 60%
  - mutacja 60% krzyżowanie 100%
  - mutacja 80% krzyżowanie 40%
  - mutacja 100% krzyżowanie 20%
- Mutacja ma większy wpływ na wynik niż krzyżowanie, co wynika z poniższego wykresu



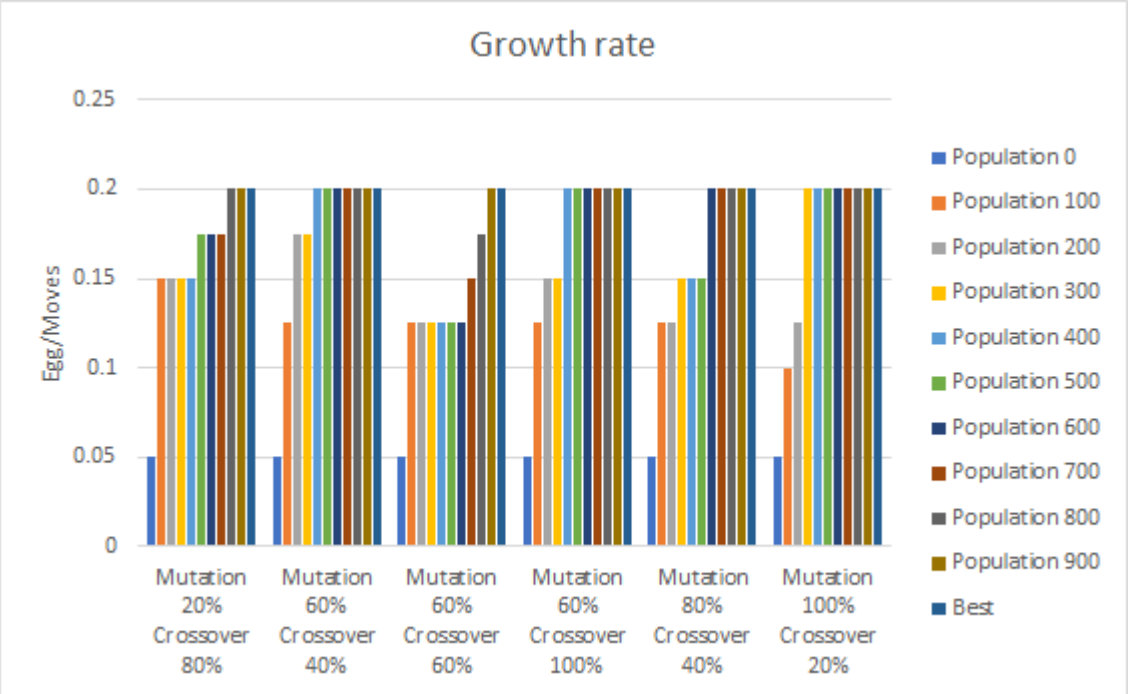
Analiza szybkości znajdywania rozwiązania

Liczba generacji: 1000
Rozmiar populacji: 20
Rozmiar osobnika: 40
Rozmiar planszy: 20x20
Ilość jajek na planszy: 40
Średni czas wykonania jednego pomiaru: 30s

Dla najlepszych wyników:

- mutacja 20% krzyżowanie 80%
- mutacja 60% krzyżowanie 40%
- mutacja 60% krzyżowanie 60%
- mutacja 60% krzyżowanie 100%
- mutacja 80% krzyżowanie 40%
- mutacja 100% krzyżowanie 20%

została zmierzona szybkość dochodzenia do rozwiązania.

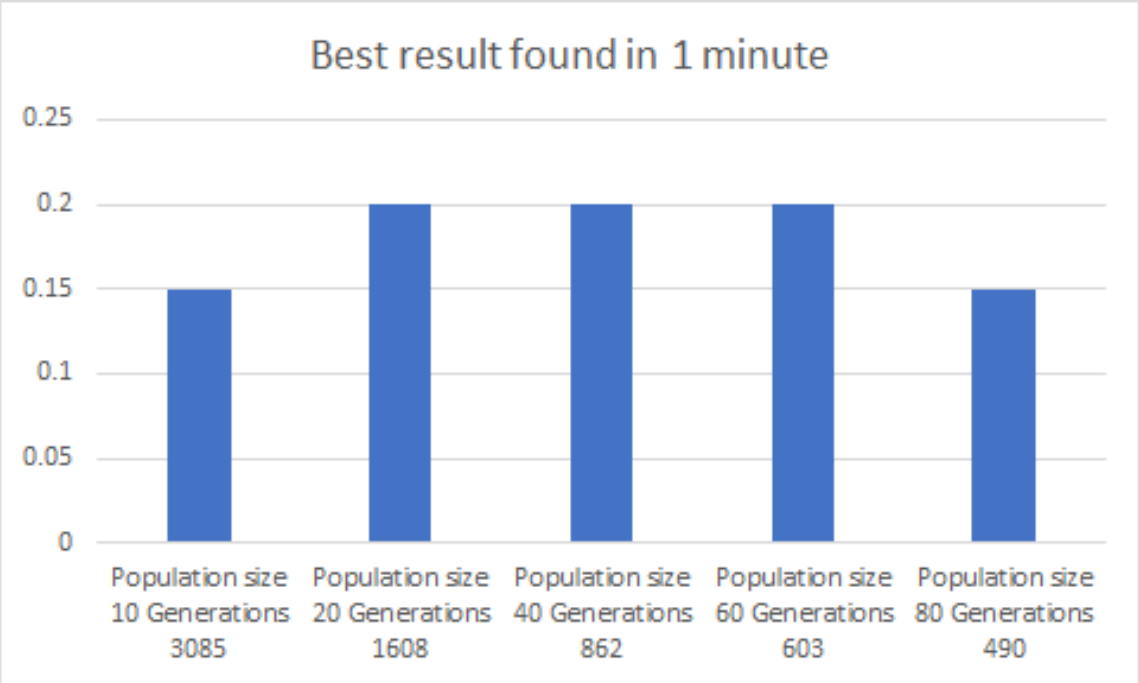


Wnioski

- Algorytm najszybciej do rozwiązania dochodzi z mutacją 100% i krzyżowaniem 20%, ale bardzo podobne wyniki zachodzą dla mutacji 60% i krzyżowania 40% oraz mutacji 60% i krzyżowania 100%
- Na tej podstawie można by powiedzieć, że program działa najlepiej dla mutacji 100% i krzyżowania 20%. Jednak niekoniecznie musi tak być, ponieważ dla różnych planszy mogą lepiej działać różne parametry, a gdyby algorytm działał trochę dłużej, rozwiązanie dużo słabsze mogłoby ,,ewoluować" w rozwiązanie dużo lepsze. Widoczne jest to chociażby na tym wykresie. Algorytm z mutacją 100% i krzyżowaniem 20% startuje powoli, a do optymalnego rozwiązania dochodzi szybko, od wartości 0.125 do 0.2, a np. algorytm z mutacją 20% i krzyżowaniem 80% szybko startuje, ale do najlepszego rozwiązania dochodzi stosunkowo powoli

Optymalny rozmiar populacji

Rozmiar osobnika: 40
Rozmiar planszy: 20x20
Ilość jajek na planszy: 40
Prawdopodobieństwo mutacji: 100%
Prawdopodobieństwo krzyżowania: 20%
Czas wykonania jednego pomiaru: 1 minuta



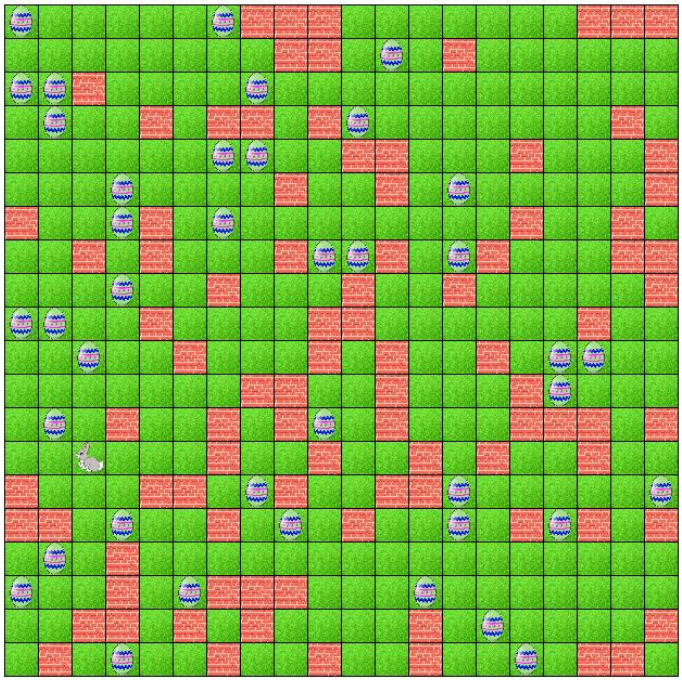
Wnioski

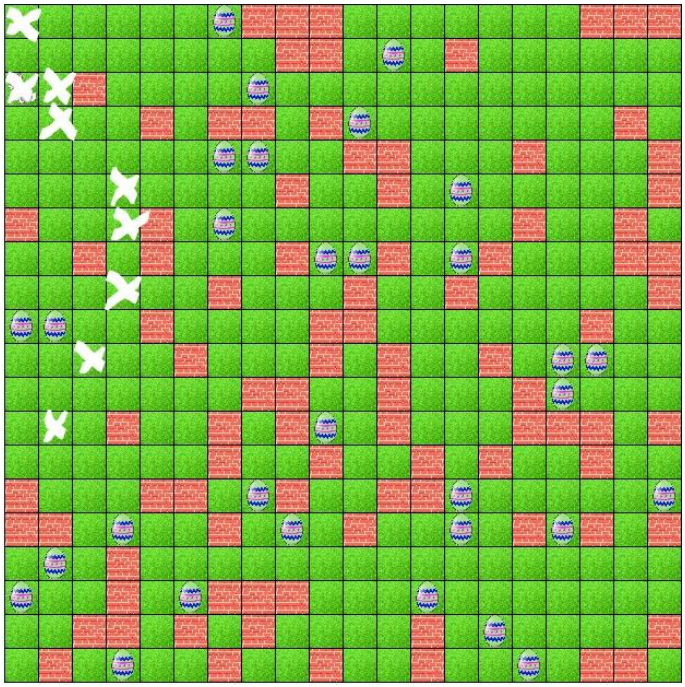
- Im większy jest rozmiar populacji, tym mniej sukcesji, krzyżowań, selekcji, mutacji jest wykonywanych w danym okresie czasu
- Im większy rozmiar populacji, tym większa szansa na znalezienie lepszego osobnika w populacji początkowej. W testach w populacji o rozmiarze 80 najlepszy osobnik startowy miał ocenę 3/40, a w populacjach o mniejszych rozmiarach miał ocenę 2/40. Nie wpłynęło to jednak na znalezienie lepszego wyniku w ciągu 1 minuty
- Na podstawie wyników, wydaje się że optymalny rozmiar populacji powinien być w przedziale 20 - 60

Uruchomienie dla najlepszych parametrów na okres 10 minut

Liczba generacji: x
Rozmiar populacji: 40
Rozmiar osobnika: 40
Rozmiar planszy: 20x20
Ilość jajek na planszy: 40
Prawdopodobieństwo mutacji: 100%
Prawdopodobieństwo krzyżowania: 20%

Czas wykonania pomiaru: 10 minut





Rozwiązanie: 9/40 jajek zdobytych w 40 ruchach.

Instrukcja uruchamiania programu

Na Windowsie aplikację można uruchomić przy użyciu QT Creator.
Na Ubuntu na początku musimy pobrać wymagane pakiety:

- sudo apt-get install g++
- sudo apt-get install make
- sudo apt-get install qt5-default
- sudo apt-get install qtmultimedia5-dev

W katalogu głównym projektu należy uruchomić narzędzie qmake:

```
qmake
```

Zostanie wygenerowany Makefile dla projektu. Poleceniem:

```
make
```

skompilujemy program. Plik wykonywalny "app" zostanie utworzony.

```
./app
```

uruchomi nasz program.

Uwagi

- Początkowo myśleliśmy aby zrobić ten projekt w Pythonie, ponieważ do tej pory nie mieliśmy większej styczności z Pythonem i pomyśleliśmy, że dzięki temu oprócz wykonania samego projektu nauczymy się podstaw języka Python. Jednak zauważyliśmy, że postęp prac jest zbyt wolny i w związku z tym zdecydowaliśmy się, że wykonamy ten projekt w języku C++, który jest nam bardzo dobrze znany.
- Lekko zmodyfikowaliśmy planowane testy, aby lepiej pokazywały działanie naszego algorytmu.