**Rozwiązywanie Fill-a-pixów**

1. **Informacje o problemie**

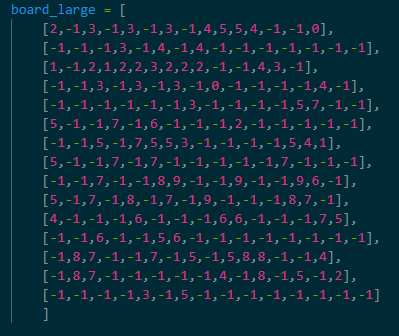
Fill-a-pix to łamigłówka graficzna podobna w pewnym stopniu do nonogramów. Mamy planszę (zazwyczaj kwadratową), podzieloną na małe kwadraty, na której należy pomalować niektóre z nich na czarno w taki sposób, aby wszystkie zamalowane kratki utworzyły spójny obrazek. O zasadzie zamalowania kratek informuje nas odpowiednia liczba w pisana w danym kwadracie która mówi, ile kratek wokół danego kwadratu (łącznie z nim samym) ma być zamalowana.

Problemem, który należało rozwiązać było znalezienie dobrego algorytmu, który rozwiązywałby ww. fill-a-pix, zaimplementowanie takiego algorytmu, przetestowanie jego działania oraz czasu jego działania.

1. **Rozwiązanie problemu**

**2.1 Przygotowania**

Na samym początku, żeby rozwiązać mój problem, potrzebowałem jakieś fill-a-pixy, na których mógłbym testować mój algorytm. Dodałem więc 3 różnej wielkości „mozaiki” przerobione tak, by program mógł je odczytywać.



1. Dodany w kodzie fill-a-pix o rozmiarze 15x15

W miejscach, gdzie postawiona jest -1, interpreter odczytuje daną kratkę jako pustą. Jeżeli jest jednak wstawiona liczba 0-9, jest to odczytywane jako wymagana liczba zamalowanych kratek wokół.

* 1. **Implementacja algorytmu - genetyczny**

Mając coś na czym mogę testować algorytm, przystąpiłem do wstępnej implementacji. Jako możliwe wartości rozwiązań, ustawiłem 0 i 1, gdzie 0 jest równe zamalowanej kratce, a 1 niezamalowanej (jest to odwrócone ustawienie, ze względu na późniejsze wyświetlanie rozwiązania algorytmu, gdzie 0 – czarna kratka, a 1 – biała kratka).

Zasada działania algorytmu jest dosyć prosta: używając algorytmu genetycznego generuje rozwiązanie. Dane rozwiązanie przerabiam na tablicę dwuwymiarową, niejako „nakładając” je na dany szablon z fill-a-pixem. Następnie sprawdzam wartości znajdujące się w szablonie. Jeżeli program wykryje liczbę w przedziale od 0 do 9, przechodzi do zliczania czarnych kratek w rozwiązaniu, na tej samej pozycji w tablicy, w której znaleziona była liczba w szablonie. Funkcja zliczająca liczbę zamalowanych kratek wokół jest najobszerniejszą w programie, ponieważ zaimplementowałem ręcznie każdy możliwy przypadek (liczba może znajdować się w środku szablonu, na danych rogach lub jego brzegach. Wtedy też zmienia się maksymalna liczba możliwych zamalowanych kratek wokół – równomiernie jest to 9, 4 i 6).

Kiedy program obliczy już liczbę zamalowanych wokół kratek, porównuje je z liczbą w szablonie i zwraca wartość bezwzględną różnicy tych dwóch liczb. Liczba ta jest odejmowana od wartości fitness (docelowo 0). Jeżeli końcowo zwracana przez funkcję fitness jest równa zeru, algorytm znalazł bezbłędne rozwiązanie.

Obraz zawierający tekst

Opis wygenerowany automatycznie

2. Funkcja fitness, na podstawie której wiemy czy dane rozwiązanie jest prawidłowe.

* 1. **Implementacja algorytmu – optymalizacja przez rój**

Funkcja fitness dla optymalizacji przez rój wygląda podobnie jak dla algorytmu genetycznego. Jedyna różnica jest taka, że wartość zwracana przez funkcję sprawdzającą zamalowane kwadraty jest dodawana do domyślnej wartości 0, a nie odejmowana. Jest to spowodowane tym, że algorytm roju szuka wartości minimalnej wśród rozwiązań, a genetyczny przeciwnie – maksymalnej.

**Text

Description automatically generated**



**3**. Funkcja fitness dla algorytmu optymalizacji przez rój.

1. **Eksperymenty**

**Text

Description automatically generated**

**4**. Parametry testowe algorytmu genetycznego.

1. **Podsumowanie**