# Generowanie labiryntów metodą Kruskala

January 22, 2019

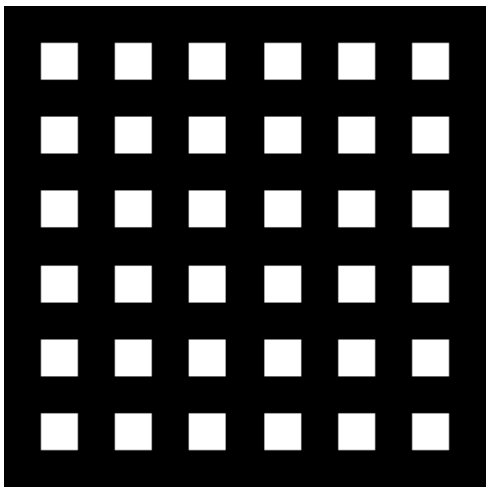
Pavlo Boidachenko

Uniwersytet Jagielloński

# Spis treści

# 1 Opis algorytmu

Algorytm Kruskala jest algorytmem grafowym wyznaczającym minimalne drzewo rozpinające dla grafu nieskierowanego ważonego. Jak się okazuję problem generowania labiryntu łatwo sprowadzić do problemu znanego: wyznaczanie minimalnego drzewa rozpinającego. Generowanie labiryntu zaczynamy od siatki w której białe komórki są przejściami, a czarne ścianami.



*Rys 1. Początkowy stan.*

Przedstawmy sobie że białe komórki reprezentują węzły grafu, a czarne są krawędziami. Przy budowaniu minimalnego drzewa rozpinającego za każdym razem jak wybieramy krawędź do naszego drzewa rozpinającego usuwamy odpowiednią ścianę w labiryncie. W wyniku otrzymujemy labirynt w którym komórki są spójne(z każdej komórki da się przejść do każdej innej) i bez cykli.

W praktyce generowanie labiryntu przebiega następująco:

1. Każdą komórkę wkładamy do swojego zbioru.
2. Wybieramy losową ścianę, jeśli komórki które ona rozdziela są w różnych zbiorach, to złączamy zbiory i usuwamy ścianę.
3. Powtórz krok 2 dopóki są nieodwiedzone ściany.

# 2 Implementacja

## 2.1 Generowanie

Komórki i ściany zaimplementowałem w jednej strukturze danych *WallCell* która ma pole *type* od wartości którego zależy czy *WallCell* reprezentuje komórkę(wartość 0), czy ścianę (wartość 0). ID są przydzielane tylko komórkom, żeby można było ich odróżniać w zbiorach.

```
class WallCell:
  def __init__(self, type, ID=None):
    self.id = ID
    self.type = type
    self.row = None
    self.col = None
```

Jądrem implementacji jest metoda *genMaze()*, która wybiera losową ścianę z listy ścian, szuka sąsiednie komórki dla tej ściany (może ich być maksymalnie dwie), sprawdza czy wybrane komórki są w różnych zbiorach i jeżeli są to usuwa ścianę pomiędzy nimi i złącza zbiory.

```
  def genMaze(self):
    while(len(self.walls) != 0):
      curr = random.choice(self.walls)
      self.walls.remove(curr)
      nb1, nb2 = self.getWallNeighbourCells(curr)
      if self.inDiffSets(nb1, nb2):
        curr.removeWall()
        self.mergeCellSets(nb1, nb2)
        self.frames.append(self.getImage())
```

## 2.2 Wizualizacja

Dla wizualizacji wyników działania programu używam biblioteki PIL (Python Image Library) - biblioteka Pythona przeznaczona dla pracy z grafiką rastrową. Oficjalna strona internetowa: `http://www.pythonware.com/products/pil/`

Biblioteka PIL pozwala na stworzenie animacji w postaci pliku .gif. Dla tego po każdym złączeniu komórek generuję obrazek i przechowuję go w liście. Dla przetwarzania macierzy labiryntu, przechowującej obiekty klasy *WallCell*, jest metoda *getImage()*, która iteruję wierszami i kolumnami macierzy i zależnie od typu komórki jest rysowany czarny lub biały piksel. Obrazek jest skalowany, żeby ładnie się otwierał w programach dla przeglądania grafiki.

# 3 Używanie programu

Po uruchomieniu skryptu użytkownik musi wprowadzić wymiar labiryntu. Jeśli podany wymiar jest parzystą liczbą $x = 2n$, to ona automatycznie jest zamieniana na najbliższą liczbę nieparzystą $x = 2n + 1$, żeby obrazek ładnie wyglądał. W tym samym folderze co i uruchamiany skrypt pojawią się dwa pliki wynikowe: *output.png* z obrazkiem wygenerowanego labiryntu i *anim.gif* z animacją generowania labiryntu.

# Bibliography

[1] Biblioteka PIL http://www.pythonware.com/products/pil/

[2] Wikipedia: Maze generation algorithm https://en.wikipedia.org/wiki/Maze_generation_algorithm

[3] Creating animated GIFs with Pillow http://www.pythoninformer.com/python-libraries/pillow/creating-animated-gif/