Wyższa Szkoła Bankowa

Uczenie Maszynowe

Ćwiczenia 6 - zadania

Wykonanie:
Dawid Szwarc 99434
Marcin Doroszko 135670
Kuba Słaboń 135514

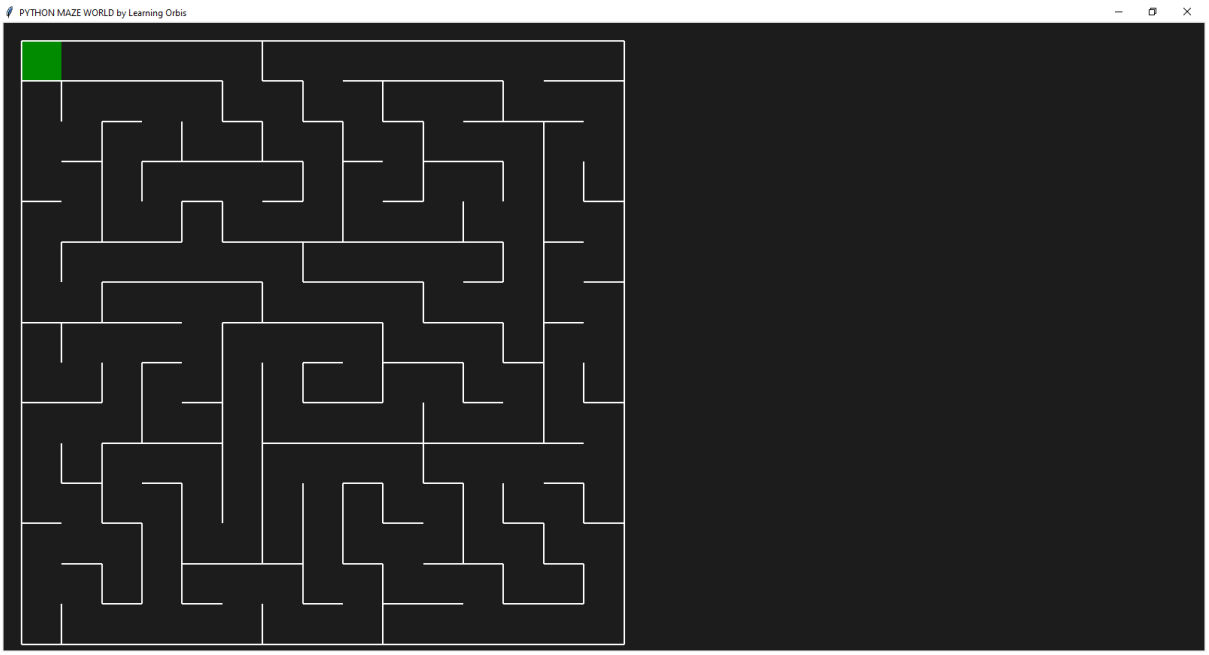
Autor: Lesław Pawlaczyk
2023/05/20

Spis treści

Rozdział 1 – Algorytmy przeszukiwania grafów

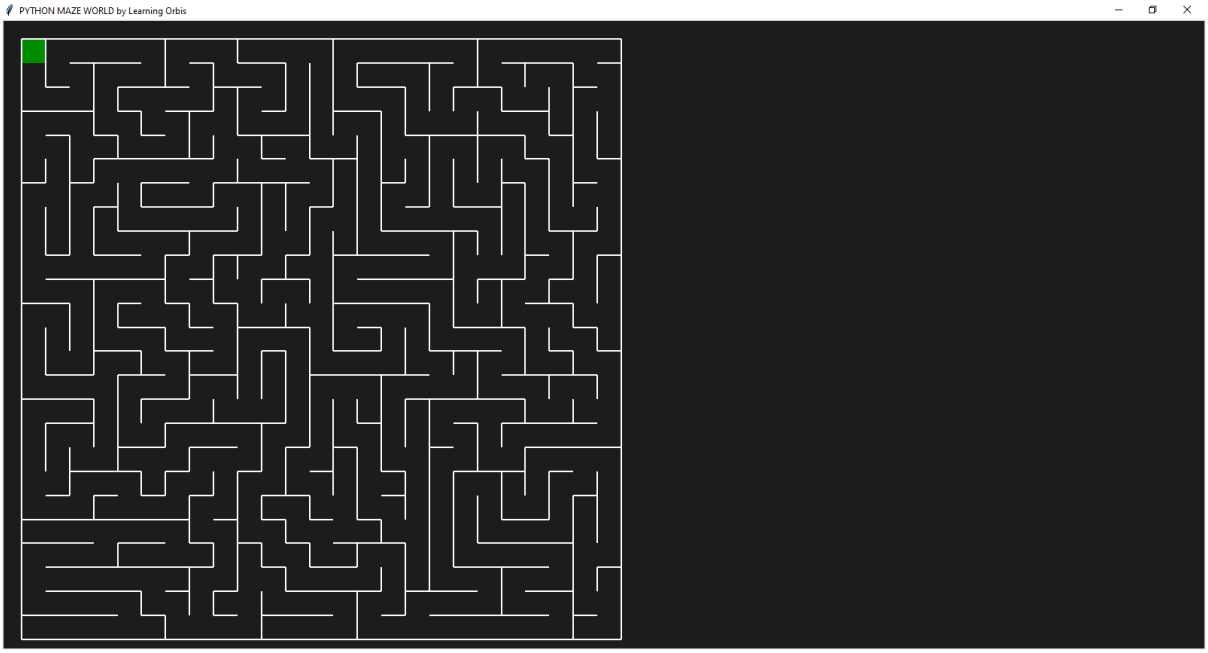
Zadanie 1.1

Na podstawie pliku maze.py przeprowadzić poniższe eksperymenty

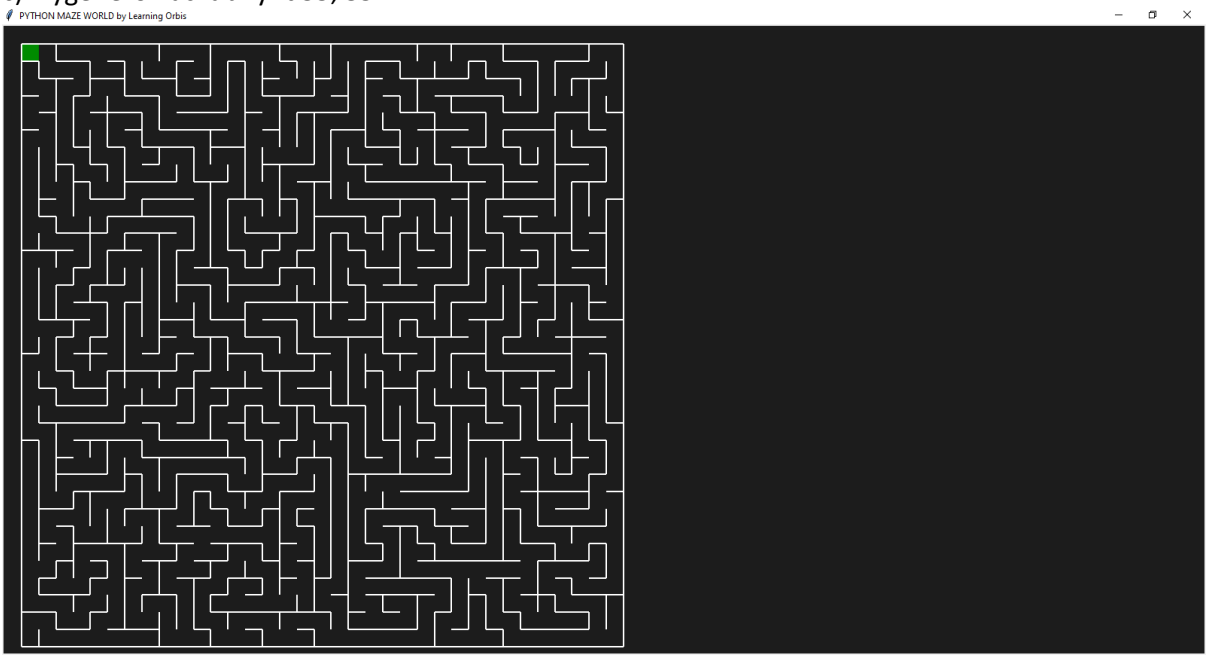


a) wygenerować labirynt 15, 15

b) wygenerować labirynt 25, 25

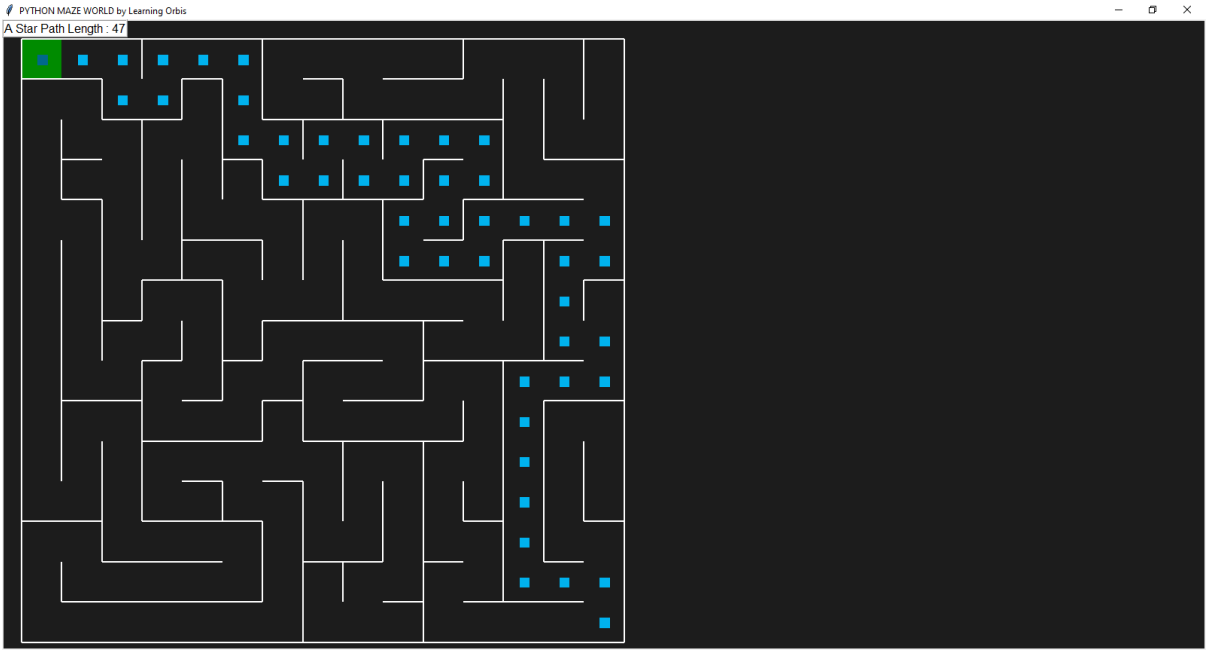


c) wygenerować labirynt 35, 35

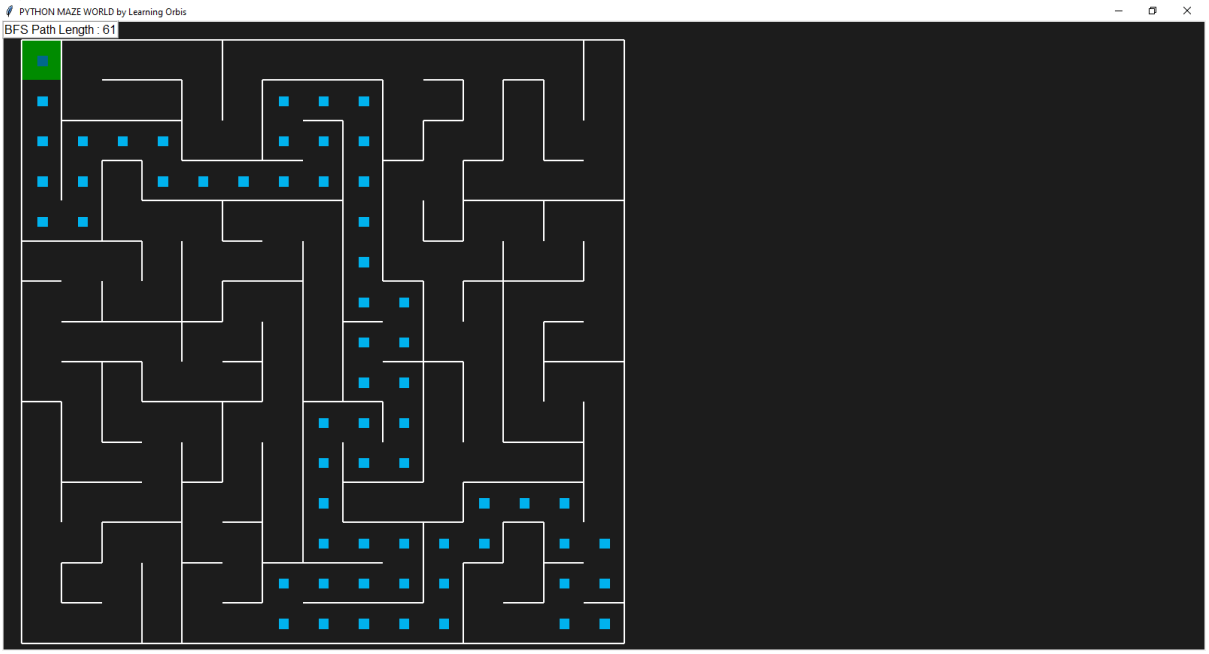


Wykonać przeszukiwanie labiryntu algorytmem A*, oraz Breadth First Search.
Pokazać rozwiązania dla wszystkich 6 przypadków w postaci zrzutów ekranów.

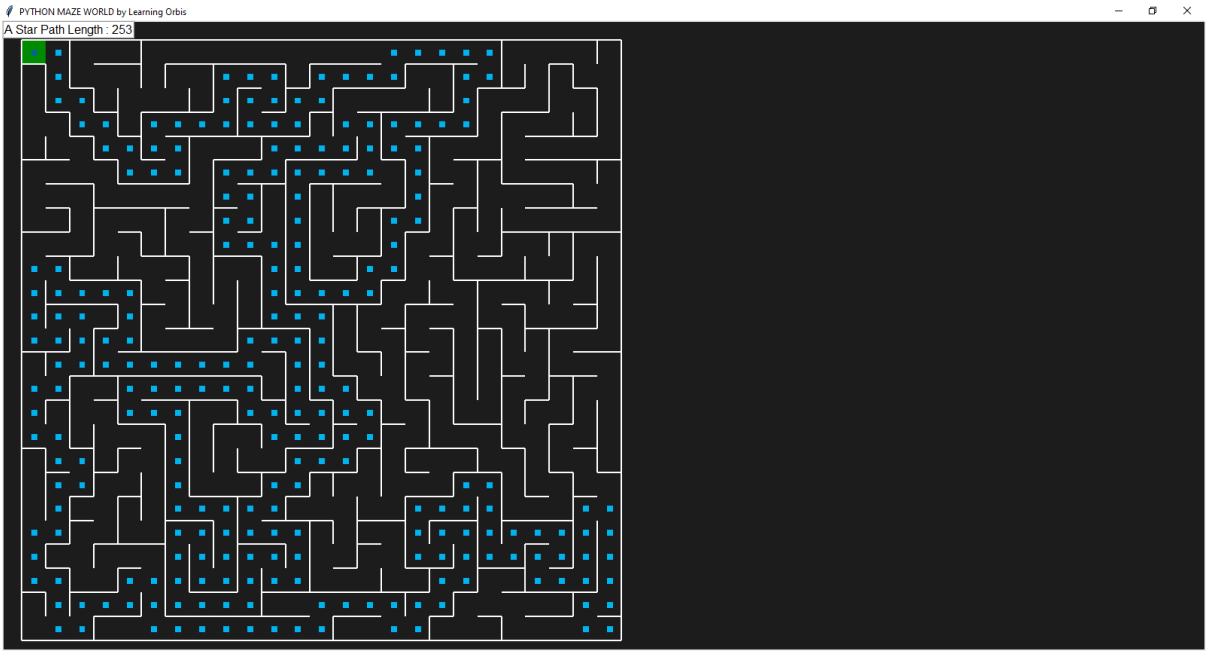
a) 15x15 A*



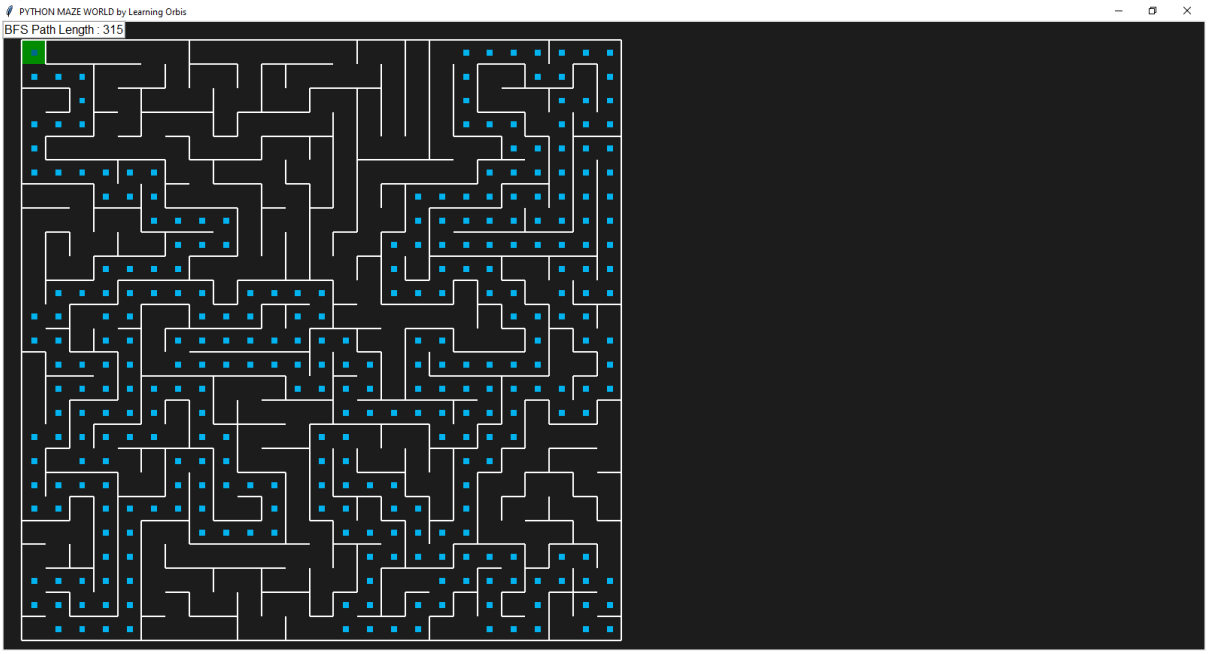
b) 15x15 BFS



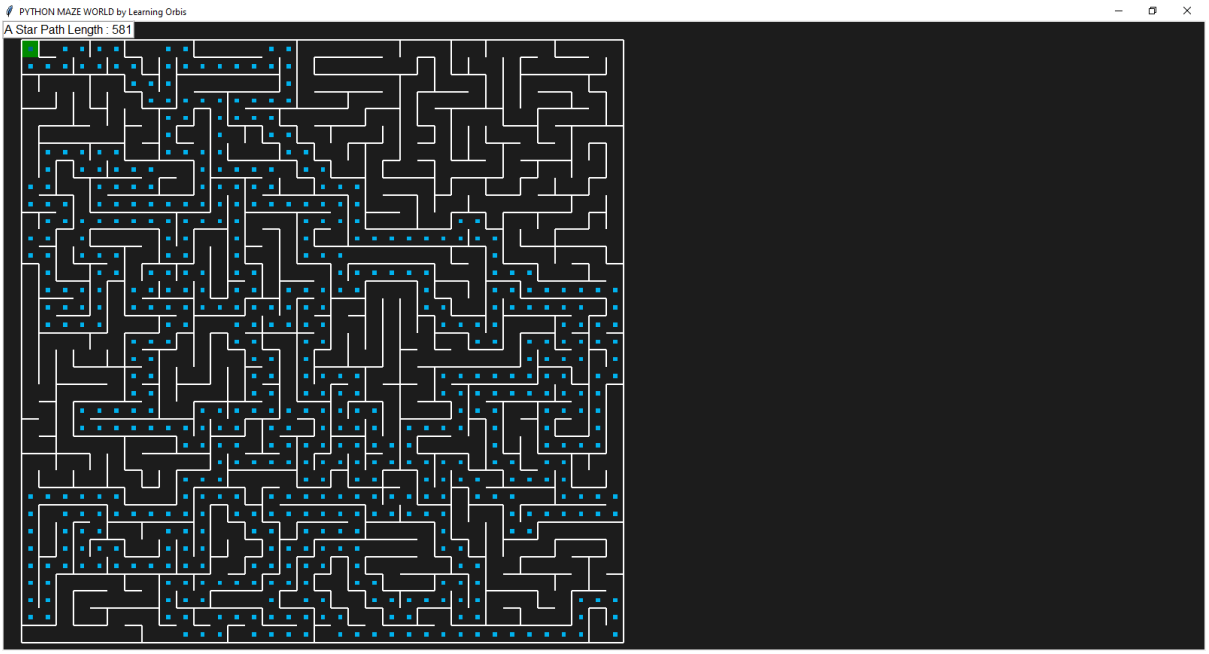
c) 25x25 A*



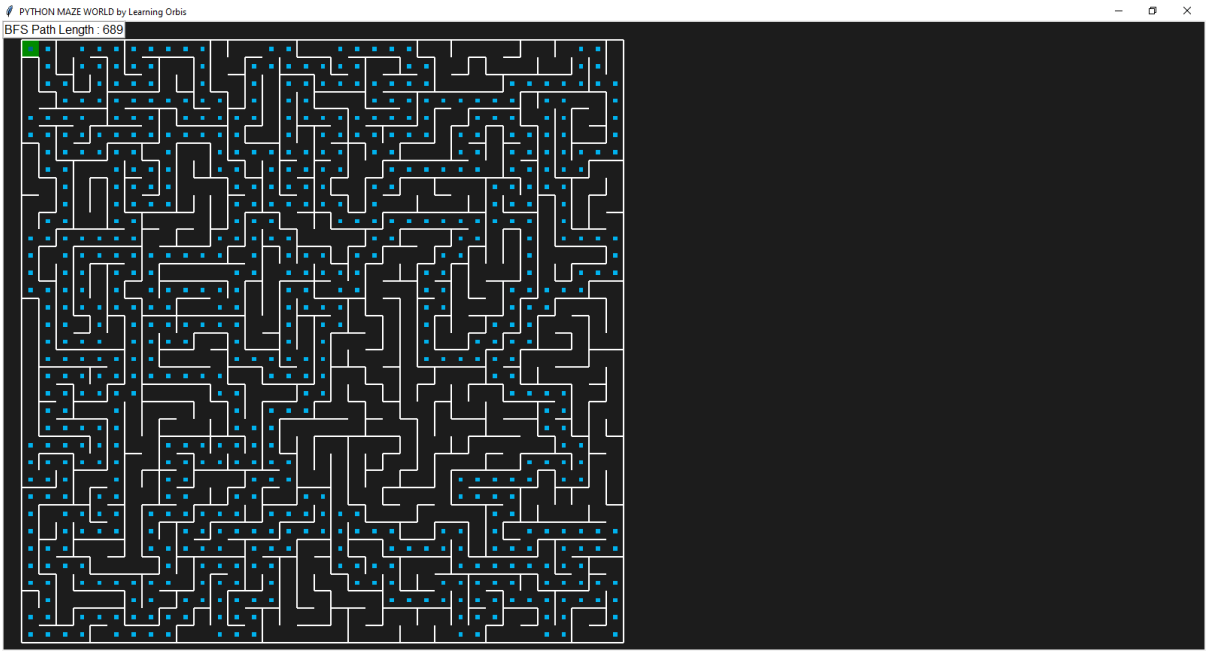
d) 25x25 BFS



e) 35x35 A*



f) 35x35 BFS



Pokazać kod BFS.

```
def bfs(m):
    start = (m.rows, m.cols)
    # Używamy kolejki do przechowywania komórek do przetworzenia.
    queue = deque([start])
    # Słownik przechowujący informacje o odwiedzonych komórkach.
    visited = {cell: False for cell in m.grid}
    # Słownik przechowujący informacje o poprzedniku każdej komórki.
    pred = {cell: None for cell in m.grid}

    visited[start] = True

    while queue:
        currCell = queue.popleft()

        # Jeżeli znaleziono wyjście, przerywamy przeszukiwanie.
        if currCell == (1, 1):
            break

        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    childCell = (currCell[0], currCell[1] + 1)
                elif d == 'W':
                    childCell = (currCell[0], currCell[1] - 1)
                elif d == 'N':
                    childCell = (currCell[0] - 1, currCell[1])
                elif d == 'S':
                    childCell = (currCell[0] + 1, currCell[1])

                if not visited[childCell]:
                    visited[childCell] = True
                    pred[childCell] = currCell
                    queue.append(childCell)

    # Odtwarzamy ścieżkę od punktu końcowego do początkowego.
    cell, path = (1, 1), {}
    while pred[cell]:
        path[pred[cell]] = cell
        cell = pred[cell]
    return path
```

Dlaczego algorytm A* jest szybszy niż BFS? Jakie są zasadnicze różnice między nimi?

Algorytm A* i BFS są oboma algorytmami przeszukiwania grafu, ale różnią się w sposobie, w jaki wybierają kolejne wierzchołki do przeszukania. Oto zasadnicze różnice i powody, dla których A* jest często szybszy niż BFS:

Heurystyka:

A* używa funkcji heurystycznej, aby oszacować koszt dotarcia od danego wierzchołka do celu. Dzięki temu może priorytetyzować wierzchołki, które prawdopodobnie znajdują się bliżej celu. Jeśli heurystyka jest dobrze dobrana, A* przeszukuje znacznie mniej wierzchołków niż BFS.
BFS nie używa heurystyki. Przeszukuje wierzchołki w kolejności ich odkrycia, co może prowadzić do przeszukiwania wielu niepotrzebnych wierzchołków, zwłaszcza jeśli cel znajduje się w jednym kierunku, a algorytm spędza dużo czasu na przeszukiwaniu wierzchołków w przeciwnym kierunku.
Kolejka priorytetowa vs. kolejka FIFO:

A* używa kolejki priorytetowej do wyboru wierzchołków do przeszukania. Wierzchołki z niższym oszacowanym kosztem dotarcia do celu (biorąc pod uwagę koszt dotarcia do bieżącego wierzchołka i heurystykę) są przeszukiwane jako pierwsze.
BFS używa kolejki FIFO (pierwszy na wejściu, pierwszy na wyjściu), co oznacza, że wierzchołki są przeszukiwane w kolejności ich odkrycia.
Koszt ścieżki:

A* uwzględnia koszt dotarcia do bieżącego wierzchołka oraz oszacowany koszt dotarcia od bieżącego wierzchołka do celu. Może to prowadzić do szybszego znalezienia najkrótszej ścieżki, zwłaszcza jeśli różne ścieżki mają różne koszty.
BFS nie uwzględnia kosztów ścieżki i zawsze traktuje każdy ruch jako równy koszt.
Podsumowując, A* jest często szybszy niż BFS, ponieważ skupia się na przeszukiwaniu wierzchołków, które prawdopodobnie znajdują się bliżej celu, podczas gdy BFS przeszukuje wszystkie wierzchołki w sposób jednolity. Jednak warto zauważyć, że A* będzie znacznie szybszy tylko wtedy, gdy używana heurystyka jest dość dokładna i nie niedoszacowuje faktycznego kosztu dotarcia do celu. Jeśli heurystyka będzie zawsze zwracać 0, A* zachowuje się dokładnie tak samo jak BFS.

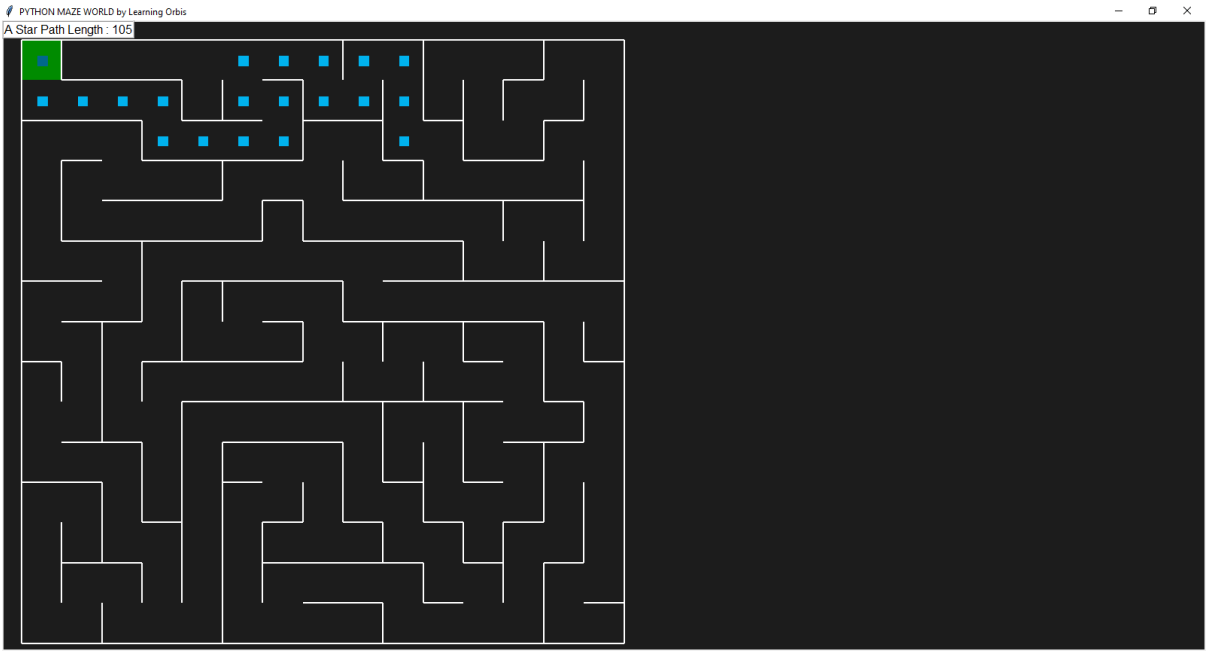
Zadanie 2.2

Wykonać modyfikację pliku maze.py (do pomocy może posłużyć poniższy adres: https://github.com/ryannel/pymaze/blob/master/maze.py) i zaimplementować algorytm poszukiwania BREADTH-FIRST

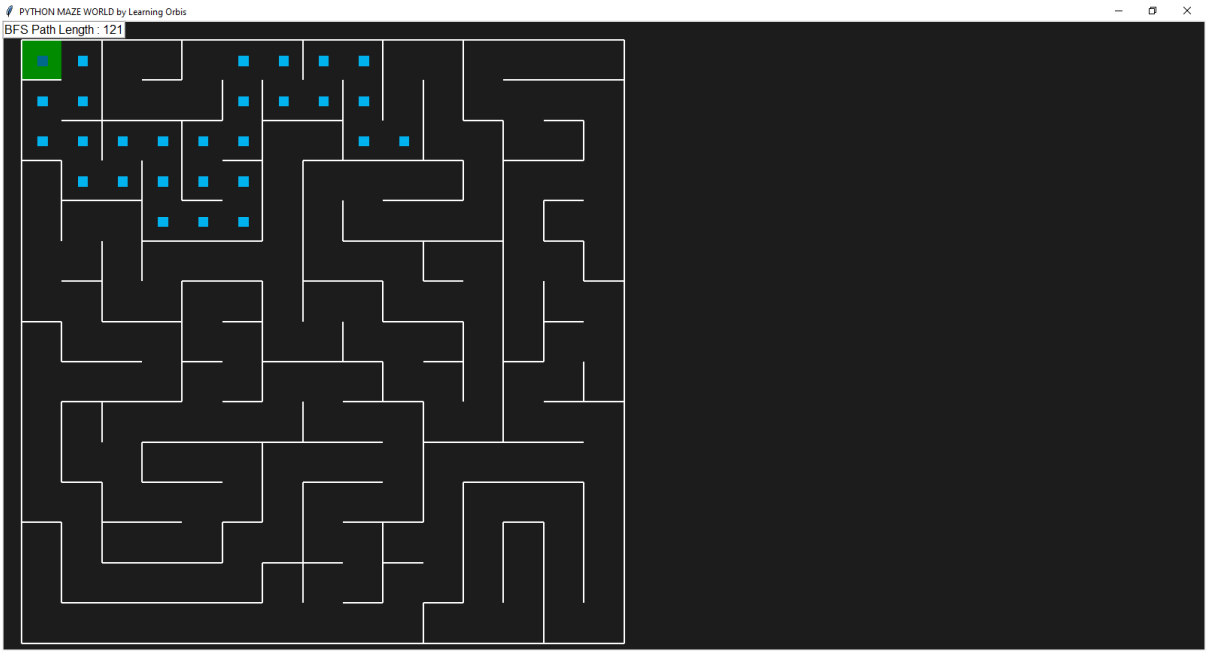
Wygenerować wyniki dla przypadków labiryntu takiego samego jak w zadaniu 1 i dla różnych pozycji startowych

Pokazać rozwiązania dla wszystkich 6 przypadków w postaci zrzutów ekranów

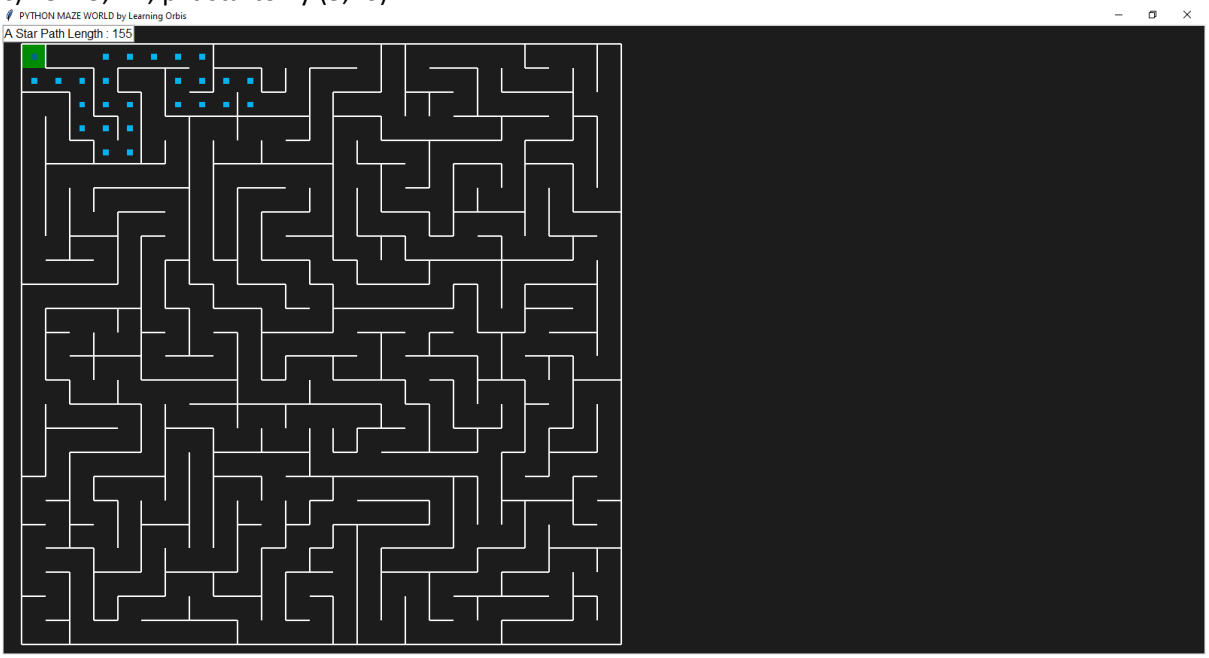
a) 15x15, A*, pkt startowy (3,10)



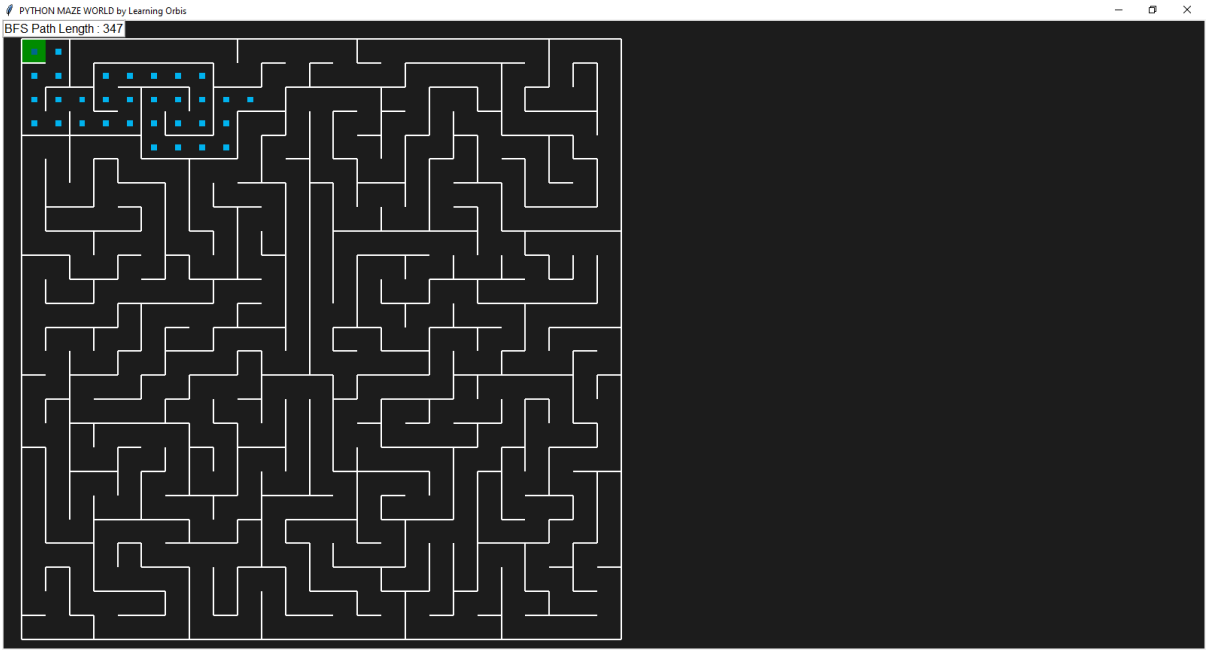
b) 15x15, BFS, pkt startowy (3,10)



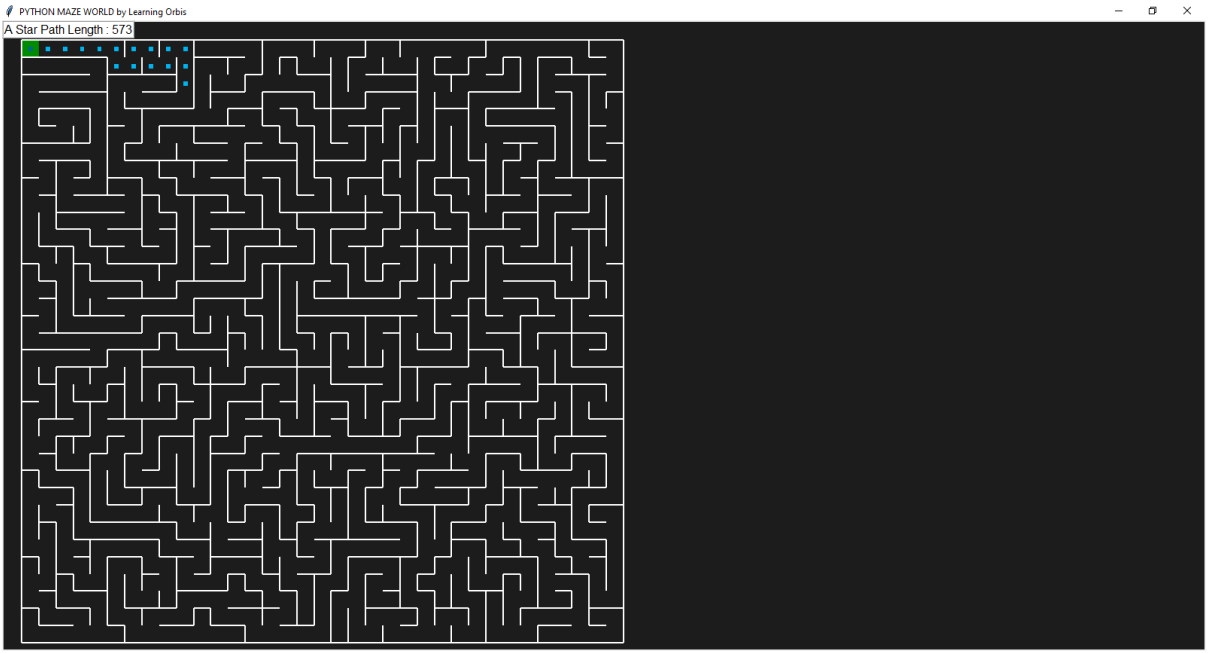
c) 25x25, A*, pkt startowy (3,10)



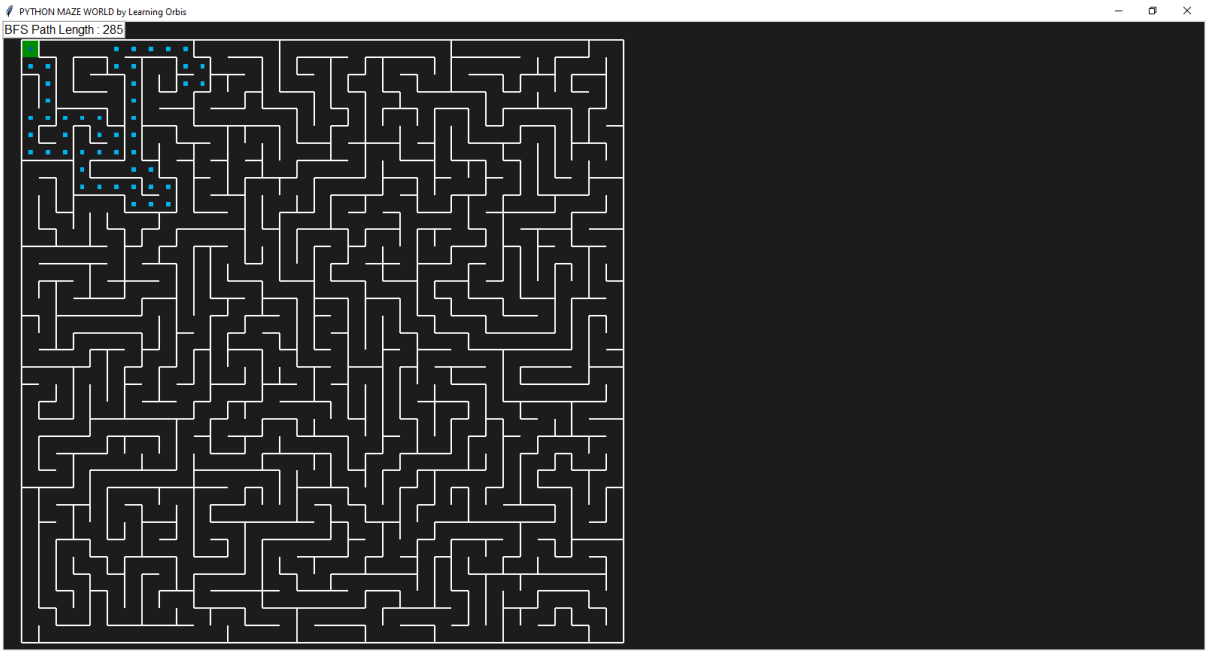
d)25x25, BFS, pkt startowy (3,10)



e)35x35, A*, pkt startowy (3,10)



f)35x35, BFS, pkt startowy (3,10)



Jak dłużej trzeba czekać na wyniki algorytmu breadth-first w porównaniu z A*.
Podać złożoności obliczeniowe obu algorytmów

BFS

Złożoność czasowa:

O(V+E), gdzie V to liczba wierzchołków, a E to liczba krawędzi.

Złożoność pamięciowa:

O(V) dla przechowywania informacji o odwiedzonych wierzchołkach oraz
O(V) dla kolejki.

A*

Złożoność czasowa:
O(VlogV)

Złożoność pamięciowa:
O(V)

W praktyce, jeśli chodzi o poszukiwanie ścieżek w grafach, A* zwykle działa szybciej niż BFS, ponieważ korzysta z dodatkowej wiedzy (heurystyki) do wyboru najbardziej obiecujących wierzchołków do przetworzenia. BFS natomiast przeszukuje wszystkie dostępne wierzchołki w kolejności, w jakiej je napotyka, bez faworyzowania tych, które wydają się być bliżej celu.

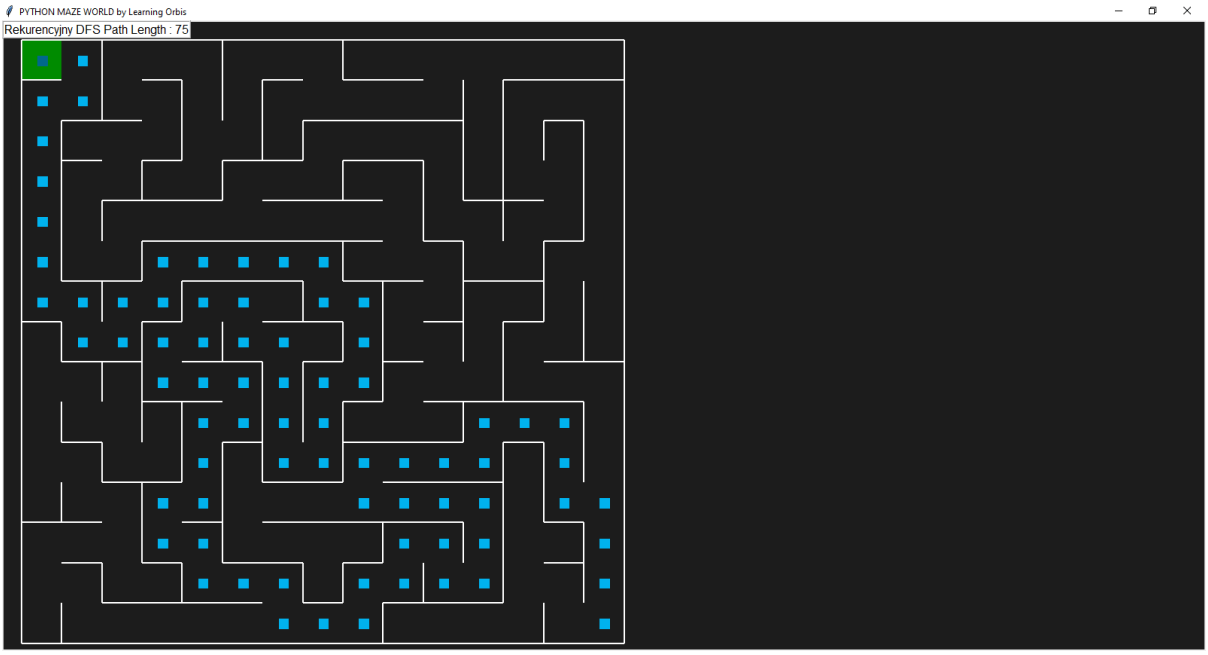
Zadanie 2.3

Wykonać modyfikację pliku maze.py i przeszukać rekurencyjnie cały labirynt.

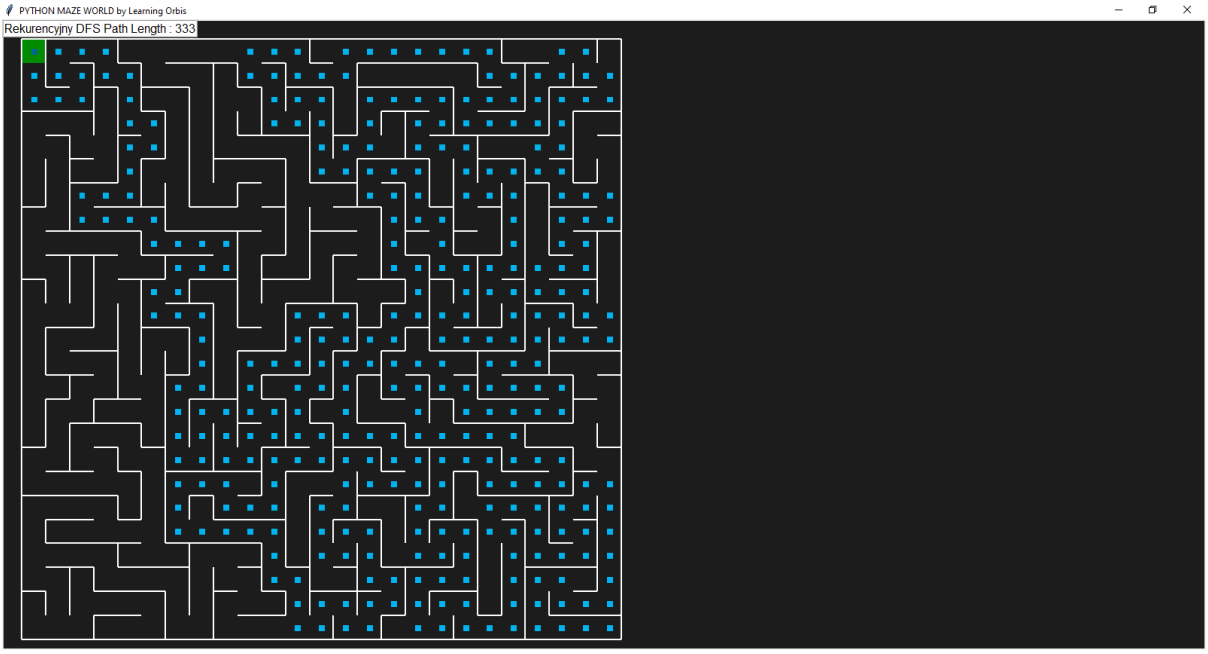
Pokazać rozwiązania dla wszystkich 6 przypadków w postaci zrzutów ekranów

Do rozwiązania wykorzystano rekurencyjny wariant algorytmu DFS:

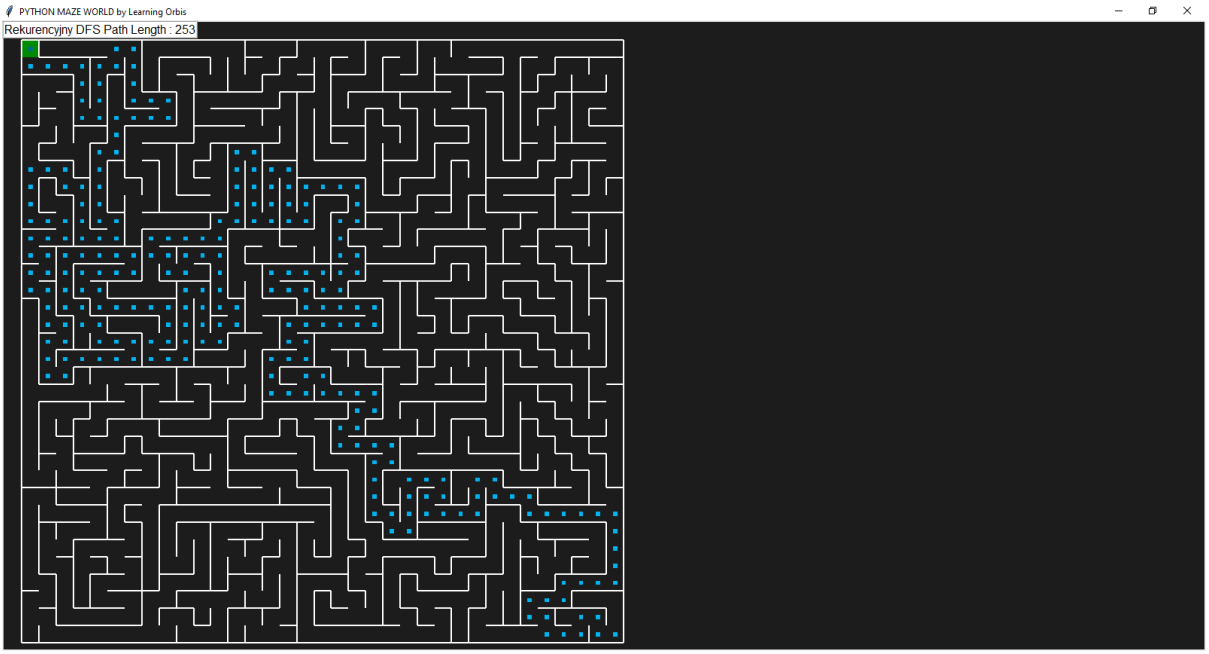
a)15x15



b) 25x25



c) 35x35


PYTHON MAZE WORLD by Learning Orbis
Rekurencyjny DFS Path Length : 253

Rozdział 2 – Uczenie ze wzmocnieniem, programowanie dynamiczne

Zadanie 2.1

Na podstawie adresu https://gist.github.com/palles77/1ee228ad15a03353439536e9ae632b20 wykonać ćwiczenia i przebadać

a) działanie algorytmu szukania drogi w jeziorze dla losowego jeziora o rozmiarach 8 x 8, wartościach gamma ze zbioru [0, 0.25, 0.5, 0.75, 1] dla metod policy_iteration oraz value_iteration

Odpowiedzieć na pytania:
1. Jaki wpływ na jakość znalezienia drogi do frisbee ma wartość gamma,
2. Czym różnią się metody policy_iteration oraz value_iteration?
3. Dlaczego value_iteration znajduje lepszą drogę niż policy_iteration

Zadanie 2.2

Wykonać to samo ćwiczenie co w zadaniu 2.1 ale dla jeziora o rozmiarze 10 x 10 i wartościach gamma [0, 0.2, 0.4, 0.6, 0.8, 1] oraz włączonym poślizgu na lodzie.

Odpowiedzieć na pytania:
1. Jak poślizg na lodzie ma wpływ na skomplikowanie trasy
2. Czy wartość gamma ma wpływ na skomplikowanie trasy, a jeżeli tak, to jakie.

Zadanie 2.3

Wykonać to samo ćwiczenie co w zadaniu 2.1 ale dla jeziora o rozmiarze 8 x 8 i wartościach gamma [0, 0.2, 0.4, 0.6, 0.8, 1].
Sprawdzić działanie metody value_iteration_2 na samym dole.

Czym się różni metoda value_iteration od value_iteration_2?

Rozdział 3. Uczenie ze wzmocnieniem – aktor, krytyk

Na podstawie https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/reinforcement_learning/actor_critic.ipynb przeprowadzić trenowanie systemu uczącego się balansować wózkiem w systemie krytyk-aktor.

Zadanie 3.1

Pokazać wizualizacje dla trenowania w postaci plików GIF wygenerowanych i dołączonych do sprawozdania jako repo na Github.

- 1000 epizodów,
- 2000 epizodów,
- 5000 epizodów,
- 10000 epizodów.

Sprawdzić jaki wpływ na parametr Gamma dla 5000 epizodów. Przebadać następujące wartości gamma:

- 0.1,
- 0.2,
- 0.5,
- 0.9,
- 0.99.

Odpowiedzieć na pytania:

1. Jak dobrze zachowuje się system po wykonaniu różnej ilości epizodów. Pokazać konkretne zachowania, które się zmieniły pomiędzy małą ilością epizodów, a dużą ilością epizodów.
2. Jak gamma ma wpływ na szybkość uczenia i zachowanie się systemu. Czy widać znaczący wpływ tego parametru?

Rozdział 4 - Ocenianie

1. Ocena **dst**: zadania z rozdziału 1
2. Ocena **db**: zadanie z rozdziałów 1, 2
3. Ocena **bdb**: zadanie z rozdziałów 1, 2 oraz 3.

Wyniki wysłać na adres leslaw.pawlaczyk@chorzow.wsb.pl jako linki do plików Jupyter Notebook, w formacie GIST GITHUB lub dokument Word z kodem i zrzutami ekranu. Tam gdzie jest konieczne wideo GIF dodać je do repo na GITHUB.