ZAKŁAD SYSTEMÓW ZŁOŻONYCH
Wydział Elektrotechniki i Informatyki
ul. Wincentego Pola 2, 35-959 Rzeszów, tel. 17 865 1340
zsz.prz.edu.pl

WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Sprawozdanie z projektu

Programowanie w języku Python

Temat zajęć: Implementacja algorytm A* w języku Python

Prowadzący: mgr. Inz. Patryk Organiściak

Imię i nazwisko	Damian Bielecki
Nr albumu	163461
Rok studiów	IV EFZI
Grupa laboratoryjna	L01
Data oddania	23.04.2023

1. Opis projektu

Celem projektu jest stworzenie programu w języku Python, którego zadaniem będzie wczytywanie mapy środowiska z zewnętrznego pliku a następnie po wciśnięciu odpowiedniego przycisku wyznaczenie najkrótszej ścieżki przy pomocy algorytmu A*. W trakcie działania programu będzie możliwość modyfikacji mapy, punku początkowego i końcowego. Program zostanie napisany przy pomocy biblioteki pygame, wykorzystanej do rysowania prostych elementów graficznych oraz pygame_gui do obsługi elementów interfejsu użytkownika.

2. Opis najważniejszych elementów w programie

A* to heurestyczny algorytm wyznaczający najkrótszą możliwą ścieżkę w grafie. Jest to algorytm zupełny i optymalny, a więc zawsze zostanie wyznaczone optymalne rozwiązanie. Ze względu na przeszukiwanie oparte na grafie algorytm działa najlepiej na strukturze drzewiastej. Zadaniem algorytmu jest minimalizacja funkcji:

$$\boldsymbol{f(x) = h(x) + g(x)} \qquad (1)$$

Gdzie:

$$f(x) - to\ minimalizowana\ funkcja$$

$$g(x) - to\ rzeczywisty\ koszt\ dojścia\ do\ punktu\ x$$

$$h(x) - to\ funkcja\ heurestyczna\ oszacowująca\ koszt\ dotarcia\ do\ punktu\ x$$

a. Implementacja algorytmu

Mechanizm wyszukiwania najkrótszej ścieżki został zamknięty w jednym module o nazwie aStar. Na moduł składa się klasa AStar ze wszystkimi potrzebnymi metodami oraz dodatkową klasą Node reprezentującą pojedynczy punkt przeszukiwanego grafu. Wyznaczanie najkrótszej ścieżki rozpoczyna się od wyznaczenia kosztu punktu startowego i utworzenia zbioru z nieprzeszukanymi wierzchołkami, do którego dopisujemy punkt początkowy.

```
startNode.h = self.heurestic(startNode.getCords(), endNode.getCords())
startNode.g = 0
heappush(openList, startNode)
```

Listing 1. Wrzócenie punktu startowego na liste

Wewnętrzna funkcja heurestic przyjmująca pozycje dwóch punktów odpowiada za wyliczenie optymistycznego kosztu przejścia od punktu x do wierzchołka docelowego. Takie podejście pozwala na szybką podmianę funkcji bez znaczących zmian w programie. Wykorzystywana funkcja heurestyczna to równanie eukalidesa W kolejnym kroku uruchamiana jest pętla, która wykonywana jest dopóki w zbiorze otwartym znajdują się nie odwiedzone elementy. Z listy

pobierany jest element o najmniejszej liczbie punktów co oznacza, że dany wierzchołek drzewa ma największe szanse być najlepszym rozwiązaniem. Jeżeli pobrany element nie jest celem to dalej pobierani i przetwarzani są wszyscy jego sąsiedzi. W tym rozwiązaniu, dla zaoszczędzenia pamięci, symulator przechowuję tylko przeszkody. W związku z tym do pobrania sąsiadów używana jest specjalna funkcja sprawdzająca czy na mapie o wskazanych koordynatach istnieje punkt. Jeżeli takowy obiekt nie istnieje to oznacza, że algorytm może użyć tych współrzędnych do trasowania ścieżki. Od tej części programu zależy, czy algorytm ma wyznaczyć ścieżkę uwzględniając skoki po skosie.

```
neighborNode = None
if Node(newCords) in openList:
    neighborNode = self._findNodeOnList(openList, newCords)
else:
    neighborNode = Node(newCords)
    neighborNode.g = COST
    neighborNode.h = self.heurestic(newCords, endNode.getCords())
    neighborNode.parent = currentNode
    openList.append(neighborNode)
    distance_from_curr_to_neighbor = COST
    scoreFromStartToCurrentNeighbor =  currentNode.g +
distance_from_curr_to_neighbor
if neighborNode.getScore() <= currentNode.getScore():
    neighborNode.g = scoreFromStartToCurrentNeighbor
    neighborNode.h = scoreFromStartToCurrentNeighbor +
self.heurestic(newCords, endNode.getCords())
     neighborNode.parent = currentNode
```

Listing 2. Wyznaczenie kolejnego węzła

Powyższy kod odpowiada za wyznaczenie kosztu przejścia do sąsiada. Jeżeli punkt nie istnieje jeszcze w otwartym zbiorze to jest tworzony i do niego dodawany. Zgodnie z założeniem całkowity koszt to suma rzeczywistego kosztu dystansu i wyniku funkcji heurestycznej. Rzeczywisty koszt (funkcja g(x)) jest ustalony na sztywno i zależy od współrzędnych, do których skaczemy. Jeżeli następny punkt jest na wprost to koszt wynosi 1. Jako iż przechodzimy po siatce z węzłami o stałej i równej odległości to koszt skoku do sąsiada po skosie został wyznaczony z twierdzenia Pitagorasa. Należy zwrócić uwagę, że jeżeli całkowity koszt jest mniejszy od aktualnego to węzeł ma ustawianego rodzica, ma to duże znaczenie w przypadku wyjścia z pętli i wyznaczenia najkrótszej trasy spośród wszystkich kosztów

```
if currentNode == endNode:
    path = []
    while currentNode is not None:
        path.append(currentNode)
        currentNode = currentNode.getParent()
        return path
    else: # nie znaleziono ścieżki
        return []
```

Listing 3. Wyznaczenie ostatecznej ścieżki

Po zakończeniu pętli, sprawdzane jest czy została znaleziona ścieżka, jeżeli takowa istnieje to w kolejnej pętli wyznaczane są przy pomocy pola wskazującego na rodzica kolejne punkty trasy. Jeżeli węzeł nie jest punktem końcowym to trasa nie została znaleziona i zwracana jest pusta lista.

b. Implementacja środowiska symulacyjnego

Dla logicznego podzielenia programu została napisana główna klasa programu o nazwie App. Takie podejście pozwoliło na odseparowanie poszczególnych funkcji aplikacji. W pierwszej kolejności konstruktor obiektu aplikacji, generuje okno, tworzy graficzne elementy użytkownika oraz ustawia odpowiednie flagi informujące o stanie aplikacji. Dalej wywoływana jest funkcja main wczytująca odpowiednią mapę z pliku, a następnie uruchamiająca główną pętlę programu działającą w 60 klatkach na sekundę.

Rysunek 1. Schemat działania aplikacji.

Aby uprościć symulowanie różnych scenariuszy i rozmiarów map otoczenia, ta ładowana jest ze wskazanego w programie pliku. Wskazany plik jest w formacie json i podzielony jest na dwie sekcje. Pierwsza dostarcza informacji o samej mapie, natomiast druga o poruszającym się po niej robocie. Sekcja mapy zawiera lokalizacje do pliku tekstowego z zapisanymi elementami.

Ładowaniem wszystkich tych informacji zajmuje się specjalna metoda w klasie MapLoader, wywołana przed uruchomieniem pętli głównej programu. Wczytywana mapa jest w postaci siatki i każdy jej element ma stały rozmiar. Symulator rozróżnia kilka typów węzłów:

- brak elementu - możliwy przejazd
- ściana - przeszkoda do ominięcia przez algorytm
- punkt początkowy - od niego rozpoczyna się wyznaczanie ścieżki, jest zdefiniowany przez użytkownika
- punkt końcowy - analogicznie jak w poprzednim przypadku - ścieżka - element wyznaczonej tras

Symulator robota pobiera pierwszy punkt, do którego ma dotrzeć. Zmiana pozycji odbywa się poprzez regulator typu P z ustawioną nastawą na wskazany punkt. Wartość wzmocnienia proporcjonalnego wczytana jest z pliku i oznaczona jest jako szybkość robota. Dalszy etap to sprawdzenie czy symulowany robot dotarł do celu.

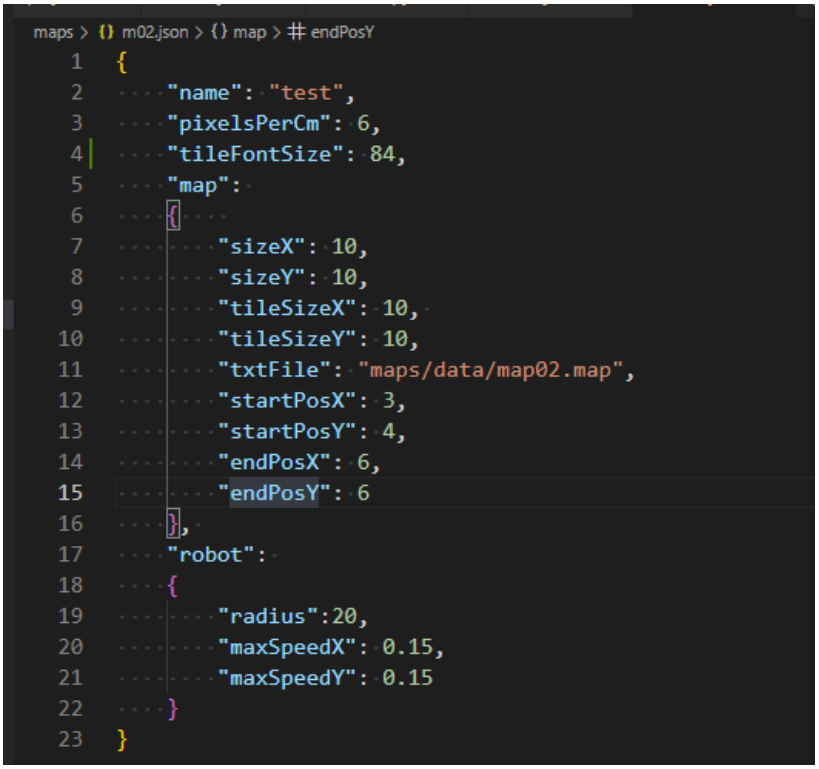
```
if abs(deltaPos0) < 0.02 and abs(deltaPos1) < 0.02:
            self._currentTarget -= 1
```

Listing 4. Zaliczenie punktu

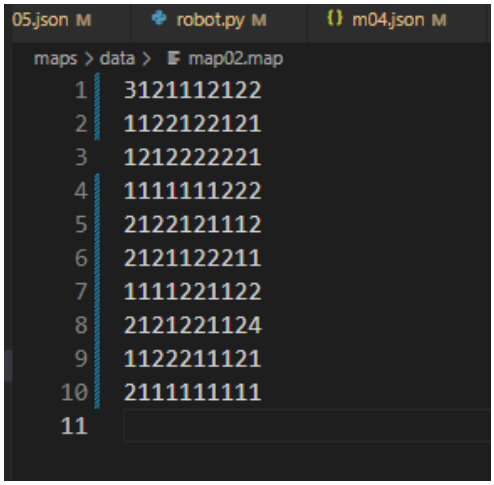
Realizowane jest to poprzez sprawdzenie czy wartość bezwzględna z różnicy aktualnej oraz docelowej pozycji jest mniejsza niż 0,1. Po osiągnięciu punktu, ustawiany jest nowy indeks tablicy wskazujący na kolejny cel. Jeżeli na początku aktualizacji indeks jest ujemny to robot przejechał po całej wyznaczonej trasie.

Cała ta symulacja robota zamknięta jest w jednej klasie Robot i metodzie update, dzięki czemu możliwe jest dodanie do symulatora własnej implementacji uwzględniającą na przykład bardziej zawansowaną fizykę. W innym przypadku możemy utworzyć taką klasę, która łączy się z rzeczywistym robotem i wysyła mu odpowiednie komend pozwalające na poruszanie się w przestrzeni.

Każda mapa wskazywana jest poprzez plik w formacie json, który zawiera 3 sekcje. Pierwsza sekcja to ustawienia ogólne jak np. pixele na 1cm (każdy kolejny wymiar podawany jest w cm). Druga sekcja zawiera parametry mapy takie jak jej rozmiar w blokach, rozmiar pojedynczego bloku i pozycja początkowa i końcowa. Dane dotyczące reprezentacji mapy zapisane są w pliku txt wskazanym przez pole txtFile. Sekcja robota posiada ustawienia o jego promieniu i prędkości.



Rysunek 2. Informacje o mapie wczytywane przez program



Rysunek 3. Plik z zapisaną mapą

Mapa reprezentowana jest poprzez wskazanie liczby oznaczającej typ poszczególnego obiektu. Mapa ładowana jest w trakcie uruchamiania programu i zapisywana jest w trakcie jego wyłączania.

3. Przedstawienie działania programu

Rysunek 4. Widok po uruchomieniu programu

Powyższy zrzut pokazuje aplikacje zaraz po uruchomieniu. Po prawej stronie widoczny jest interfejs użytkownika z kilkoma przyciskami a po lewej wczytana mapa, z oznaczonymi przeszkodami do ominięcia oraz punktem startowym i końcowym.

Rysunek 5. Program w trakcie symulacji, domyślne ustawienia

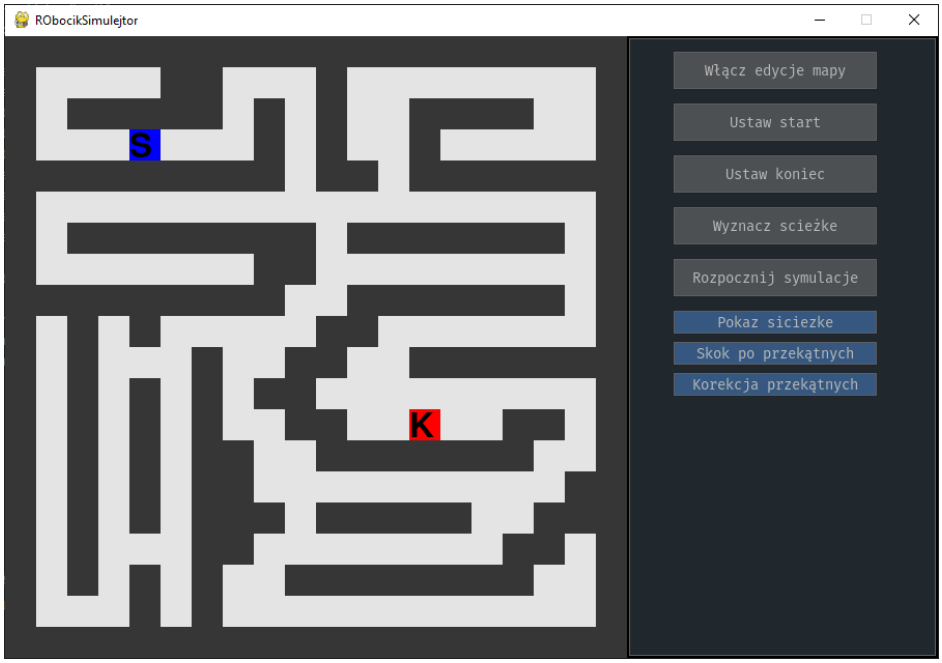
Po wciśnięciu przycisku Rozpocznij symulacje, automatycznie wyznaczana jest ścieżka a dalej uruchomiana jest symulacja przejazdu robota po danej ścieżce. Prędkość i promień robota ustawione są w pliku konfiguracyjnym mapy. Po dojechaniu do punktu końcowego symulacja jest kończona.

Rysunek 6. Ścieżka bez korekcji przekątnych

Powyżej widoczna jest ta sama mapa z wyznaczoną ścieżką bez korekcji przekątnych. Widać że robot przechodzi po przekątnych nawet jeżeli po obu bokach ma przeszkody. Takie rozwiązanie nie zadziałało by w rzeczywistym przypadku.

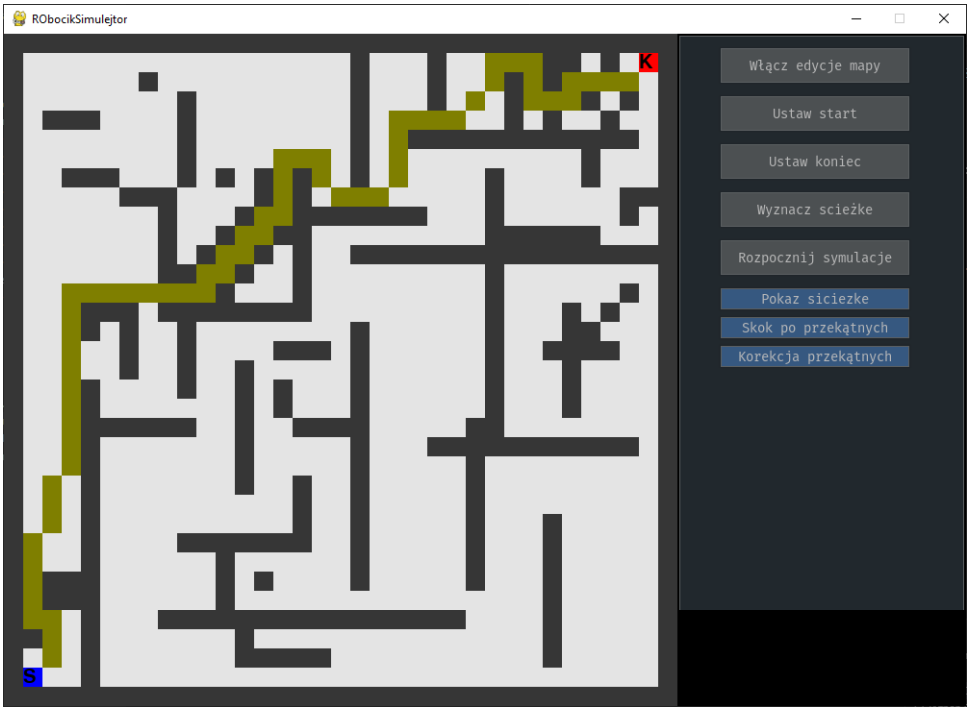
Rysunek 7. Wyłączony skok po przekątnych

Powyżej widoczna jest ścieżka z wyłączonym skokiem po przekątnych. Można zauważyć że ścieżka jest automatycznie dużo dłuższa.



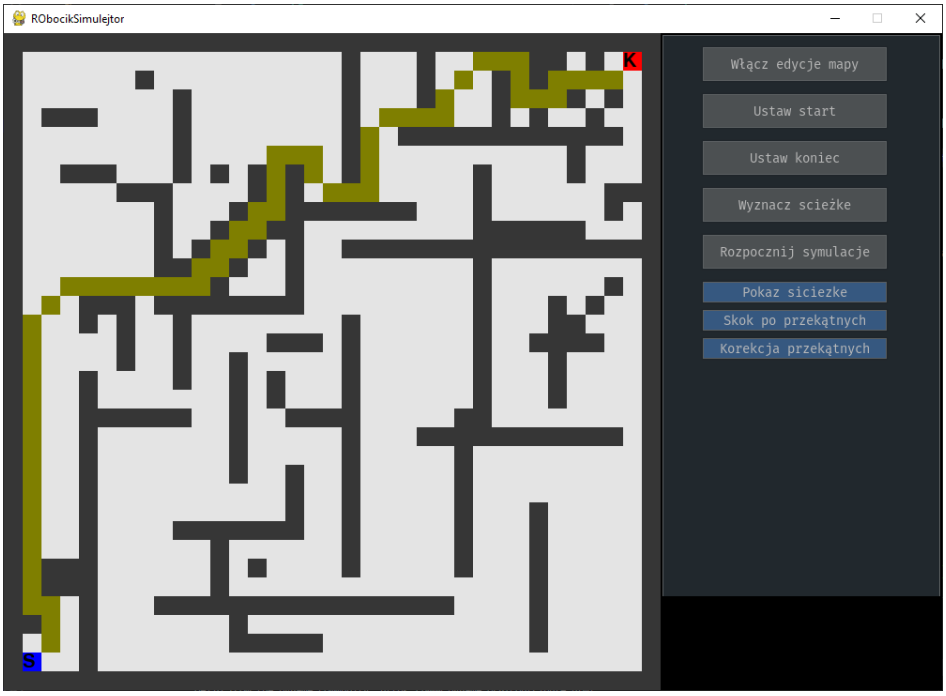
Rysunek 8. Inna wczytana mapa

Mapy ładowane są z pliku co oznacza że w zależności od symulowanego środowiska można wczytać mapę o innym kształcie lub rozmiarze.



Rysunek 9. Heurestyka eukalidesowa

Powyżej widać że w przypadku heurestyki eukalidesowej rozwiązanie jest optymalne.



Rysunek 10. Funkcja heurestyczna to kwadrat różnicy dwóch punktów

W przypadku funkcji heurestycznej z sumą różnic kwadratów początek ścieżki jest linią prostą i przejście po skosie jest jednolite na końcu prostej linii. W przypadku rozwiązań rzeczywistych stałość ścieżki może wpływać na osiąganą prędkość robota.

4. Wnioski

Analizując wyniki otrzymane podczas testowania algorytmu możemy zauważyć że wyznaczona ścieżkę jest optymalna. Podczas testów sprawdzono wpływ różnych funkcji heurestycznych na otrzymaną ścieżkę. Na ich podstawie widać że takie funkcje mają duże znaczenie i w zależności od skomplikowania otoczenia, odległości oraz dostępnych ruchów(możliwość przejścia po przekątnej) otrzymujemy ścieżki o różnym poziomie skomplikowania. Zaimplementowanie algorytmu w pythonie pozwoliło na szybkie i sprawne jego przetestowanie, jednak w przypadku dużych i skomplikowanych map można było odczuć coraz dłuższe przeliczanie. W znacznej mierze wynika to ze ścisłego połączenia algorytmu z symulatorem i co za tym idzie wymuszone ciągłe przeszukiwanie tablicy z przeszkodami.