Dokumentacja

Plikiem z kodem źródłowym jest plik **zad_bug_final.py**. Uruchamiając program należy podać następujące parametry:

-x – współrzędna x punktu docelowego

-y – współrzędna y punktu docelowego

-a – żądany algorytm(możliwe wartości: ‘bug1’, ‘alg1’, ‘rev1’)

We wszystkich grafach i opisach umieszczonych poniżej użyto następujących oznaczeń:

S – punkt startowy(punkt z którego robot zaczyna swój ruch)

T – punkt docelowy

Oi – i-ta przeszkoda napotkana przez robota

Li – i-ty punkt opuszczenia przeszkody

Hi – i-ty punkt zderzenia z przeszkodą

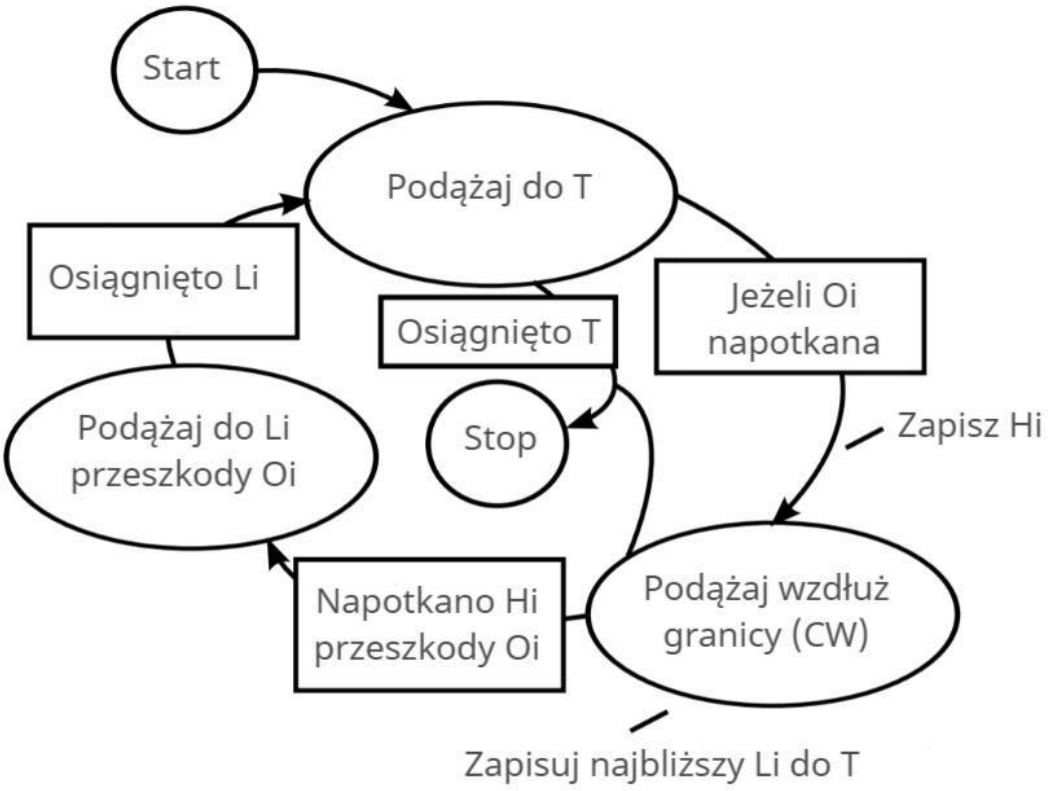
Di – kierunek otaczania i-tej przeszkody(0 – CW, 1 - CCW)

CW(Clockwise) – zgodnie z ruchem wskazówki zegara

CCW(Counterclockwise) – przeciwko ruchu wskazówki zegara

Bug1

Graficzny schemat działania algorytmu został przedstawiony poniżej:



Algorytm Bug1 jest jednym z najprostszych algorytmów typu BUG. Zasada działania polega na tym, że robot podąża do punktu docelowego(T), jeżeli napotyka przeszkodę zapamiętuje punkt w którym nastąpiło wykrycie tej przeszkody(Hi) i zaczyna ją otaczać. W trakcie otaczania robot cały czas wylicza dystans do punktu T, jeżeli jest on mniejszy niż dystans od punktu zapisanego wcześniej, robot ustawia nowy punkt opuszczenia przeszkody(Li). Gdy robot po otoczeniu przeszkody wraca do punktu Hi podąża stąd wzdłuż przeszkody do wyliczonego wcześniej Li, gdzie kończy ruch wzdłuż przeszkody i rozpoczyna ruch do punktu docelowego. W przypadku wykrycia nowej przeszkody algorytm się powtarza do momentu osiągnięcia punktu docelowego.

Droga, którą pokona robot nigdy nie przekroczy limitu:

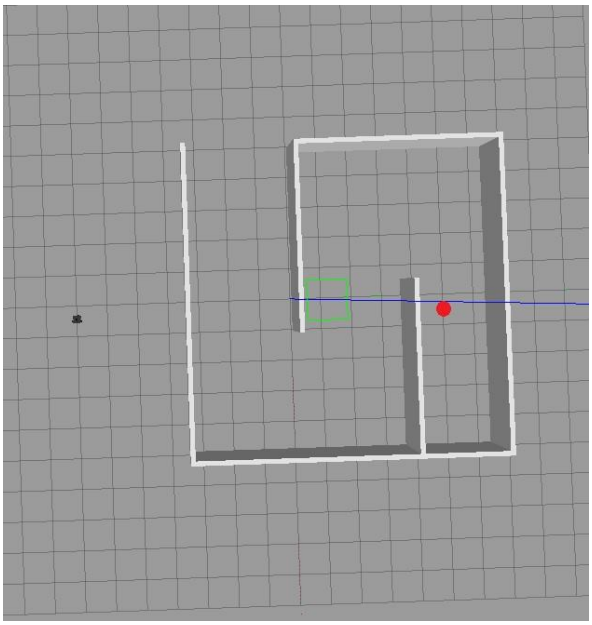
$$P = d(S,T) + 1.5 \cdot \sum p_i$$, gdzie

P – cała droga, d(S,T) - bezpośrednia odległość między punktem startowym(S) i docelowym(T), p_i – długość granicy i-tej przeszkody.

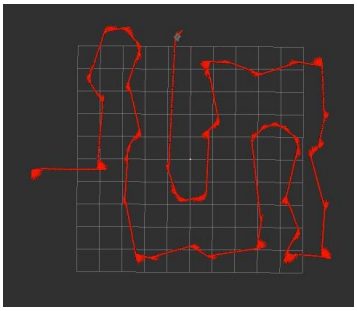
W swojej implementacji bazowałem na opisie algorytmu przedstawionym powyżej. Komentarze i opis mojej implementacji znajdują się w pliku z kodem źródłowym. Z powodu długiego czasu wykonywania algorytmu ścieżka w R-viz zaczynała znikać i by pokazać tor ruchu musiałem zrobić kilka zrzutów ekranu.

Dla labiryntu 1(czerwoną kropką jest oznaczony punkt docelowy(0.5, 5)):

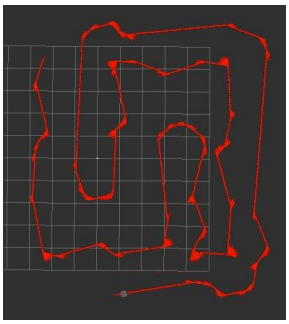
```
student@wrwaet:~$ python ~/wr_ws/src/wr_init/zad_bug_final.py -x 0.5 -y 5 -a bug1
```



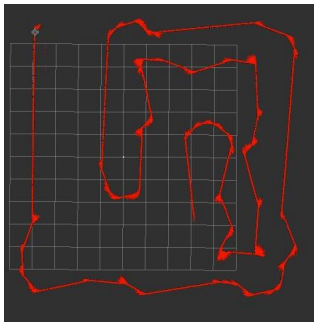
Labirynt 1



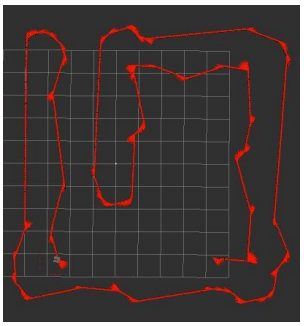
1



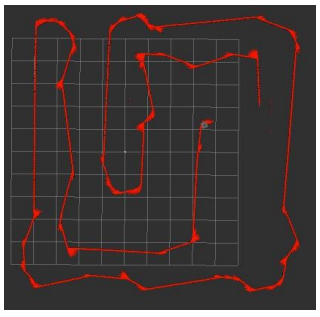
2



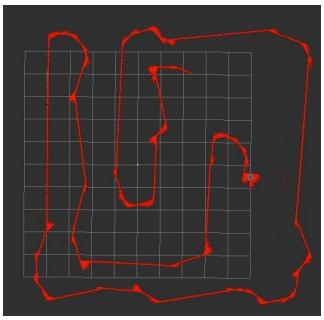
3



4



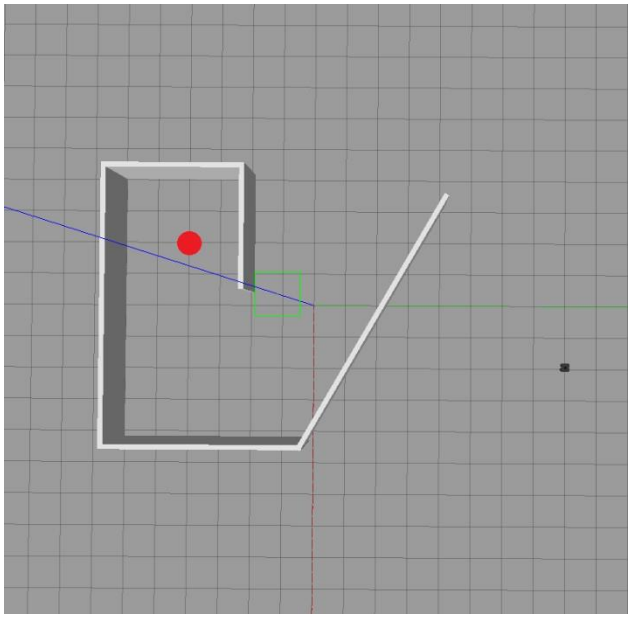
5



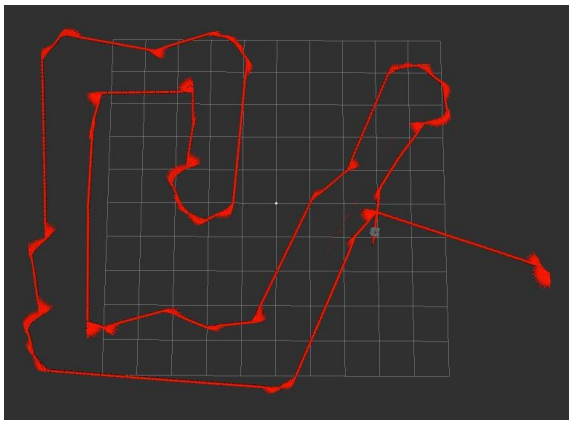
6

Dla labiryntu 2(czerwoną kropką jest oznaczony punkt docelowy(-2, -4)):

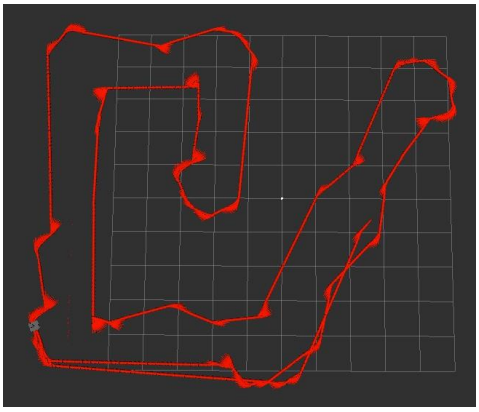
```
student@wrwaet:~$ python ~/wr_ws/src/wr_init/zad_bug_final.py -x -2 -y -4 -a bug1
```



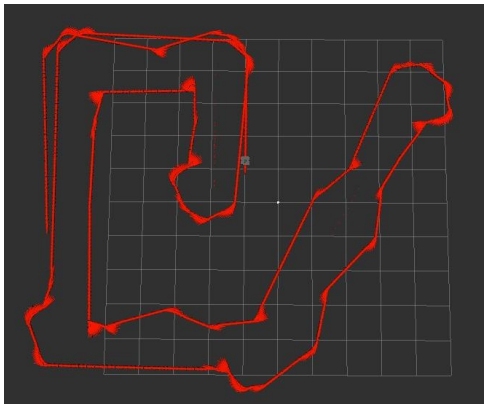
Labirynt 2



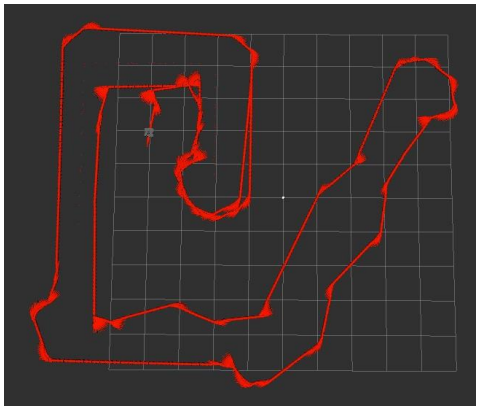
1



2



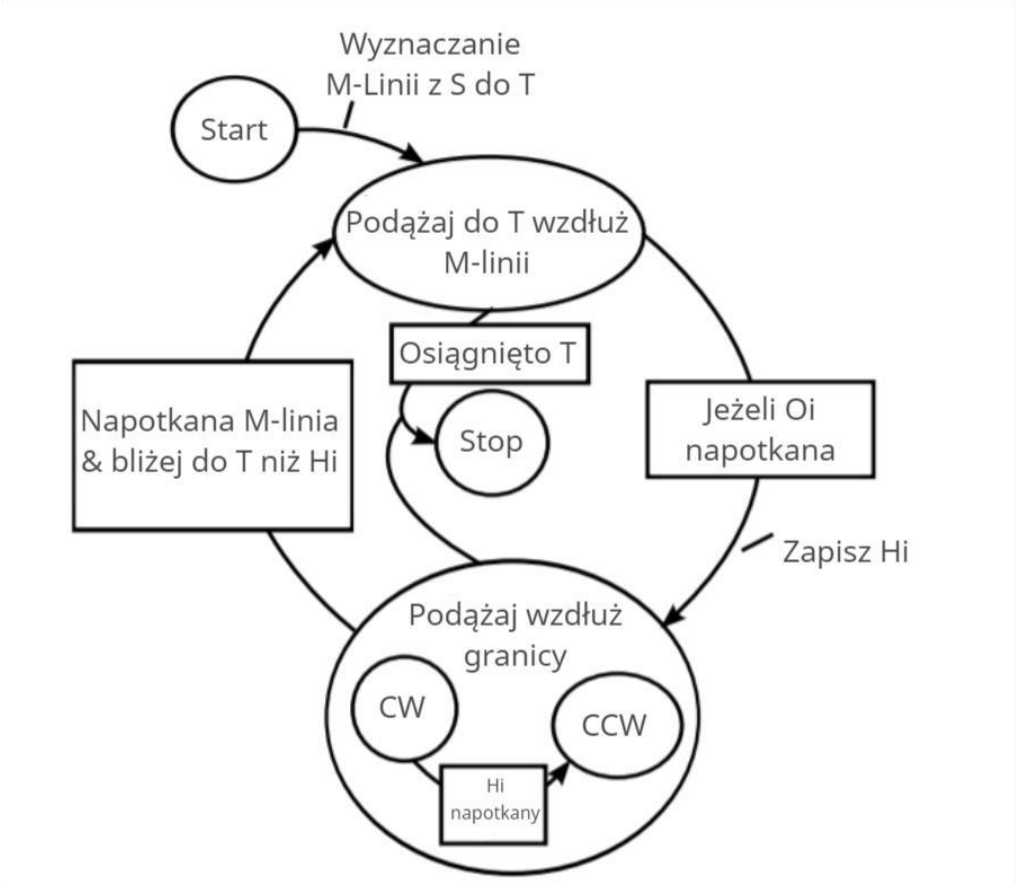
3



4

Alg1

Graficzny schemat działania algorytmu został przedstawiony poniżej:



Zasada działania algorytmu Alg1 polega na tym, że najpierw robot wyznacza M-Linię, jest to linia, łącząca punkt startowy(S) i docelowy(T). Po wyznaczeniu M-linii robot podąża do punktu T wzdłuż tej linii, jeżeli napotyka przeszkodę - zapamiętuje punkt w którym nastąpiło wykrycie tej przeszkody(Hi) i zaczyna ją otaczać. W trakcie otaczania robot cały czas sprawdza czy nie jest znów na M-linii i jeżeli jest i dystans do punktu T jest mniejszy niż d(Hi, T) – robot podąża do punktu docelowego wzdłuż M-linii. Jeżeli podczas otaczania przeszkody robot powróci do punktu Hi – zmienia kierunek otaczania przeszkody. W przypadku wykrycia nowej przeszkody algorytm się powtarza do momentu osiągnięcia punktu docelowego.

Droga, którą pokona robot nigdy nie przekroczy limitu:

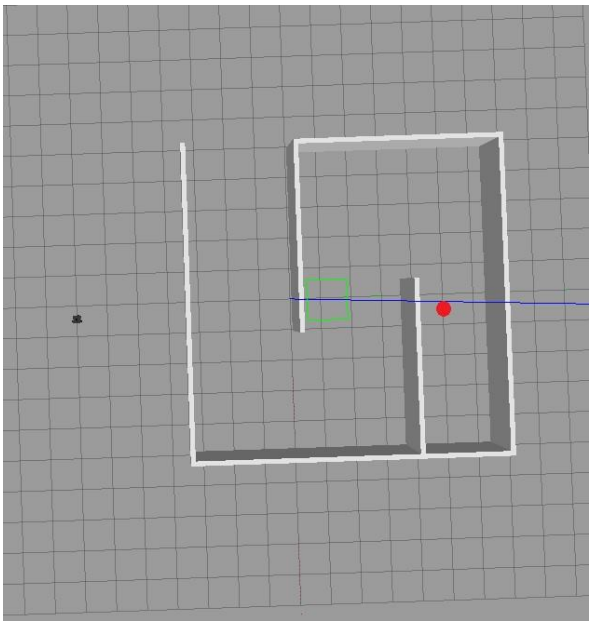
$$P = d(S,T) + 2 \cdot \sum p_i$$, gdzie

P – cała droga, d(S,T) - bezpośrednia odległość między punktem startowym(S) i docelowym(T), p_i – długość granicy i-tej przeszkody.

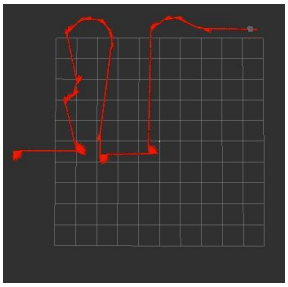
W swojej implementacji bazowałem na opisie algorytmu przedstawionym powyżej. Komentarze i opis mojej implementacji znajdują się w pliku z kodem źródłowym. Z powodu długiego czasu wykonywania algorytmu ścieżka w R-viz zaczynała znikać i by pokazać tor ruchu musiałem zrobić kilka zrzutów ekranu.

Dla labiryntu 1(czerwoną kropką jest oznaczony punkt docelowy(0.5, 5)):

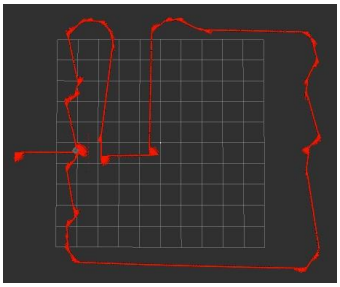
```
student@wrwaet:~$ python ~/wr_ws/src/wr_init/zad_bug_final.py -x 0.5 -y 5 -a alg1
```



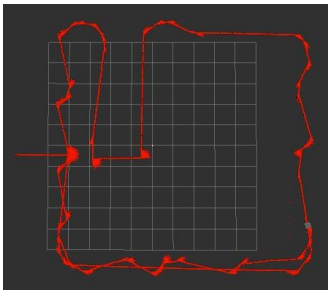
Labirynt 1



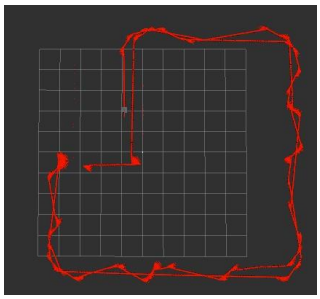
1



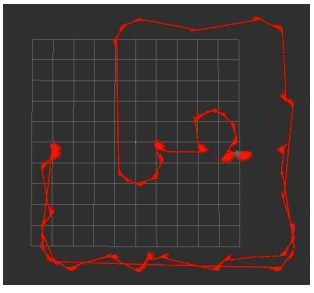
2



3



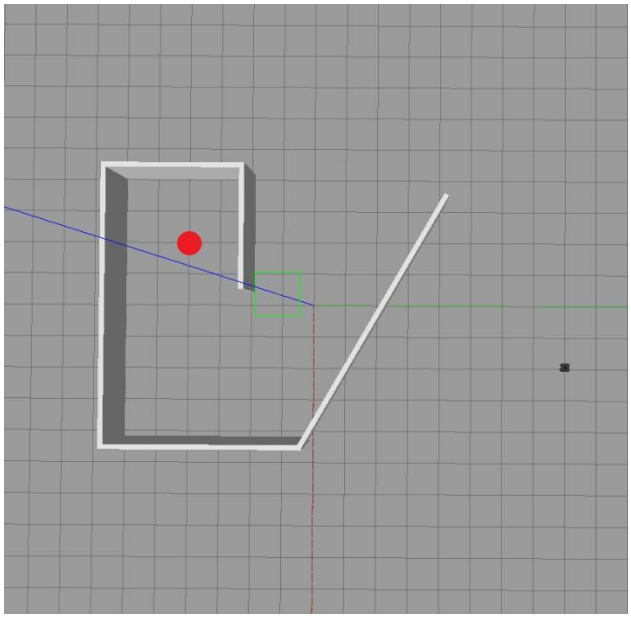
4



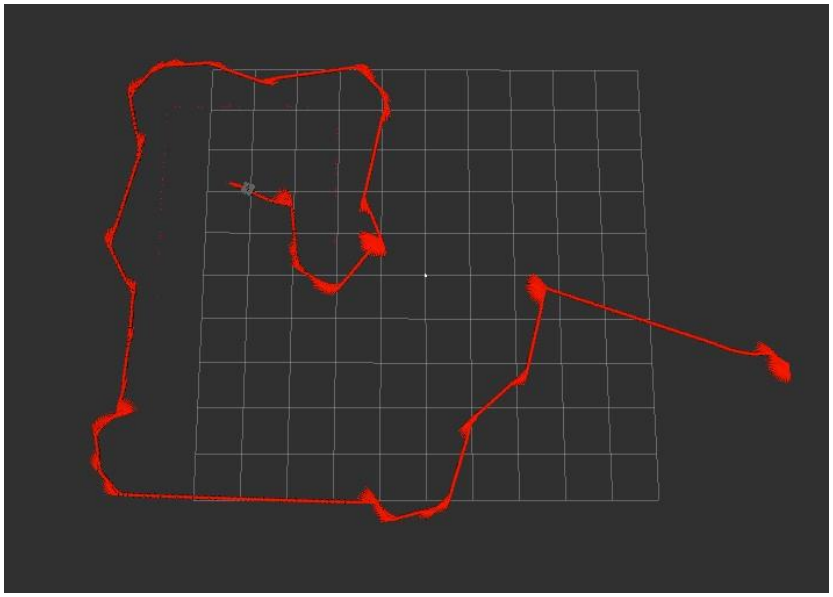
5

Dla labiryntu 2(czerwoną kropką jest oznaczony punkt docelowy(-2, -4)):

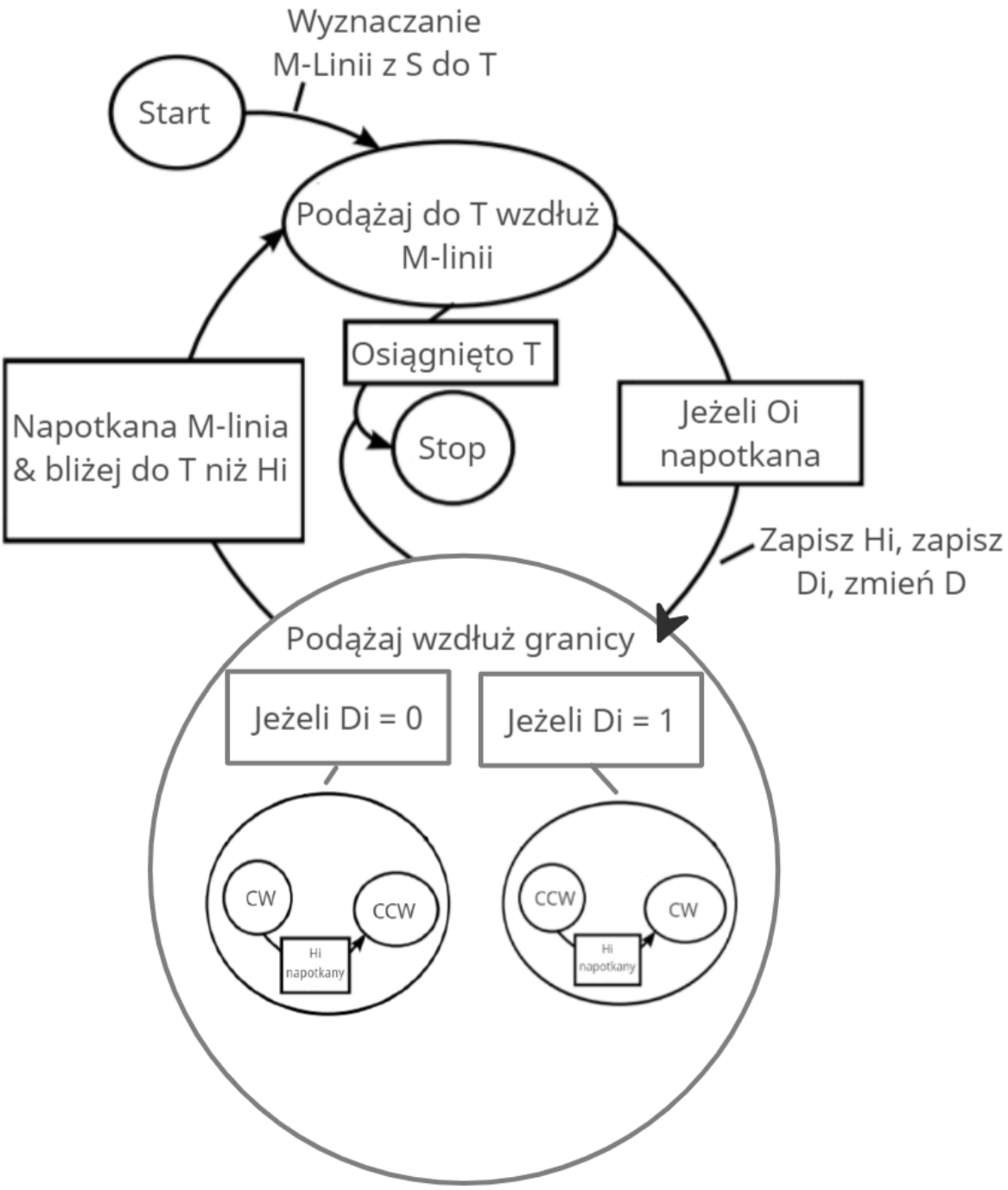
```
student@wrwaet:~$ python ~/wr_ws/src/wr_init/zad_bug_final.py -x -2 -y -4 -a alg1
```



Labirynt 2



Graficzny schemat działania algorytmu został przedstawiony poniżej:



Zasada działania algorytmu Rev1 polega na tym, że najpierw robot wyznacza M-Linię, jest to linia, łącząca punkt startowy(S) i docelowy(T). Po wyznaczeniu M-linii robot podąża do punktu T wzdłuż tej linii, jeżeli napotyka przeszkodę - zapamiętuje punkt w którym nastąpiło wykrycie tej przeszkody(Hi) i kierunek w którym zaczął otaczać przeszkodę(Di) oraz zmienia wartość zmiennej globalnej, odpowiadającej za wybór kierunku otaczania - na przeciwną(co pozwala na przemian zmieniać kierunek otaczania przeszkód). W trakcie otaczania robot cały czas sprawdza czy nie jest znów na M-linii i jeżeli jest i dystans do punktu T jest mniejszy niż d(Hi, T) – robot podąża do punktu docelowego wzdłuż M-linii. Jeżeli podczas otaczania przeszkody robot powróci do punktu Hi – sprawdza w jakim kierunku już otaczał tą przeszkodę i zmienia go na przeciwny. W przypadku wykrycia nowej przeszkody algorytm się powtarza do momentu osiągnięcia punktu docelowego Droga, którą pokona robot nigdy nie przekroczy limitu:

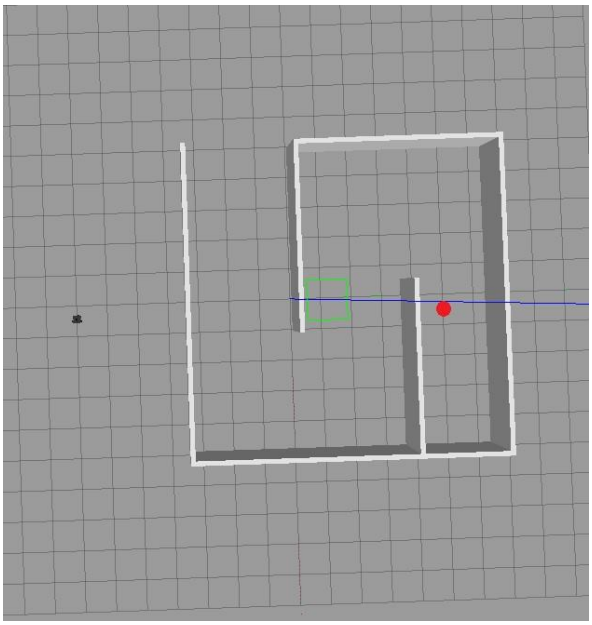
$$P = 2 \cdot d(S, T) + 2 \cdot \sum p_i \quad \text{, gdzie}$$

P – cała droga, d(S,T) - bezpośrednia odległość między punktem startowym(S) i docelowym(T), p_i – długość granicy i-tej przeszkody. W swojej implementacji bazowałem na opisie algorytmu przedstawionym powyżej.

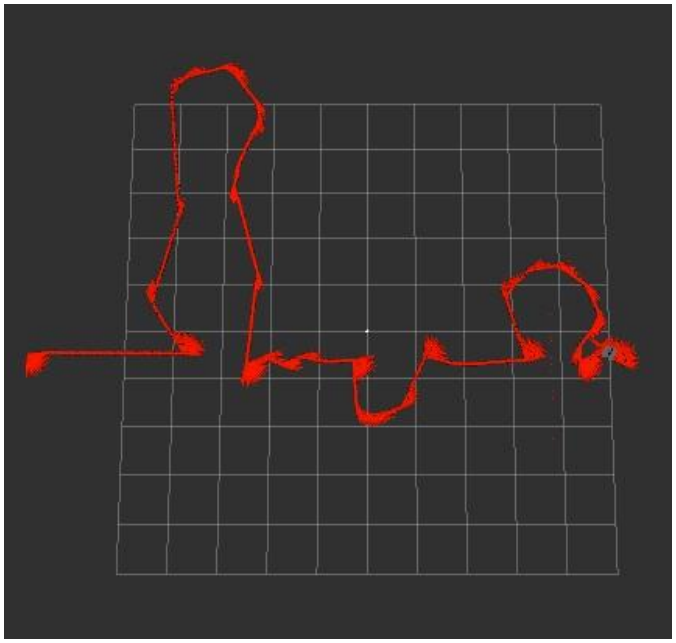
Komentarze i opis mojej implementacji znajdują się w pliku z kodem źródłowym. Z powodu długiego czasu wykonywania algorytmu ścieżka w R-viz zaczynała znikać i by pokazać tor ruchu musiałem zrobić kilka zrzutów ekranu.

Dla labiryntu 1(czerwoną kropką jest oznaczony punkt docelowy(0.5, 5)):

```
student@wrwaet:~$ python ~/wr_ws/src/wr_init/zad_bug_final.py -x 0.5 -y 5 -a rev1
```

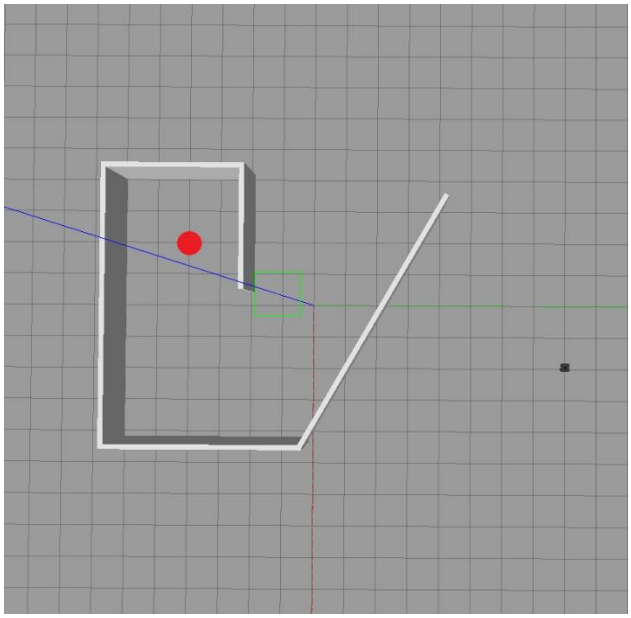


Labirynt 1

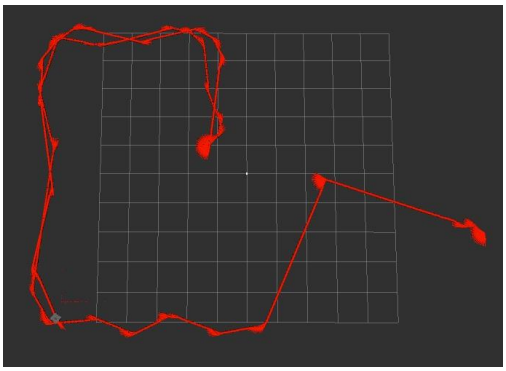


Dla labiryntu 2(czerwoną kropką jest oznaczony punkt docelowy(-2, -4)):

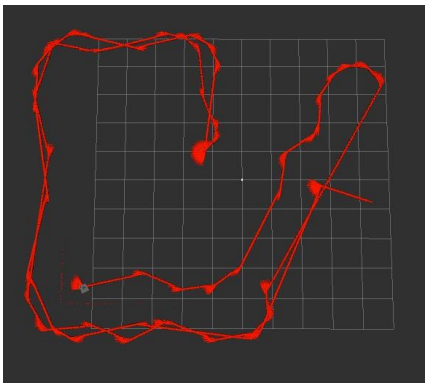
```
student@wrwaet:~$ python ~/wr_ws/src/wr_init/zad_bug_final.py -x -2 -y -4 -a rev1
```



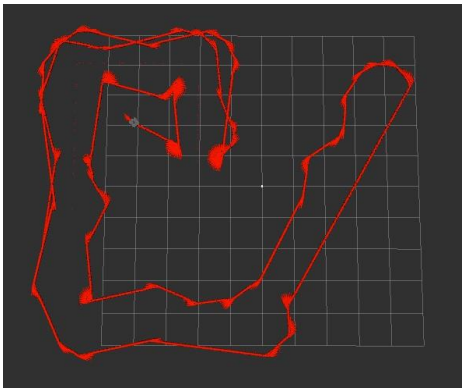
Labirynt 2



1



2



3

Podsumowanie

Jak widać na podstawie nawet tych 2 labiryntów nie da się stwierdzić, który algorytm BUG jest lepszy, bo np. dla labiryntu 1 jest to **rev1**, dla labiryntu 2 lepszy jest **rev1**, a patrząc na maksymalną możliwą drogę – lepszy jest **bug1.** Dla każdego algorytmu możemy znaleźć środowisko, w którym będzie on najlepszym, jednak nie możemy jednoznacznie określić który z tych algorytmów jest lepszy, nie mając informacji o środowisku.

Dla każdego algorytmu robot pokonywał labirynt w sposób zgodny z oczekiwaniami, co świadczy o poprawności algorytmów. Lecz brakowało mu płynności, czego objawem jest trajektoria w postaci linii łamanej. Wynika to z niedoskonałości algorytmu śledzenia ściany.

Kolejnym minusem jest czas pokonania labiryntu, dla algorytmu **bug1** jest to ~24 min. dla labiryntu 1 i ~18 min. dla labiryntu 2. Możliwie, że wynika to ze stosunku skali labiryntu do skali i prędkości robota. Gdy zwiększałem prędkość – algorytm, sterujący robotem przestawał działać, powodem tego może być inercja, ale nie jestem pewien. Mówiąc o algorytmie sterowania w większości wypadków działa dobrze, ale czasami, gdy np. robot musi dojechać do punktu docelowego i nie ma na jego drodze żadnych przeszkód, zamiast linii prostej porusza się linią łamaną.

W trakcie realizacji zadań zapoznałem się z podstawami ROS'a. Nauczyłem się implementować proste algorytmy sterowania robotem. Wykorzystałem zdobytą wcześniej wiedzę z zakresu programowania w Python'ie.