



**WYDZIAŁ
BUDOWY MASZYN
I LOTNICTWA**
POLITECHNIKI RZESZOWSKIEJ

Damian Bielecki

Implementacja sterowania hierarchicznego mobilnym robotem
kołowym z wykorzystaniem języka Python

Praca dyplomowa inżynierska

Opiekun pracy:
dr. Paweł Penar

Rzeszów, 2023

Spis treści

1. Wprowadzenie	5
2. Przegląd literatury	6
2.1. Przegląd istniejących rozwiązań	6
2.2. Opis przyjętego rozwiązania	6
3. Algorytm A*	8
3.1. Geneza powstania	8
3.2. Opis działania algorytmu	8
4. Implementacja	10
4.1. Implementacja algorytmu	10
4.2. Implementacja środowiska testowego	12
5. Projekt robota	17
5.1. Założenia projektowe	17
5.2. Projektowanie zarysu robota w środowisku CAD	17
5.3. Projekt, wykonanie oraz podłączenie elektroniki robota	19
5.4. Oprogramowanie robota	22
5.4.1. Sterowanie silnikami	22
5.4.2. Serwer TCP	24
6. Przeprowadzone testy	27
6.1. Test poprawności trasowania	27
6.2. Wpływ ustawień na wyznaczoną ścieżkę	28
6.3. Sprawdzenie wpływu funkcji heurystycznej na ścieżkę	29
6.4. Test komunikacji z fizycznym robotem	29
Literatura	30

1. Wprowadzenie

Od lat można zauważyć zwiększającą się popularność różnych robotów, które bazując na wprowadzonej do systemu mapie autonomicznie poruszają się po terenie tak aby osiągnąć określony cel. Celem projektu inżynierskiego będzie zaimplementowanie algorytmu wyszukującym najkrótszą ścieżkę. Do przetestowania algorytmu zostanie napisany prosty program symulujący robota poruszającego się po wyznaczonej ścieżce. Kolejnym etap projektu to zbudowanie robota mającego zweryfikować zaproponowaną przez algorytm ścieżkę.

Głównym powodem podjęcia się implementacji takiego algorytmu jest chęć zapoznania się z systemami planującymi ruch robota i próba zaimplementowania takiego rozwiązania na fizycznym robocie.

Na zakres pracy składa się:

- Przegląd literatury
- Implementacja algorytmu A* w języku Python
- Symulacja działania pracy algorytmu
- Weryfikacja na obiekcie rzeczywistym

Omówienie rozdziałów

Rozdział pierwszy - zawiera przegląd literatury oraz istniejących rozwiązań. Na ich podstawie określona zostanie ogólna struktura własnego rozwiązania.

Rozdział drugi - skupia się na genezie, charakterystyce i opisie działania wybranego algorytmu wyszukiwania najkrótszej ścieżki. Dodatkowo zostaną przedstawione inne algorytmy zwiększającą autonomię robota mobilnego.

Rozdział trzeci - w tym rozdziale zostanie przedstawiony napisany algorytm oraz środowisko symulacyjne w języku Python.

Rozdział czwarty - omawia budowę robota mobilnego oraz jego program sterujący. W szczególności skupiono się na komunikacji oraz sterowaniu silnikami.

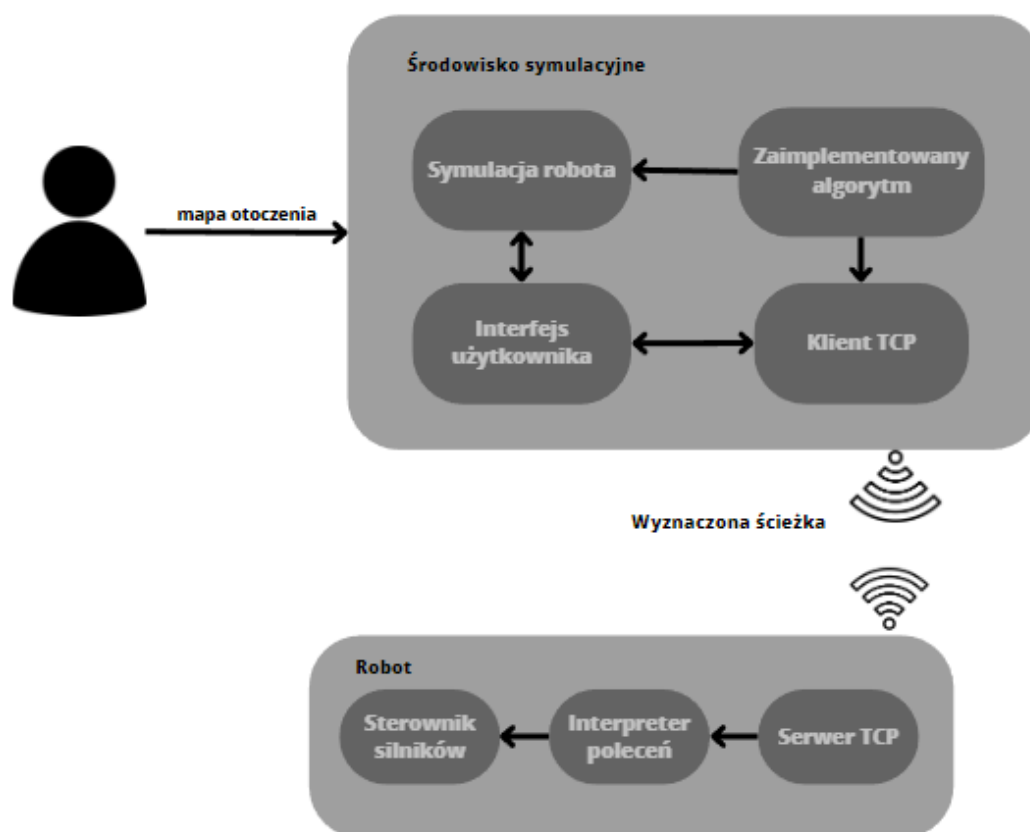
Rozdział piąty - przedstawia przeprowadzone testy wykonane w środowisku symulacyjnym oraz rzeczywistym.

2. Przegląd literatury

2.1. Przegląd istniejących rozwiązań

2.2. Opis przyjętego rozwiązania

Algorytm wyznaczania najkrótszej ścieżki wraz z środowiskiem testowym został napisany w języku Python. Implementacja algorytmu nie wymaga żadnych dodatkowych bibliotek. Do zbudowania środowiska symulacyjnego zostały użyte biblioteki graficzne do rysowania kształtów geometrycznych oraz podstawowych elementów graficznego interfejsu użytkownika. Weryfikacja działania algorytmu została przeprowadzona na zbudowanym trójkątowym robocie. Oprogramowanie robota zostało napisane w języku C++ i dodatkowych bibliotekach udostępnionych przez producentów użytych sterowników.



Rysunek 2.1: Ogólny schemat projektu

Użytkownik po uruchomieniu środowiska symulacyjnego będzie miał możliwość

edycji mapy opartej na siatce, określenia punktu początkowego i końcowego. Kolejnym krokiem jest wyznaczenie ścieżki oraz jej graficznej reprezentacji. Tak wyznaczoną ścieżkę można wysłać do robota operującego w środowisku rzeczywistym.

3. Algorytm A*

3.1. Geneza powstania

Algorytm A* powstał w ramach projektu Shakey, zapoczątkowanego w 1966 roku przez Charles Rosen'a. Celem projektu było zbudowanie robota, który potrafiłby planować własne działania. Zbudowany robot wyróżniał się na tle innych tym że integrował kilka różnych modeli sztucznej inteligencji pracującej jako jeden system.

Robot był zbudowany z: - kamery telewizyjnej i dalmierza optycznego - system wizyjny do obserwacji środowiska - łącze radiowe - służącego do komunikacji z bazą, odbierania i wysyłania komend - detektor uderzeń - pozwalający na zatrzymanie robota w przypadku kolizji

Komunikacja odbywała się poprzez wysyłane radiowo, tekstowe polecenia mające określoną strukturę np.: GOTO D4 - co oznaczało automatyczne przemieszczenie się robota do wskazanej pozycji



Rysunek 3.2: Robot Shakey i Charles Rosen, inicjator projektu [2]

3.2. Opis działania algorytmu

A* to heurystyczny 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:

$$f(x) = h(x) + g(x) \quad (3.1)$$

gdzie: $f(x)$ - minimalizowana funkcja, $g(x)$ - to rzeczywisty koszt dojścia do punktu x .

Funkcja $h(x)$ to funkcja heurystyczna, oszacowuje ona koszt dotarcia od punktu x do wierzchołka docelowego

Zalety:

- jest kompletny i optymalny
- może przeszukiwać skomplikowane mapy
- jest najwydajniejszym takim algorytmem

Wady:

- jego wydajność w znacznej mierze zależy od funkcji heurystycznej
- Każda akcja ma stały koszt wykonania
- nie nadaje się do często zmieniającego się otoczenia robota, wymaga ponownego przeliczenia

Przykładowe funkcje heurystyczne:

- Funkcja euklidesowa

$$h(x) = 10 * \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (3.2)$$

- Geometria Manhattanu (innaczej metryka miejska)

$$h(x) = |x_2 - x_1| + |y_2 - y_1| \quad (3.3)$$

Gdzie: x_1 i y_1 to współrzędne wyznaczanego punktu, x_2 i y_2 to koordynaty celu

4. Implementacja

4.1. 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 dodatkowa klasa Node reprezentująca 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.

```
1  openList = []
2  startNode.h = self.heuristic(startNode.getCords(), endNode.
   getCords())
3  startNode.g = 0
4  heappush(openList, startNode)
```

Listing 1: Przygotowanie danych

Wewnętrzna funkcja heuristic 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 heuristiczna to równanie nr. (3.2)

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 są wszyscy jego sąsiedzi i pętla przechodzi po pobranych sąsiadach aktualnego punktu. W tym rozwiązaniu dla zaoszczędzenia pamięci, symulator przechowuje 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.

```
1 neighborNode = self._findNodeOnList(openList, newCords)
2 if neighborNode is None:
3     neighborNode = Node(newCords)
4     neighborNode.g = COST
5     neighborNode.h = self.heuristic(newCords, endNode.getCords())
   )
```

```

6     neighborNode.parent = currentNode
7     openList.append(neighborNode)
8
9     distance_from_curr_to_neighbor = COST
10    scoreFromStartToCurrentNeighbor =  currentNode.g +
        distance_from_curr_to_neighbor
11
12    if neighborNode.getScore() <= currentNode.getScore():
13        neighborNode.g = scoreFromStartToCurrentNeighbor
14        neighborNode.h = scoreFromStartToCurrentNeighbor + self.
        heuristic(newCords, endNode.getCords())
15        neighborNode.parent = currentNode

```

Listing 2: Wyznaczenie kosztu ścieżki

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 heurystycznej. 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 10. 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.

```

1    if currentNode == endNode:
2        path = []
3        while currentNode is not None:
4            path.append(currentNode)
5            currentNode = currentNode.getParent()
6        return path
7    else:
8        return []

```

Listing 3: Przygotowanie danych

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.

4.2. Implementacja środowiska testowego

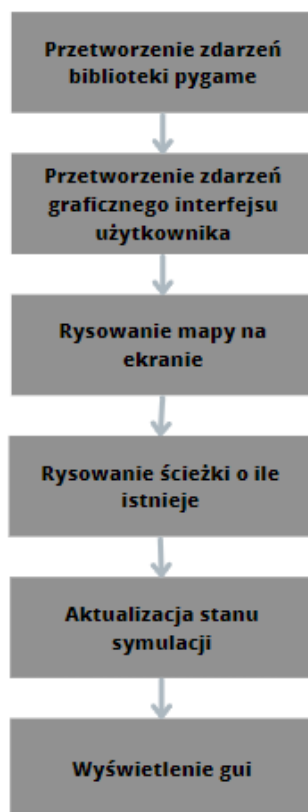
Środowisko symulacyjne zostało napisane w języku Python i bibliotekach pygame[3] i pygame_gui[4], zapewniających obsługę graficznego interfejsu użytkownika.

```
1 if __name__ == '__main__':  
2     app = App()  
3     app.main()
```

Listing 4: Uruchomienie aplikacji

Obsługa mapy

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 4.3: Schemat głównej pętli programu

Aby uprościć sobie 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 informacje o samej mapie, natomiast druga o poruszającym się po niej robocie. Sekcja mapy zawiera lokalizacje do pliku tekstowego z zapisanymi elementami.

```
1 (self.map, self.robot) = MapLoader.fromJson("maps/m02.json")
```

Listing 5: Uruchomienie aplikacji

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 trasy

Wszystkie wymienione typy zapisane są w klasie Tile, reprezentującej pojedynczy rysowany i zapisywany w pliku kwadrat. Użytkownik może w każdej chwili działania aplikacji edytować mapę a przed zamknięciem programu zaktualizowana mapa jest zapisywana do pliku. Wyjątkiem jest obiekt będący elementem ścieżki, ten nie jest zapisywany do pliku, ponieważ generowany jest dynamicznie i służy tylko do reprezentacji wygenerowanej ścieżki.

Wyznaczanie ścieżki i symulacja

Po załadowaniu program oczekuje na akcje użytkownika i ewentualne uruchomienie wyznaczania ścieżki lub całej symulacji.

```
1 elif ev.ui_element == self._startAStar:
2     star = AStar(self.map.map, self.map.getSize())
3     star.diagonalJump(self._diagonalJump)
4     star.diagonalCorrection(self._diagonalCorrection)
5     self.path = star.findPath2(Node(self.map.getStartCords()), Node((self.map.getEndCords())))
```

```

6 self._pathExist = True
7 self.robot.stopSimulation()
8 self.robot.hideRobot()

```

Listing 6: Uruchomienie aplikacji

Po kliknięciu w przycisk odpowiadający za wyznaczenie i pokazanie ścieżki, program tworzy obiekt wyznaczający trasę i ustawia dodatkowe opcje algorytmu (np. skok po przekątnych). Dalej pobierana jest wyznaczona trasa będąca tablicą kolejnych punktów, po których należy przejść aby dojść do celu. Na końcu ustawiana jest odpowiednia flaga programu, zatrzymywana jest symulacja i ukrywany jest robot (mamy nowo wyznaczoną trasę). Przycisk uruchamiający symulację sprawdza czy istnieje ścieżka i w razie potrzeby ją generuje. Następnie trasa podawana jest do obiektu robota odpowiadającego za symulację i dalej jest ona uruchamiana.

Przejsie robota po ścieżce

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.

```

1 if abs(deltaPos0) < 0.2 and abs(deltaPos1) < 0.2:
2     self._currentTarget -= 1

```

Listing 7: Uruchomienie aplikacji

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.

Komunikacja z robotem

Po wyznaczeniu ścieżki symulator na polecenie użytkownika może połączyć się ze zbudowanym robotem aby ten osiągnął założony cel. Zgodnie ze schematem [?] robot po połączeniu z siecią WiFi i utworzeniu serwera TCP oczekuje na wysyłane komendy. Aplikacja podczas uruchamiania, tworzy gniazdo, które potem zostanie wykorzystane do połączenia.

```

1 self._robotSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

```

Listing 8: Utworzone gniazdo

Dla zwiększenia elastyczności utworzone są dwa pola tekstowe pozwalające na wprowadzenie adresu ip oraz portu z uruchomionym serwerem. Po wprowadzeniu wymaganych danych operator może spróbować nawiązać połączenie z robotem poprzez specjalny przycisk.

```
1 if self._robotSocketFlag is True:
2     self._robotSocket.close()
3
4 print(f"Laczenie z robotem {(self._textRobotIp.get_text(), int(
5     self._textRobotPort.get_text()))}")
6 self._robotSocket.settimeout(1)
7 try:
8     self._robotSocket.connect((self._textRobotIp.get_text(), int
9     (self._textRobotPort.get_text())))
10    self._robotSocketFlag = True
11    self._sendCommandToRobot("setMode auto")
12    print("Podlaczone!!!")
13 except Exception as err:
14     self._robotSocketFlag = False
15     print(f"Blad polaczenia z robotem")
16     print(err)
```

Listing 9: Nawiązanie połączenia

W pierwszej kolejności sprawdzana jest flaga informująca o stanie połączenia, jeżeli jest aktywne to jest ono zamykane. Na utworzonym gnieździe wywoływana jest funkcja connect z wprowadzonymi przez użytkownika parametrami. Cała operacja zamknięta jest w bloku try..except a więc jeżeli program nie połączy się z robotem to wyświetlana jest odpowiednia informacja. Podczas łączenia należy zwracać uwagę na podsieci, w których jest robot i sterujący laptop. Dodatkowo robot posiada dynamiczne ip co zostało szerzej opisane w rozdziale o oprogramowaniu robota. Jeżeli żaden wyjątek nie został rzucony to wysyłana jest pierwsza komenda ustawiająca tryb robota na automatyczny.

```
1 def _sendCommandToRobot(self, cmd):
2     if self._robotSocketFlag is False:
3         print("Robot nie podlaczony!!")
4         return
5
6     try:
7         self._robotSocket.send(str.encode(cmd))
8     except OSError as err:
9         print("Robot nie podlaczony!!!")
10        self._robotSocket.close()
11        self._robotSocketFlag = False
12        return
```

Listing 10: Wysyłanie komendy do robota

Program może wysłać dowolną komendę robota poprzez metodę `sendCommandToRobot`. Podobnie jak przy połączeniu, metoda wysyłająca dane może rzucić wyjątek o braku połączenia. W takim przypadku wyświetlana jest odpowiednia informacja i resetowana jest flaga połączenia.

Finalna wersja programu

Na poniższym zrzucie widać uruchomiony program z trwającą symulacją. Po prawej stronie został umieszczony panel z interfejsem użytkownika. Po lewej stronie widoczna jest wczytana mapa. Kolorem niebieskim został oznaczony punkt startowy a czerwonym końcowy. Żółte kratki oznaczają wyznaczoną przez algorytm ścieżkę, po której przejdzie robot. Szare pola to przeszkody, które robot ma ominąć. Robot oznaczony jest przez jasno-zielone koło, które przesuwa się po mapie.



Rysunek 4.4: Widok finalnej wersji programu

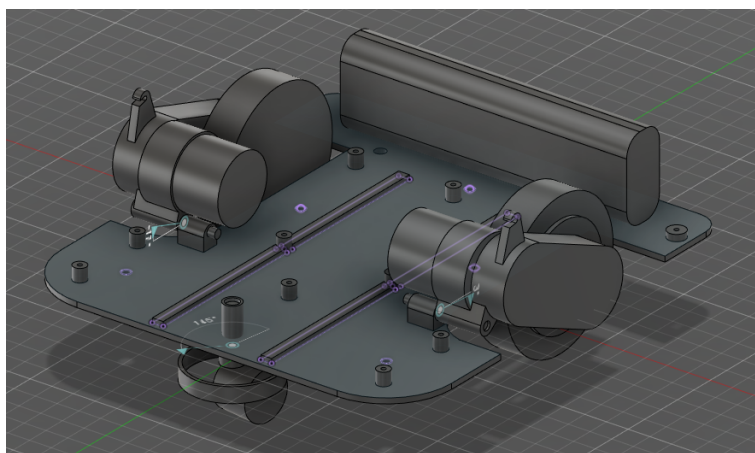
5. Projekt robota

5.1. Założenia projektowe

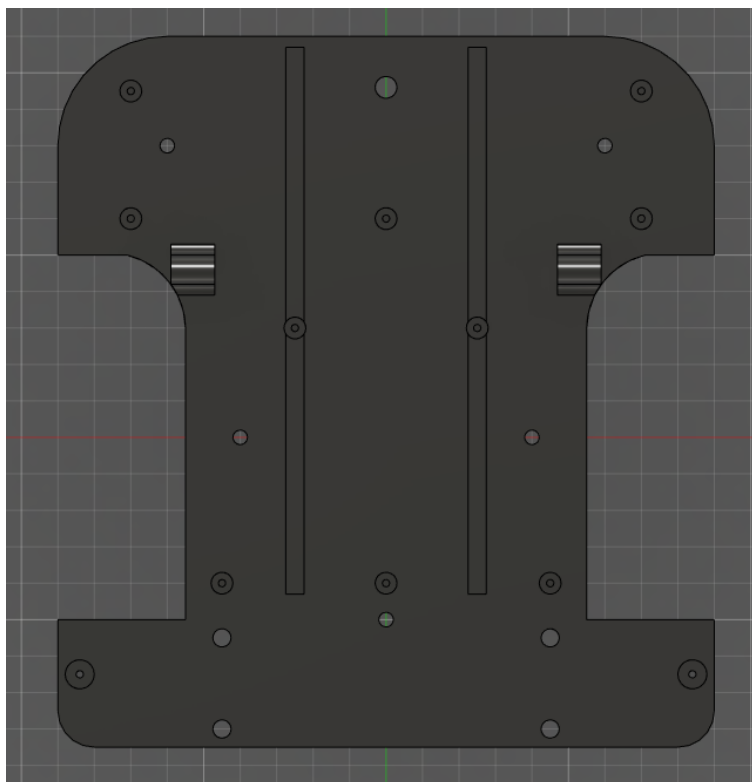
Wykonany robot powinien być jak najmniejszy i najprostszy w wykonaniu oraz sterowaniu tak aby móc przetestować przy jego pomocy działanie algorytmu A*. Robot będzie zbudowany z platformy, do której zostaną przyczepione napędy, elektronika sterująca oraz bateria. Do platformy zostaną przymocowane dwa gotowe moduły napędowe składające się z silnika, przekładni oraz dużego koła. Aby pojazd stał stabilnie, doczepione zostanie trzecie koło obracające się swobodnie w każdym kierunku. Całość będzie sterowana przy pomocy mikroprocesora ESP32 oraz dwukanałowego sterownika silników DC opartym na układzie L298n. Za zasilanie będzie odpowiadał litowo-jonowy akumulator 4S.

5.2. Projektowanie zarysu robota w środowisku CAD

Robot ma być prosty w budowie i wykonaniu a więc założyłem że podstawa utrzymująca pozostałe komponenty zostanie wydrukowana na drukarce 3D. Model platformy i pozostałych posiadanych elementów został wykonany w programie Fusion 360. Modele modułów napędowych, przedniego kółka oraz baterii pozwoliły na optymalne wyznaczenie pod względem wielkości lokalizacji wszystkich elementów.

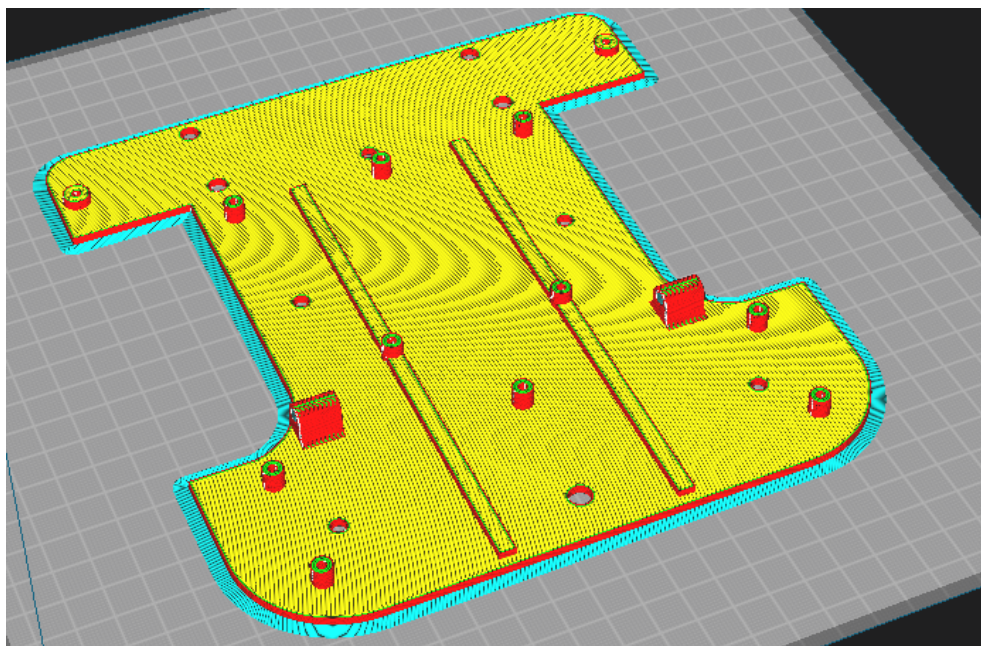


Rysunek 5.5: Zaprojektowane podwozie robota w programie Fusion360



Rysunek 5.6: Widok z góry na podwozie robota

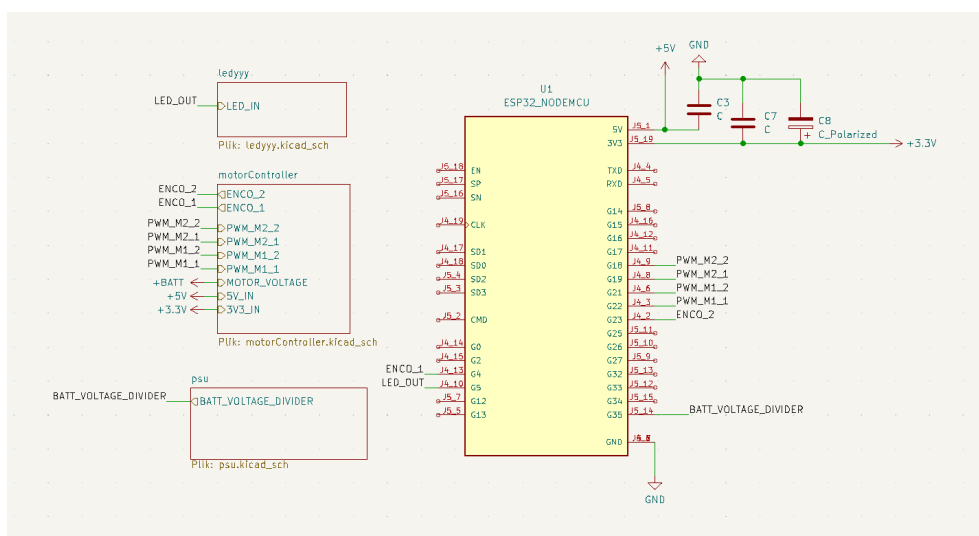
Model został przygotowany do druku w programie Ultimaker Cura a parametry druku zostały dobrane eksperymentalnie. Podstawę wydrukowano z PLA w temperaturze 220°C i wysokości warstwy 0,2mm. Żeby zwiększyć wytrzymałość temperatura głowicy została lekko zawyżona względem wymagań producenta filamentu.



Rysunek 5.7: Widok przygotowanego do druku modelu

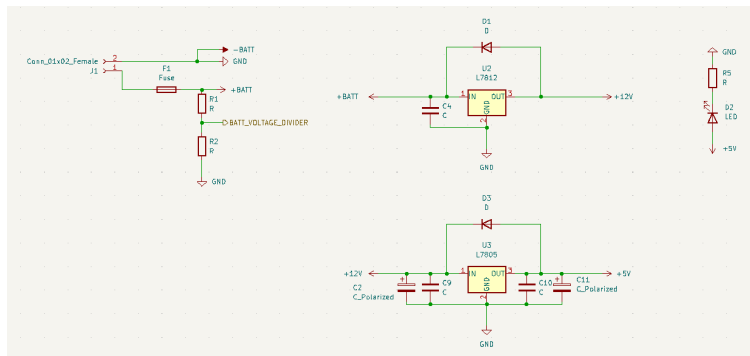
5.3. Projekt, wykonanie oraz podłączenie elektroniki robota

Ze względu na chęć bezprzewodowego sterowania robotem zostanie wykorzystany moduł ESP32, dla którego zostanie przygotowana odpowiednia płytki z wyprowadzeniami do enkoderów silnika oraz ich sterownika. Schemat elektroniczny i projekt pcb został wykonany w programie KiCad.



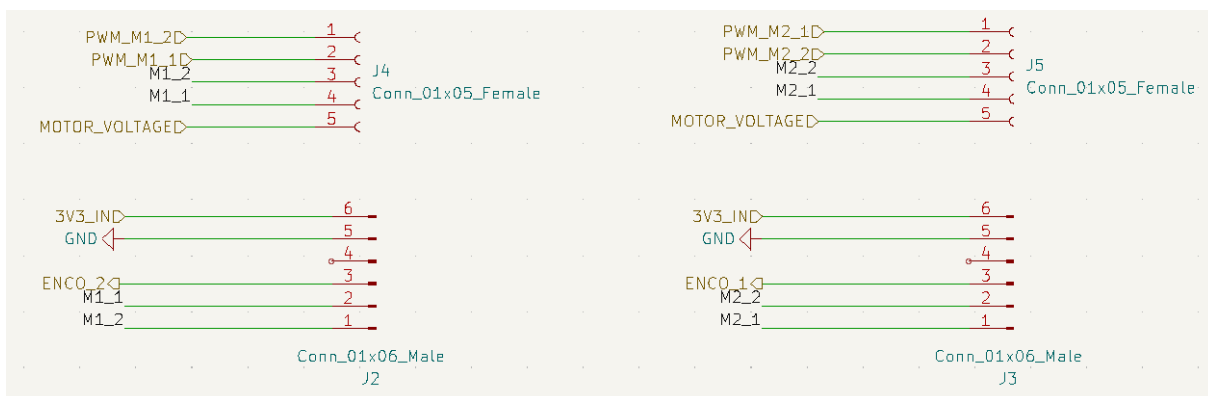
Rysunek 5.8: Ogólny schemat połączeń

Wszystkie potrzebne elementy podsystemów mikroprocesora zostały już włączone w module deweloperskim a więc mój schemat zawiera jedynie odpowiednie połączenia z modułami roboczymi.



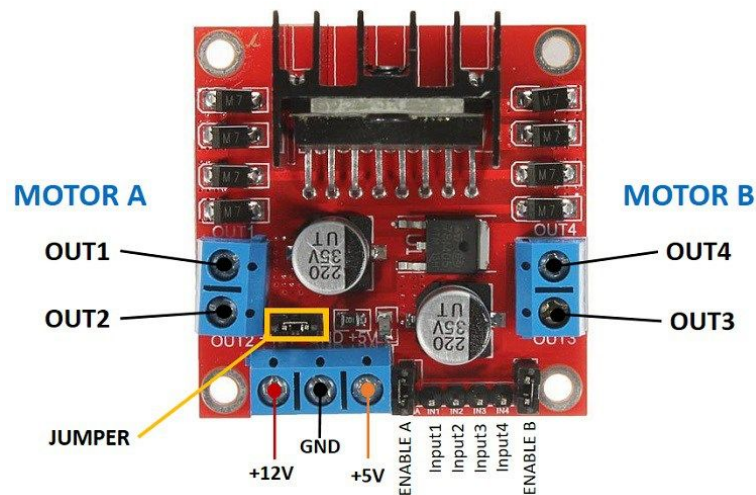
Rysunek 5.9: Ogólny schemat połączeń

Użyta bateria posiada napięcie maksymalne 16,8V a więc musi zostać odpowiednio zmniejszone przed podaniem go na piny zasilania. Ze względu na wykorzystywaną łączność bezprzewodową a więc zwiększone zużycie prądu, zmniejszanie napięcia zostało podzielone na trzy sekcje. Najpierw zmniejszane jest poprzez stabilizator LM7812 z 16.8V do 12V a następnie poprzez LM7805 z 12V napięcie redukowane jest do 5V. ESP32 zasilane jest napięciem 3.3V tworzonym poprzez stabilizator AMS1117 w module deweloperskim. Aby zredukować spadki napięć powstałe przy nagłym poborze prądu dodałem dodatkowe kondensatory filtrujące. Dodatkowo na schemacie widoczny jest dzielnik napięcia pozwalający na poziom naładowania baterii przez sterownik oraz programowalne diody świecące ws2812b, jednak ostatecznie nie zostały wykorzystane w projekcie.

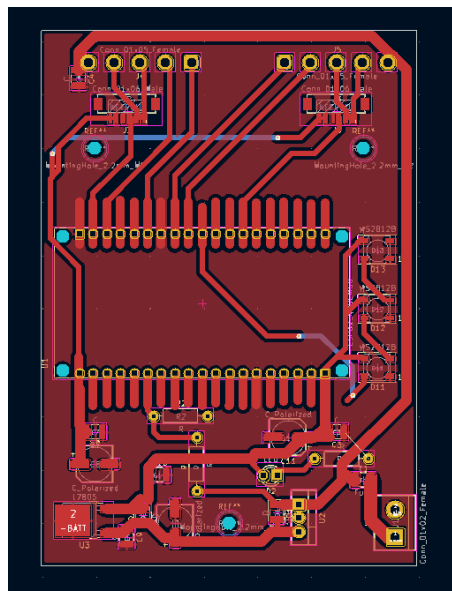


Rysunek 5.10: Ogólny schemat połączeń silników

Użyte moduły napędowe posiadają enkoder inkrementalny zbudowany z czujnika halla i tarczy magnetycznej zamocowanej na wale silnika. Czujnik halla jest zasilany napięciem 3.3V i na wyjściu otrzymujemy sygnał ze szpilkami"proporcjonalny do aktualnych obrotów silnika. Silniki zasilane są poprzez moduł z układem L298n a prędkość ustalana jest poprzez odpowiednio generowany sygnał PWM.



Rysunek 5.11: Wykorzystany moduł do sterowania silnikami, L298n

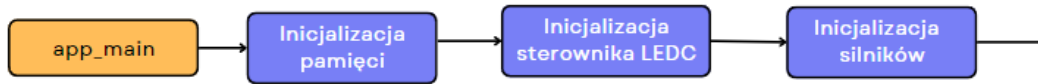


Rysunek 5.12: Projekt PCB

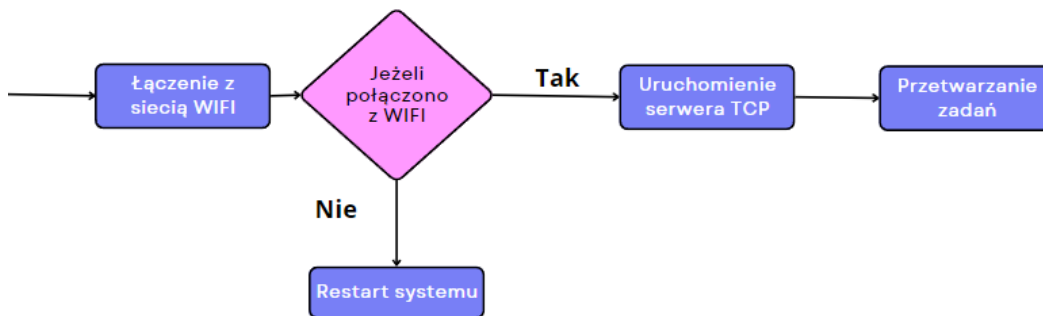
Na widocznym powyżej zdjęciu widać ostateczną wersję projektu pcb.

5.4. Oprogramowanie robota

Program sterujący robotem został napisany w C++. Szkielet aplikacji bazuje na projekcie utworzonym przez framework IDF w wersji 4.4 udostępnionym przez producenta użytego procesora. Po za tym użyta została biblioteka implementująca system czasu rzeczywistego FreeRTOS i ASIO do obsługi połączenia TCP.



Rysunek 5.13: Schemat programu cz.1



Rysunek 5.14: Schemat programu cz.2

Działanie programu rozpoczyna się od wywołania funkcji `app_main`, inicjalizacji funkcji systemowych i sterowników silników. W dalszej kolejności nawiązywana jest łączność z siecią WiFi. W przypadku braku połączenia system resetuje się. Po poprawnym połączeniu uruchamiany jest kontekst biblioteki ASIO a następnie uruchomienie serwera TCP. Klienci po połączeniu do utworzonego serwera wysyłają komendy tekstowe wraz z odpowiednimi argumentami, które robot odpowiednio przetwarza.

5.4.1. Sterowanie silnikami

Sterowanie silnikami odbywa się poprzez klasę `MotorController`, która dziedziczy po klasie `PIDController` implementującej regulator PID. Obiekt automatycznie tworzy timer uruchamiający co 100ms metodę aktualizującą wyjścia sterujące silnikiem. Aktualna prędkość wyznaczana jest na podstawie przerwania wyzwalanego przez enkoder silnika. Przerwanie inkrementuje licznik, czyszczony przez wcześniej opisany

timer. Nastawy regulatora PID zostały dobrane eksperymentalnie, człon proporcjonalny wynosi 8 a całkujący i różniczkujący 0,1. Regulator PID możemy opisać przy pomocy wzoru:

$$u(x) = e(x) * P + \int e(x) * I + \partial e(x) * D \quad (5.4)$$

Gdzie: $e(x)$ – jest błędem; P,I,D – to stałe odpowiednio członu proporcjonalnego, całkującego i różniczkującego

$$e(x) = y_{nast}(x) - y_{aktu}(x) \quad (5.5)$$

Gdzie: $y_{nast}(x)$ – to nastawa regulatora, $y_{aktu}(x)$ – jest rzeczywistą zmierzoną wartością

Powyżej przedstawiony regulator został zaimplementowany w funkcji calcOutput klasy PIDController.

```

1 float PIDController::calcOutput(float current, float set)
2 {
3     float error = set - current;
4     float der = (error - this->_lastValue)/(_timeStep* 0.001);
5
6     float output = (this->_p * error) +
7                   (this->_i * error * this->_timeStep * 0.001) +
8                   (this->_d * der);
9
10    this->_lastValue = error;
11    return output;
12 }

```

Listing 11: Zaimplementowany w C++ regulator PID

Do wyznaczenia części różniczkującej potrzebna wartość błędu z poprzedniego wywołania pętli a ta zapisywana jest do zmiennej prywatnej klasy lastValue. Całkowanie zrealizowane jest poprzez pomnożenie przez krok dyskretyzacji. Stałe regulatora ustawiane są poprzez wywołanie konstruktora klasy.

Tak zaimplementowany regulator używany jest do wyznaczenia sygnału PWM, bezpośredniego sterującego układem L298n a ten silnikami.

```

1 if(abs(setSpeed) - 1 > 0) // jezeli predkosc jest wieksza to pid
   jak nie to hamulec bo pwm=0
2 {
3     pidToPwm = (int) this->calcOutput(abs(this->
4     getCalculatedEngineRadialSpeed()), abs(setSpeed));
5     if(setSpeed < 0)
6     {
7         pwmCh = 1;
8     }
9 }

```

```

7   }
8   else
9       pwmCh = 0;
10
11   pidToPwm += ledc_get_duty(LEDCLOW_SPEED_MODE, this->_channels[
12       pwmCh]);
13   pidToPwm = std::min(pidToPwm, 8192);
14   pidToPwm = std::max(pidToPwm, 0);
15 }
16
17 // ustawienie wyjsc
18 ledc_set_duty(LEDCLOW_SPEED_MODE, this->_channels[pwmCh], (int)
19     pidToPwm);
20 ledc_update_duty(LEDCLOW_SPEED_MODE, this->_channels[pwmCh]);
21
22 // ustawienie drugiego kanalu
23 ledc_set_duty(LEDCLOW_SPEED_MODE, this->_channels[(pwmCh==0)
24     ?1:0], 0);
25 ledc_update_duty(LEDCLOW_SPEED_MODE, this->_channels[(pwmCh==0)
26     ?1:0]);

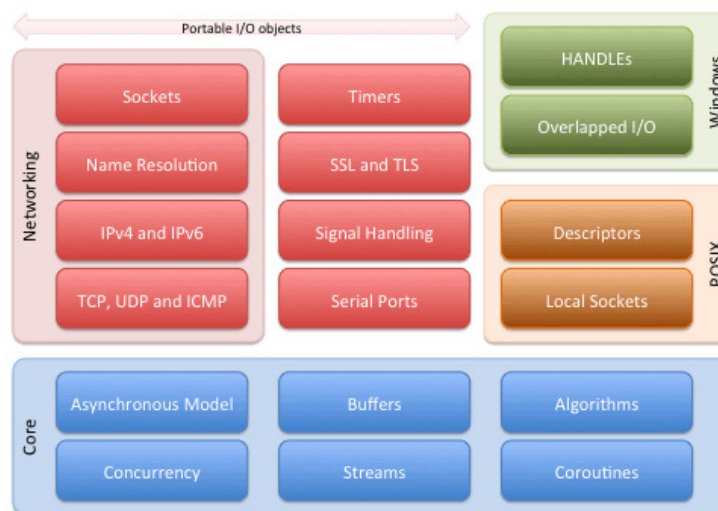
```

Listing 12: Wyznaczenie modulacji PWM

W pierwszej kolejności sprawdzana jest zadana prędkość, jeżeli ta jest zbyt niska to na piny wystawiane są bezpośrednio stany niskie. Jeżeli ustawiona prędkość jest poprawna to wyznaczamy wyjście regulatora. Sterowanie kierunkiem obrotów odbywa się poprzez znak zadanej prędkości a więc regulator otrzymuje wartości bezwzględne. Wyjście regulatora PID sumowane jest z aktualną nastawą. Wyjścia pwm skonfigurowane są w częstotliwości 5kHz i rozdzielczości 13bitów. Przed wysłanie nastaw do kontrolera pwm, te są obcinane do obsługiwanych zakresów (tj. 0 - 8192).

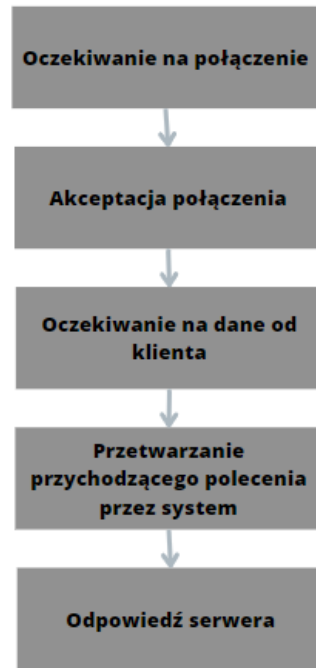
5.4.2. Serwer TCP

Po podłączeniu zasilania i uruchomieniu systemu robot oczekuje na polecenia wysłane do robota poprzez protokół internetowy TCP. Serwer został napisany w oparciu o bibliotekę ASIO, pozwalającą na asynchroniczną obsługę wejścia i wyjścia (w tym sieci).



Rysunek 5.15: Schemat biblioteki ASIO [5]

Biblioteka została napisana w C++ i pracuje w oparciu o standard POSIX wspierany również przez wykorzystywane ESP32. Do akceptacji przychodzących połączeń została napisana klasa Server. Konstruktor przyjmuje referencje do kontekstu utworzonego w funkcji głównej programu. Uruchomienie kontekstu jest ważną częścią biblioteki ASIO, przetwarzane są w niej wszystkie asynchroniczne operacje. Aby zapewnić jak najlepszy czas przetwarzania, kontekst uruchomiony jest w oddzielnym wątku przyłączonym do drugiego fizycznego rdzenia.



Rysunek 5.16: Schemat działania serwera TCP

Po zaakceptowaniu przychodzącego połączenia, obiekt połączenia dodawany jest do specjalnej tablicy przechowującej wszystkich klientów. Dane mogą być równocześnie przetwarzane w kilku miejscach a więc używane są dzielone wskaźniki, które automatycznie usuwają dane jeżeli nikt z nich nie korzysta. Na każdym aktywnym połączeniu prowadzony jest nasłuch, dzięki czemu możemy mieć równocześnie podłączony program do generowania ścieżki i drugi pozwalający na podgląd parametrów robota. Klient wysła polecenia w formie tekstowej, te przekazywane są do klasy systemowej odpowiednio interpretujące ich przeznaczenie. Jeżeli komenda tego wymaga (np. zwrócenie prędkości obrotowej silników) to dane są odsyłane do klienta w postaci tekstu i oddzielonych przerwą liczb.

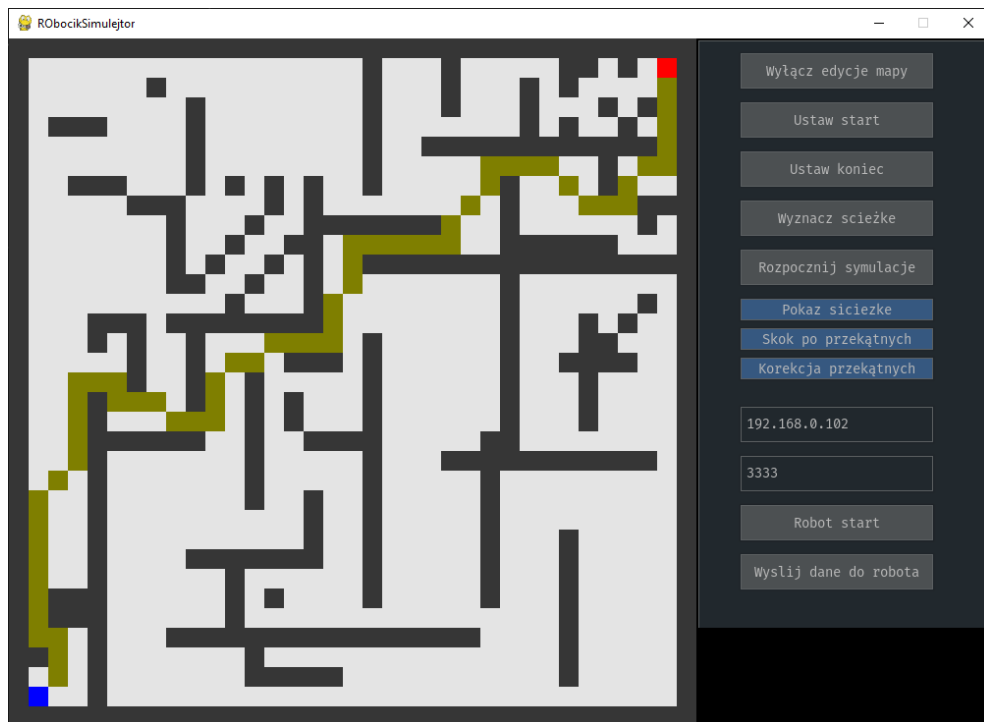
6. Przeprowadzone testy

6.1. Test poprawności trasowania

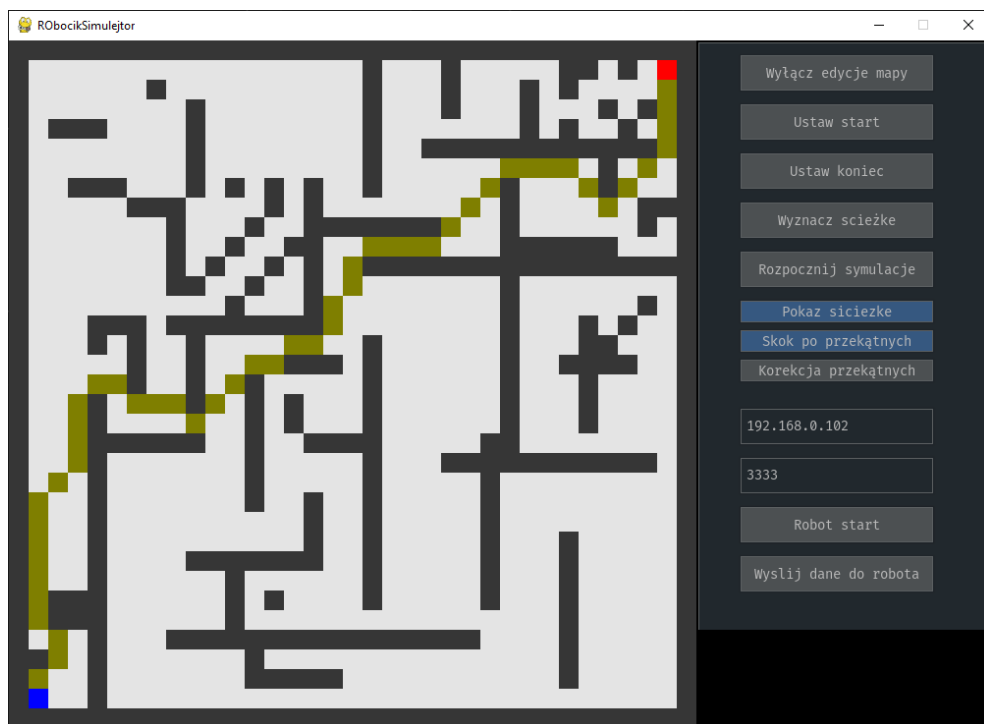


Rysunek 6.17: Podstawowy test trasowania

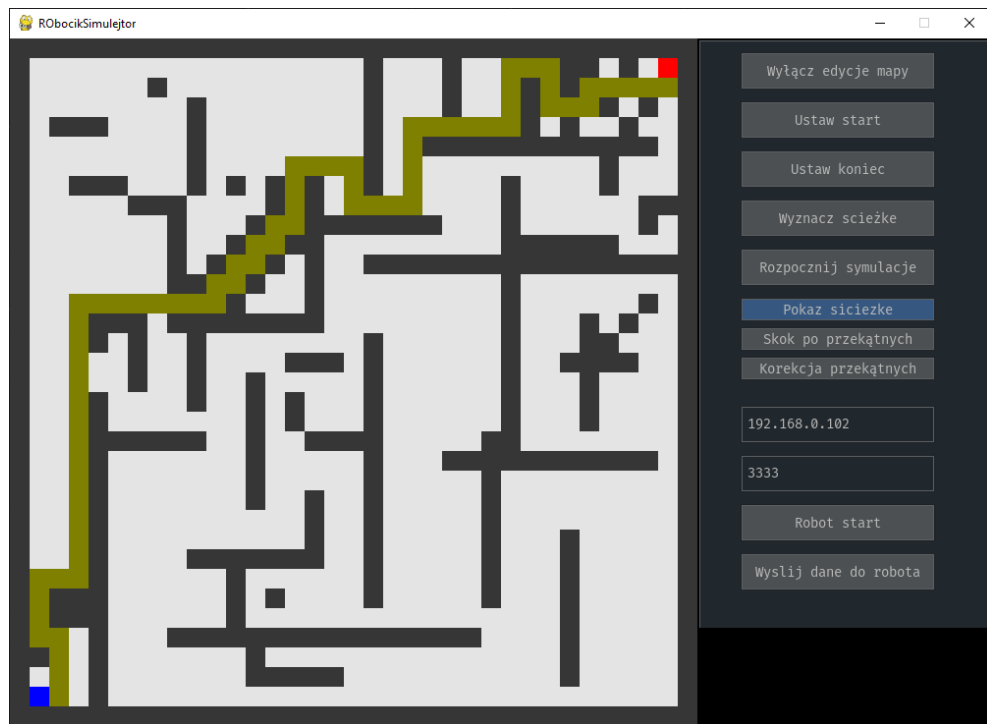
6.2. Wpływ ustawień na wyznaczoną ścieżkę



Rysunek 6.18: Wyznaczona ścieżka, włączona korekcja i przejście po przekątnej



Rysunek 6.19: Wyznaczona ścieżka, korekcja przekątnych jest wyłączona



Rysunek 6.20: Najkrótsza trasa, wyłączone przejście po przekątnych

6.3. Sprawdzenie wpływu funkcji heurystycznej na ścieżkę

6.4. Test komunikacji z fizycznym robotem

Literatura

- [1] <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html>. Dostęp 04.01.2023.
- [2] <https://medium.com/dish/75-years-of-innovation-shakey-the-robot-385af2311ec8>
Dostęp 04.01.2023.
- [3] <https://www.pygame.org/docs/> Dostęp 06.01.2023
- [4] <https://pygame-gui.readthedocs.io/en/latest/> Dostęp 06.01.2023
- [5] <https://think-async.com/Asio/> Dostęp 06.01.2023

STRESZCZENIE PRACY DYPLOMOWEJ INŻYNIERSKIEJ
IMPLEMENTACJA STEROWANIA HIERARCHICZNEGO
MOBILNYM ROBOTEM KOŁOWYM Z WYKORZYSTANIEM
JĘZYKA PYTHON

Autor: Damian Bielecki, nr albumu: ME-163461

Opiekun: dr. Paweł Penar

Słowa kluczowe: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po polsku

BSC THESIS ABSTRACT
TEMAT PRACY PO ANGIELSKU

Author: Damian Bielecki, nr albumu: ME-163461

Supervisor: (academic degree) Imię i nazwisko opiekuna

Key words: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po angielsku