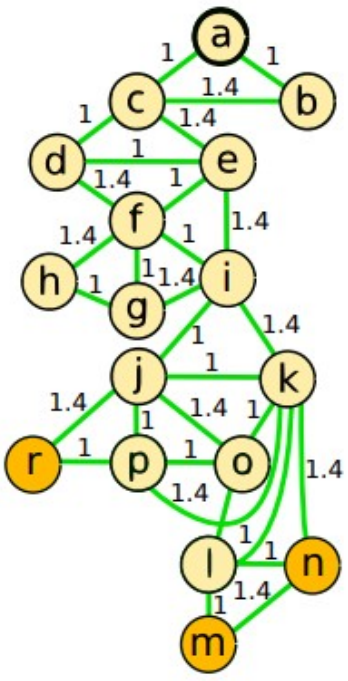
Sprawozdanie			
Nawigacja i planowanie ruchu robotów - projekt			
Temat: Planowanie ruchu robota o kinematyce monocykla z użyciem algorytmu A*			
Imię nazwisko, nr albumu:	**Wydział:**	**Kierunek**	**Prowadzący**
	WARiE	AiR	dr hab. inż. Dariusz Pazderski
Tomasz Smaruj, 131371	**Specjalność:**	**Semestr:**	**Data oddania ćwiczenia:**
Szymon Kacperek, 131989	SSiR	II	17.02.2021

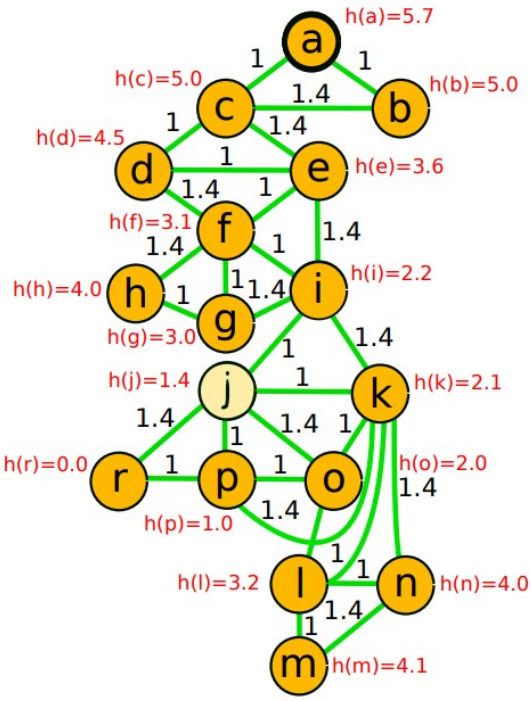
1. Opis zadania

Zadaniem projektowym było zamodelowanie robota o kinematyce monocykla, który znajduje i porusza się na planszy 2D. Ma on realizować dojazd do punktu zadanego na mapie wykorzystując algorytm A*.

- **Algorytm A***

Jest to metoda przeszukiwania grafu. Aby wprowadzić tematykę algorytmu A*, najpierw należy zapoznać się z algorytmem Djikstry. Znajduje on w grafie wszystkie najkrótsze ścieżki pomiędzy wybranym wierzchołkiem, a wszystkimi pozostałymi przy okazji wyliczając również koszt przejścia każdej z tych ścieżek. Ta ścieżka która będzie miała najmniejszy koszt, jest najbardziej optymalna.
Na mapie z 8-sąsiedztwami, przyjmuje się, że dla pól sąsiadujących pionowo lub poziomo koszt wynosi wynosi 1, natomiast dla pozostałych 4 pól znajdujących się na skos, jest to 1.4.



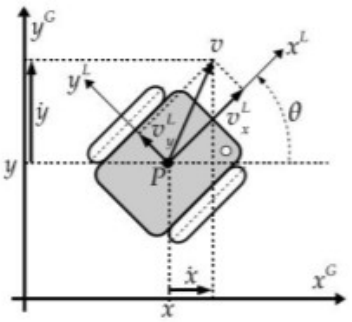


Rys. 1. Przykładowy graf dla algorytmu Djikstry Przykładowy graf dla algorytmu A*

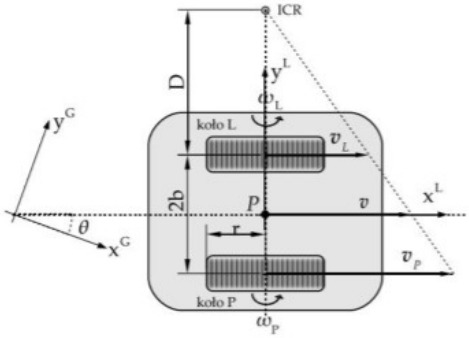
Alorytm A* jest rozwinięciem algorytmu Djikstry. Jest to heurystyczny algorytm służący do znajdowania najkrótszej ścieżki w grafie. Jest to algorytm zupełny i optymalny, co oznacza, że zawsze zostanie znalezione najlepsze rozwiązanie. W metodzie tej istnieje zorganizowany model pamięciowy, który gwarantuje, że każdy punkt może zostać odwiedzony. A* jest przykładem metody „najpierw najlepszy”. Algorytm A* działa najlepiej, gdy przestrzeń przeszukiwań jest przestrzenią drzewiastą. [1]
Działanie algorytmu oparte jest na minimalizacji funkcji celu , zdefiniowanej jako suma funkcji kosztu g(n) oraz funkcji heurystycznej h(n).

$$f(n)=g(n)+h(n)$$

- **Model kinematyczny robota** - został zaimplementowany wzorując się na skrypcie z laboratorium sterowania robotów mobilnych [2].



Rys. 2. Platforma robota mobilnego w ruchu na płaszczyźnie



Kinematyczny model robota

Kinematyka monocykla jest przykładem robota o kinematyce (2,0).
Dla bryły sztywnej na płaszczyźnie definiujemy dwie składowe prędkości, które w pełni opisują zmianę konfiguracji robota w układzie globalnym. Jest to prędkość kątowa platformy ω oraz prędkość postępowa v. Przyjmując model platformy robota (Rys. 2.) równania kinematyki przybierają postać tak jak poniżej:

$$\begin{aligned}\dot{\theta} &= \omega \\ \dot{x} &= v_x^L\cos(\theta)=v\cos(\theta) \\ \dot{y} &= v_y^L\sin(\theta)=v\sin(\theta)\end{aligned}$$

Po przekształceniu do równań macierzowych równania kinematyki robota wyglądają następująco:

$$\begin{bmatrix}\dot{\theta}\\ \dot{x}\\ \dot{y}\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & \cos(\theta)\\ 0 & \sin(\theta)\end{bmatrix}\begin{bmatrix}\omega\\ v\end{bmatrix} \quad \text{czyli,} \quad \dot{q}=G(q)u$$

gdzie θ, jest bieżącym kątem orientacji platformy.

2. Implementacja

Do zadania wykorzystany został język programowania Python, a program wykonywano w środowisku Pycharm. Do samej wizualizacji ruchu pojazdu użyta została biblioteka Pygame[3], natomiast do operacji na macierzach - biblioteka Numpy.

2.1 Algorytm A*

- Sposób liczenia heurystyki

```
def heuristics(st, end):
    # distance = abs(st[0] - end[0]) + abs(st[1] - end[1])  # Manhattan
    distance = ((st[0] - end[0]) ** 2 + (st[1] - end[1]) ** 2) ** 0.5  # Euclidean
    return distance
```

Listing 1 – Obliczanie heurystyki

Do wyznaczenia odległości ścieżki przetestowano dwa sposoby liczenia odległości na współrzędnych kartezjańskich – Manhattan oraz Euklidesowa. Odległość Manhattan oblicza długość ścieżki tylko w poziomie i pionie, natomiast odległość Euklidesowa bierze pod uwagę również ścieżkę na skos. Dla algorytmu A* z 8 sąsiedztwami optymalniejszą drogę wskaże odległość Euklidesowa.

- Sposób wyznaczania koszt i ścieżki

```
def find_path_a_star(start, end):
    came_from = {}
    current = []
    gscore = {start: 0}
    fscore = {start: heuristics(start, end)}
    oheap = []
    heapq.heappush(oheap, (fscore[start], start))

    while oheap:
        current = heapq.heappop(oheap)[1]
        if current == end:
            break
        if map[int(current[1] / block)][int(current[0] / block)] < 50:
            neighbours = []
            for new in [(0, -block), (0, block), (-block, 0), (block, 0),
                        (block, block), (block, -block), (-block, -block), (-block, block)]:
                position = (current[0] + new[0], current[1] + new[1])
                if int(position[1] / block) >= 40 or int(position[0] / block) >= 60:
                    continue
                else:
                    if map[int(position[1] / block)][int(position[0] / block)] < 50:
                        neighbours.append(position)

            for neigh in neighbours:
                gscore[current] = 1.0
                x_1 = neigh[0] - current[0]
                y_1 = neigh[1] - current[1]
                for i in [-block, block]:
                    for j in [-block, block]:
                        if (x_1, y_1) == (i, j):
                            gscore[current] = 1.4

                cost = heuristics(current, neigh) + gscore[current]  # cost of the path
                if cost < gscore.get(neigh, 0) or neigh not in gscore:
                    came_from[neigh] = current
                    gscore[neigh] = cost
                    fscore[neigh] = cost + heuristics(neigh, end)
                    pq.heappush(oheap, (fscore[neigh], neigh))
```

Listing 2 – Definicja funkcji A*

Funkcja A* przyjmuje jako argumenty wejściowe współrzędne punktu początkowego oraz docelowego. Pierwsza zostaje policzona funkcja heurystyczna h(n). Do przeszukania całej planszy użyto biblioteki heapq, która pozwala na przeiterowanie się po całym zbiorze i dodaniu trasy do listy.
Następnie wyznaczane zostają ograniczenia, przeszkody mają wartości pól powyżej 50, więc algorytm musi pomijać te rozwiązania i szukać innej ścieżki przez dostępne pola.
Kolejna pętla ma za zadanie policzyć funkcję kosztu g(n). Jeśli ścieżka prowadzi poziomo lub pionowo funkcja będzie miała koszt 1, natomiast gdy prowadzi na skos(jednoczesna zmiana x i y) funkcja kosztu będzie równa 1,4.
Ostateczny koszt algorytmu jest sumą h(n) jak i g(n). Ta ścieżka, która będzie miała najmniejszy koszt zostaje przechowana najpierw w liście came_from, a następnie trafia do wyjścia funkcji i zwraca współrzędne wszystkich pół potrzebnych do dojazdu do punktu docelowego.

2.2 Kinematyka monocykla

```
# ---------- Kinematic Model ----------
        # Calculate position of the z point
        xz = xp - d * math.cos(theta)
        yz = yp + d * math.sin(theta)

        # Planner - calculate theta and z_dot
        (x_dest, y_dest) = (x1 + int(block / 2), y1 + int(block / 2))  # Destined next block (x,y) from A* (center)
        (x_diff, y_diff) = (x_dest - xz, y_dest - yz)  # Difference between the points
        theta = math.atan2(y_diff, -x_diff)  # Calculate theta of the robot, 180 because of mirror view
        z_dot = np.array([[math.cos(theta)], [-math.sin(theta)]]) * V  # Velocity of the robot z point
        print('z_dot: ', z_dot)

        # Linearization method - calculate velocities Vx and omega
        P = np.array([[math.cos(theta), -d * math.sin(theta)],
                      [math.sin(theta), d * math.cos(theta)]])
        inv_P = np.linalg.inv(P)
        (Vx, omega) = np.dot(inv_P, z_dot)
        print('(Vx, omega): ', (Vx, omega))

        # Calculate position change - configuration coordinates q
        q1 = np.array([[-math.cos(theta)],
                       [math.sin(theta)],
                       [0]])
        q2 = np.array([[0],
                       [0],
                       [1]])
        q = q + (q1 * V + q2 * omega) * Ts  # calculate new robot coords
        (xp, yp, theta) = q  # new coords of the robot
```

Listing 3 – Algorytm kinematyki robota wraz z planerem ruchu

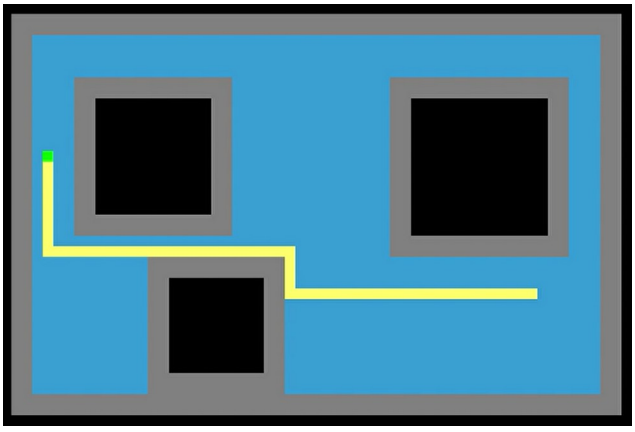
Model kinematyczny monocykla przedstawiono w Listingu 3.
Jako wejście dla modelu potrzebne jest określenie pozycji punktu Z robota. Obliczany on jest na podstawie współrzędnych konfiguracyjnych.
Współrzędne punktu Z trafiają do planera ruchu, który wyznacza punkt docelowy, w kierunku którego musi poruszać się robot – są to kolejne pola algorytmu A*. Na podstawie punktu zadanego oraz punktu Z obliczana jest różnica, czyli uchyb. Pozwala on określić kąt theta, czyli kierunek w którym będzie poruszał się robot. Mając te informacje obliczona zostaje prędkość dla punktu Z robota. Korzystając z metody linearyzacji projektuje się pewną macierz odwrotną P, która pozwala określić poszczególne prędkości sygnałów sterujących. Na ich podstawie określa się nowe współrzędne konfiguracyjne robota.

3. Wyniki symulacji

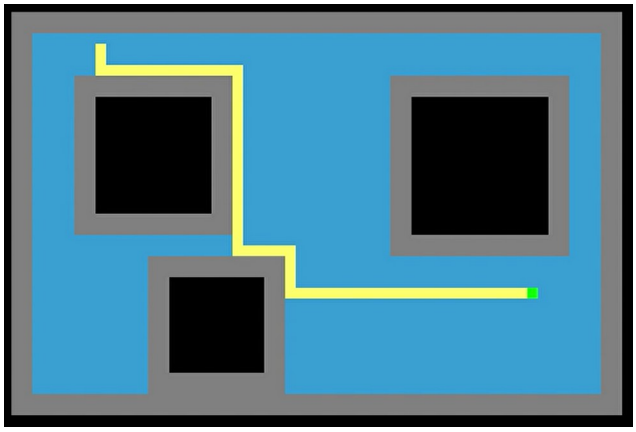
Do testowania środowiska została wczytana została mapa z ograniczeniem na krawędziach oraz trzema kwadratowymi przeszkodami o różnych rozmiarach.
Przestrzeń robocza, w której pracuje algorytm A*, to wszystkie niebieskie pola. Przeszkody zostały zaznaczone kolorem czarnym. W celu uniknięcia kolizji modelu kinematycznego monocykla wokół przeszkód, zostało narysowane szare pola(uzależnione one są od długości pojazdu). Są one możliwe do jazdy dla robota, jednak algorytm będzie ich unikać, przez co pojazd nie będzie mógł wjechać w przeszkodę.

3.1 Algorytm A*

Symulacja prezentuje samo działanie algorytmu A*. Punktem początkowym jest pierwszy z żółtych punktów, natomiast punkt końcowy symbolizuje zielony kwadrat.

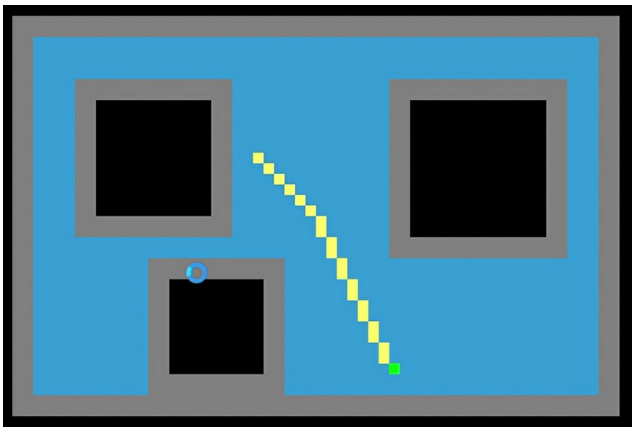


Rys. 3.a

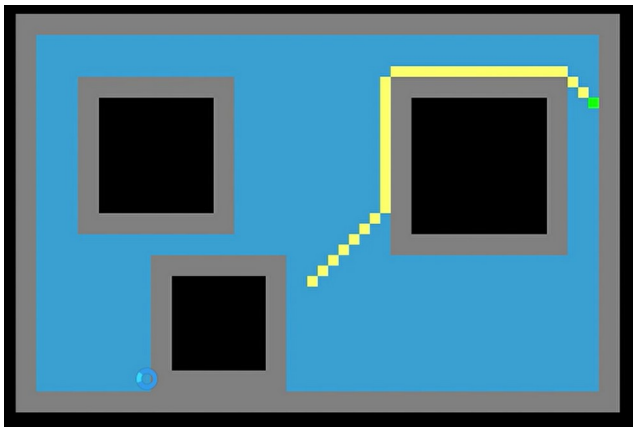


Rys 3.b

Znajdowanie ścieżki algorytmem A* dla heurystyki wg. odległości Manhattan



Rys. 4.a



Rys. 4.b

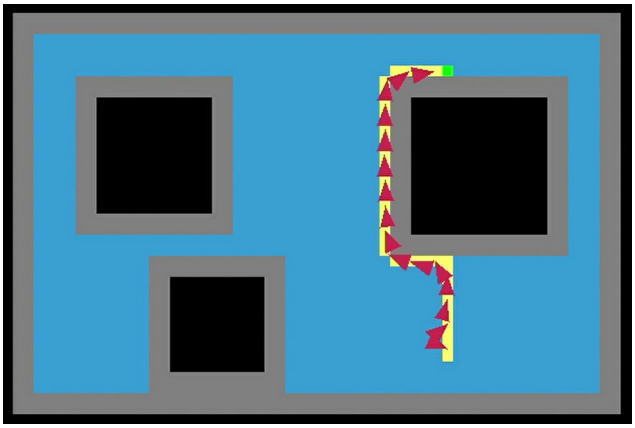
Znajdowanie ścieżki algorytmem A* dla heurystyki wg. odległości Euklidesowej

Do sprawdzenia poprawności znajdowania ścieżki użyto dwóch typów liczenia heurystyki. Na Rys.3. przedstawione zostały znalezione optymalne ścieżki dla odległości Manhattan, zgodnie z założeniami teoretycznymi optymalne znalezione ścieżki będą zawsze poprowadzone horyzontalnie lub wertykalnie. Droga na Rys.4. jest natomiast krótsza, jednak uwzględnia możliwość jazdy na skos, gdyż heurystyka obliczona została na podstawie odległości Euklidesowej.
W obydwu przykładach, biorąc pod uwagę ograniczenia(np. brak jazdy na skos) obliczenia zwróciły najbardziej optymalną drogę, co wskazuje na poprawność działania algorytmu A*.

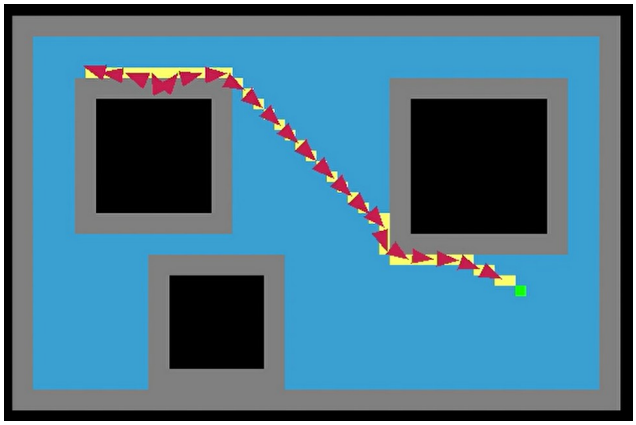
3.2 Algorytm A* wraz z modelem o kinematyce monocykla

Dla tej symulacji ścieżka jest obliczana na podstawie odległości Euklidesowej, czyli umożliwia robotowi ruchy na skos. Robot porusza się w kierunku żółtej ścieżki, która ostatecznie prowadzi do zielonego punktu końcowego.

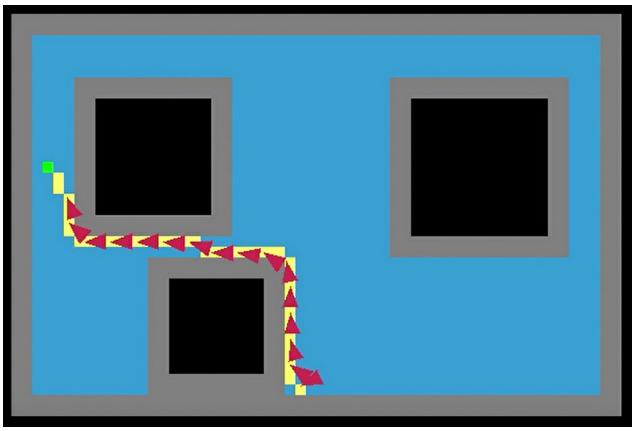
Parametry modelu:
Początkowe współrzędne konfiguracyjne robota: $q=[305,\ 205,\ 90°]$, jednak program w momencie dojazdu do punktu zadanego generuje nowy punkt docelowy, więc współrzędne konfiguracyjne są zawsze inne.
Czas odświeżania pętli algorytmu: $T_s=0.1[s]$, prędkość platformy: $V=25$.



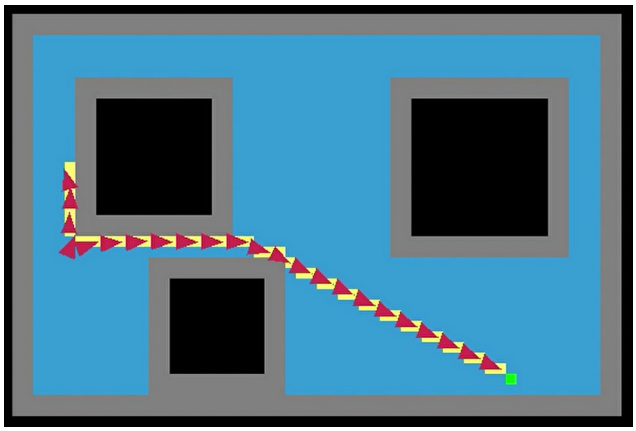
Rys. 5.a



Rys. 5.b



Rys. 5.c



Rys. 5.d

Model kinematyczny robota w ruchu po ścieżce z algorytmu A*

Robot poprawnie realizuje zadanie poruszania się po ścieżce oraz dojazd do punktu docelowego. Dzięki plannerowi, który wskazuje każde pole i liczy względem niego orientacje robot może w optymalny sposób dojechać do punktu końcowego. Jak widać na każdym z rysunków kinematyka modelu decyduje kiedy nastąpi najlepszy moment na obrócenie się pojazdu. Rys. 5.b pokazuje poprawne działanie i sens działania obramówek dla przeszkód. W momencie braku szarych pól robot mógłby wjechać w szarą przeszkodę, gdyż możliwie tak poprowadziłby go algorytm.

4. Wnioski

Projekt bardzo dobrze i ciekawie zapoznał z tematyką jednego z najpopularniejszych algorytmów znajdowania ścieżki i planowania ruchu. Model robota pozwolił zasymulować i przeanalizować w jaki sposób ma poruszać się pojazd, aby trafić do wyznaczonego punktu na mapie w optymalny sposób. Należało przemyśleć jak odwzorować model kinematyczny w przestrzeni ciągłej na dyskretną siatkę planszy oraz uwzględnić pewne ograniczenia, tak aby ruch był możliwy bez kolizji.

Programowanie za pomocą języka Python oraz symulacja dzięki jego bibliotekom pozwoliła również rozwinąć umiejętności programowania i tworzenia algorytmów. Regularne konsultacje z prowadzącym umożliwiały rozwianie wątpliwości i problemów na wcześniejszym etapie, co znacznie ułatwiło pracę nad projektem.

5. Materiały źródłowe

[1] Algorytm A* - https://elektron.elka.pw.edu.pl/~jarabas/ALHE/notatki3.pdf
[2] Sterowanie robotów mobilnych. Laboratorium, Maciej Michałek, Dariusz Pazderski, 2012.
[3] Dokumentacja biblioteki Pygame - https://www.pygame.org/docs/