UNIVERSITÀ
degli STUDI
di CATANIA

Dipartimento di Matematica e Informatica
Corso di Laurea Magistrale in Informatica

Progetto di Sistemi Robotici

Prof. Corrado Santoro

AA.2022/20223

Visibility Graph on 2D 2-Wheeled Robot

Studentessa: Francesca Ragazzi

Matricola: 1000038267

20/07/2023

Sommario

Introduzione

Nell'ambito della robotica, la pianificazione del movimento è un elemento cruciale per garantire che i robot possano navigare in modo sicuro ed efficiente attraverso ambienti complessi e dinamici. In questo contesto, l'implementazione dell'algoritmo del ‘Visibility Graph’ rappresenta un modo per consentire ad un robot a due dimensioni, dotato di due ruote, di pianificare e seguire una traiettoria ottimale.

Il Visibility Graph è un metodo ampiamente utilizzato per la pianificazione di percorsi in ambienti con ostacoli. Esso sfrutta la costruzione di un grafo, in cui i nodi rappresentano i punti critici dell'ambiente e gli archi definiscono le linee di visibilità tra di essi. L'algoritmo determina quindi il percorso più breve, utilizzando l’algoritmo di Dijkstra, sulla base di questo grafo, permettendo al robot di navigare attraverso ostacoli in modo efficiente, evitando collisioni e minimizzando il tempo di percorrenza.

Inoltre, è stato sviluppato uno script che permette al robot 2d con due ruote di seguire con precisione la traiettoria calcolata dal Visibility Graph. Per ottenere questo risultato, sono stati utilizzati controller PID (Proporzionale-Integrale-Derivativo) per le velocità delle ruote destra e sinistra e il controllo polare per posizione e l'orientamento del robot rispetto ad un punto di riferimento e la sua direzione di spostamento. Questa combinazione di tecniche di controllo ha consentito di regolare accuratamente la velocità del robot durante la navigazione, garantendo la stabilità e il mantenimento del percorso desiderato.

Nella seguente relazione, dunque, si presenteranno in dettaglio l'implementazione dell'algoritmo del Visibility Graph per la pianificazione del movimento del robot a due dimensioni. Si descriverà anche il funzionamento dello script sviluppato per controllare le ruote del robot utilizzando i controller polari e il PID.

1. Visibility Graph

Il Visibility Graph (grafo di visibilità) è un algoritmo utilizzato per la pianificazione di percorsi per robot o altri oggetti mobili in ambienti con ostacoli. L'obiettivo principale del Visibility Graph è quello di trovare il percorso più breve e sicuro tra due punti all'interno di un ambiente 2D, tenendo conto degli ostacoli presenti.

La creazione di un Visibility Graph avviene in tre fasi principali:

1. Generazione dei punti critici:
   In questa fase, vengono individuati i punti critici dell'ambiente. I punti critici sono i punti che rappresentano i vertici del grafo di visibilità e corrispondono ai vertici degli ostacoli e ai punti di partenza e arrivo del robot. Si tratta di punti strategici che influenzano il percorso del robot.

2. Costruzione del grafo di visibilità:
   Una volta generati i punti critici, si procede a costruire il grafo di visibilità. Ogni punto critico diventa un nodo del grafo.
   Si collegano gli archi tra i nodi (punti critici) solo se la linea di visibilità tra i due punti non interseca gli ostacoli presenti nell'ambiente. Questa è la chiave del metodo: solo i collegamenti visibili sono considerati nella creazione del grafo.
   Un arco nel grafo di visibilità rappresenta quindi una possibile traiettoria libera da ostacoli tra due punti critici.

3. Pianificazione del percorso:
   Dopo la costruzione del grafo di visibilità, si utilizza un algoritmo di ricerca del percorso, come l'algoritmo di Dijkstra, per trovare il percorso più breve tra i punti di partenza e arrivo del robot.
   Questo percorso è il risultato dell'implementazione del Visibility Graph ed è quello che il robot seguirà per raggiungere la destinazione, evitando gli ostacoli in modo efficiente e sicuro.

Il Visibility Graph è adatto per ambienti statici, dove gli ostacoli non cambiano posizione nel corso del tempo. In ambienti dinamici, l'algoritmo dovrebbe essere integrato con strategie di rilevamento e reattività agli ostacoli in movimento.

L'efficienza dell'algoritmo dipende dalla complessità della scena e dal numero di punti critici. Per ambienti molto grandi o con numerosi ostacoli, potrebbe essere necessario utilizzare tecniche di ottimizzazione o semplificazione.

Una volta costruito il grafo di visibilità, le fasi successive di ricerca del percorso sono relativamente veloci, poiché il numero di nodi e archi nel grafo è ridotto rispetto alla densità dell'ambiente.

L’implementazione del Visibility Graph offre un approccio robusto e efficiente per la pianificazione del movimento di robot e oggetti mobili in ambienti 2D con ostacoli, consentendo di trovare un percorso ottimale e sicuro tra due punti specifici all'interno di un ambiente complesso.

2. Implementazione Visibility Graph

In questo capitolo, verrà esaminata l'implementazione in Phython dell'algoritmo del Visibility Graph (Grafo di Visibilità) per la pianificazione del percorso di un robot a due ruote in un ambiente bidimensionale. Saranno forniti dettagli sulla struttura del codice e sulle principali funzioni implementate, concentrandosi sulle logiche alla base del grafo di visibilità e sulla generazione dei percorsi ottimali.

2.1 Preparazione dell'Ambiente e delle Strutture di Dati

Prima di procedere con l'implementazione dell'algoritmo, è necessario preparare l'ambiente di simulazione e definire le strutture di dati necessarie. Sono state create due classi personalizzate: 'Environment' e 'Obstacle' per rappresentare rispettivamente l'ambiente di simulazione e gli ostacoli all'interno dello spazio. Queste classi permettono di gestire in modo flessibile l'interazione con l'ambiente e la rappresentazione degli ostacoli.

La classe 'VisibilityGraph' viene inizializzata con i seguenti attributi:

- 'environment': Un'istanza della classe 'Environment', che rappresenta l'ambiente in cui il robot opera. La classe 'Environment' contiene informazioni sugli ostacoli presenti nell'ambiente.
- 'start': Una tupla che rappresenta la posizione di partenza (x, y) da cui inizia la pianificazione del percorso.
- 'goal': Una tupla che rappresenta la posizione di destinazione (x, y) che il robot si propone di raggiungere.
- 'fastest_path': Una lista per memorizzare il percorso più veloce finale dal punto di partenza al punto di destinazione.

2.2 Funzione visibility graph

La funzione 'visibility_graph' è responsabile della genazione del visibility graph basato sulle informazioni dell'ambiente e degli ostacoli. Il visibility graph è una collezione di archi (segmenti di linea) che connettono tutti i punti visibili nell'ambiente. Esso include gli archi che collegano i punti di partenza e destinazione e gli archi all'interno di ciascun poligono rappresentante gli ostacoli.
La funzione raccoglie innanzitutto tutti i vertici degli ostacoli nell'ambiente e aggiunge i punti di partenza e destinazione alla lista dei punti. Quindi, itera attraverso tutte le coppie di punti per determinare se possono essere connessi senza intersecare gli ostacoli. Se esiste una connessione valida tra due punti, un arco viene aggiunto alla lista degli archi.

```
def visibility_graph(self):
        points = [self.start, self.goal]
```

```
        for obstacle in self.environment.obstacles:
            points.extend(obstacle.vertices)

        edges = []
        for i in range(len(points)):
            for j in range(i + 1, len(points)):
                point1 = points[i]
                point2 = points[j]
                if self.can_connect(point1, point2):
                    edges.append((point1, point2))

        # Aggiungi gli archi tra i vertici dell'ostacolo stesso
        for obstacle in self.environment.obstacles:
            vertices = obstacle.vertices
            num_vertices = len(vertices)
            for i in range(num_vertices):
                point1 = vertices[i]
                point2 = vertices[(i + 1) % num_vertices]
                edges.append((point1, point2))
        self.edges = edges

        return
```

2.3 Funzione can connect

La funzione ‘can_connect’ verifica se due punti dati, ‘point1’ e ‘point2’, possono essere collegati da una linea retta senza intersecare alcun ostacolo nell'ambiente. Prima controlla se entrambi i punti sono validi (all'interno dei limiti dell'ambiente) e non si trovano all'interno di alcun ostacolo. Successivamente, suddivide il segmento di linea tra ‘point1’ e ‘point2’ in piccoli passi e controlla se alcuni di questi passi intersecano un ostacolo, per far questo viene utilizzata la funzione is_in_obstacle della classe Environment. Se non vengono trovate intersezioni, la linea è considerata valida e la funzione restituisce ‘True’. In caso contrario, restituisce ‘False’.

```
def can_connect(self, point1, point2):
        if not self.environment.is_valid_point(point1) or not self.environment.is_valid_point(point2):
            return False

        if self.environment.is_in_obstacle(point1) or self.environment.is_in_obstacle(point2):
            return False

        x1, y1 = point1
        x2, y2 = point2
        dx = x2 - x1
        dy = y2 - y1
        distance = math.sqrt(dx ** 2 + dy ** 2)
        step = 0.1
```

```
        num_steps = int(distance / step)

        for i in range(num_steps + 1):
            t = i / num_steps
            x = x1 + t * dx
            y = y1 + t * dy
            if self.environment.is_in_obstacle((x, y)):
                return False
        return True
```

2.4 Funzione find fastest path

La funzione ‘find_fastest_path’ trova il percorso più veloce tra tutte le traiettorie possibili utilizzando l'algoritmo di Dijkstra. Questo algoritmo calcola le distanze minime da un nodo di partenza a tutti gli altri nodi del grafo.

Viene utilizzata una coda prioritaria (PriorityQueue) per mantenere i nodi in ordine crescente di distanza. Partendo dal nodo di partenza, l'algoritmo visita tutti i nodi e aggiorna le distanze più corte man mano che trova percorsi migliori. Una volta raggiunto il nodo di destinazione, l'algoritmo costruisce il percorso più veloce partendo dalla destinazione e risalendo i nodi precedenti utilizzando l'informazione memorizzata nella struttura dati ‘previous’.

```
    def find_fastest_path(self):

        self.visibility_graph()

        graph = {}

        for edge in self.edges:

            point1, point2 = edge

            if point1 not in graph:

                graph[point1] = []

            if point2 not in graph:

                graph[point2] = []

            graph[point1].append(point2)

            graph[point2].append(point1)



        distances = {point: math.inf for point in graph}

        distances[self.start] = 0

        previous = {point: None for point in graph}
```

```
queue = PriorityQueue()
queue.put((0, self.start))

while not queue.empty():
    dist, current_node = queue.get()
    if current_node == self.target:
        break
    if dist > distances[current_node]:
        continue
    for neighbor in graph[current_node]:
        new_dist = distances[current_node] + self.distance(current_node, neighbor)
        if new_dist < distances[neighbor]:
            distances[neighbor] = new_dist
            previous[neighbor] = current_node
            queue.put((new_dist, neighbor))

# Costruzione del percorso dal target al punto di partenza
path = []
current_node = self.target
while current_node != self.start:
    path.append(current_node)
    current_node = previous[current_node]
path.append(self.start)
path.reverse()

self.fastest_path = path
```

2.5 Funzione distance

La funzione ‘distance’ calcola la distanza euclidea tra due punti nel piano. È utilizzata nella funzione ‘find_fastest_path’ per calcolare le distanze tra i nodi del grafo.

2.6 Funzione visualize

La funzione ‘visualize’ disegna una rappresentazione grafica dell'ambiente e dei percorsi trovati. Utilizzando la libreria ‘matplotlib’, disegna gli ostacoli, i percorsi possibili in tratteggiato e il percorso più veloce in una traccia continua. I punti di partenza e destinazione vengono rappresentati rispettivamente con un cerchio blu e un cerchio rosso.

La funzione verifica inizialmente se sono stati calcolati i percorsi prima di eseguire la visualizzazione. Se i percorsi non sono stati calcolati, viene stampato un messaggio di avviso.

La funzione ‘visualize’ permette di ottenere una rappresentazione visiva dell'ambiente e dei percorsi calcolati, aiutando a comprendere meglio il processo di pianificazione del percorso.

2.7 Avvio applicazione

L’avvio del codice implica le seguenti visualizzazioni:

Visualizzazione del percorso più veloce

Il codice inizia stampando il percorso più veloce del robot. Viene utilizzato un loop ‘for’ per iterare attraverso i punti del percorso contenuti nell'attributo ‘fastest_path’ dell'oggetto ‘robot’. Ogni punto viene stampato a schermo per consentire la visualizzazione del percorso pianificato.

Ridimensionamento degli ostacoli

Successivamente, vengono eseguite operazioni per ridimensionare gli ostacoli presenti nell'ambiente. Viene definito un fattore di scala ‘scale_factor’, che rappresenta la percentuale di ridimensionamento. Questo fattore viene applicato a ciascun ostacolo presente nella lista ‘obstacles’, generando una nuova lista chiamata ‘scaled_obstacles’. L'obiettivo è ridimensionare gli ostacoli per adattarli alle dimensioni e alle esigenze specifiche dell'applicazione o della simulazione.

Avvio dell'applicazione GUI

Infine, il codice avvia un'applicazione GUI utilizzando PyQt (PyQt è un modulo di Python che offre binding per il framework Qt per sviluppare interfacce grafiche). Il cuore dell'applicazione è costituito da una finestra (‘MyCartWindow’) che visualizza il robot e gli ostacoli nell'ambiente. Prima di avviare l'applicazione, viene chiamata la funzione ‘visualize()’ per mostrare il grafico del robot con il percorso più veloce calcolato.

Complessivamente, questo codice combina la pianificazione del percorso più veloce per un robot mobile con la creazione di un'interfaccia grafica per l'interazione e la visualizzazione dell'ambiente. Questo permette di comprendere meglio come il robot si muoverà nell'ambiente, superando gli ostacoli in base al percorso calcolato, e offre la possibilità di sperimentare e ottimizzare il percorso in modo interattivo.

3. Script robot 2d two-wheels

La classe ‘MyCart2D rappresenta un robot mobile a 2 dimensioni con 2 ruote (un sistema robotico) e definisce il suo comportamento e controllo durante l'esecuzione dell'algoritmo del Visibility Graph per la creazione di percorsi pianificati. Spieghiamo le funzioni all'interno della classe:

1. ‘__init__(self, path)’: questo è il costruttore della classe. Viene chiamato quando si crea un'istanza della classe ‘Cart2DRobot’. Riceve come argomento un percorso pianificato ‘path’ composto da una serie di punti (coordinate x, y). Inizializza tutti i componenti del modello del robot denominato TwoWheelsCart2DEncodersOdometry, che si basa sull'odometria delle ruote encoder a due ruote del robot.
   Il modello del robot TwoWheelsCart2DEncodersOdometry tiene traccia della posizione del robot nel piano 2D e delle velocità lineari e angolari in base ai dati forniti dai suoi encoder. Viene inizializzato nel costruttore della classe Cart2DRobot, come parte del suo componente principale.
   Nel costruttore della classe, il modello TwoWheelsCart2DEncodersOdometry viene inizializzato con vari parametri:

   - 20: La distanza tra gli assi delle ruote del robot (distanza tra le due ruote).
   - 0.15: Raggio delle ruote sinistra e destra.
   - 0.8: Distanza dal centro del robot al punto di misurazione dell'encoder per entrambe le ruote.
   - 0.025: Variazione massima dell'accelerazione della ruota sinistra.
   - 0.025: Variazione massima dell'accelerazione della ruota destra.
   - 0.2: Coefficiente di attrito per il modello della dinamica del robot.
   - 0.02: Variazione massima della velocità angolare.
   - 0.02: Variazione massima della velocità lineare.
   - 0.24: Velocità massima della ruota del robot (in rad/s).
   - 2*math.pi/4000.0: Risoluzione dell'encoder, utilizzata per calcolare gli spostamenti delle ruote.

   Sono inclusi, nel costruttore, i controller (di velocità e polar), il modello del robot (‘cart’), il controllore di traiettoria (‘path_controller’) e le variabili per il tracciamento e la visualizzazione dei dati (‘plotter’).

2. ‘run(self)’: questa funzione rappresenta il ciclo di controllo principale del robot durante l'esecuzione dell'algoritmo del Visibility Graph. Durante l'esecuzione, il robot calcola il grafo di visibilità, cioè i collegamenti tra i punti critici dell'ambiente, utilizzando l'algoritmo del Visibility Graph. Successivamente, il robot valuta il percorso più veloce da seguire all'interno del grafo di visibilità, utilizzando il controllore di traiettoria (‘path_controller’). Quindi, calcola le velocità delle ruote (‘vref_l’ e ‘vref_r’) utilizzando il controllore polare (‘polar_controller’). I controllori di velocità (‘left_controller’ e ‘right_controller’) vengono utilizzati per calcolare le azioni di controllo da applicare alle ruote sinistra e destra. Infine, il modello del robot (‘cart’) viene utilizzato per aggiornare la posizione e la velocità del robot in base alle azioni di controllo calcolate.

3. ‘get_pose(self)’: questa funzione restituisce la posizione attuale del robot nel formato (x, y, theta), dove x e y rappresentano le coordinate della posizione e theta rappresenta l'angolo di orientamento.

4. ‘get_speed(self)’: questa funzione restituisce la velocità lineare ‘v’ e la velocità angolare ‘w’ del robot, ottenute dal modello del robot (‘cart’).

Le istanze di questa classe rappresentano il robot mobile all'interno dell'ambiente, e il metodo ‘run()’ consente al robot di eseguire l'algoritmo del Visibility Graph per la creazione di percorsi pianificati, tenendo conto dei vincoli imposti dai controllori e dal modello del robot. Durante l'esecuzione, il robot calcola il grafo di visibilità e utilizza il controllore di traiettoria per determinare il percorso più veloce all'interno del grafo. Infine, si muove lungo il percorso pianificato, aggiornando la sua posizione e velocità con il modello del robot.

```
class MyCart2D(RoboticSystem):

    def __init__(self, path):

        super().__init__(1e-3) # delta_t = 1e-3
        self.cart = TwoWheelsCart2DEncodersOdometry(20, 0.15, 0.8, 0.8,
                                                    0.025, 0.025, 0.2,
                                                    0.02, 0.02, 0.24, 2*math.pi/4000.0)

        self.left_controller = PIDSat(6.0, 4.0, 0.0, 2, True)
        self.right_controller = PIDSat(6.0, 4.0, 0.0, 2, True)

        self.polar_controller = Polar2DController(2, 2, 2, 2)

        self.path_controller = Path2D(0.2, 0.5, 0.5, 0.01)  # _vmax, _acc, _dec, _threshold
```

```
        converted_path = []
        for point in path:
            x, y = point
            converted_x = x / 1000
            converted_y = 0.5 - (y / 1000)
            converted_point = (converted_x, converted_y)
            converted_path.append(converted_point)
            print(converted_point)

        self.path_controller.set_path(converted_path)

        (x, y, _) = self.get_pose()
        self.path_controller.start((x, y))
        self.plotter = DataPlotter()



    def run(self):
        pose = self.get_pose()
        target = self.path_controller.evaluate(self.delta_t, pose)

        self.plotter.add('x', pose[0])#effettivo pos
        self.plotter.add('y', pose[1])
        self.plotter.add('x_t', self.path_controller.x_current)
        self.plotter.add('y_t', self.path_controller.y_current)
        if target is None: #target raggiunto
            self.plotter.plot(['x', 'x'], [['y', 'y']])
            self.plotter.plot(['x_t', 'x_t'], [['y_t', 'y_t']])
            self.plotter.show()
            return False

        (x_target, y_target) = target

        # polar control
        (v_target,    w_target)    =    self.polar_controller.evaluate(self.delta_t,    x_target,    y_target,
self.get_pose())
        vref_l = v_target - w_target * self.cart.encoder_wheelbase / 2.0
        vref_r = v_target + w_target * self.cart.encoder_wheelbase / 2.0
        (vl, vr) = self.cart.get_wheel_speed()

        # speed control (left, right)
        Tleft = self.left_controller.evaluate(self.delta_t, vref_l, vl)
        Tright = self.right_controller.evaluate(self.delta_t, vref_r, vr)

        # robot model
        self.cart.evaluate(self.delta_t, Tleft, Tright)

        return True

    def get_pose(self):
        return self.cart.get_pose()
```

```
    def get_speed(self):
        return self.cart.v, self.cart.w
```

4. Classi Environment ed Obstacle

4.1 Classe Environment

La classe ‘Environment’ ha un ruolo cruciale nel contesto del Visibility Graph, fornendo l'ambiente tridimensionale in cui il robot mobile si muove. I suoi attributi principali, ‘width’ e ‘height’, definiscono le dimensioni dell'ambiente, stabilendo i confini in cui il robot può pianificare il suo percorso. Inoltre, l'ambiente contiene una lista di ostacoli, rappresentati da oggetti, i quali possono essere aggiunti dinamicamente tramite il metodo ‘add_obstacle()’.

Durante la generazione del Visibility Graph, la classe ‘Environment’ svolge due compiti fondamentali. Innanzitutto, il metodo ‘is_valid_point()’ verifica se un punto specifico è valido per essere incluso nel grafo di visibilità. Un punto valido deve soddisfare due condizioni: deve trovarsi all'interno dei limiti dell'ambiente e non deve essere contenuto all'interno di alcun ostacolo. Ciò consente di escludere i punti non accessibili o quelli occupati dagli ostacoli, assicurando che solo i punti validi vengano considerati come nodi del grafo.

Inoltre, l'ambiente gestisce gli ostacoli presenti. Durante la generazione del Visibility Graph, ogni ostacolo rappresentato da un oggetto separato viene considerato per determinare come influenzi i segmenti di visibilità tra i punti. Il metodo ‘is_in_obstacle()’ verifica se un punto si trova all'interno di un qualsiasi ostacolo presente nell'ambiente. Questa informazione è essenziale per evitare che i segmenti di visibilità attraversino gli ostacoli, garantendo che il grafo rifletta solo i collegamenti tra i punti accessibili.

La classe ‘Environment’ contribuisce alla pianificazione del percorso tramite Visibility Graph fornendo l'ambiente e gestendo gli ostacoli. Questi aspetti consentono di creare un grafo di visibilità che rappresenti solo i punti accessibili e le connessioni tra di essi, fornendo una base solida per il robot per pianificare il percorso migliore in modo efficiente e sicuro, evitando gli ostacoli durante il movimento.

```
class Environment:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.obstacles = []

    def add_obstacle(self, obstacle):
        self.obstacles.append(obstacle)

    def is_valid_point(self, point):
        x, y = point
        if 0 <= x < self.width and 0 <= y < self.height and not self.is_in_obstacle(point):
            return True
        return False
```

```
    def is_in_obstacle(self, point):
        for obstacle in self.obstacles:
            if obstacle.is_inside(point):
                return True
        return False
```

4.2 Classe Obstacle

La classe ‘Obstacle’ definisce un ostacolo nel contesto del Visibility Graph. Un ostacolo è rappresentato da un insieme di vertici, che definiscono i punti che compongono il suo contorno. Il costruttore ‘__init__()’ inizializza l'oggetto ostacolo con i vertici forniti come input.

Il metodo ‘get_vertices()’ restituisce l'elenco dei vertici che compongono l'ostacolo, consentendo di accedere e utilizzare questa informazione da altre parti del programma.

La funzione ‘scale(self, scale_factor)’ è utilizzata per ridimensionare l'ostacolo rispetto a un fattore di scala dato. Questo metodo crea una copia dell'ostacolo originale, applica il fattore di scala ai suoi vertici e restituisce il nuovo ostacolo scalato. Ciò permette di adattare gli ostacoli a diverse dimensioni o risoluzioni senza alterarne la forma.

Il metodo ‘is_inside(self, point)’ verifica se un determinato punto si trova all'interno dell'ostacolo. Per fare ciò, utilizza l'algoritmo del punto in poligono, che traccia una linea orizzontale dal punto in questione e conta quante volte la linea interseca i lati del poligono. Se il numero di intersezioni è dispari, il punto si trova all'interno dell'ostacolo; se è pari, il punto è all'esterno.

La funzione ‘scale_polygon(vertices, scale_factor)’ è un'implementazione dell'operazione di ridimensionamento di un poligono. Prende in input una lista di vertici che rappresentano il poligono originale e un fattore di scala. La funzione trova il centro del poligono originale, quindi trasla il poligono in modo che il centro sia all'origine (0, 0). Successivamente, applica il fattore di scala ai vertici traslati e quindi trasla nuovamente il poligono scalato al centro originale. In questo modo, la funzione restituisce la lista dei nuovi vertici del poligono scalato.

```
class Obstacle:
    def __init__(self, vertices):
        self.vertices = vertices

    def get_vertices(self):
        return self.vertices

    def scale(self, scale_factor):
        scaled_vertices = scale_polygon(self.vertices, scale_factor)
        return Obstacle(scaled_vertices)

    def is_inside(self, point):
        x, y = point
        num_vertices = len(self.vertices)
        inside = False
        p1x, p1y = self.vertices[0]
```

```
        for i in range(num_vertices + 1):
            p2x, p2y = self.vertices[i % num_vertices]
            if y > min(p1y, p2y):
                if y <= max(p1y, p2y):
                    if x <= max(p1x, p2x):
                        if p1y != p2y:
                            xinters = (y - p1y) * (p2x - p1x) / (p2y - p1y) + p1x
                        if p1x == p2x or x <= xinters:
                            inside = not inside
            p1x, p1y = p2x, p2y
        return inside

def scale_polygon(vertices, scale_factor):
    # Trova il centro del poligono originale
    center_x = sum(x for x, _ in vertices) / len(vertices)
    center_y = sum(y for _, y in vertices) / len(vertices)

    # Trasla il poligono in modo che il centro sia all'origine (0, 0)
    translated_vertices = [(x - center_x, y - center_y) for x, y in vertices]

    # Applica il fattore di scala ai vertici del poligono traslato
    scaled_vertices = [(x * scale_factor, y * scale_factor) for x, y in translated_vertices]

    # Trasla nuovamente il poligono scalato al centro originale
    scaled_vertices = [(x + center_x, y + center_y) for x, y in scaled_vertices]

    return scaled_vertices
```

5. Risultati

Di seguito è presente il risultato dell'esecuzione del codice, rappresentato nell'immagine del grafico.

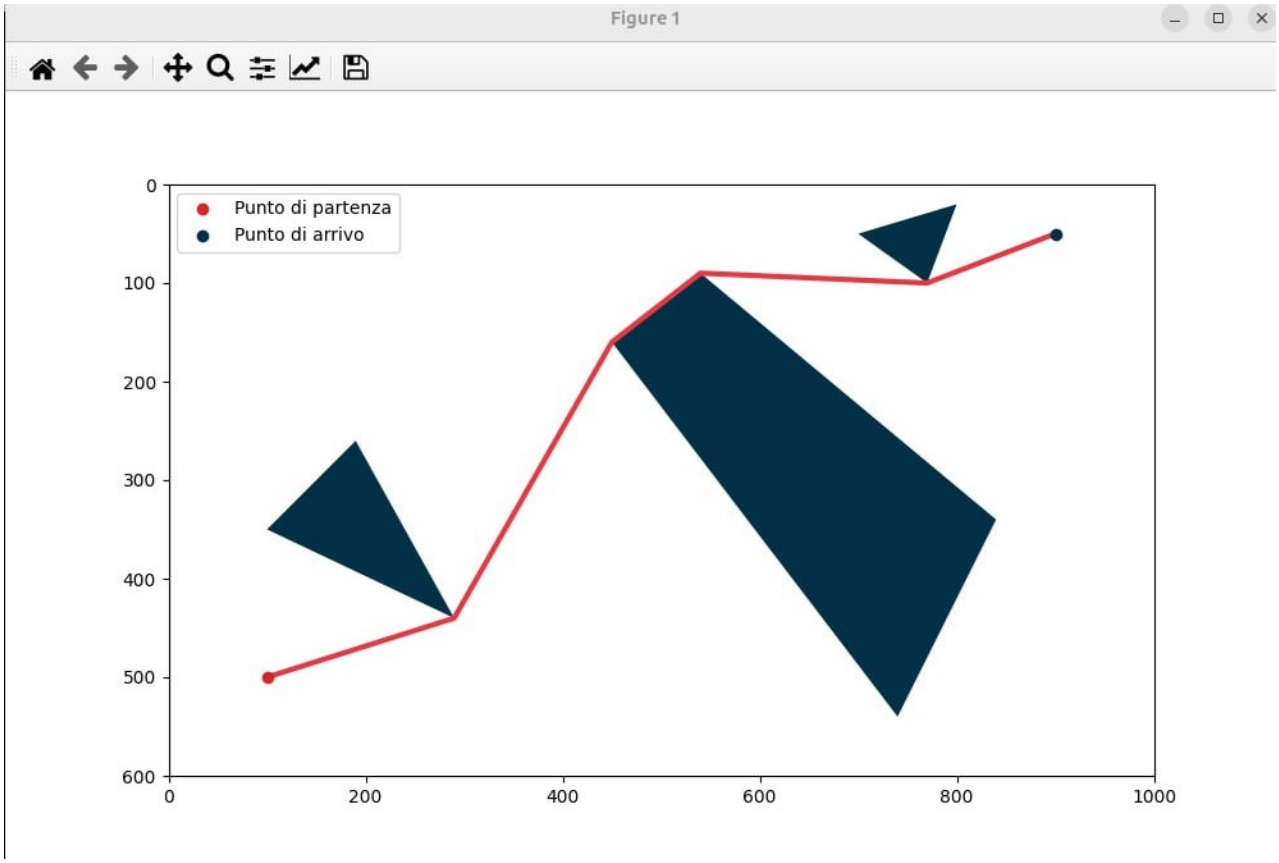

Figura 1 - Grafico Visibility Graph.

Il grafico risultante ha svolto un ruolo fondamentale nel visualizzare in modo chiaro e intuitivo il processo di pianificazione del percorso tramite il Visibility Graph. L'immagine mostra chiaramente la disposizione degli ostacoli nell'ambiente, evidenziando i punti critici che costituiscono i possibili cammini in giallo nel grafo di visibilità.

In particolare, il percorso più veloce, rappresentato in rosso, ha fornito una chiara rappresentazione del percorso ottimale individuato dall'algoritmo. La visualizzazione del percorso ha permesso di comprendere come il robot o l'oggetto mobile si muoveranno all'interno dell'ambiente, superando gli ostacoli in modo efficiente e sicuro lungo il cammino più breve.

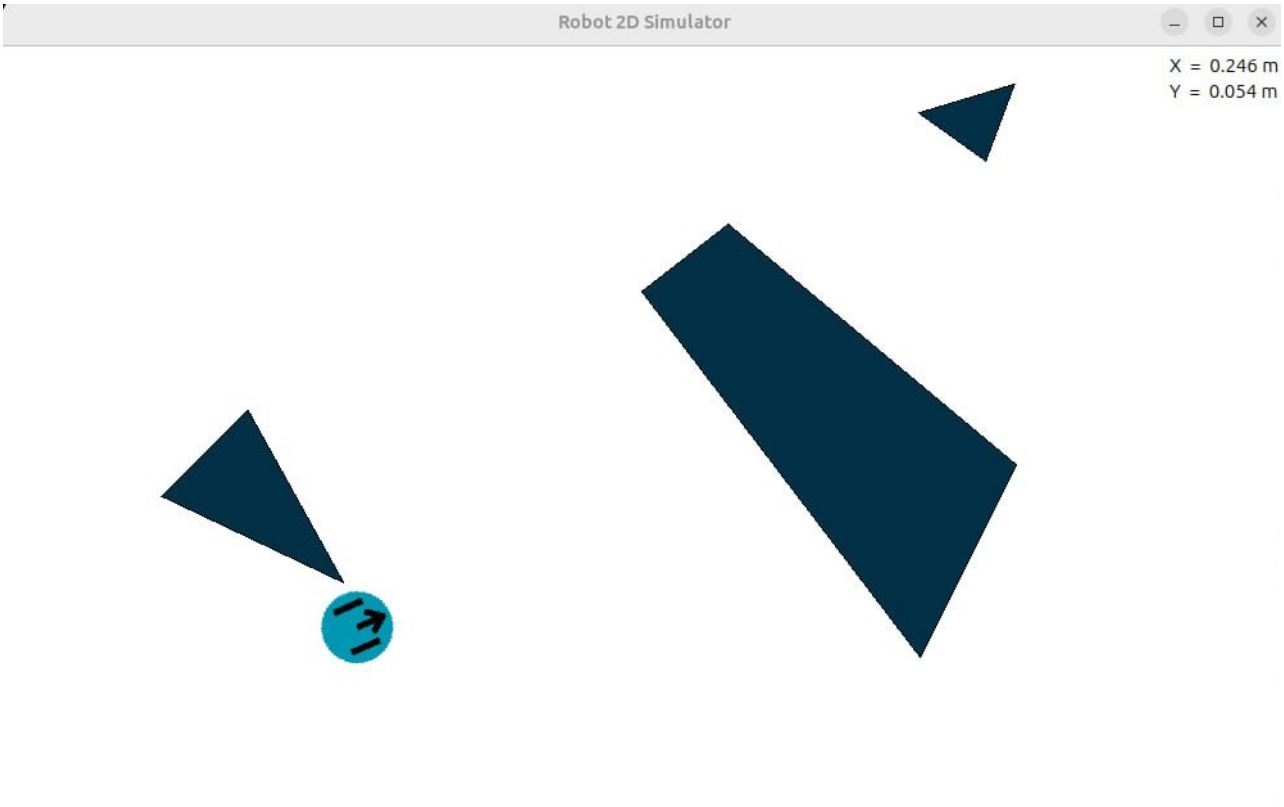

Figura 2 - Screen del movimento del robot.

La figura 2 è uno screen della simulazione con il robot che segue la traiettoria del Visibility Graph e si muove tra gli ostacoli nell'ambiente è stato fondamentale per mostrare l'efficacia dell'algoritmo di pianificazione del percorso implementato. L'immagine cattura un momento del movimento del robot, evidenziando come esso sia in grado di evitare gli ostacoli lungo il percorso pianificato e seguire il cammino più veloce e sicuro. Questa visualizzazione ha permesso di comprendere in modo tangibile e visivo il funzionamento dell'algoritmo e la capacità del robot di navigare in un ambiente 2D complesso, superando gli ostacoli in modo efficiente.

Conclusioni

Il progetto aveva come obiettivo la pianificazione del percorso utilizzando il Visibility Graph, si è osservato che l'algoritmo offre un approccio robusto e efficiente per la pianificazione del movimento di robot e oggetti mobili in ambienti bidimensionali con ostacoli. Si è notato che il grafo di visibilità, una volta costruito, permette di trovare il percorso più breve tra i punti di partenza e arrivo del robot, evitando gli ostacoli in modo efficiente e sicuro. L'efficienza dell'algoritmo dipende dalla complessità della scena e dal numero di punti critici, e in ambienti molto grandi o con numerosi ostacoli, potrebbe essere necessario utilizzare tecniche di ottimizzazione o semplificazione.

È stato osservato che l'implementazione dell'algoritmo in Python ha dimostrato la sua efficacia nella pianificazione del movimento del robot, con una struttura di codice ben organizzata che consente la creazione di un ambiente di simulazione e la gestione degli ostacoli. L'approccio del Visibility Graph è particolarmente adatto per ambienti statici, dove gli ostacoli non cambiano posizione nel corso del tempo. In ambienti dinamici, è necessario integrare l'algoritmo con strategie di rilevamento e reattività agli ostacoli in movimento.

In conclusione, l'utilizzo del Visibility Graph rappresenta un approccio flessibile e potente per la pianificazione del percorso di robot e oggetti mobili in ambienti complessi, garantendo un movimento efficiente e sicuro attraverso la generazione di percorsi ottimali e l'evitamento di ostacoli.