Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Triennale in Ingegneria Informatica

Esplorazione Multi-Robot di Ambienti non Convessi con ostacoli

Candidato
Luca Leuter

Relatore
Prof. Giorgio Battistelli

Anno Accademico 2020/2021

Alla mia famiglia e ai miei parenti, per avermi sostenuto sempre. In particolare a mio Babbo, e a mia Mamma, la mia roccia e il mio modello fin da quando ero piccolo;

A Chiara, il mio amore e la mia compagna di vita, senza di te non sarei arrivato fin qui;

Ai miei amici: TBF i miei amici da sempre, i dell'Arte del Ricamo e i PratoM compagni di viaggio in quest'avventura, i Ragazzi di Lagomaggio, per avermi regalato estati stupende; In particolare a Dario, Simone, Alessandro e Paolo, grazie per avermi sostenuto sempre ed essermi stati accanto;

Al professore Giorgio Battistelli, per avermi concesso il suo tempo e aver condiviso le sue conoscenze con me;

A chiunque abbia condiviso anche un minimo momento di studio o di risate con me;

A me stesso, per la tenacia nonostante tutte le difficoltà nel cammino

Non bastano le parole per ringraziarvi.

Luca

Indice

Introduzione

Il problema dell'**esplorazione** è uno dei problemi principali della robotica moderna, il cui obiettivo è mappare un ambiente sconosciuto con più robot. Le applicazioni sono diverse e includono missioni di ricerca e salvataggio, come la ricerca di sopravvissuti dopo una catastrofe in ambienti inaccessibili all'uomo o l'esplorazione della superficie di altri pianeti o corpi celesti. In questo elaborato limiteremo la nostra attenzione alle strategie di esplorazione basate sulle **frontiere**. Nell'esplorazione basata sulle frontiere, una frontiera è il confine che separa lo spazio conosciuto (già esplorato) da quello sconosciuto (ancora da esplorare). Finché esiste una frontiera, il robot si sposta in una posizione sulla frontiera e mappa il nuovo ambiente. La ripetizione di questo schema porta alla fine a una completa esplorazione dell'ambiente.

L'aspetto principale nell'esplorazione multi-robot è il coordinamento efficiente di un gruppo di robot. Essi formano un sistema **multi-agent**, in cui ogni agente mantiene la propria autonomia collaborando però con gli altri per raggiungere il comune obiettivo. Ciò riduce molto gli indici di complessità e di costo, poiché ogni agente è in grado di risolvere un sottoproblema in parallelo con gli altri, migliorando anche l'affidabilità del sistema rispetto a uno single-agent. D'altra parte però si introducono nuovi problemi come la complessità di coordinazione dei robot e la loro gestione.

In questo elaborato considereremo un **ambiente sconosciuto**, **non-convesso** e **con la presenza di ostacoli**.

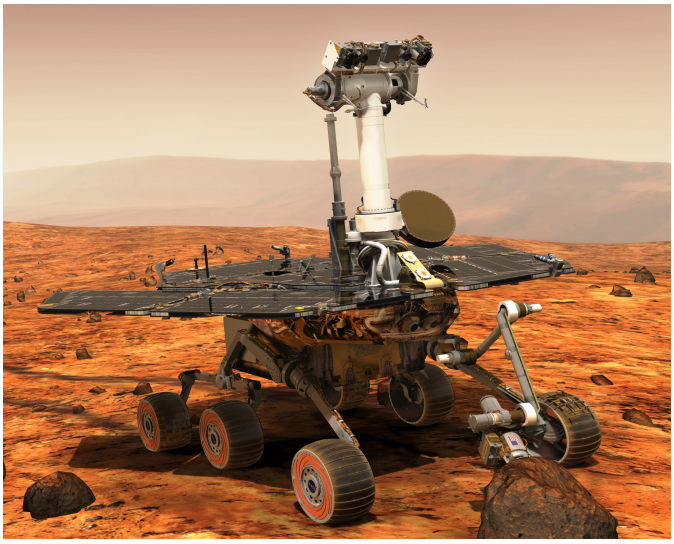

Figura 1: Nasa Mars Rover

Capitolo 1

Concetti e Definizioni

In questo capitolo verranno descritti i concetti e le definizioni utili per affrontare il problema dell'esplorazione.

1.1 Ambiente non convesso

L'ambiente che prenderemo in considerazione è un ambiente non convesso e con la presenza di ostacoli. In geometria euclidea un **insieme convesso** è un insieme nel quale, per ogni coppia di punti, il segmento che li congiunge è interamente contenuto nell'insieme. Più precisamente, sia V uno spazio vettoriale. Un insieme A si dice convesso se per ogni coppia di punti x, y in A il segmento che li congiunge:$(1-t)x + ty : t \in (0,1)$
è **interamente contenuto in** A.

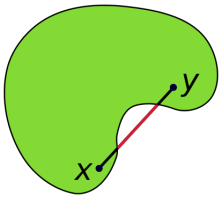

Figura 1.1: Insieme non Convesso

In questo elaborato l'ambiente è presentato come un rettangolo di dimensioni variabili diviso in tanti quadratini che chiameremo celle, ognuno dei quali rappresenterà una porzione di ambiente da scoprire. La presenza di ostacoli rende l'insieme non convesso, poiché un segmento che congiunge 2 punti qualsiasi potrebbe passare per un ostacolo, andando contro la definizione di insieme convesso.

1.2 Frontiera

In topologia, la **frontiera** o contorno di un sottoinsieme S di uno spazio topologico X è la chiusura dell'insieme meno il suo interno. Un elemento della frontiera di S è chiamato punto di frontiera di S. Un altro modo per definire la frontiera è il seguente: si definisce frontiera di S l'intersezione fra la chiusura di S e la chiusura del suo complementare.

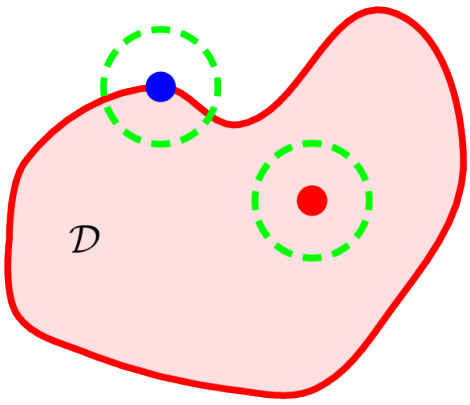

Figura 1.2: Esempio di Frontiera di un Insieme

Nella figura 1.2 vediamo un esempio di frontiera di un insieme D: la parte rosa rappresenta l'insieme D, la linea rossa è la frontiera e il punto blu è un punto di essa.

1.3 Esplorazione basata sulle frontiere

Descriviamo ora la **dinamica** di ogni robot: ognuno di essi ha una porzione di area scoperta, (insieme di celle) rispetto alla quale viene calcolata la frontiera. Successivamente ogni robot sceglie un punto della frontiera verso cui muoversi, il **target point**, che sarà la cella di frontiera più vicina al robot. Successivamente viene calcolato il percorso più breve per raggiungere il target point tramite l'**algoritmo di Dijkstra**, e il robot inizierà il movimento. Muovendosi verso il target point il robot potrà mappare nuove zone. Raggiunto il target point viene calcolata la nuova frontiera e si ripete il ciclo, fino a che può essere calcolata una nuova frontiera. Quando il robot non può più calcolare una frontiera allora si disattiva e quando tutti i robot sono disattivati allora l'esplorazione termina.

1.3.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve il problema dei cammini minimi in un grafo orientato pesato, nel caso in cui tutti i pesi degli archi non siano negativi. Fu inventato nel 1956 dall'informatico olandese Edsger Dijkstra. Tale algoritmo trova applicazione in molteplici contesti quali l'ottimizzazione nella realizzazione di reti (idriche, telecomunicazioni, stradali, circuitali, ecc.) o l'organizzazione e la valutazione di **percorsi runtime nel campo della robotica**, come nel caso in esame. Si riporta lo pseudo-codice dell'algoritmo:

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Si può considerare 3 parti dell'algoritmo:

- Inizializzazione e creazione della coda di priorità Q (Righe 1-3)
- Estrazione del minimo (Righe 5-6)
- Rilassamento degli archi (Riga 8)

La complessità computazionale dell'algoritmo di Dijkstra può essere espressa in funzione di $|V|$ ed $|E|$ ossia, rispettivamente, il numero di nodi e degli archi appartenenti al grafo sul quale viene eseguito. L'algoritmo mantiene la coda di priorità Q chiamando 3 operazioni delle code di priorità: INSERT (implicita

nella riga 3), EXTRACT-MIN (riga 5) e DECREASE-KEY (implicita in Relax, riga 8). Il tempo di esecuzione dell'algoritmo di Dijkstra dipende dal modo in cui viene implementata la coda di priorità. In questo elaborato l'algoritmo permette di trovare il cammino minimo dal punto in cui si trova il robot al target point, usando come coda di priorità un MIN-HEAP BINARIO. Un heap è una struttura dati basata sugli alberi che soddisfa la cosiddetta "proprietà di heap": se un nodo A è un genitore di un nodo B, allora la chiave di A è ordinata rispetto alla chiave di B conformemente alla relazione d'ordine applicata all'intero heap (min o max). La scelta di usare l'Heap-Binario invece che un semplice array è stata fatta per sfruttare la libreria HEAPQ di Python per la gestione degli Heap-Binari, e per il tempo di esecuzione minore. In particolare ogni operazione di EXTRACT-MIN richiede un tempo di $O(lnV)$ e ci sono $|V|$ operazioni come questa. Il tempo di costruzione di un min-heap è $O(V)$. Ogni operazione DECREASE-KEY richiede un tempo $O(lnV)$ e ci sono al più $|E|$ operazioni come questa. Il tempo totale di esecuzione è dunque $O((V+E)lnV)$, migliore rispetto alla semplice implementazione con l'array che risulta essere $O(V^2+E)$

Capitolo 2

Implementazione

In questo capitolo viene presentata la metodologia di soluzione al problema dell'esplorazione con il relativo codice. L'elaborato è stato svolto con il linguaggio di programmazione **Python** e la relativa libreria **Pygame**.

2.1 Mappa

L'ambiente non convesso è stato rappresentato come una griglia composta da tanti quadrati o celle di dimensione 20 pixel ciascuno. Ognuno di essi ha un diverso colore, che rappresenta la sua tipologia e ne facilita la visualizzazione:

- **Grigio Chiaro o Grigio Scuro:** la cella non è stata ancora esplorata
- **Bianco**: la cella è stata esplorata e non è un ostacolo, quindi i robot possono passarci sopra
- **Nero**: la cella è stata esplorata ed è un ostacolo, quindi i robot non possono attraversarla
- **Rosso**: la cella appartiene alla frontiera di un robot

- **Blu**: cella di frontiera scelta dal robot come punto target

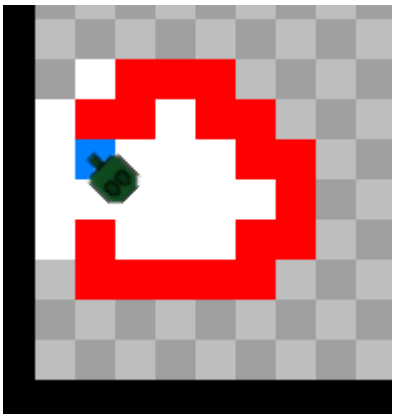

Figura 2.1: Particolare della mappa

All'inizio del programma viene creata la mappa con la funzione MAP-CREATE() del file MAP.PY, che sfrutta oggetti di tipo MAPRECT che rappresentano le celle. Ad eccezione del bordo mappa che viene subito contrassegnato come ostacolo e visualizzato, le altre celle vengono create come da scoprire. Inoltre per ognuna di esse vengono trovati i vicini nelle 8 possibili direzioni e salvati i loro numeri identificativi in un vettore proprio di ogni cella, insieme alla distanza da ognuno di essi che servirà poi nel calcolo del percorso minimo.

```
//Map.py
import pygame as pg
import math
GRIDSIZE = 20
class MapRect(pg.sprite.Sprite):
    def __init__(self, x, y, color):
        super().__init__()
        self.rect = pg.Rect((x, y), (GRIDSIZE, GRIDSIZE))
        self.color = color
        self.found = 0
        self.is_obstacle = False
        self.id = [int(x/GRIDSIZE), int(y/GRIDSIZE)]
        self.neighbour_ids = []
        self.neighbour_distances = {}

def map_create(window):
    surface = pg.Surface([window.get_width(), window.get_height()])
    map = []
    n_total_tiles = 0
    for i in range(0, window.get_width(), GRIDSIZE):
        for j in range(0, window.get_height(), GRIDSIZE):
            if ((i + j)/GRIDSIZE) % 2 == 0:
                r = MapRect(i, j, GREY)
            else:
                r = MapRect(i, j, LIGHTGREY)
            if i == 0 or i == window.get_width() - GRIDSIZE or j == 0
                or j == window.get_height() - GRIDSIZE:
                r.is_obstacle = True
```

```
            r.color = BLACK
        pg.draw.rect(surface, r.color, r.rect)
        map.append(r)
        n_total_tiles += 1

# Ostacolo 1
i = GRIDSIZE * 20
for j in range(400, window.get_height(), GRIDSIZE):
    for m in map:
        if m.rect.x == i and m.rect.y == j:
            m.is_obstacle = True

# Neighbour
for m in map:
    i += 1
    if not m.is_obstacle:
        m.neighbour_ids.append((m.id[0] - 1, m.id[1] - 1))
        m.neighbour_distances[1] = math.dist(m.rect.center,
                                             search_by_id(map,
                                                 (m.id[0] - 1,
                                                 m.id[1] -
                                                 1)).rect.center)

# Viene ripetuto per ogni direzione
return surface, map, n_total_tiles
```

2.2 Algoritmo di Dijkstra

Per calcolare il percorso minimo dalla posizione del robot al target viene usato l'algoritmo di Dijkstra. L'elemento principale del file DIJKSTRA.PY è la classe GRAFO, che rappresenta appunto un **Grafo**. In matematica e in informatica si dice grafo una coppia ordinata $G = (V, E)$ di insiemi, con V insieme dei nodi ed E insieme degli archi, tali che gli elementi di E siano coppie di elementi di V. Due vertici u, v sono connessi da un arco se esiste $(u, v) \in E$. Ad ogni arco può essere assegnato un valore $a(u, v) > 0$, detto peso dell'arco. Nel caso dell'elaborato ogni cella della mappa è un nodo, mentre ogni arco è la connessione tra una cella e i suoi vicini, che sono gli unici raggiungibili da essa. I pesi degli archi rappresentano la distanza tra la cella considerata e i suoi vicini. Usando l'algoritmo di Dijkstra viene quindi trovato il percorso minimo. All'iterazione iniziale e successivamente ogni volta che viene raggiunto un target, ogni robot crea un grafo usando gli identificativi delle celle bianche che ha scoperto e la cella target. Gli verrà fornito dalla funzione SHORTEST-PATH() della classe GRAFO un vettore contenente gli identificativi delle caselle che compongono il percorso minimo. La classe GRAFO è stata realizzata usando la libreria HEAPQ di Python.

```
//Dijkstra.py
import heapq
class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, name, edges):
        self.vertices[name] = edges

    def shortest_path(self, start, finish):
        distances = {}
        previous = {}
        nodes = []
        for vertex in self.vertices:
            if vertex == start: # Mette il nodo sorgente alla
                distanza 0
                distances[vertex] = 0
                heapq.heappush(nodes, [0, vertex])
            else:
                distances[vertex] = sys.maxsize
                heapq.heappush(nodes, [sys.maxsize, vertex])
            previous[vertex] = None
        while nodes:
            smallest = heapq.heappop(nodes)[1] # Vertice in "nodes"
                con la distanza minore in "distances"
            if smallest == finish: # Se il nodo pi vicino  il target
                allora fine
                path = []
```

```
            while previous[smallest]:
                path.append(smallest)
                smallest = previous[smallest]
            path.reverse()
            return path
        if distances[smallest] == sys.maxsize: # Tutti i vertici
            rimanenti sono inaccessibili dalla sorgente
            break
        for neighbor in self.vertices[smallest]: # Guarda tutti i
            vicini del nodo considerato
            alt = distances[smallest] +
                self.vertices[smallest][neighbor]
            if alt < distances[neighbor]: # Se c' un nuovo
                percorso pi breve aggiorna la coda di priorit
                distances[neighbor] = alt
                previous[neighbor] = smallest
                for n in nodes:
                    if n[1] == neighbor:
                        n[0] = alt
                        break
                heapq.heapify(nodes)
    return distances
```

2.3 I Robot

In questo capitolo verrà illustrata la classe che rappresenta i robot, in particolare come viene creata la frontiera e come avviene il movimento. La classe robot è una classe derivata dalla classe SPRITE della libreria PYGAME. Ciò ha facilitato la gestione delle collisioni tra i robot e le celle della mappa, che sono gestite dal programma principale, e ha permesso l'uso di un immagine per ogni robot.

2.3.1 Update

La funzione UPDATE() è la funzione principale di ogni robot e viene eseguita per ogni robot ad ogni iterazione del programma principale. E' una funzione propria della classe SPRITE. Se il robot non è nello stato di movimento viene calcolata la frontiera e il punto di start con la funzione CALCULATE-START-AND-FRONTIER() e il target con la funzione CALCULATE-TARGET(). Se il target non viene trovato allora vuol dire che la frontiera è vuota, quindi il programma principale provvede a disattivare il robot con la funzione DEACTIVATE(). Se viene trovato invece viene calcolato il percorso minimo con l'algoritmo di Dijkstra descritto nel paragrafo 2.2, usando la funzione CALCULATE-PATH(). A questo punto il robot viene messo nello stato di movimento, e viene usata la prima cella del percorso minimo come target temporaneo del robot. Il robot poi calcola l'angolo δ tra lui e il target temporaneo, ruota usando quest'informazione e si muove verso di esso.

La dinamica del robot è stata implementata con la seguente formula:

$$p^{k+1} = p^k + v * \begin{pmatrix} cos\delta \\ sin\delta \end{pmatrix}$$

dove p è la posizione del robot, al tempo discreto k e $k+1$, v è la velocità uguale per ogni robot e δ è l'angolo tra il robot e il target.

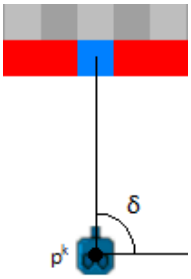

Figura 2.2: δ e p^k

Ad ogni aggiornamento della posizione il robot controlla se la distanza tra di esso e il target attuale è minore di una certa tolleranza fissata a priori. In caso positivo allora estrae un altro target dal percorso minimo e si rimette in movimento fino a raggiungerlo. Tutto il processo viene iterato fino a che esiste una frontiera.

2.3.2 Start e Frontiera

La funzione CALCULATE-START-AND-FRONTIER() calcola il punto di start e la **frontiera** basandosi sulle informazioni raccolte dal robot fino al momento della chiamata. Per calcolare lo start semplicemente il robot guarda qual è la cella più vicino a lui tra quelle contenute nel vettore WHITE-TILES, che rappresenta le celle che il robot stesso ha scoperto. Tale vettore viene usato anche nel calcolo della frontiera: il robot esamina ogni cella di WHITE-TILES

guardando se i vicini della cella non sono degli ostacoli, e se essa ha almeno una cella vicina non ancora scoperta. In caso positivo allora la cella è un punto di frontiera, quindi viene aggiunto al vettore FRONTIER e colorata di rosso.

2.3.3 Target

La funzione CALCULATE-TARGET() calcola il punto target usando le celle della frontiera. Semplicemente il robot sceglie come target la cella di frontiera più vicina a lui, che viene poi colorata di blu. Se la frontiera è vuota viene ritornato il valore NULL dalla funzione.

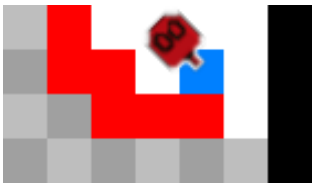

Figura 2.3: Frontiera e target

2.3.4 Calcola Percorso

La funzione CALCULATE-PATH() crea un grafo usando la classe GRAFO del file DIJKSTRA.PY. Vengono usati come nodi gli identificatori delle celle contenute in WHITE-TILES() (che sono le celle scoperte dal robot) e come archi e i relativi pesi le distanze tra ogni cella e i suoi vicini. La funzione SHORTEST-PATH() poi restituirà il percorso minimo tra la cella di start e la cella target.

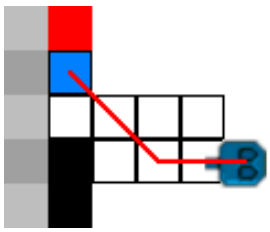

Figura 2.4: Percorso minimo per un punto target

```
//Robot.py
import pygame as pg
import math
import Dijkstra

# Colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREEN = (0, 153, 76)
RED = (255, 0, 0)
BLUE = (0, 128, 255)

robot_image = pg.image.load("robot1.png")
robot_image_2 = pg.image.load("robot2.png")
robot_image_3 = pg.image.load("robot3.png")
# Speed
v = 1
# Delta
pi = math.pi
converter = 180 / pi
tollerance = 0.1

class Robot(pg.sprite.Sprite):
    def __init__(self, id, x, y, line_color):
        super().__init__()
        self.id = id
        self.image = robot_image
        self.x = x
```

```
        self.y = y
        self.radius = 35
        self.rect = self.image.get_rect(center=(x, y))
        self.delta = standard_delta[0]
        self.white_tiles = []
        self.frontier = []
        self.is_moving = False
        self.target = None
        self.target_vector = []
        self.current_target = None
        self.path = []
        self.empty_frontier = False
        self.line_color = line_color
        self.first = True
        self.disabled = False

    def update(self, surface, map):
        if not self.is_moving:
            start = self.calculate_start_and_frontier(surface, map)
            if self.first:
                self.target_vector.append(start.rect.center)
                self.first = False
            self.target = self.calculate_target()
            if self.target is None:
                self.empty_frontier = True
                self.is_moving = False
            if not self.empty_frontier:
                self.target_vector.append(self.target.rect.center)
                pg.draw.rect(surface, BLUE, self.target.rect)
```

```
            self.path = self.calculate_path(start)
            trg = self.path.pop(0)
            for m in map:
                if m.id[0] == trg[0] and m.id[1] == trg[1]:
                    self.current_target = m
                    break
            self.is_moving = True
        if not self.empty_frontier:
            if math.dist(self.rect.center, self.target.rect.center) <
            tollerance:
            self.is_moving = False
            self.current_target = None
            for f in self.frontier:
                pg.draw.rect(surface, WHITE, f.rect)
            self.frontier.clear()
            self.path.clear()
        if self.current_target is not None:
            self.delta =
                math.atan2(self.current_target.rect.center[1] -
                self.rect.center[1],
                                    self.current_target.rect.center[0]
                                        - self.rect.center[0])
            x, y = self.rect.center
            if self.id % 3 == 1:
                self.image, self.rect = rotate(robot_image, x, y,
                    self.delta)
            elif self.id % 3 == 2:
                self.image, self.rect = rotate(robot_image_2, x,
                    y, self.delta)
```

```
            elif self.id % 3 == 0:
                self.image, self.rect = rotate(robot_image_3, x,
                    y, self.delta)
            dx = math.cos(self.delta) * v
            dy = math.sin(self.delta) * v
            self.x += dx
            self.y += dy
            self.rect = self.image.get_rect(center=(self.x,
                self.y))
            if math.dist(self.rect.center,
                self.current_target.rect.center) < tollerance and
                \
                    self.current_target.id != self.target.id:
                trg = self.path.pop(0)
                for m in map:
                    if m.id[0] == trg[0] and m.id[1] == trg[1]:
                        self.current_target = m
                        break

    def calculate_start_and_frontier(self, surface, map):
        start = None
        min_dist = 10000
        for m in self.white_tiles:
            if self.rect.colliderect(m.rect):
                if math.dist(self.rect.center, m.rect.center) <
                    min_dist:
                    start = m
                    min_dist = math.dist(self.rect.center,
                        m.rect.center)
```

```
frontier = []
for t in self.white_tiles:
    grey_neighbour = False
    black_neighbour = False
    for m in map:
        n = 0
        for i in range(0, len(t.neighbour_ids)):
            if m.id[0] == t.neighbour_ids[i][0] and m.id[1]
                == t.neighbour_ids[i][1]:
                n += 1
                if m.is_obstacle:
                    black_neighbour = True
                    grey_neighbour = False
                elif m.found == 0:
                    grey_neighbour = True
                break
        if black_neighbour:
            break
        if n == 4:
            break
    if grey_neighbour:
        frontier.append(t)
for f in frontier:
    if f not in self.frontier:
        self.frontier.append(f)
        f.color = RED
        pg.draw.rect(surface, f.color, f.rect)
return start
```

```
def calculate_target(self):
    minimum = None
    min_dist = 0
    for f in self.frontier:
        if math.dist(self.rect.center, f.rect.center) < min_dist
            or min_dist == 0:
            min_dist = math.dist(self.rect.center, f.rect.center)
            minimum = f
    return minimum


def calculate_path(self, start):
    g = Dijkstra.Graph()
    nodes = {}
    i = 0
    for t in self.white_tiles:
        nodes[i] = t.id
        i += 1
    for n in self.white_tiles:
        l = -1
        vertex = {}
        for k in range(0, len(nodes)):
            if nodes[k] == n.id:
                l = k
                break
        for i in range(1, len(n.neighbour_ids) + 1):
            for j in range(0, len(nodes)):
                if n.neighbour_ids[i - 1][0] == nodes[j][0] and
                    n.neighbour_ids[i - 1][1] == nodes[j][1]:
                    tmp = str(j)
```

```
                    vertex[tmp] = n.neighbour_distances[i]
                    break
            l = str(l)
            g.add_vertex(l, vertex)
        s, t = 0, 0
        for i in range(0, len(nodes)):
            if nodes[i] == start.id:
                s = str(i)
            if nodes[i] == self.target.id:
                t = str(i)
        tmp_path = g.shortest_path(s, t)
        path = []
        for v in tmp_path:
            path.append(nodes.get(int(v)))
        return path

    def deactivate(self):
        self.disabled = True

def rotate(image, x, y, angle):
    angle = angle * converter
    angle = -angle
    rotated_image = pg.transform.rotozoom(image, angle, 1)
    rotated_rect = rotated_image.get_rect(center=(x, y))
    return rotated_image, rotated_rect
```

2.4 Main

Nel file MAIN.PY si trova il ciclo principale dell'elaborato. Prima di entrare in esso però vengono inizializzati l'array dei robot, la mappa e la SURFACE, un oggetto della libreria PYGAME che serve per rappresentare le immagini e permettere la loro visualizzazione. Entrando poi nel ciclo WHILE che tiene la finestra attiva troviamo il ciclo che controlla per ogni robot se nel suo raggio d'azione (che corrisponde al parametro RADIUS della classe robot) sono presenti celle non scoperte. In caso positivo le scopre rivelando il loro colore e se non sono ostacoli le aggiunge al vettore WHITE-TILES del robot che le ha scoperte. Inoltre viene settato il parametro FOUND a 1, in modo che gli altri robot capiscano che tale cella è già stata scoperta. Successivamente il programma principale controlla se qualcuno dei robot ha finito l'esplorazione, cioè se la frontiera del robot è vuota. In tal caso viene disattivato con la funzione DEACTIVATE() e stampato il suo percorso sulla mappa. Altrimenti per ogni robot viene chiamata la funzione UPDATE(). Viene poi aggiornata la mappa con gli eventuali nuovi colori delle celle e le posizioni dei robot. Infine viene controllato se tutti i robot hanno finito l'esplorazione, in tal caso viene stampato che l'esplorazione ha avuto successo e il tempo di esecuzione.

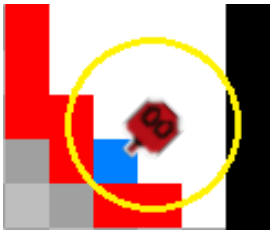

Figura 2.5: Raggio di scoperta del robot

```
//main.py
import pygame as pg
import math
import Map
import Robot
import time

# Colors
BLACK = (0, 0, 0)
LIGHTBLACK = (32, 32, 32)
WHITE = (255, 255, 255)
DARKWHITE = (224, 224, 224)
GREY = (160, 160, 160)
LIGHTGREY = (190, 190, 190)
GREEN = (0, 153, 76)
RED = (255, 0, 0)
BLUE = (0, 128, 255)
# Window
pg.display.init()
window = pg.display.set_mode((800, 600))
robot_image = pg.image.load("robot1.png")
GRIDSIZE = Map.GRIDSIZE
ROWS = int(window.get_width() / GRIDSIZE)
COLUMNS = int(window.get_height() / GRIDSIZE)

# Delta
pi = math.pi
converter = 180 / pi
```

```
def main():
    pg.display.set_caption("Non-Convex Exploration")
    icon = pg.image.load("robot1.png")
    pg.display.set_icon(icon)
    is_running = True
    surface, map, n_total_tiles = Map.map_create(window)
    # Robots
    robots = [Robot.Robot(1, 110, window.get_height() - 110, BLACK),
              Robot.Robot(2, 300, window.get_height() - 310, GREEN),
              Robot.Robot(3, 600, window.get_height() - 190, RED)
              ]
    start = time.time()
    l = 0
    # Main Loop
    while is_running:
        for event in pg.event.get():
            if event.type == pg.QUIT:
                is_running = False
        for m in map:
            for robot in robots:
                if not m.found:
                    if pg.sprite.collide_circle(robot, m):
                        if m.is_obstacle:
                            m.color = BLACK
                        elif m.neighbour_ids:
                            m.color = WHITE
                            robot.white_tiles.append(m)
                        m.found = 1
```

```
                pg.draw.rect(surface, m.color, m.rect)
        for robot in robots:
            if not robot.empty_frontier:
                robot.update(surface, map)
            else:
                if not robot.disabled:
                    l += 1
                    for i in range(1, len(robot.target_vector)):
                        pg.draw.line(surface, robot.line_color,
                            robot.target_vector[i],
                            robot.target_vector[i - 1], 2)
                    print(l)
                    robot.deactivate()
        if l == len(robots):
            stop = time.time()
            t = stop - start
            print("SUCCESS EXPLORATION")
            print("\nNumero di Robot:", len(robots))
            print("Tempo impiegato:", math.trunc(t / 60), " minuti",
                round(t - math.trunc(t / 60) * 60), " secondi")
            l = 0
        # Draw
        window.blit(surface, (0, 0))
        for robot in robots:
            window.blit(robot.image, robot.rect)
        pg.display.update()
```

Capitolo 3

Risultati

Prima di procedere con i risultati si rendono note le specifiche della macchina su cui sono stati effettuati i test:

Sistema Operativo: Windows 10 Home 20H2
Processore: AMD A8-6410 APU with AMD Radeon R5 Graphics 2.00 GHz
Memoria: RAM 8GB DDR3 800MHz
Scheda Grafica: AMD Radeon R5 Graphics 1024 MB

Inizialmente vengono mostrate delle fasi di esplorazione, usando una mappa di 800x600 pixel, con 3 robot e con 2 pareti di ostacoli.

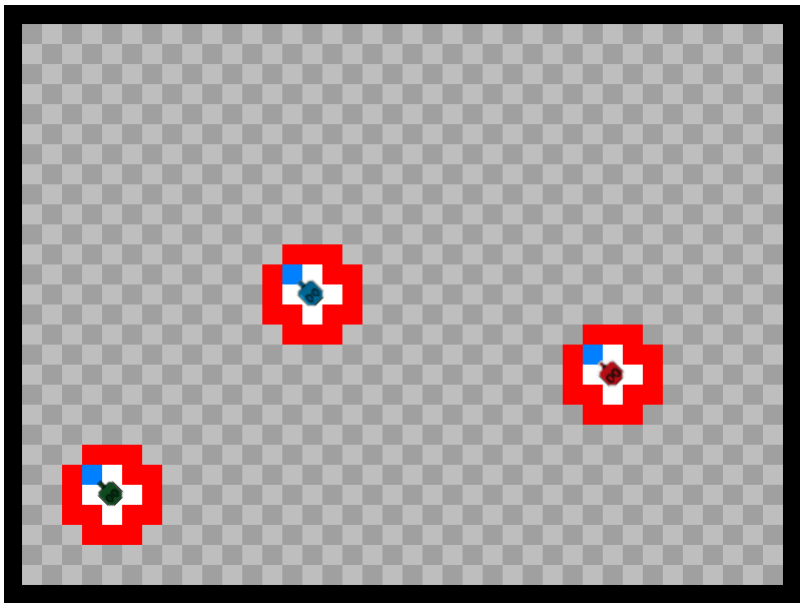

Figura 3.1: Inizializzazione

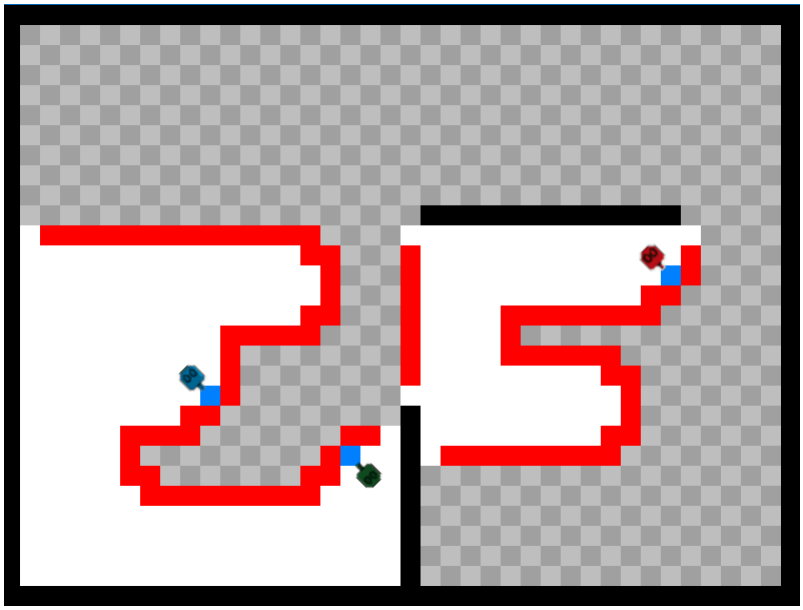

Figura 3.2: Fase 2

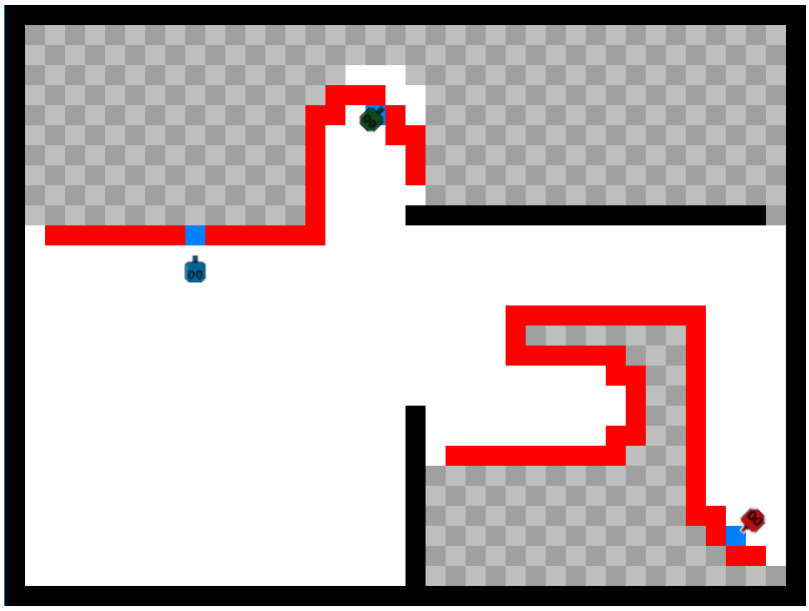

Figura 3.3: Fase 3

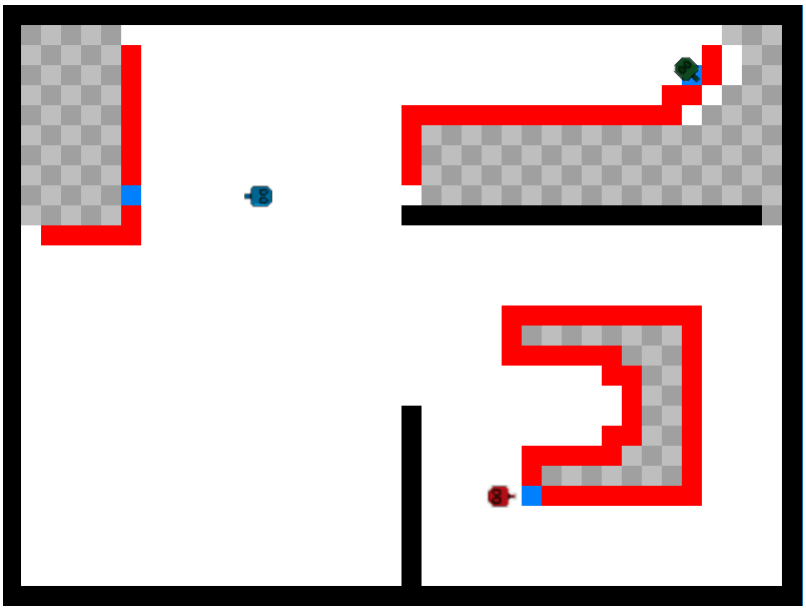

Figura 3.4: Fase 4

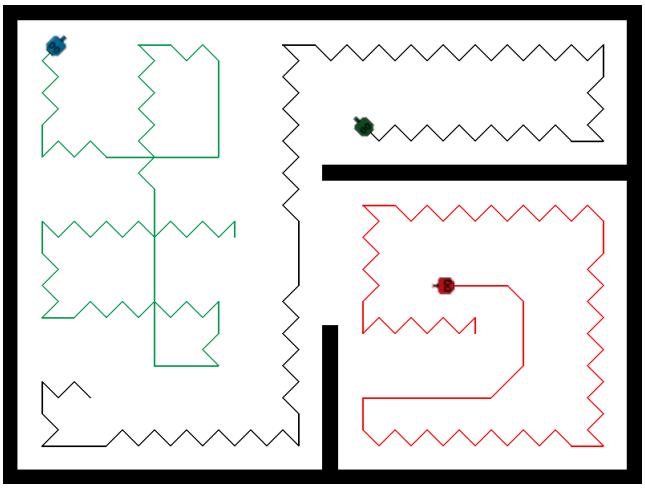

Figura 3.5: Finale

Per compiere quest'esplorazione i robot hanno impiegato un tempo di 8 minuti e 26 secondi. Il test è stato ripetuto lasciando la mappa invariata e aumentando e diminuendo il numero di robot. I risultati sono mostrati nel grafico di figura 3.6. Come si può notare diminuendo il numero di robot l'esplorazione è più lenta mentre aumentandone il numero l'esplorazione risulta più veloce, anche se subisce molte interruzioni per il calcolo delle frontiere dei vari robot. Per questo motivo non risulta tanta differenza tra le esplorazioni effettuate con 5, 6 e 7 robot. Di quest'ultima esplorazione si riporta anche il risultato finale.

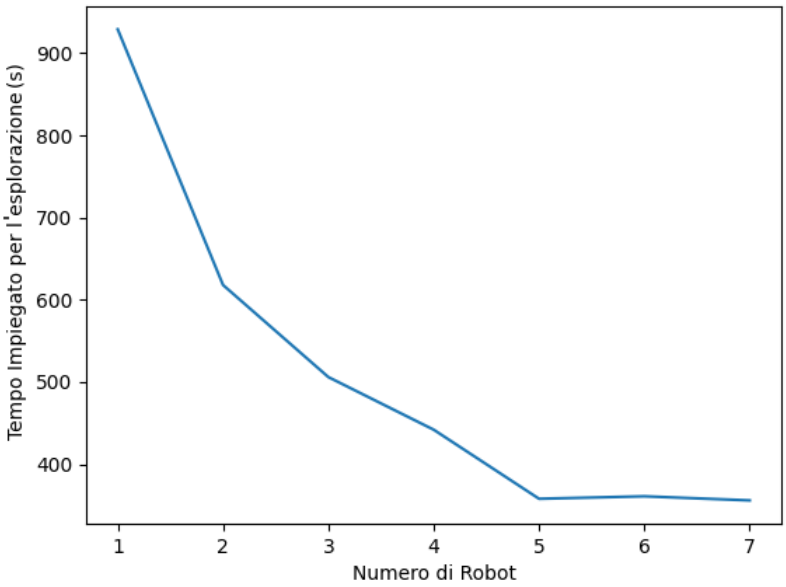

Figura 3.6: Confronto tempo impiegato e numero di robot

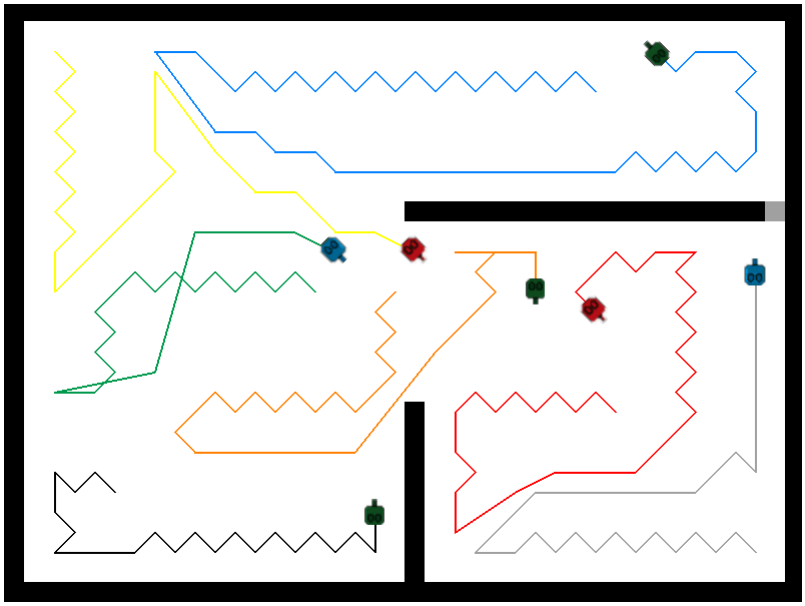

Figura 3.7: Esplorazione della mappa 800x600 con 7 robot

Si riportano poi i risultati di altre esplorazioni, effettuate cambiando il numero di robot, la loro posizione iniziale, la grandezza della mappa e modificando gli ostacoli in numero e in posizione.

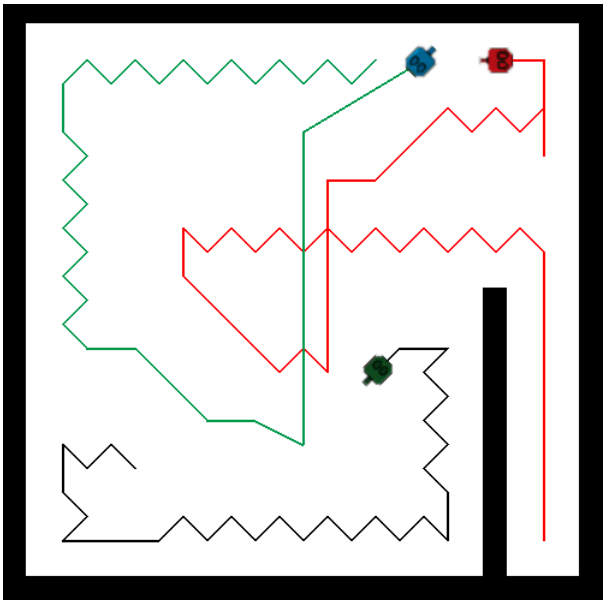

Figura 3.8: Esplorazione di una mappa 500x500 con 3 robot

Dimensioni mappa: 500x500; **Numero di robot:** 3 **Tempo Impiegato:** 2 minuti 6 secondi

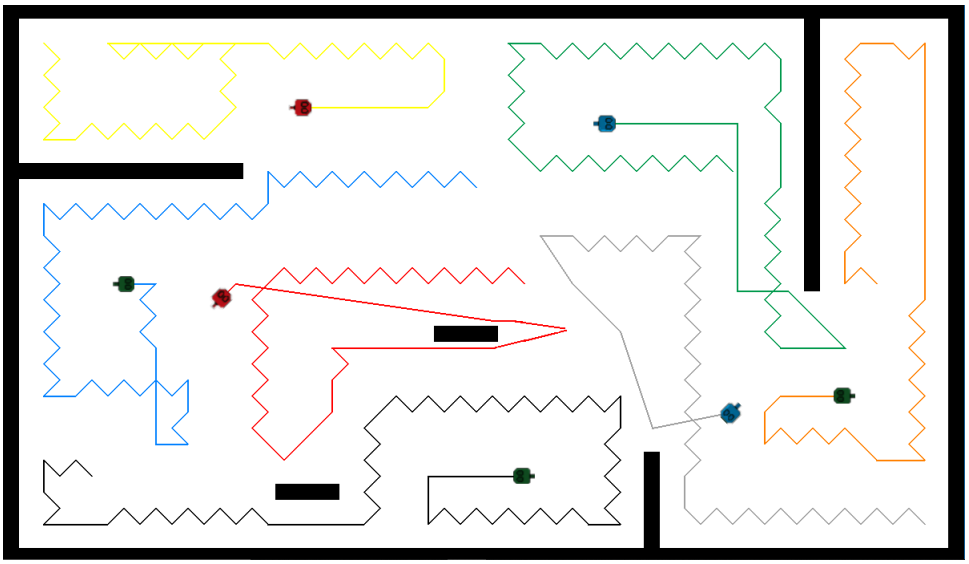

Figura 3.9: Esplorazione di una mappa 1200x700 con 7 robot

Dimensioni mappa: 1200x700; **Numero di robot:** 7 **Tempo Impiegato:** 18 minuti 55 secondi

Capitolo 4

Considerazioni Finali

Come visto nel capitolo precedente non è detto che aumentando il numero di robot si abbia un'esplorazione più veloce. Questo infatti dipende dalla grandezza della mappa: un numero elevato di robot porta a un esplorazione più veloce di una mappa grande (ad esempio 7 robot sula mappa 1200x700), mentre per una mappa più piccola bastano meno robot (3 robot sulla mappa 500x500). Questo perché, nonostante ogni robot possa calcolare autonomamente la propria frontiera e il proprio target, nell'attuale implementazione il calcolo avviene in modo centralizzato, rendendo tutto il processo più lento. Eventuali sviluppi futuri dell'elaborato potrebbero interessare l'aumento della complessità della dinamica dei robot e aggiungere un'effettiva parallelizzazione.

Bibliografia

[1] Kim D. Listmann A. Dominik Haumann and Volker Willert. Discoverage: A new paradigm for multi-robot exploration. *IEEE International Conference on Robotics and Automation*, 2010.

[2] Volker Willert Kim Listmann Dominik Haumann, Andreas Breitenmoser and Roland Siegwart. Discoverage for non-convex environments with arbitrary obstacles. *IEEE International Conference on Robotics and Automation*, 2011.

[3] Peter Shinners. Pygame docs.

[4] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduzione agli algoritmi e strutture dati-Terza Edizione*. McGraw-Hill Education, 2010.