UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Triennale in
Ingegneria delle Tecnologie per l'Impresa Digitale

Relazione di Sistemi Intelligenti

SORBITERMICA PATH-FINDER

Giuliano Ronchi
Davide Sorbi

Anno Accademico 2023/2024

Problema in astratto

Nel contesto attuale, ottimizzare le risorse e migliorare l'efficienza operativa sono due punti cruciali per il successo e la competitività di ogni azienda. Tuttavia, molte realtà affrontano sfide quotidiane che vanno ad ostacolare tali obiettivi; tra queste è presente anche l'azienda "Sorbitermica", ditta storica di Brescia specializzata nella gestione e conduzione degli impianti di riscaldamento e teleriscaldamento che noi abbiamo preso in considerazione per la ricerca e lo sviluppo di questo progetto.
L'azienda nonostante disponga di dipendenti competenti e di un magazzino ben fornito e strutturato in modo da agevolare il proprio staff, si scontra regolarmente con il problema del dispendio di tempo e risorse da parte del personale nel reperire i pezzi necessari per rifornire i propri furgoni aziendali. Inoltre, questo processo che è ovviamente di vitale importanza per la continuità delle attività quotidiane all' interno della ditta presa in esame, spesso diventa complesso in quanto i dipendenti sono costretti a scegliere tra queste due soluzioni:
o memorizzare la posizione di ogni componente o attrezzo idraulico, o cercare quest'ultimi tra i vari scaffali.
La prima soluzione comporta il problema di dover concentrare una considerevole quantità di risorse cognitive per ricordare la posizione di tutto il materiale all'interno del magazzino, questo si traduce in minore produttività, causata dallo sforzo mentale, e in rischio di errori con conseguenza perdita di tempo.
La seconda soluzione invece si traduce in una ricerca del materiale abbastanza lunga che non solo ritarda le operazioni di manutenzione e di assistenza, ma genera anche una serie di inefficienze produttive, in quanto ogni minuto speso in questa attività di ricerca si traduce in una perdita di risorse che potrebbero essere impiegate in modo più proficuo altrove.
Di fronte a questi problemi elencati emerge chiaramente la necessità di un approccio innovativo ed efficiente che possa ottimizzare il processo di rifornimento dei vari furgoni, alleviando i dipendenti da inutili sforzi mentali consentendo ad essi di concentrare le proprie energie sulle attività che generano valore aggiunto all'azienda.
In questa relazione trattiamo la ricerca e lo sviluppo della soluzione più ottimale per arrivare ad ottenere quanto appena descritto, in modo da porre le basi per un miglioramento significativo delle performance a livello di produttività dell'azienda "Sorbitermica".

1 Approccio al problema

Per risolvere il problema del dispendio di tempo e risorse nel reperimento del materiale nel magazzino per il rifornimento dei vari furgoni dell'azienda "Sorbitermica" descritto sopra, abbiamo adottato un approccio basato sull'utilizzo di algoritmi di pathfinding.
Il concetto di pathfinding si basa sull'idea di individuare il tragitto ottimale da un punto di partenza ad un punto di destinazione, attraversando un ambiente complesso e potenzialmente ostacolato.

1.1 Pathfinding

Il pathfinding è un problema fondamentale dell'informatica e come anticipato sopra consiste nel trovare il percorso più breve o più efficiente tra due punti.
Un algoritmo di pathfinding di solito inizia rappresentando il problema come un grafo. Quest'ultimo è un insieme di nodi collegati da spigoli; I nodi rappresentano le posizioni nello spazio di base, mentre gli spigoli rappresentano i possibili percorsi tra di essi. Una volta rappresentato il problema gli algoritmi di pathfinding utilizzano varie tecniche per trovare il percorso tra due punti.

1.2 Implementazione nel nostro scenario

Nel nostro scenario specifico, il pathfinding viene applicato al contesto operativo dell'azienda "Sorbitermica", dove i dipendenti devono navigare tra i vari scaffali per rifornire il furgone con i pezzi necessari per le attività quotidiane. Attraverso il suo utilizzo l'obiettivo è quello di determinare i percorsi più efficienti e rapidi all'interno del magazzino, consentendo ai dipendenti di completare la ricerca dei pezzi in modo ottimale. Tramite il pathfinding si va ad eliminare il problema relativo alla necessità per i dipendenti di dover ricordare la posizione specifica di ciascun articolo in modo da liberarli da oneri mentali superflui, consentendo loro di concentrarsi esclusivamente sull'esecuzione delle attività di valore aggiunto, e si riduce drasticamente il tempo impiegato nel reperimento dei pezzi, tutto ciò inoltre semplifica notevolmente le operazioni di rifornimento del furgone.
Inoltre, sfruttando la potenza degli algoritmi di pathfinding possiamo ottimizzare i percorsi che i vari dipendenti potrebbe intraprendere anche adattandoli dinamicamente in base alle condizioni in tempo reale del magazzino, come la disponibilità dei pezzi e la presenza di ostacoli o aree congestionate.

Per riuscire a fare ciò la nostra idea è quella di realizzare un software tramite il linguaggio di programmazione python che ci mostri una finestra con il percorso trovato tramite gli algoritmi di pathfinding; per fare ciò vorremmo utilizzare una libraria python chiamata pygame, la quale dopo l'avvio del software va a creare una finestra in cui può essere generata una griglia in cui vengono colorati in un determinato colore tutti i quadratini della griglia che corrispondo al punto di partenza, al punto di arrivo, ai punti intermedi e alle scaffalature; una volta fissati tutti questi punti, tramite comando, può essere generato il percorso tra i punti che assume un determinato colore in modo che si possa vedere in modo chiaro il percorso più veloce per reperire tutti i materiali per il rifornimento del furgone.

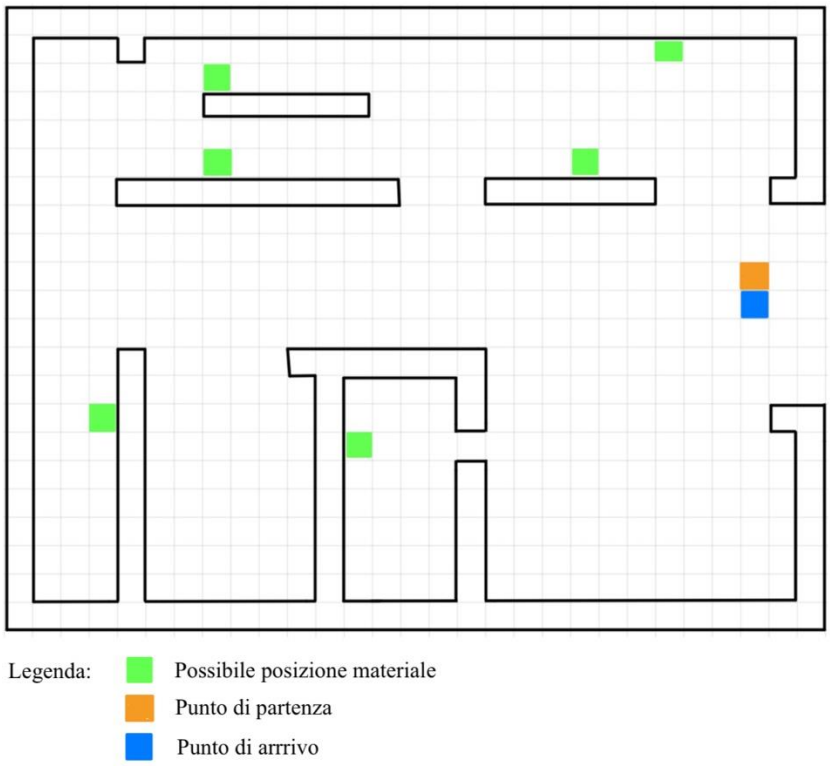
2 Modellazione del problema

In questo capitolo parliamo di come abbiamo sviluppato il pathfinding all'interno del nostro progetto; I passi per fare ciò sono stati i seguenti:

- Definizione dell'ambiente
- Scelta dell'algoritmo più opportuno
- Implementazione algoritmo con le nostre esigenze

2.1 Definizione dell'ambiente

Il primo passo che abbiamo affrontato è stato quello di definire in modo efficace l'ambiente complesso del magazzino in cui l'algoritmo deve operare.
In base alla spiegazione del pathfinding fatta nel capitolo precedente abbiamo rappresentato l'area del magazzino come una griglia di nodi in 2d includendo la presenza delle scaffalature che verranno viste come ostacoli da evitare, il terreno percorribile dai dipendenti e la distribuzione del materiale necessario per il rifornimento dei furgoni.
Di seguito una prima bozza di come abbiamo definito il nostro ambiente:



2.2 Scelta dell'algoritmo più opportuno

Il secondo passo è stato quello di scegliere l'algoritmo più opportuno da utilizzare per trovare i vari percorsi all'interno del magazzino; al giorno d'oggi vi sono diversi algoritmi di pathfinding ognuno con le proprie caratteristiche:

- *Ricerca in ampiezza (BFS):*
  Algoritmo semplice utilizzato in scenari di ricerca del percorso più semplici. Esplora tutti i nodi vicini alla profondità attuale prima di passare ai nodi alla prossima profondità, garantendo una ricerca completa dell'area.
- *Ricerca in profondità (DFS):*
  Algoritmo semplice che esplora il più lontano possibile lungo ogni ramo prima di tornare indietro, rendendolo utile in scenari in cui è necessario un completo attraversamento dell'ambiente. È meno efficiente per trovare il percorso più breve, ma utile in compiti di ricerca più complessi.
- *Algoritmo di Dijkstra:*
  Noto per la sua semplicità ed efficacia, in particolare in ambienti basati su grafici. Viene ampiamente utilizzato in scenari in cui tutti i bordi hanno pesi non negativi, garantendo che venga trovato il percorso più breve.
- *Ricerca Greedy Best-First:*
  Questo algoritmo dà priorità al movimento verso l'obiettivo nel modo più diretto possibile. È veloce ma a volte può portare a percorsi subottimali, poiché non considera sempre l'intero percorso in anticipo.
- *Algoritmo A*:*
  Rinomato per bilanciare efficienza ed accuratezza. Combina le caratteristiche dell'algoritmo di Dijkstra e della Ricerca del Migliore Primo, fornendo un metodo veloce ed efficace per trovare il percorso più breve

L'idea chiave alla base di tutti questi algoritmi è che teniamo traccia di un anello in espansione chiamato frontiera, ma ovviamente ogni algoritmo però ha le proprie caratteristiche di funzionamento. La nostra scelta è ricaduta su A* in quanto come possiamo vedere dalla descrizione è l'algoritmo che ci permette di avere efficienza ed accuratezza, caratteristiche di fondamentale importanza per lo sviluppo del nostro software in quanto la prima ci garantisce la generazione del percorso nel minor tempo possibile, mentre la seconda ci garantisce la generazione del percorso più veloce in base all'ambiente circostante.
Per ottenere le caratteristiche che abbiamo elencato, A* fonde due approcci: Dijkstra e Greedy Best First Search.
Il primo individua il percorso più breve tenendo conto dei costi differenti degli archi. Ad esempio, se consideriamo le scaffalature come ostacoli da evitare, esse avranno un costo più elevato. Tuttavia, questo metodo espande la frontiera in tutte le direzioni possibili, risultando quindi accurato ma inefficiente.
Il secondo metodo, Greedy Best First Search, si focalizza su direzioni promettenti. Espande la frontiera verso l'obiettivo più rapidamente rispetto ad altre direzioni, sfruttando una funzione euristica che valuta la vicinanza all'obiettivo. Questo approccio è notevolmente più veloce rispetto a Dijkstra, ma non tiene conto dei costi, potrebbe non garantire il percorso più breve, concentrandosi invece sull'efficienza.
Così combinando l'accuratezza di Dijkstra e l'efficienza di Greedy Best First Search, A* offre un equilibrio tra precisione e velocità nell'individuare il percorso ottimale.

2.3 Implementazione algoritmo con le nostre esigenze

Una volta definito l'algoritmo da utilizzare abbiamo strutturato il codice in base alle nostre esigenze definendo diversi moduli di cui di seguito spieghiamo in modo dettagliato:

- Search_algorithm
- Path_finding
- A_Star
- Heuristisc
- Gui

Search_algorithm

```
class Node:
    def __init__(self, state, parent=None, action=None, g=0, h=0):
        """
        Inizializza un nuovo nodo.

        Args:
            state: Lo stato rappresentato dal nodo.
            parent: Il nodo genitore del nodo corrente (default: None).
            action: L'azione che ha portato a questo nodo (default: None).
            g: Il costo effettivo dal nodo di partenza al nodo corrente (default: 0).
            h: L'euristica dal nodo corrente al nodo di destinazione (default: 0).
        """
        self.state = state
        self.parent = parent
        self.action = action
        self.g = g
        self.h = h

    def __lt__(self, other):
        """
        Confronta due nodi in base alla loro somma di g e h.

        Args:
            other: Altro nodo con cui confrontare.

        Returns:
            True se il nodo corrente ha una somma di g e h minore rispetto all'altro nodo, altrimenti False.
        """
        return (self.g + self.h) < (other.g + other.h)

class SearchAlgorithm:
    def __init__(self, view=False):
        """
        Inizializza un nuovo algoritmo di ricerca.
```

```
    Args:
        view: Indica se visualizzare i nodi espansi durante la ricerca (default: False).
    """
    self.expanded = 0
    self.expanded_states = set()
    self.view = view

def update_expanded(self, state):
    """
    Aggiorna il conteggio dei nodi espansi.

    Args:
        state: Lo stato del nodo espanso.
    """
    if self.view:
        self.expanded_states.add(state)
    self.expanded += 1

def reset_expanded(self):
    """Resetta il conteggio dei nodi espansi."""
    if self.view:
        self.expanded_states = set()
    self.expanded = 0

def extract_solution(self, node) -> list:
    """
    Estrae la soluzione dal nodo finale risalendo ai nodi genitori.

    Args:
        node: Il nodo finale della soluzione.

    Returns:
        Una lista di azioni che rappresentano la soluzione.
    """
    sol = []
    while node.parent is not None:
        sol.insert(0, node.action)
        node = node.parent
    return sol
```

Questo primo modulo fornisce una struttura fondamentale per l'implementazione e l'esecuzione degli algoritmi di ricerca su grafi, offrendo funzionalità essenziali per la gestione dei nodi e il calcolo delle soluzioni ottimali. Nel dettaglio presenta due classi: Node e SearchAlgoritm.
La prima rappresenta un nodo nel grafo di ricerca, il quale contiene informazioni cruciali come lo stato attuale, il nodo genitore (ovvero il nodo da cui è stato generato), l'azione che ha portato a questo nodo, il costo effettivo dal nodo di partenza a questo nodo (`g`), e l'euristica dal nodo corrente al nodo di destinazione (`h`).

Il metodo __lt__ è implementato per consentire il confronto tra due nodi in base alla somma dei loro costi effettivi e delle loro euristiche. Questo è essenziale per l'ordinamento dei nodi all'interno della coda a priorità durante l'esecuzione dell'algoritmo A*.
La seconda classe fornisce una struttura generica per gli algoritmi di ricerca su grafi.
È dotata di funzionalità per il conteggio e l'aggiornamento dei nodi espansi durante la ricerca, nonché per l'estrazione della soluzione dal nodo finale.
Presenta tre metodi:

- I metodi update_expanded e reset_expanded sono responsabili della gestione del conteggio dei nodi espansi e degli stati visitati.
- Il metodo extract_solution è utilizzato per estrarre la soluzione dal nodo finale, risalendo ai nodi genitori e ottenendo la sequenza di azioni che rappresentano il percorso dalla radice alla soluzione.

Path_finding

```
class SearchProblem(object):
    def __init__(self, init, goal, actions :set, world, cost :dict,barrier):
        """
        Inizializza un nuovo problema di ricerca.

        Args:
            init: Lo stato iniziale del problema.
            goal: Lo stato obiettivo del problema.
            actions: Insieme di azioni disponibili.
            world: Oggetto che rappresenta il mondo in cui avviene la ricerca.
            cost: Dizionario che assegna un costo a ciascuna azione.
            barrier: Barriera rappresentativa del mondo.
        """
        self.init = init
        self.goal = goal
        self.actions = actions
        self.world = world
        self.cost = cost
        self.barrier = barrier

class PathFinding(SearchProblem):
    def __init__(self, world, init, goal,barrier):
        actions = ['N','S','W','E']
        cost = [(a,1) for a in actions]
        super().__init__(init, goal, actions, world, cost,barrier)


    def getSuccessors(self, state):
        """
        Ottiene gli stati successori dello stato attuale in base alle azioni disponibili.
```

```
    Args:
        state: Lo stato corrente.

    Returns:
        Un insieme di coppie (azione, stato_successivo) rappresentanti gli stati successori validi.
    """
    successors = set()
    for a in self.actions:
        if a == 'N':
            next_state = (state[0], state[1]+1)
        elif a == 'S':
             next_state = (state[0], state[1]-1)
        elif a == 'W':
            next_state = (state[0]-1, state[1])
        elif a == 'E':
            next_state = (state[0]+1, state[1])
        if next_state not in self.world.walls and self.isInTheLimits(next_state):

            successors.add((a, next_state))
    return successors

def isInTheLimits(self, state :tuple):
    """
    Verifica se lo stato si trova nei limiti del mondo.

    Args:
        state: Lo stato da verificare.

    Returns:
        True se lo stato è all'interno dei limiti del mondo, altrimenti False.
    """
    return state[0] >=0 and state[0] <= self.world.x_limit and state[1] >= 0 and state[1] <= self.world.y_limit


def isGoal(self,state):
    """
    Verifica se lo stato corrente è lo stato obiettivo.

    Args:
        state: Lo stato corrente.

    Returns:
        True se lo stato corrente coincide con lo stato obiettivo, altrimenti False.
    """
    return state[0] == self.goal[0] and state[1] == self.goal[1]

class World:

    def __init__(self, x_limit, y_limit, walls):
```

```
        """
    Inizializza un nuovo mondo.

    Args:
        x_limit: Limite dell'asse x.
        y_limit: Limite dell'asse y.
        walls: Lista di muri nel mondo.
    """
        self.x_limit = x_limit
        self.y_limit = y_limit
        self.walls = walls
```

Questo modulo fornisce una struttura per definire un ambiente di ricerca (il mondo) e un problema di ricerca specifico (la navigazione attraverso il labirinto), con funzioni per generare gli stati successori e verificare se uno stato è obiettivo.
Nel dettaglio presenta tre classi: SearchProblem, PathFinding e World.
La prima classe rappresenta un problema di ricerca generico.
I parametri iniziali includono lo stato iniziale (init), lo stato obiettivo (goal), un insieme di azioni disponibili (actions), un oggetto rappresentativo del mondo (world), un dizionario di costi associati ad ogni azione (cost), e una barriera rappresentativa del mondo (barrier)
La classe PathFinding specifica un problema di ricerca per il percorso di ricerca in un labirinto. All'inizializzazione, imposta le azioni disponibili come i movimenti cardinali (nord, sud, ovest, est) e assegna loro un costo unitario.
Sovrascrive il metodo getSuccessors, che restituisce gli stati successori validi dello stato corrente in base alle azioni disponibili. Controlla se lo stato successivo è all'interno dei limiti del mondo e non è un muro.
Implementa il metodo isInTheLimits per verificare se uno stato è nei limiti del mondo e il metodo isGoal per verificare se uno stato è lo stato obiettivo.

Infine, la classe World rappresenta il mondo in cui avviene la ricerca.
È inizializzata con i limiti dell'asse x e y e una lista di muri nel mondo (walls).

A_Star

```
from search_algorithm import SearchAlgorithm
from queue import PriorityQueue
from search_algorithm import Node

class AstarNode(Node):
    def __init__(self, state, parent=None, action=None, g=0, h=0):
        """
        Inizializza un nuovo nodo per l'algoritmo A*.

        Args:
            state: Lo stato rappresentato dal nodo.
```

```
            parent: Il nodo genitore del nodo corrente (default: None).
            action: L'azione che ha portato a questo nodo (default: None).
            g: Il costo effettivo dal nodo di partenza al nodo corrente (de-
fault: 0).
            h: L'euristica dal nodo corrente al nodo di destinazione (default:
0).
        """
        self.h = h
        super().__init__(state, parent, action, g)

    def __lt__(self, other):
        """
        Confronta due nodi A* in base alla loro somma di g e h.

        Args:
            other: Altro nodo con cui confrontare.

        Returns:
            True se il nodo corrente ha una somma di g e h minore rispetto
all'altro nodo, altrimenti False.
        """
        return (self.g + self.h) < (other.g + other.h)

class AStar(SearchAlgorithm):
    def __init__(self, heuristic=lambda: 0, view=True, w=1):
        """
        Inizializza un nuovo algoritmo di ricerca A*.

        Args:
            heuristic: La funzione euristica da utilizzare (default: funzione
costante 0).
            view: Indica se visualizzare i nodi espansi durante la ricerca
(default: True).
            w: Fattore di pesatura per l'euristica (default: 1).
        """
        self.heuristic = heuristic
        self.w = w
        super().__init__(view)

    def solve(self, problem) -> list:
        """
        Risolve il problema utilizzando l'algoritmo A*.

        Args:
            problem: Il problema di ricerca.

        Returns:
```

```
            Una lista di azioni che rappresentano la soluzione.
        """
        reached = set()
        frontier = PriorityQueue()

        reached.add(problem.init)
        h = self.heuristic(problem.init, problem.goal, problem.barrier)
        frontier.put((h, AstarNode(problem.init, None, None, 0, h)))

        while not frontier.empty():
            _, node = frontier.get()
            if problem.isGoal(node.state):
                return self.extract_solution(node)

            for action, state in problem.getSuccessors(node.state):
                if state not in reached:
                    self.update_expanded(state)
                    reached.add(state)
                    g = node.g + 1
                    h = self.heuristic(state, problem.goal, problem.barrier)
                    f = g + h * self.w
                    frontier.put((f, AstarNode(state, node, action, g, h)))

        return None
```

Questo modulo implementa l'algoritmo A* per risolvere problemi di ricerca, utilizzando una combinazione di costo effettivo e euristica per guidare l'esplorazione verso la soluzione ottimale.

Nel dettaglio abbiamo due classi: AstarNode e Astar
La prima estende la classe base Node, aggiungendo un attributo h per l'euristica. Sovrascrive il metodo __lt__ per confrontare i nodi A* in base alla somma di g e h.
La classe AStar eredita da SearchAlgorithm e implementa il metodo solve, che utilizza l'algoritmo A* per risolvere un dato problema di ricerca.
A* implementato in solve funziona nel seguente modo:
Crea un insieme reached per tenere traccia degli stati raggiunti e una coda a priorità frontier per gestire i nodi da esaminare.
Inizializza il nodo iniziale con lo stato iniziale del problema e l'euristica calcolata tramite la funzione euristica specificata, questo nodo viene inserito nella coda a priorità.
Finché la coda a priorità non è vuota, il ciclo continua:
estrae il nodo con il valore f (costo effettivo più euristica) minore dalla coda.
Se lo stato del nodo corrente è lo stato obiettivo, restituisce la soluzione chiamando il metodo extract_solution.
Altrimenti, per ogni azione disponibile nello stato corrente, calcola il costo effettivo g, l'euristica h, e il valore f (g + h * w).Infine, aggiunge il nuovo nodo alla coda a priorità e aggiorna l'insieme degli stati raggiunti.

Se la coda a priorità si svuota senza trovare una soluzione, restituisce None.

Euristics

```
import math

def manhattan_with_barriers(punto1, punto2, barriere) -> int:
    """
    Calcola la distanza di Manhattan tra due punti con l'aggiunta del costo
delle barriere lungo il percorso.

    Args:
        punto1: Le coordinate del primo punto (x, y).
        punto2: Le coordinate del secondo punto (x, y).
        barriere: Lista di coordinate delle barriere nel formato [(x1, y1),
(x2, y2), ...].

    Returns:
        La distanza di Manhattan tra i due punti con il costo delle barriere
lungo il percorso.
    """
    x1, y1 = punto1
    x2, y2 = punto2
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    # Calcolo del costo delle barriere lungo il percorso
    barrier_cost = sum(0.1 for barriera in barriere if (barriera[0] > min(x1,
x2) and barriera[0] < max(x1, x2) and
                                                        barriera[1] > min(y1,
y2) and barriera[1] < max(y1, y2)))
    return dx + dy + barrier_cost
```

Questo modulo comprende solo la funzione manhattan_with_barriers che calcola la distanza di Manhattan tra due punti, considerando anche il costo delle barriere lungo il percorso. Prende in input le coordinate di due punti e una lista di coordinate delle barriere, restituendo la distanza di Manhattan tra i due punti con l'aggiunta del costo delle barriere lungo il percorso.
Il costo delle barriere lungo il percorso è calcolato come la somma delle distanze tra i punti in cui il percorso interseca le barriere, moltiplicato per un fattore di 0.1.

Gui

Nel modulo gui (graphical user interface) abbiamo implementato le funzioni che permettono all'algoritmo A* di calcolare il percorso prendendo in considerazione dei punti intermedi che corrispondono alle posizioni degli attrezzi da lavora che bisogna reperire.
Le funzioni implementate sono le seguenti : get_punti_intermi, trova_punto_vicino e trova_percorso.

```
def get_punti_intermedi(punti_intermedi):
    prefixed_points = {
        "Chiave a tubo": ((16, 20)),
        "Pompa ad aria compressa": ((60, 24)),
        "Tagliatubi": ((37, 48)),
        "Idropulitrice": ((80, 20)),
        "Chiave a catena": ((35, 66)),
        "Fresa a mano": ((35,20)),
        "Torcia a gas": ((3,49)),
        "Guarnizioni assortite": ((75,3)),
        "Pinze": ((80,66)),
        "Rubinetto a sfera": ((15,3))
    }

    grid_maps = ["mappa_1.json", "mappa_2.json", "mappa_3.json"]

    root = tk.Tk()
    root.withdraw()

    dialog = tk.Toplevel(root)
    dialog.title("Selezione punti e mappa")
    dialog.geometry("425x400")

    checkboxes = {}
    selected_points = []
    selected_map = None

    def toggle_point(name, var):
        if var.get() == 1:
            if prefixed_points[name] not in selected_points:
                selected_points.append(prefixed_points[name])
        elif var.get() == 0:
            if prefixed_points[name] in selected_points:
                selected_points.remove(prefixed_points[name])

    def add_selected_points():
        punti_intermedi.update(selected_points)
        nonlocal selected_map
```

```
        selected_maps = [map_name for map_name, var in checkboxes_map.items()
if var.get() == 1]
        try:
            if selected_maps:
                selected_map = selected_maps[0]
            else:
                raise ValueError("nessuna mappa selezionata")
        except ValueError as va:
            print(va)

        print("Nome della mappa selezionata:", selected_map)
        print("Punti intermedi aggiunti:", punti_intermedi)
        root.destroy()

    def select_all_points():
        for name in prefixed_points.keys():
            var = checkboxes[name]
            var.set(1)
            if prefixed_points[name] not in selected_points:
                selected_points.append(prefixed_points[name])

    def on_close():
        punti_intermedi.clear()
        root.destroy()

    dialog.protocol("WM_DELETE_WINDOW", on_close)


    main_frame = tk.Frame(dialog)
    main_frame.pack(pady=5, padx=5, fill=tk.Y, side=tk.LEFT)


    for i, name in enumerate(prefixed_points.keys()):
        var = tk.IntVar()
        checkbox = tk.Checkbutton(main_frame, text=name, variable=var, onva-
lue=1, offvalue=0,
                                  command=lambda n=name, v=var: tog-
gle_point(n, v))
        checkbox.grid(row=i, column=0, sticky=tk.W)
        checkboxes[name] = var

    select_all_button = tk.Button(main_frame, text="Seleziona Tutti", com-
mand=select_all_points)
    select_all_button.grid(row=len(prefixed_points), column=1, pady=5)

    done_button = tk.Button(main_frame, text="Aggiungi", command=add_selec-
ted_points)
```

```
    done_button.grid(row=len(prefixed_points) + 1, column=1, pady=5)


    map_frame = tk.Frame(dialog)
    map_frame.pack(pady=10, padx=10, fill=tk.Y, side=tk.RIGHT)


    checkboxes_map = {}
    for i, map_name in enumerate(grid_maps):
        var_map = tk.IntVar()
        checkbox_map = tk.Checkbutton(map_frame, text=map_name, va-
riable=var_map, onvalue=1, offvalue=0)
        checkbox_map.grid(row=i, column=0, sticky=tk.W, padx=5, pady=2)
        checkboxes_map[map_name] = var_map

    dialog.wait_window()

    return punti_intermedi, selected_map
```

Questa funzione crea una finestra di dialogo per selezionare i punti intermedi e la mappa per la navigazione. Utilizza una serie di checkbox per selezionare i punti intermedi e le mappe disponibili. Quando viene premuto il pulsante "Aggiungi", aggiunge i punti intermedi selezionati alla lista punti_intermedi restituendola insieme al nome della mappa selezionata.

```
def trova_punto_vicino(start_point,punti_intermedi,walls):
    min_euristica = float('inf')
    prossimo_punto = None

    for punto in punti_intermedi:
        euristica = heuristics.manhattan_with_barriers(start_point,
punto,walls)
        if euristica < min_euristica:
            min_euristica = euristica
            prossimo_punto = punto

    return prossimo_punto
```

Questa funzione cerca il punto più vicino a un punto di partenza tra una lista di punti intermedi. Per fare ciò itera attraverso tutti i punti intermedi, calcolando l'euristica per ciascuno utilizzando la funzione manhattan_with_barriers, il punto più vicino è quello che ha l'euristica minore; infine restituisce il punto più vicino trovato.

```
def trova_percorso(rows,wall,start,end,punti_intermedi,search_algorithm):

    world = PathFinding.World(rows-1,rows-1,wall)
    percorso = []
    # Inizializzazione del punto corrente come punto di partenza
    initial_point = (start.row, start.col)
    final_point = (end.row,end.col)

    if punti_intermedi.__len__() != 0:
        # Calcolo dell'euristica tra il punto di partenza e il primo punto in-
termedio
        prossimo_punto = trova_punto_vicino(initial_point,punti_inter-
medi,wall)
        p = PathFinding(world,(initial_point),(prossimo_punto),wall)
        current_point= prossimo_punto
        plan = search_algorithm.solve(p)

        if plan:
            percorso.extend(plan)
            punti_intermedi.remove(prossimo_punto)
        # Calcolo dell'euristica tra il punto più vicino lo start e un punto
intermedio

            while punti_intermedi:
                # Trova il punto intermedio più vicino al punto corrente
                prossimo_punto = trova_punto_vicino(current_point, punti_in-
termedi,wall)
                # Calcola il percorso tra il punto corrente e il punto inter-
medio
                p = PathFinding(world, current_point, prossimo_punto,wall)
                plan = search_algorithm.solve(p)

                if plan:
                    percorso.extend(plan)
                    # Aggiorna il punto corrente al punto appena trovato
                    current_point = prossimo_punto
                    # Rimuovi il punto intermedio appena trovato dalla lista
dei punti intermedi
                    punti_intermedi.remove(prossimo_punto)
        #Calcolo dell'euristica tra l'ultimo punto e il punto finale
        int_p = PathFinding(world,(current_point),(final_point),wall)
        int_plan = (search_algorithm.solve(int_p))

        if int_plan:
            percorso.extend(int_plan)

    else :
```

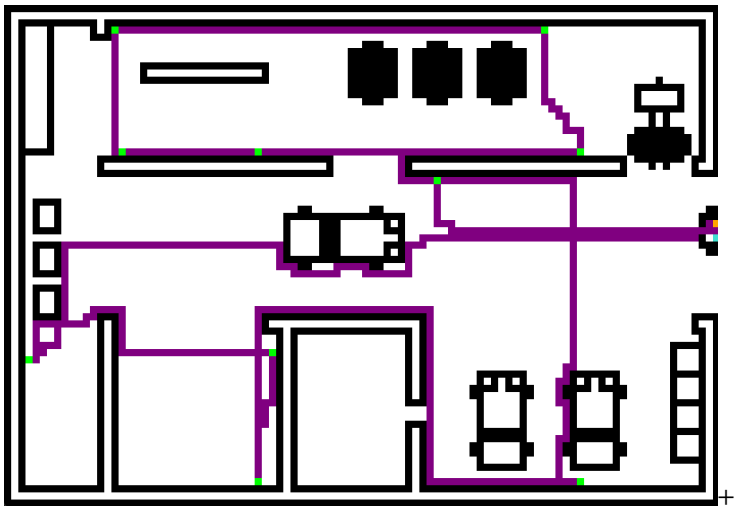
```
        final_p = PathFinding(world,(initial_point),(final_point),wall)
        final_plan = (search_algorithm.solve(final_p))

        if final_plan:

            percorso.extend(final_plan)

    return percorso
```

Questa funzione calcola un percorso tra un punto di partenza e un punto di arrivo, passando attraverso una serie di punti intermedi. Utilizza un algoritmo di ricerca specificato che nel nostro caso è A*. Se non ci sono punti intermedi, calcola direttamente il percorso dal punto di partenza al punto di arrivo. Infine, restituisce il percorso calcolato.

3 Valutazione

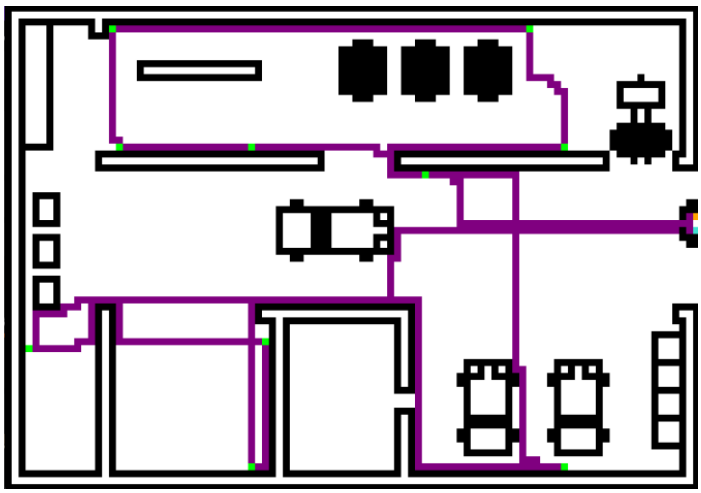
Per valutare al meglio il codice che abbiamo creato sono stati fatti tre esperimenti che valutano l'efficienza dell'algoritmo.
Gli esperimenti si basano sul cambio di certi parametri che cambiano il comportamento di A* e di conseguenza il percorso finale generato.
Il percorso quando viene generato ha un costo che si basa sulla quantità di movimenti che il codice esegue tra tutti i punti selezionati
Il parametro principale che è stato cambiato nel corso degli esperimenti è il costo per barriera, il quale incide molto sul percorso causando possibili sprechi.

Nell'esperimento numero 1 abbiamo impostato un costo per barriera a 10, il risultato è un costo finale del percorso di 582.



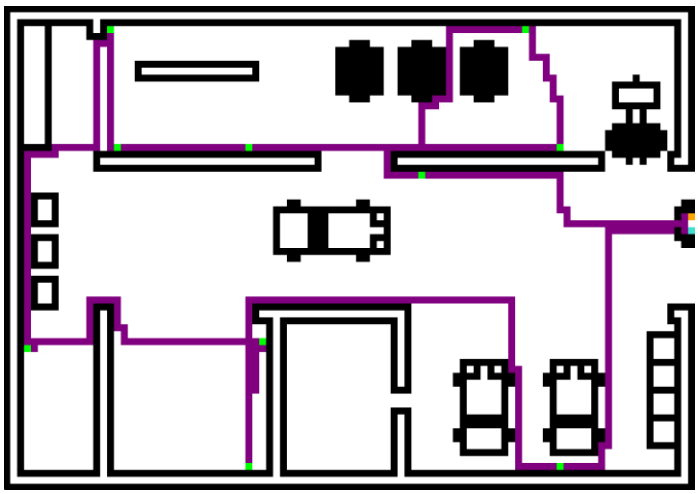
Come si puo notare il percorso in viola generato ha qualche problema, ripassa su percorsi gia visitati e ha delle perdite di efficienza.

Nell'esperimento numero 2 abbiamo impostato il parametro costo per barriera a 1, il risultato è un costo finale del percorso di 566



Come si nota ci sono stati dei miglioramenti rispetto a l'esperimento numero 1 riguardo certe svolte che sono state prese e abbiamo ottenuto un costo finale inferiore.

Nell 'esperimento numero 3 abbiamo impostato un costo per barriera a 0.1, il risultato è un costo finale del percorso di 488



Possiamo dire che l'esperimento 3 ci fornisce il valore di costo per barriere definitivo in quanto ha il costo finale minore e come possiamo vedere dall'immagine il percorso generato prende soluzioni intelligenti basate su tutto l'ambiente.