UNIVERSITÀ DEGLI STUDI DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN
INGEGNERIA INFORMATICA

PF4EA: Path Finding for an entry agent

Studenti:
Nicola Bettinzoli
Nicholas Dumas

01-02-2024

Indice

Elenco delle Figure

Elenco dei listati

Capitolo 1

Introduzione

L'argomento di questo elaborato è lo sviluppo di un programma per la risoluzione dei problemi della classe di *Path Finding for an entry agent* (PF4ea). Problemi in cui l'obbiettivo è la ricerca di un cammino a minor costo per un agente all'interno di una griglia bidimensionale, di dimensioni prefissate, da una cella iniziale detta *init* a una finale detta *goal*.

La griglia in cui si muove l'agente non è completamente attraversabile, al suo interno sono presenti celle dette ostacoli. Inoltre, sempre al suo interno, sono presenti altri agenti, ognuno di essi con un percorso predeterminato.

L'obbiettivo sarà quello di trovare un percorso, ovvero una serie si azioni a partire dalla posizione iniziale, di costo minimo che porti l'agente fino al goal e che rispetti tutti i vincoli di traversabilità.

Nei capitoli successivi andremo a trattare in maniera più approfondita il problema nelle sue varie componenti, verranno presentate, dopodiché, le principali soluzioni adottare per lo sviluppo del programma. In fine verranno presentati e discussi i risultati.

1.1 Il problema

Un istanza di problema PF4EA è definita da una tupla nella forma

$$\langle G, w, \{\pi_i\}_{0\,=\,1..n}, init, goal, max\rangle$$

dove:

- G rappresenta un grafo orientato, nel nostro caso una griglia dotata di ostacoli, in cui i vertici rappresentano i vertice, mentre gli archi le connessioni tra le celle attraversabili (le mosse da una cella all'altra).
- $w : E[G] \to \mathbb{R}$ è una funzione peso, dove il peso di un arco rappresenta il costo di una mossa.
- $\{\pi_i\}_{0\,=\,1..n}$ è un insieme dei percorsi di n (con $n \geq 0$) agenti α_i su tale griglia, che rispettano i vincoli delle mosse di attraversabilità, si presuppone che una volta terminato il percorso l'agente rimanga fermo nella posizione finale.
- $init \in V[G]$ è il vertice che rappresenta la cella iniziale dell'agente aggiuntivo α_{n+1}.
- *max* è il limite superiore per la lunghezza/per la durata di ciascun percorso entro la griglia (assumendo che tutti i percorsi inizino all'istante 0)

1.1.1 Griglia

La griglia presa in considerazione in questo elaborato è una *8-connected gridmap*, ovvero una spazio bidimensionale entro il quale si possono muovere uno o più agenti. Ogni mossa cardinale (N,S,E,W) ha come costo 1, mentre ogni mossa diagonale (NE,NW,SE,SW) comporta un costo pari a $\sqrt{2}$. Ogni cella all'interno della griglia può essere traversabile oppure non traversabile (rappresentando quindi un ostacolo agli agenti). Importate ricordare che è possibile attraversare gli spigoli degli ostacoli attraverso le mosse diagonali.

1.1.2 Grafo

La griglia vista in precedenza induce un grafo orientato pesato G dove:

- Le celle attraversabili corrispondono ai vertici di G
- Le adiacenze tra le celle attraversabili sono gli archi di G.
- il peso $w(v_1, v_2)$ di una arco (v_1, v_2) rappresenta il costo della mossa per passare dal vertice v_1 al vertice adiacente v_2 (e quindi il costo per passare dalla prima cella alla seconda nella griglia)
- Per ciascun vertice v presente in G vi è un cappio (v, v) con peso pari a 1, questo rappresenta la mossa di *wait*, ovvero la mossa con cui l'agente rimane ad aspettare nella cella attuale.

1.1.3 Percorso

Ogni agente all'interno della griglia segue un percorso π, questo è costituito da una sequenza di archi (mosse) $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. Si assume che tutte le mosse di spostamento, perciò lo spostamento di un agente da una cella a una adiacente, avvengano in tempo unitario e ,come già detto in precedenza, che i percorsi degli agenti inizino tutti al tempo 0

1.1.4 Vincoli

Anche se esiste un arco da un vertice ad un altro non è detto che all'istante t tale mossa sia disponibile. In un istante temporale un agente può $l_t(\alpha_i) = l_{t+1}$ *oppure* $(l_t(\alpha_i), l_{t+1}(\alpha_i)) \in E[G]$, perciò in un intervallo di tempo unitario può o rimanere nella stessa cella (rimanere fermo con un azione di *wait*) oppure spostarsi tra due celle adiacenti distinte.
In ogni caso gli spostamenti sono soggetti ai vincoli:

- In qualsiasi istante però la stessa cella non può essere occupata da due o più agenti distinti, formalmente : $l_t(\alpha_i) \neq l_t(\alpha_j)$ per ogni istante temporale t, $i \neq j$
- Quando due agenti occupano celle adiacenti non possono scambiarsi di posto.

Se i percorsi di due agenti distinti violano uno dei due vincoli, si dice che tali percorsi entrano in **collisione**.

Capitolo 2

strutture dati

2.1 gridGraph

Nel problema *PF4EA* l'algoritmo di ricerca opera su un grafo non orientato a griglia con grado massimo 9 (8 nodi vicini più se stesso). Il grafo viene rappresentato dalla classe `gridGraph`, che si occupa di creare i nodi e gli archi.

Ipotizzando che il nostro problema lavori con grafi di grandi dimensioni e sparsi, abbiamo scelto di usare le *liste di adiacenza* come struttura dati per memorizzare il grafo, in modo da ottimizzare lo spazio in memoria e diminuire i tempi d'accesso.

Per motivare ulteriormente la nostra scelta, di seguito confrontiamo le liste di adiacenza con la matrice di adiacenza.

2.1.1 Liste di adiacenza vs Matrice di adiacenza

Complessità spaziale

Le liste di adiacenza hanno una complessità spaziale di $O(|V| + |E|)$, dove $|V|$ è il numero di nodi e $|E|$ è il numero di archi (collegamenti tra i nodi). Nel nostro caso, ogni nodo ha al più 9 nodi adiacenti, quindi la complessità spaziale si riduce a $O(|V| + 9|V|)$. La matrice di adiacenza, invece, ha complessità spaziale $O(|V|^2)$, indipendentemente dal numero di archi.

Complessità temporale

Le liste di adiacenza hanno tempi d'accesso nel caso pessimo di $O(n)$, ma nel nostro caso sono costanti, perché hanno lunghezza massima 9 ed inizialmente abbiamo un array di `n` elementi.

Risulta quindi che per il nostro problema specifico le liste di adiacenza sono più efficienti della matrice di adiacenza sia in termini di spazio sia di tempo.

2.1.2 Implementazione in Python delle liste di adiacenza

Per implementare le liste di adiacenza abbiamo valutato due opzioni, in modo da sfruttare le strutture dati native di Python: il dizionario di tipo `dict` e la lista di tipo `list` con elementi le tuple.

La scelta è ricaduta sul *dizionario*, perché dispone di metodi adatti alle operazioni di inserimento, ricerca, rimozione e modifica che dovevamo svolgere sui nodi e archi del grafo, con tempi d'accesso in media di $O(1)$, quindi molto efficienti. L'utilizzo della lista di tuple,

invece, avrebbe comportato per l'operazione di ricerca una complessità temporale media di $O(n)$.

Per rappresentare la lista di adiacenza, abbiamo quindi utilizzato una Lista di dizionari, ordinando quest'ultimi in base agli indici. I nodi sono numeri interi da 0 a `n-1` dove `n` è il numero totale dei nodi. Ogni dizionario contiene le copie nodo-peso, dove il nodo è la chiave e il peso è il valore.

In questo modo, possiamo accedere facilmente ai nodi adiacenti e ai pesi degli archi senza dover memorizzare esplicitamente gli indici e con tempi d'accesso costanti.

```
class GridGraph:
    def __init__(self, rows: int, cols: int, traversability_ratio: float
    , obstacle_agglomeration_ratio: float):
        """
        Inizializza una griglia di nodi con le dimensioni specificate.

        :param rows: Il numero di righe della griglia.
        :param cols: Il numero di colonne della griglia.
        :param traversability_ratio: La percentuale di attraversabilità
    della griglia.
        :param obstacle_agglomeration_ratio: La percentuale di
    agglomerazione degli ostacoli nella griglia.
        """

        self.rows = rows
        self.cols = cols
        self.size = rows * cols
        self.traversability_ratio = traversability_ratio

        self.obstacle_agglomeration_ratio = obstacle_agglomeration_ratio
        self.nodes: List[int] = list(range(0, self.size))
        self.adj_list: List[Dict[int, float]] = [{} for node in self.
    nodes]
        self.num_obstacles = self.calculate_num_obstacles()
        self.generate_neighbors()
        self.generate_obstacles()
```

Il costruttore della classe `GridGraph` riceve i seguenti parametri in ingresso:

- `rows`: numero di righe della griglia.
- `cols`: numero di colonne della griglia.
- `traversability_ratio`:percentuale di traversabilità della griglia. Rappresenta la percentuale di celle nella griglia che non sono ostacoli.
- `obstacle_agglomeration_ratio`: Definisce la distribuzione degli ostacoli nella griglia, essendo la generazione un processo semi-casuale.

Oltre ai parametri, il costruttore inizializza vari attributi:

- `self.nodes`: lista che enumera tutti i nodi presenti nella griglia.
- `self.adj_list`: lista di dizionari che rappresenta la lista di adiacenza del grafo. Ogni dizionario tiene traccia dei nodi adiacenti a un nodo specifico e i relativi pesi degli

archi. I dizionari vengono inizializzati vuoti per poi essere popolati dai metodi invocati successivamente.

- `self.num_obstacles` calcola il numero di ostacoli presenti nella griglia.

Il costruttore chiama infine i seguenti due metodi:

1. `self.generate_neighbors()`
2. `self.generate_obstacles()`

che hanno il compito rispettivamente di generare i nodi adiacenti e i nodi-ostacolo. Una descrizione dettagliata di tali metodi sarà presentata nel Capitolo 3

2.2 State

La classe `State` rappresenta gli stati del problema. Le sue istanze vengono utilizzate dalla classe `PathFinder`, in particolare dagli algoritmi di ricerca `ReachGoal` e `ReachGoal variant`, per definire lo stato iniziale in cui si trova il problema, gli stati successivi generati durante la ricerca e lo stato finale, quello che soddisfa il goal-test.

```
1  class State:
2      """
3      Rappresenta uno stato nel problema.
4
5      Attributes:
6          node: Il nodo corrispondente allo stato.
7          time: Il tempo associato allo stato.
8          parent: Lo stato genitore.
9          path_cost: Il costo del percorso per raggiungere lo stato.
10     """
11     def __init__(self, node: int, time: int, parent=None, path_cost:
       float = 0):
12         self.time = time
13         self.node = node
14         self.parent = parent
15         self.path_cost = path_cost
```

Un oggetto della classe `State` ha quattro attributi:

- `node`: nodo del grafo in cui si trova lo stato in un particolare istante di tempo
- `time`: istante di tempo il cui noto è stato scoperto
- `parent`: stato padre da cui si proviene.
- `path_cost`: costo del cammino dallo stato iniziale all'istanza

2.3 Open

Per tenere traccia degli stati scoperti non ancora esplorati, abbiamo utilizzato la `open list` sia nell'algoritmo di ricerca *Reach Goal* sia nella sua variante. La struttura dati di `open` è una *coda di priorità*, una scelta comune negli algoritmi di ricerca. La coda di priorità è una istanza della classe `PriorityQueue`, che abbiamo realizzato apposta, la quale implementa

un min heap appoggiandosi alla libreria Python `heapq`. Per inizializzare un oggetto di tipo `PriorityQueue` si passa al costruttore due parametri: lo stato iniziale e una funzione *lambda function* per ordinare gli elementi all'interno della coda. L'elemento con valore più basso è posto in cima alla coda.

```python
class PriorityQueue:
    """
    Una coda prioritaria che mantiene gli elementi in ordine di priorità.
    Gli elementi vengono ordinati in base alla funzione di priorità specificata durante l'inizializzazione.
    """

    def __init__(self, item: State, f: Callable[[Any], Any] = lambda x: x):
        """
        Inizializza la coda prioritaria con lo stato iniziale e la funzione di priorità.
        La funzione di priorità viene utilizzata per calcolare la priorità di ogni elemento.
        L'elemento iniziale viene inserito nella coda con la sua priorità calcolata.
        """
        self.f = f
        self.queue = [(f(item), item)]  # lista di coppie (score,item)
```

Per mantenere la coda ordinata, abbiamo usato i metodi `heapq.heappush` e `heapq.heappop`. Il primo si occupa di inserire un elemento mantenendo la struttura ordinata. Accetta come input lo stato da inserire e il suo valore di priorità, che viene calcolato in anticipo.

L'operazione di *pop*, invece, viene svolta all'inizio del ciclo `while` e restituisce l'elemento in cima alla coda, ovvero quello a priorità maggiore.

```python
def add(self, item: State):
        """
        Aggiunge un elemento alla coda prioritaria.
        L'elemento viene inserito nella coda con la sua priorità calcolata.
        """
        pair = (self.f(item), item)
        heapq.heappush(self.queue, pair)

    def pop(self) -> State:
        """
        Rimuove e restituisce l'elemento con la priorità minima.
        """
        return heapq.heappop(self.queue)[1]
```

2.4 Closed

Per memorizzare gli stati esplorati dall'algoritmo di ricerca, abbiamo sperimentato due diverse strutture dati native del linguaggio Python: `list` e `set`. Inizialmente abbiamo usato la *lista* tuttavia ci siamo resi conto che i tempi d'accesso non erano ottimali, in quanto ogni volta che si espande un nodo bisogna controllare se lo stato generato sia già presente in *Closed*, e in caso positivo, scartarlo per procedere con lo stato successivo. Tale operazione viene svolta

di frequente e comporta una scansione lineare della lista, con una complessità temporale media di $O(n)$ dove n è la lunghezza della lista. Per migliorare le prestazioni abbiamo deciso di sostituire il tipo `list` con il tipo `set`. Infatti l'*insieme* rappresenta al meglio *Closed* in quanto dotata delle seguenti caratteristiche:

- Unicità degli elementi di un set
- Non ordinamento
- Accesso agli elementi con complessità temporale media costante di $= O(1)$ e nel caso peggiore di $O(n)$

Andando nel dettaglio, abbiamo scoperto che dietro il tipo `set` c'è una struttura dati a noi nota, ovvero la *tabella hash*.

Quindi l'uso di `set` ha reso l'algoritmo di ricerca più efficienti e ha ridotto i tempo di esecuzione.

2.5 Path

Per rappresentare i percorsi degli agenti pre-esistenti e il percorso individuato dall'entry-agent abbiamo utilizzato la *Lista*, una struttura nativa di Python. Una lista è una sequenza ordinata di elementi, che nel nostro caso sono i nodi del grafo che costituiscono il cammino degli agenti. Ogni elemento della lista ha un indice che indica la sua posizione nella sequenza, e che corrisponde all'istante di tempo in cui l'agente visita il nodo. La scelta di utilizzare la Lista è stata determinata valutando le operazioni che avremmo svolto su di essa, in particolare:

- Inserimento in coda
- Accesso ad un elemento tramite indice
- Verifica della lunghezza della lista

Tali operazioni sono messe a disposizione dai metodi che operano sul tipo `list` con una complessità temporale costante di $O(1)$. Pertanto, sono molti efficienti rispetto ad altre strutture dati.

Capitolo 3

Algoritmi

3.1 Generazione del problema

La classe `Problem` rappresenta un problema di ricerca PF4EA come una grafo a griglia, in cui ogni agente deve seguire un percorso da un nodo di partenza ad un nodo di arrivo.

Il costruttore di `Problem` riceve come parametri i valori che caratterizzano una specifica istanza del problema. Successivamente si appoggia alle classe `GridGraph` per rappresentare la griglia in cui gli agenti operano e alla classe `Agents` per la creazione degli agenti e la generazione dei loro percorsi sulla griglia.

Inoltre fornisce un metodo per verificare se un nodo è un goal e una serie di metodi per rappresentare l'istanza del problema in formato stringa.

3.2 Generazione della griglia

Di seguito vengono scritti i principali metodi che si occupano di generare la griglia. In particolare verranno trattati i metodi `generate_neighbors()` e generate_obstacles().

3.2.1 Generazione dei vicini

Il metodo `generate_neighbors` ha il compito di generare le liste di adiacenza per ogni nodo della griglia.

```
1 def generate_neighbors(self):
2   """
3   Genera gli elenchi di adiacenza per ogni nodo nella griglia.
4   """
5   for current_node in self.nodes:
6   self.connect_adjacent_nodes(current_node)
```

Il metodo `connect_adjacent_nodes` accetta come input un nodo specifico e stabilisce le connessioni con i nodi adiacenti sulla griglia. Questo processo viene eseguito attraverso i seguenti passaggi:

1. Determina la posizione del nodo sulla griglia, rappresentata dalle coordinate cartesiane (col, row).
2. Definisce un insieme di direzioni possibili per identificare i nodi adiacenti. Questo insieme include le direzioni cardinali e diagonali.

3. Per ciascuna direzione, calcola la posizione del nodo adiacente m sommando la posizione del nodo corrente n alla direzione corrente. Il peso dell'arco w varia in base alla direzione: è pari a 1 per le direzioni cardinali e a $\sqrt{2}$ per le direzioni diagonali.

4. Dopo aver determinato la posizione dei nodi n e m, e del peso w, il metodo `add_edge(n,m,w)` viene utilizzato per stabilire la connessione. Questo metodo aggiunge l'arco '(n,m)' con peso 'w' alla lista di adiacenza di 'n', rappresentata da un dizionario che utilizza il nodo adiacente come chiave e il peso dell'arco come valore.

```
def connect_adjacent_nodes(self, node: int):
    """
    Connette il nodo specificato ai suoi nodi adiacenti nella griglia.
    """
    row = (node) // self.cols
    col = (node) % self.cols

    # Definisco le direzioni dei nodi
    directions = [
    (-1, 0),  # nord
    (1, 0),  # sud
    (0, -1),  # ovest
    (0, 1),  # est
    (-1, -1),  # nord-ovest
    (-1, 1),  # nord-est
    (1, -1),  # sud-ovest
    (1, 1),  # sud-est
    ]

    # connetto i nodi adiacenti in base alle direzioni
    for direction_row, direction_col in directions:
    new_row, new_col = row + direction_row, col + direction_col
    if 0 <= new_row < self.rows and 0 <= new_col < self.cols:
        neighbor = new_row * self.cols + new_col
        weight = WEIGHT_CARDINAL_DIRECTION
        if direction_row != 0 and direction_col != 0:
        weight = WEIGHT_DIAGONAL_DIRECTION
        self.add_edge(node, neighbor, weight)
    self.add_edge(node, node, WEIGHT_CARDINAL_DIRECTION)

#---
def add_edge(self, node1: int, node2: int, weight: float = 1.0) -> None:
    self.adj_list[node1][node2] = weight
```

3.2.2 Generazione degli ostacoli

`generate_obstacles` genera gli ostacoli nella griglia, ovvero dei nodi senza archi, svolgendo i seguenti passaggi:

1. Calcola il numero di ostacoli da generare con il metodo `calculate_num_obstacles`, basandosi sulla percentuale di attraversabilità della griglia.

2. Crea un insieme di nodi-ostacolo con il metodo `build_obstacles`, che verranno trasformati in nodi-ostacolo dal metodo `set_as_obstacles` andando a rimuovere le connessioni tra i nodi-ostacolo e i loro nodi adiacenti.

3. `build_obstacles` genera i nodi-ostacolo in maniera semi-casuale raggruppandoli in cluster. Il metodo prevede i seguenti passaggi:
   (a) Inizializza un set vuoto di nodi-ostacolo `obstacles`
   (b) Determina la dimensione del cluster con il metodo `calculate_cluster_size`
   (c) Ogni cluster viene costruito a partire da un nodo iniziale scelto casualmente tra tutti i nodi che non sono già ostacoli e non hanno vicini gli ostacoli tramite il metodo `find_start_node`.
   (d) il metodo `generate_obstacle_cluster` a partire dal nodo di partenza forma il cluster andando ad aggiungere, ad ogni iterazione, un nodo scelto casualmente tra i nodi adiacenti al nodo corrente che non sia già ostacolo o che abbia vicini ostacoli.
4. Aggiunge il cluster all'insieme `obstacles`
5. Una volta generati tutti i cluster ritorna `obstacles`

```
def generate_obstacles(self):
        """
        Genera gli ostacoli nella griglia in base alla percentuale di attraversabilità e di agglomerazione degli ostacoli.
        """
        if (self.num_obstacles != 0):
            obstacles = self.build_obstacles()
            for node in obstacles:
                self.set_as_obstacle(node)

def build_obstacles(self) -> Set[int]:
  """
  Genera gli ostacoli nella griglia in base alla percentuale di agglomerazione degli ostacoli.
  """
  obstacles: Set[int] = set()
  cluster_size = self.calculate_cluster_size()
  num_clusters = self.calculate_num_clusters(cluster_size)
  for i in range(num_clusters):
  start = self.find_start_node(obstacles)
  if start is not None:
      cluster = self.generate_obstacle_cluster(
      cluster_size, obstacles, cluster=[start]
      )
      obstacles.update(cluster)
  return obstacles

def find_start_node(self, obstacles: Set[int]) -> int:
  """
  Trova un nodo di partenza per generare un nuovo cluster di ostacoli.
  """
  available = [
  node
  for node in self.nodes
  if node not in obstacles
  and self.are_neighbors_obstacle_free(node, obstacles)
  ]
  if not available:
  return None
  start = random.choice(available)
```

```
39    return start
40
41  def are_neighbors_obstacle_free(self, node: int, obstacles: Set[int]) ->
        bool:
42    neighbors = self.get_adj_list(node)
43    for neighbor in neighbors:
44    if neighbor in obstacles:
45        return False
46    return True
47
48  def generate_obstacle_cluster(
49    self, dim_cluster: int, obstacles: Set[int], cluster: List[int]
50  ) -> List[int]:
51    """
52    Genera un nuovo cluster di ostacoli a partire dal nodo specificato.
53    """
54    while len(cluster) < dim_cluster:
55    available = []
56    for node in cluster[::-1]:
57        neighbors = self.get_adj_list(node)
58        available = [
59        node
60        for node in neighbors
61        if node not in cluster
62        and self.are_neighbors_obstacle_free(node, obstacles)
63        ]
64        if available:
65        break
66    if not available:
67        return cluster
68    next_node = random.sample(available, 1)[0]
69    cluster.append(next_node)
70    if len(cluster) == dim_cluster:
71    return cluster
72    return cluster
73
74      def set_as_obstacle(self, node: int) -> None:
75          neighbors = list(self.adj_list[node].keys())
76          for neighbor in neighbors:
77              self.delete_edge(node, neighbor)
```

3.3 Generazione dei percorsi degli agenti

La classe `Agents` si occupa di generare i percorsi degli agenti. Il costruttore richiede due parametri:

- `max_time` è un intero che indica la durata massima di un percorso
- `num_paths` è il numero di percorsi che la classe `Agents` deve produrre

Inoltre, viene inizializzata una lista `paths`, all'inizio è vuota, che contiene i percorsi generati.

3.3.1 generate_paths

La classe `Problem`, dopo avere creato l'oggetto `agents_paths` della classe `Agents`, invoca su di esso il metodo `generate_paths` per generare i percorsi. Tale metodo riceve come parametri l'oggetto di tipo `GridGraph` e la lista `available_nodes`, che contiene i nodi disponibili.

```
def generate_paths(self, grid: GridGraph, available_nodes: List[int]) ->
    List[List[int]]:
    random.shuffle(available_nodes)
    cols = grid.get_dim()[1]

    for _ in range(self.num_paths):
        if not available_nodes:
            print("Non ci sono più nodi disponibili")
            break

        path = self._generate_single_path(grid, available_nodes, cols)
        if path is not None:
            self.paths.append(path)
```

Il metodo, ad ogni iterazione del ciclo, genera un singolo percorso e lo aggiunge alla lista `paths`.

Genera Single Path Il metodo `_generate_single_path` seleziona un nodo casuale dalla lista `available_nodes`, lo rimuove da essa, e lo assegna come nodo di partenza al percorso in costruzione. Successivamente, determina la durata percorso campionando un valore da una distribuzione normale. Tale espediente serve a evitare percorsi troppo brevi o troppo lunghi. Il metodo entra quindi in un ciclo che ha lo scopo di generare un percorso casuale privo di collisioni con i percorsi precedenti e che termina quando il tempo stabilito è scaduto, oppure in assenza di nodi disponibili. Infine, viene restituito il percorso generato.

```
    def _generate_single_path(
        self, grid: GridGraph, available_nodes: List[int], cols: int
    ) -> Optional[List[int]]:
        current_node = available_nodes.pop()
        path = [current_node]

        mu = (1 + self.max_time) / 2
        sigma = self.max_time / 4
        agent_path_duration = int(random.gauss(mu, sigma))

        for time in range(agent_path_duration):
            if not available_nodes:
                break

            neighbors = list(grid.get_adj_list(current_node))
            random.shuffle(neighbors)

            for next_node in neighbors:
                if is_free_collision(self.paths, current_node, next_node
    , time, cols):
                    path.append(next_node)
                    current_node = next_node
                    break
```

```
23
24          return path
```

Is Free Collision La funzione `is_free_collision` è un componente fondamentale dell'algoritmo di ReachGoal, che viene usata generare i percorsi degli agenti evitando collisioni tra di essi.

```
def is_free_collision(
    agent_paths: List[List[int]], node: int, next_node: int, time: int,
    cols: int
) -> bool:
    """
    Verifica se ci sono collisioni tra il percorso degli agenti e il
    nodo successivo.

    Args:
        agent_paths (List[List[int]]): Lista dei percorsi degli agenti.
        node (int): Nodo corrente.
        next_node (int): Nodo successivo.
        time (int): Tempo corrente.
        cols (int): Numero di colonne nella griglia.

    Returns:
        bool: True se non ci sono collisioni libere, False altrimenti.
    """
    for path in agent_paths:
        agent_location, agent_next_location = calculate_agent_locations(
    path, time)
        if check_collision(
            node,
            next_node,
            agent_location,
            agent_next_location,
            cols,
        ):
            return False
    return True
```

La funzione `calculate_agent_locations` si occupa di determinare la posizione di un agente nell'istante di tempo corrente e in quello immediatamente successivo. Tali informazioni sono sfruttate dalla funzione `check_collision`, che controlla se esiste una collisione tra due agenti in base ai vincoli stabiliti dai requisiti.

```
def calculate_agent_locations(path: List[int], time: int) -> Tuple[int,
    int]:
    path_length = len(path) - 1
    if time >= path_length:
        node = next_node = path[-1]
    else:
        node = path[time]
        next_node = path[time + 1]
    return node, next_node
#---
def check_collision(
```

```
    node: int, next_node: int, agent_location: int, agent_next_location:
     int, cols: int
) -> bool:
    return any(
        [
            agent_next_location == next_node,
            agent_next_location == node and agent_location == next_node,
            node - 1 == agent_next_location and agent_location ==
    next_node,
            agent_location - cols == next_node and node - cols ==
    agent_next_location,
            node == agent_next_location - 1 and agent_location ==
    next_node - 1,
            node + cols == agent_next_location and agent_location + cols
     == next_node,
        ]
    )
```

3.3.2 Nodo iniziale e finale dell'entry agent

Il nodo di partenza e il nodo di arrivo dell'*entry agent* sono selezionati in modo casuale dalla lista `empty_node` della griglia al momento della creazione dell'oggetto della classe `Problem`.

3.3.3 PathFinder

La classe `PathFinder` implementa l'algoritmo di ricerca *ReachGoal*, la sua variante, e l'algoritmo per la ricostruzione del percorso *ReconstructPath*. Essa viene instanziata nel modulo `__main__.py` ricevendo come parametri, un oggetto della classe `Problem`, un oggetto della classe `Heuristic` , e un flag booleano `use_variant` che indica se usare la variante `_ReachGoal_variant`.

```
class PathFinder:
    def __init__(self, problem, heuristic, use_variant=False):
        """
        Inizializza un nuovo oggetto risolutore per il problema PF4EA.

        :param problem: Il problema da risolvere.
        :param heuristic: L'euristica da utilizzare per valutare i nodi.
        :param use_variant: Indica se utilizzare la variante di
    ReachGoal.
        """
        self.problem = problem
        self.heuristic = heuristic
        self.use_variant = use_variant
        self.f_score = lambda state: state.path_cost + self.heuristic(
    state.node)
        self.path = None
        self.open = None
        self.closed = None
        self.wait = None
        self.path_cost = None
        self.__start_time = time.process_time()
#---
    def search(self):
        search_algorithm = (
```

```
23              self._ReachGoal_variant() if self.use_variant else self.
    _ReachGoal()
24          )
25          return search_algorithm
```

3.3.4 _ReachGoal

Il metodo `_ReachGoal` implementa l'algoritmo descritto sotto forma di pseudo codice nell'*elaborato*. Di fatto esegue una versione modificata della ricerca A^* per trovare il percorso ottimo dallo stato iniziale allo stato goal, evitando le collisioni con gli altri agenti presenti nella griglia.

Possiamo suddividere il metodo che implementa *ReachGoal* in sezioni in modo da facilitarne l'analisi.

Inizializzazione All'inizio vengono create le strutture dati necessarie per l'algoritmo, che sono state trattate nel secondo capitolo.

```
1   def _ReachGoal(self) -> Result:
2   """ Trova il percorso ottimo per raggiungere l'obiettivo nel problema
      specificato.
3           """
4   init = State(self.problem.init, 0)
5   self.open = PriorityQueue(init, f=self.f_score)
6   self.closed = set()
```

Ciclo principale L'algoritmo entra in un ciclo finché la *open list*. non risulta vuota. Ad ogni iterazione: -

- Viene estratto lo stato con il valore minore funzione di costo `f_score` dalla coda `open` e assegnato a `current_state`.
- Si controlla che lo stato corrente non superi il tempo massimo consentito dal problema, il tal caso l'algoritmo salta l'iterazione corrente e passa alla successiva.
- Altrimenti lo stato corrente viene aggiunto nel set `closed`
- Poi si verifica se lo stato corrente soddisfa il *goal test*, e in caso affermativo, si ricostruisce il percorso partendo dallo stato iniziale e si restituisce la soluzione trovata. Nel nostro caso si utilizza la classe Result Factory che implementa il pattern Factory per la creazione di oggetti `Result`, una struttura dati per memorizzare i risultati prodotti dalla ricerca.

```
1 while self.open:
2   current_state = self.open.pop()
3   time = current_state.time
4
5   if time > self.problem.maximum_time:
6       continue
7
8   self.closed.add(current_state)
```

```
9
10    if self.problem.is_goal(current_state.node):
11        self.path, self.wait = self.ReconstructPath(init, current_state)
12        self.path_cost = current_state.path_cost
13
14        return ResultFactory.create_result(self, self.__execution_time)
```

esplorazione stati figli In questa sezione vengono esplorati gli stati figli. La funzione ausiliaria `expand` prende due parametri: il problema e lo stato corrente, e restituisce una lista con tutti i possibili stati figli dello stato corrente che sono stati generati.

Successivamente, per ogni stato figlio generato, si controlla se è stato già esplorato, verificando se è presente nel set `closed`. Si effettua poi un controllo sulle collisioni, utilizzando la funzione `is_free_collision`. Se il percorso risulta attraversabile, l'algoritmo controlla se lo stato figlio è già presente nella coda `open` e in caso negativo lo aggiunge alla coda. Se invece è già presente ma il nuovo percorso ha un costo determinato da `f_score` inferiore, allora lo stato figlio viene aggiornata in `open`, con il nuovo costo.

```
1    for child_state in expand(self.problem, current_state):
2        if child_state not in self.closed:
3        current_node = current_state.node
4        child_node = child_state.node
5
6        traversable = is_free_collision(
7        self.problem.agent_paths,
8        current_node,
9        child_node,
10       time,
11       self.problem.cols,
12       )
13
14       if traversable:
15       if child_state not in self.open:
16           self.open.add(child_state)
17       elif self.f_score(child_state) < self.open[child_state]:
18           del self.open[child_state]
19           self.open.add(child_state)
20
21 return ResultFactory.create_result(self, self.__execution_time)
```

Se la *open list* risulta vuota e lo stato goal non è stato raggiunto, alla fine verrà restituito un risultato che indice che non è stato trovato alcun percorso.

ReconstructPath

`ReconstructPath` è la funzione che viene chiamata dagli algoritmi `ReachGoal` e `ReachGoal_variant`, dopo avere concluso la ricerca, per ricostruire il percorso ottimo dallo nodo iniziale allo nodo goal. In sostanza, la funzione ricostruisce il percorso seguendo i riferimenti ai genitori di ciascuno stato, e calcola il tempo totale d'attesa dell'agente. La funzione inizia creando una *deque* `path` inizializzata con il nodo goal. Abbiamo scelto come struttura dati per memorizzare la soluzione *deque* in modo da avere complessità temporale costante di

0(1). Successivamente, la funzione entra in un ciclo che termina quando il nodo dello stato corrente coincide con il nodo iniziale. Ad ogni iterazione, si aggiunge il nodo genitore ricavato dallo stato corrente in testa alla *deque* `path`, poi si controlla se il nodo dello stato corrente e il nodo dello stato genitore sono uguali, perché in tal caso significa che l'agente rimane fermo in quel nodo per un istante di tempo, di consequenza si incrementa di uno il contatore `wait` che indica il tempo totale in cui l'entry agent è rimasto fermo. Infine, si aggiorna lo stato corrente con `next_state`. La funzione termina ritornando il percorso (convertito in una lista) e il valore `wait`.

```
def ReconstructPath(self, init: State, goal: State):
  path = deque([goal.node])
  current_state = goal
  wait = 0
  while current_state.node != init.node:
  next_state = current_state.parent
  path.appendleft(next_state.node)
  if current_state.node == next_state.node:
      wait += 1
  current_state = next_state
  return list(path), wait
```

3.3.5 ReachGoal Variant

Il metodo `Reachgoal_variant` adotta una strategia di ricerca alternativa basata sull'*euristica del cammino dei percorsi rilassati* che stima per ogni nodo il costo del cammino minimo che porta al nodo goal in assenza di agenti. Vediamo di seguito gli ulteriori criteri di terminazione introdotti dalla variante.

Nessun predecessore per il nodo iniziale Se il nodo iniziale non ha un predecessore, ciò significa che non esiste un cammino dal nodo iniziale al nodo goal, di conseguenza l'algoritmo termina con un fallimento.

```
predecessors = self.heuristic.predecessors
init = State(self.problem.init, 0)
self.open = PriorityQueue(init, f=self.f_score)
self.closed = set()

if predecessors[init.node] is None:
return ResultFactory.create_result(self, self.__execution_time)

while self.open:
  ...
  ...
```

Esistenza percorso libero da collisioni Dal punto di vista computazionale,

`_ReachGoal _variant` è più efficiente di `_ReachGoal`, in quanto se esiste un percorso privo di collisioni dal nodo iniziale al nodo goal, l'algoritmo evita di esplorare ulteriormente lo spazio degli stati e può determinare subito la soluzione. Vediamo di seguito i dettagli implementati.

```
while self.open:
  ...
  ...
  current_node = current_state.node
  path_free = is_path_free(self.problem, current_node, time,
    predecessors)

  if path_free:
      path_from_current = self.heuristic.return_path(
      predecessors[current_state.node], self.problem.goal
      )
      path_to_current, self.wait = self.ReconstructPath(init,
    current_state)
      self.path = path_to_current + path_from_current
      self.path_cost = (
      current_state.path_cost
      + self.heuristic.distances[current_state.node]
      )
```

Is Path Free La funzione `is_path_free` ha il compito di verificare l'esistenza di un percorso libero da un nodo specificato al nodo goal. Essa riceve in input il problema, un nodo, l'istante di tempo corrente, e i predecessori. (Precisiamo che i predecessori sono stati implementati con il tipo `dict` di Python e tale struttura ha il compito di mappare ogni nodo al suo predecessore nel cammino minimo verso il nodo goal.) La funzione entra in un ciclo che termina negativamente se si raggiunge la durata massima stabilita o se si riscontrano collisioni. Ad ogni iterazione, si determina il nodo successivo sul cammino dal dizionario dei predecessori e si verifica se il movimento dal nodo corrente al nodo successivo è privo di collisioni. Tale compito è svolto dalla funzione `is_free_collision` precedentemente analizzata.

Se `is_free_collision` ritorna *True* si verifica che il `next_node` soddisfi il `goal_test` e in caso affermativo la funzione principale`is_path_free` ritorna *True*, indicando che esiste un percorso libero.

```
def is_path_free(problem, node: int, time: int, predecessors: List[int])
    -> bool:
    while time < problem.maximum_time:
        next_node = predecessors[node]
        if is_free_collision(problem.agent_paths, node, next_node, time,
    problem.cols):
            if problem.is_goal(next_node):
                return True
            node = next_node
            time += 1
        else:
            return False
    return False
```

Costruzione del percorso finale Il `path` finale viene ricostruito concatenando i due sottopercorsi restituiti dai metodi `return_path` e `ReconstructPath`.

`return_path` è un metodo della classe `HeuristicRelaxPath` che ha il compito di ricostruire il percorso libero e prende come parametri in ingresso il nodo di partenza (nel nostro caso il predecessore del nodo corrente) e il nodo goal.

```
    def return_path(self, starting_node, end_node):
        path = []
        while starting_node != end_node:
            path.append(starting_node)
            starting_node = self.predecessors[starting_node]
        path.append(end_node)
        return path
```

Mentre il sotto-percorso dal nodo iniziale al nodo corrente viene ricostruito dalla funzione `ReconstructPath` precedentemente analizzata.

3.4 Generazione delle euristiche

Il modulo `heuristic.py` definisce le euristiche che possono essere applicate agli algoritmi di ricerca per il problema PF4EA. Le euristiche presenti sono: distanza di Manhattan, distanza euclidea, euristica di Chebyshev, distanza diagonale e euristica del cammino rilassato. Queste euristiche sono implementate come sottoclassi della classe astratta `Heuristic`, che fornisce una interfaccia comune per tutte. Ogni sottoclasse concreta implementa il metodo astratto `estimate` che calcola il valore euristico di uno stato.

```
class DiagonalDistance(Heuristic):
    def __init__(self, grid, goal):
        super().__init__(grid, goal)
        self.col = self.grid.get_dim()[1]
        self.D = WEIGHT_CARDINAL_DIRECTION
        self.D2 = WEIGHT_DIAGONAL_DIRECTION
        self.goal = goal
        self.x_goal, self.y_goal = get_coordinates(self.goal, self.col)

    def estimate(self, init):
        """
        diagonal: calcola la distanza diagonale
        param col: colonne della griglia
        param init: nodo di partenza
        param goal: nodo di arrivo
        param D: costo per movimento cardinale
        param D2: costo per movimento diagonale
        return: euristica secondo la formula diagonale
        """
        x_init, y_init = get_coordinates(init, self.col)

        dx = abs(x_init - self.x_goal)
        dy = abs(y_init - self.y_goal)
        return self.D * (dx + dy) + (self.D2 - 2 * self.D) * min(dx, dy)

    def __call__(self, init):
        return self.estimate(init)


class ChebyshevDistance(Heuristic):
    def __init__(self, grid, goal):
```

```
32          super().__init__(grid, goal)
33          self.col = self.grid.get_dim()[1]
34          self.x_goal, self.y_goal = get_coordinates(self.goal, self.col)
35          self.D = WEIGHT_CARDINAL_DIRECTION
36          self.D2 = WEIGHT_DIAGONAL_DIRECTION
37
38      def estimate(self, init):
39          """
40          param goal: nodo di arrivo
41          param D: costo per movimento cardinale
42          param D2: costo per movimento diagonale
43          return: euristica secondo la formula diagonale
44          """
45          x_init, y_init = get_coordinates(init, self.col)
46
47          dx = abs(x_init - self.x_goal)
48          dy = abs(y_init - self.y_goal)
49          return self.D * (dx + dy) + (self.D2 - 2 * self.D) * min(dx, dy)
50
51      def __call__(self, node):
52          return self.estimate(node)
53
54
55  class ManhattanDistance(Heuristic):
56      def __init__(self, grid, goal):
57          super().__init__(grid, goal)
58          self.col = self.grid.get_dim()[1]
59          self.x_goal, self.y_goal = get_coordinates(self.goal, self.col)
60          self.D = WEIGHT_CARDINAL_DIRECTION
61
62      def estimate(self, init):
63          """
64          manhattam: calcola la distanza di manhattam
65          param col: colonne della griglia
66          param init: nodo di partenza
67          param goal: nodo di arrivo
68          param D: costo per movimento cardinale
69          return: euristicaa secondo la formula di manhattam
70          """
71          x_init, y_init = get_coordinates(init, self.col)
72          x_goal, y_goal = get_coordinates(self.goal, self.col)
73
74          dx = abs(x_init - self.x_goal)
75          dy = abs(y_init - self.y_goal)
76
77          return self.D * (dx + dy)
78
79      def __call__(self, init):
80          return self.estimate(init)
81
82
83  class EuclideanDistance(Heuristic):
84      def __init__(self, grid, goal):
85          super().__init__(grid, goal)
86          self.col = self.grid.get_dim()[1]
87          self.D = WEIGHT_CARDINAL_DIRECTION
88          self.x_goal, self.y_goal = get_coordinates(self.goal, self.col)
89
90      def estimate(self, init):
```

```
91          """
92          Calcola la distanza euclidea
93          param col: colonne della griglia
94          param init: nodo di partenza
95          param goal: nodo di arrivo
96          param D: costo per movimento cardinale
97          return: euristica secondo la formula euclidea
98          """
99
100         x_init, y_init = get_coordinates(init, self.col)
101         dx = abs(x_init - self.x_goal)
102         dy = abs(y_init - self.y_goal)
103
104         return self.D * math.sqrt(dx ^ 2 + dy ^ 2)
105
106     def __call__(self, init):
107         return self.estimate(init)
```

3.4.1 Euristica del cammino rilassato

In questo paragrafo si analizza l'implementazione dell'*euristica del cammino rilassato*. Questa euristica è stata utilizzata nella variante di *ReachGoal*.

Il costruttore della classe `HeuristicRelaxPath` prende in input come argomenti la griglia, il nodo goal e inizializza le distanze e i predecessori di tutti i nodi utilizzando il metodo `estimate`. `distances` è un dizionario che mappa ogni nodo della griglia con la sua stima, ovvero la distanza minima dal nodo goal, mentre `predecessors` è un dizionario che associa ad ogni nodo il suo predecessore nel cammino minimo verso il nodo goal.

```
class HeuristicRelaxPath(Heuristic):
    def __init__(self, grid, goal):
        super().__init__(grid, goal)
        self.distances, self.predecessors = self.estimates()
```

Il metodo `estimates` implementa l'algoritmo di *Dijkstra all'indietro*, ovvero una variante dell'algoritmo di *Dijkstra standard* in cui al posto di procedere dal nodo iniziale, inizia dal nodo goal e si muove verso i nodi adiacenti, aggiornando le loro distanze dal goal se il percorso al nodo di destinazione attraverso il nodo corrente è più breve. Questo processo continua fino a quando tutti i nodi sono stati visitati. Analizziamo di seguito l'implementazione:

All'inizio del metodo vengono inizializzate le strutture dati `distances` impostando le distanze di ogni nodo a *inf*, ad eccezione del nodo di destinazione, la cui distanza è impostata a 0. Viene inoltre inizializzato il dizionario dei predecessori impostando i predecessori di ogni nodo a `None`.

Viene poi creata una coda di priorità `pq` implementata con uno *min heap* che contiene una tupla per ogni nodo, composta dalla sua distanza corrente e da un intero che rappresenta il nodo stesso. Inizialmente, la coda di priorità contiene solo il nodo di destinazione con distanza 0.

```
def estimates(self):
  distances = {node: float("inf") for node in self.grid.nodes}
```

```
3   distances[self.goal] = 0
4   predecessors = {node: None for node in self.grid.nodes}
5   pq = [(0, self.goal)]
```

L'algoritmo entra poi in un ciclo che continua fino a quando la coda di priorità non è vuota. In ogni iterazione del ciclo, viene estratto dalla coda di priorità il nodo con la distanza minima. Se la distanza estratta è maggiore della distanza corrente del noto, l'Iterazione viene interrotta per passare alla prossima.

Per ogni nodo adiacente a quello corrente, l'algoritmo calcola la distanza dal nodo di destinazione attraverso quello corrente. Se questa distanza è minore della distanza corrente del nodo adiacente, la distanza e il predecessore di quello adiacente vengono aggiornati. Il nodo adiacente viene poi inserito nella coda di priorità con la nuova distanza

```
1             while pq:
2             current_distance, current_node = heapq.heappop(pq)
3             if current_distance > distances[current_node]:
4                 continue
5
6             # Aggiornamento delle distanze dei nodi adiacenti
7             adj_list = self.grid.get_adj_list(current_node)
8             for neighbor, _ in adj_list.items():
9                 distance = current_distance + self.grid.get_edge_weight(
10                    current_node, neighbor
11                )
12                if distance < distances[neighbor]:
13                    distances[neighbor] = distance
14                    predecessors[neighbor] = current_node
15                    heapq.heappush(pq, (distance, neighbor))
16        return distances, predecessors
```

Capitolo 4

Istruzioni per l'uso

Di seguito si riporta la struttura dei file e delle cartelle.

- `benchmarks` contiene i file necessari per generare, testare e risolvere i problemi di ricerca.
- `pf4ea` contiene il codice sorgente del programma.

```
 [benchmarks]
    +-- [generators]        # file csv con le specifiche dei problemi
    +-- [output_csv]        # file csv di output usati nei test
    +-- [problems]          # salva istanze problema
    L-- [report]            # file report
        L-- [media]
[pf4ea]
    +-- __main__.py          # Punto di ingresso dell'applicazione
    +-- agents.py            # Generazione di percorsi agenti
    +-- cli.py               # Interfaccia a riga di comando
    +-- constants.py         # Costanti utilizzate nel progetto
    +-- gridGraph.py         # Generazione griglia
    +-- heuristic.py         # Euristiche del problema
    +-- input_handler.py     # Gestore input nella modalità manuale
    +-- plotGraph.py         # Rappresenta graficamente la griglia
    +-- problem.py           # Definisci il problema PF4EA
    +-- repository.py        # Gestisce le operazioni su file
    +-- result.py            # Gestisce i risultati della ricerca
    +-- search.py            # Implementa algoritmi ReachGoal e variante
    +-- state.py             # Generazioni istanze stato del problema
    +-- utils.py             # Funzioni ausiliarie
    +-- visualize.py         # Visualizzazione grafica griglia

README.md                    # Questo file
requirements.txt             # Dipendenze Python
test.ipynb                   # notebook Jupyter per i test sulle performance
```

4.1 Requisiti

Per avviare il programma è richiesto Python 3.10.11 o superiore. Per il corretto funzionamento del programma devi installare le dipendenze indicate in `requirements.txt`. Per

farlo, puoi usare il gestore di pacchetti `pip`, con il comando:

```
pip install -r requirements.txt
```

4.1.1 Avvio applicazione

Una volta installate le dipendenze, puoi avviare l'applicazione usando l'interprete di Python, specificando uno dei seguenti comandi:

- `gen` : genera da file i problemi e li risolve
- `run` : carica da file un problema e lo risolve
- `man` : genera e risolve un problema interagendo da terminale.

In qualsiasi momento è possibile possibile utilizzare l'opzione `-h` (o `--help`) per ottenere una descrizione delle opzioni disponibili.

Per esempio, per mostrare l'help del comando principale:

```
python gen -h
```

Di seguito vediamo nel dettaglio il funzionamento dei comandi

4.1.2 gen

Il comando `gen` legge da file le specifiche dei problemi che si vogliono sottoporre all'algoritmo di ricerca , crea le relative istanze e le risolve. Le opzioni disponibili sono:

- `-f`, `--file`: file di input da cui leggere le specifiche dei problemi.
- `h`, `--hueristic`: per l'euristica da adottare nella ricerca (h1,h2,h3,h4).
- `-v`,`--variant`: seleziona la variante di *ReachGoal*.
- `-r`, `--report`: salva il risultato dell'esecuzione su file markdown.
- `--show`: mostra a video la soluzione graficamente, usando matplotlib.
- `-s`, `--save`: salva l'istanza del problema su file pickle.
- `--csv_output`: scrive l'output su un file csv. (usato per il test sulle performance)

Per esempio, una volta definiti i problemi sul file, puoi sottoporli al programma nel seguente modo:

```
python pf4ea gen -f exp_0.csv --h h1 -r
```

Questo comando genererà e risolverà i problemi specificati nel file exp_0.csv, usando l'euristica h1 e salvando il report su file markdown.

Chiariamo che l'euristica viene inserita nel seguente modo:

- `h1`: Diagonal Distance
- `h2`: Chebyshev Distance
- `h3`: Manhattan Distance
- `h4`: Euclidean Distance

- `h5`: Heuristic Relax Path

4.1.3 run

Il comando `run` consente di caricare nel programma una istanza del problema precedentemente generata e di risolverla con l'algoritmo `ReachGoal` o la sua variante. Le opzioni supportate sono:

- `-i`, `--input`: file da cui caricare l'istanza del problema.
- `-o`, `--output`: file su cui salvare il risultato dell'algoritmo.
- `h`, `--hueristic`: per l'euristica da adottare (h1,h2,h3,h4)
- `-v`,`--variant`: seleziona la variante di *ReachGoal*.
- `-r`, `--report`: salva il risultato dell'esecuzione su file markdown
- `--show`: mostra a video la soluzione graficamente.

Per esempio, per caricare e risolvere un problema dal file pickle, puoi scrivere:

```
python pf4ea run -i 50x50_08_01_20_60_2190_2460.pkl --h h1 -r
```

Questo comando caricherà il problema dal file pickle, lo risolverà con l'euristica h1 e salverà il report su file markdown.

4.1.4 man

Il comando `man` consente di interagire con il programma attraverso il terminale, inserendo i dati del problema a mano. La soluzione verrà comunque mostrata sul file markdown.

4.2 Formato file

4.2.1 File di input

File csv

Il file di input contenente le specifiche dei problemi deve essere in formato csv.

```
    # exp_0.csv
rows,cols,traversability_ratio,obstacle_agglomeration_ratio
,num_agents,maximum_time
10,10,0.1,1,0,30
100,10,1,1,30,30
106,10,1,1,30,30
10,105,1,1,30,30
105,107,1,1,30,30
108,104,1,1,30,30
10564,102,1,1,30,30
```

Il file di input di default si trova nel percorso `benchmarks\generators\exp_0.csv`.

File pickle

il file pickle è un formato di Python per memorizzare un oggetto serializzato. I file contenenti le istanze si trova di default nel percorso

`benchmarks\problems\50x50_08_01_20_60_2190_2460.pkl`.

4.2.2 File di output

File markdown

I report sono dei file markdown che contengono le seguenti informazioni:

- Dati del problema
- Risultati della ricerca
- Performance
- Immagine con la soluzione grafica
- Video animazione della soluzione

Il file di report di default si trova nel percorso

`benchmarks\report\50x50_08_01_20_60_247_73_DiagonalDistance.md`.

File csv

Hai finiti del test sulle performance è possibile salvare l'output su un file csv. Ad esempio:

```
rows,cols,traversability_ratio,obstacle_agglomeration_ratio,
num_agents,maximum_time,init,goal,h_type,path_length,
path_cost,tot_states,percentage_visited_nodes,
unique_node_visited,wait,problem_time,
heuristic_time,search_time,mem_grid,
mem_heuristic,mem_open,mem_closed,mem_path

50,50,0.8,0.1,20,60,645,2285,DiagonalDistance,34,37.14213562373095,
947,12.15,243,0,0.0,0.0,0.109375,0.046875,
0.046875,0.046875,8.2109375,0.3203125
```

Di default viene salvato nel percorso `benchmarks\output_csv\output_exp_0.csv`.

Capitolo 5

Risultati

In questo capitolo analizziamo i risultati sperimentali relativi alle performance degli algoritmi di ricerca ReachGoal e variante, applicati a diverse istanze del problema PF4EA.

L'obiettivo è quello di esaminare l'impatto di alcuni parametri chiave, quali la dimensione della griglia, il numero di agenti, il numero e il fattore di agglomerazione degli ostacoli e l'euristica adottata per la ricerca.

Per ogni parametro oggetto della sperimentazione, abbiamo mantenuto costanti gli altri o lì abbiamo modificati in base a dei criteri prestabiliti, al fine di valutare il loro effetto sulle prestazioni. Ad esempio, nel "test sulla grandezza della griglia" abbiamo pensato ragionevolmente di fare variare in modo proporzionale alla grandezza della griglia la lunghezza massima del cammino minimo o il numero di agenti.

Abbiamo poi confrontato le prestazioni dell'algoritmo *ReachGoal* con la sua variante ottimizzata, ad esclusione del primo esperimento. Per quest'ultimo, in cui abbiamo indagato l'impatto dell'euristica sulle prestazioni ci siamo limitati ad usare solo l'algoritmo ReachGoal standard.

Per ottenere dei risultati più affidabili, tenendo conto dei fattori randomici, abbiamo ripetuto ogni esperimento più volte calcolando la media dei valori ottenuti. Tali dati sono stati raccolti in un file csv, che abbiamo poi elaborato con il modulo pandas di Python utile ad eseguire in modo rapido le operazioni di analisi e a mostrare i risultati ottenuti tramite grafici.

5.1 Test sull'euristica

In questi test sono state prese le 3 euristiche e gli sono stati sottoposti problemi di dimensione e complessità crescente 5.1: partendo da una griglia di dimensione 5x5, a una di 50x50.

rows	cols	traversability_ratio	obstacle_agglomeration_ratio	num_agents	maximum_time
5	5	0.8	0.1	3	5
10	10	0.8	0.1	5	10
15	15	0.8	0.1	8	15
20	20	0.8	0.1	10	20
25	25	0.8	0.1	13	25
30	30	0.8	0.1	15	30
40	40	0.8	0.1	20	40
50	50	0.8	0.1	25	50

Figura 5.1: configurazioni test sulle euristiche

In primo luogo prendiamo in considerazione il tempo di generazione dell'istanza del problema. In questo caso come si più vedere 5.2 le tre euristiche si comportano in modo praticamente

equivalente.

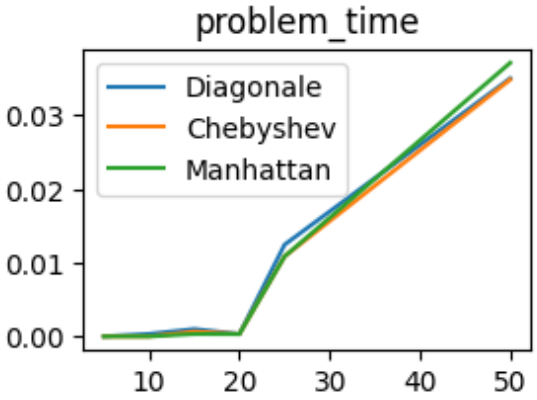

Figura 5.2: Sulle ascisse abbiamo il numero di colonne mentre sulle ordinate il tempo impegnato in secondi

Dopodiché per quanto riguarda invece il tempo di ricerca della soluzione, come possiamo vedere 5.3 incominciano a vedersi alcune differenze: la distanza di Chebyshev risulta essere, all'aumentare della complessità del problema, quella che porta alla soluzione in un minor tempo; subito dopo viene seguita dalla distanza diagonale e da quella di Manhattan.

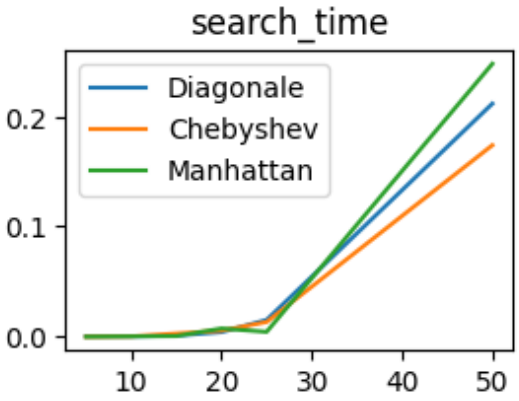

Figura 5.3: Sulle ascisse abbiamo il numero di colonne e sulle ordinate il tempo in secondi

Infine viene presa in considerazione la percentuale di nodi visitati 5.4, come si può vedere anche qui le tre euristiche portano a un comportamento abbastanza simile: con tutte e tre all'aumentare della complessità e della grandezza la percentuale di nodi visitati diminuisce.

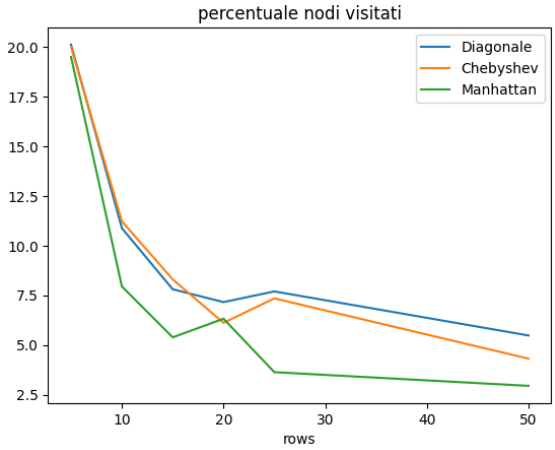

Figura 5.4: Sull'asse delle ascisse abbiamo il numero di colonne, sulle ordinate la percentuale di griglia visitata

5.2 Test sulla grandezza della griglia

Come detto in precedenza nella sperimentazione riguardante la dimensione della griglia 5.5 i parametri di interesse sono ovviamente le dimensioni, ma allo stesso tempo sono stati fatti variare anche altri fattori come il numero di agenti e la lunghezza massima dei percorsi, tutto però in proporzione alle dimensioni. Inoltre per semplicità è stato deciso di utilizzare sempre griglie quadrate e dalle dimensioni non eccessive: si parte anche qui da una di 5x5 e si arriva ad una 50x50; questo più che altro per questioni di tempistiche (avere la possibilità di eseguire più run).

rows	cols	traversability_ratio	obstacle_agglomeration_ratio	num_agents	maximum_time
5	5	0.8	0.1	3	5
10	10	0.8	0.1	5	10
15	15	0.8	0.1	8	15
20	20	0.8	0.1	10	20
25	25	0.8	0.1	13	25
30	30	0.8	0.1	15	30
40	40	0.8	0.1	20	40
50	50	0.8	0.1	25	50

Figura 5.5: configurazioni test sulla grandezza ella griglia

Dal primo grafico 5.6 si può vedere come il tempo di generazione del problema rimane pressoché lo stesso per entrambe le versioni.

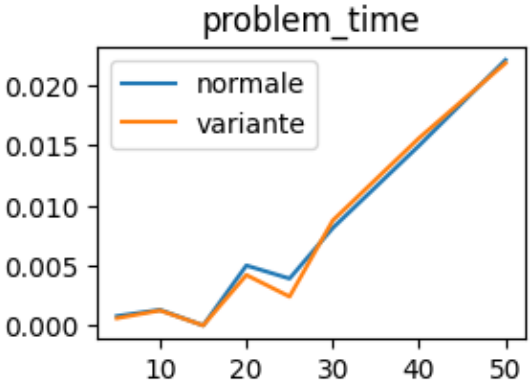

Figura 5.6: Tempo generazione problema

Per quanto riguarda invece il tempo di generazione dell'euristica 5.7 il tempo legato alla versione normale è nullo (non viene pre-calcolato nulla), mentre per la variante abbiamo del tempo speso che aumenta all'aumentare della dimensione della griglia.

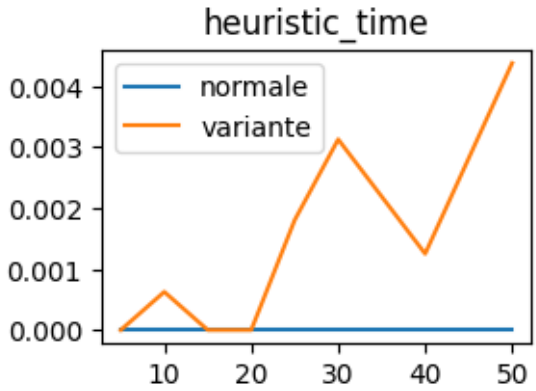

Figura 5.7: Tempo generazione euristica

Per quanto riguarda invece il tempo di ricerca 5.8 possiamo vedere che tra le due versioni vi è una netta differenza:la variante di ReachGoal ha delle performance migliori grazie al criterio di terminazione anticipata che, in buona parte dei casi, consente non appena trova un percorso libero dal nodo corrente al nodo goal di interrompe e di restituire la soluzione. Questo permette di restringere lo spazio degli stati da esplorare, riducendo di conseguenza il tempo di esecuzione.

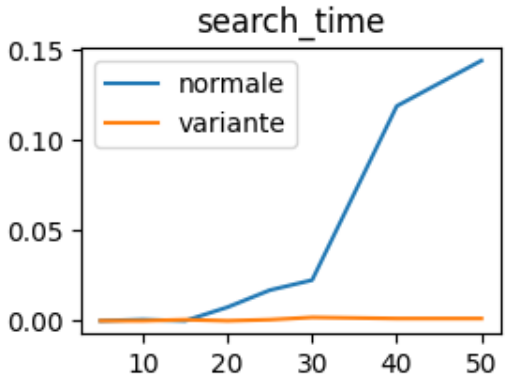

Figura 5.8: Tempo ricerca soluzione

Dal punto di vista invece della memoria occupata abbiamo che: la variante deve dedicare dello spazio in memoria aggiuntivo per i percorsi rilassati5.10, però allo stesso tempo la lista closed della versione normale risulta molto più impattante 5.9(esplorando più nodi è normale). Invece per quanto riguarda la memoria occupata dalla soluzione è praticamente equivalente in entrambe le versioni, come ci si può aspettare.

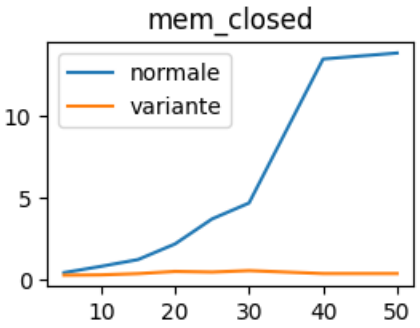

Figura 5.9: Sull'asse delle ascisse abbiamo il numero di colonne della griglia, mentre sulle ordinate la memoria occupata in kilobytes

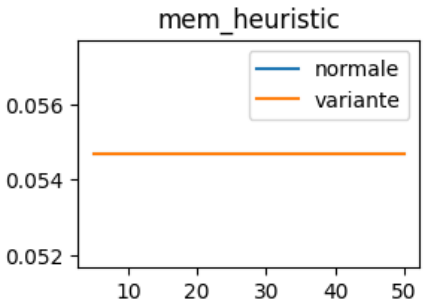

Figura 5.10: Sull'asse delle ascisse abbiamo il numero di colonne della griglia, mentre sulle ordinate la memoria occupata in kilobytes

Infine per quanto riguarda la percentuale di nodi visitati 5.11, in accordo a quello visto precedentemente, anche qui la variante risulta più efficiente, arrivando alla soluzione esplorando meno della griglia (probabilmente sempre grazie al check sul cammino rilassato).

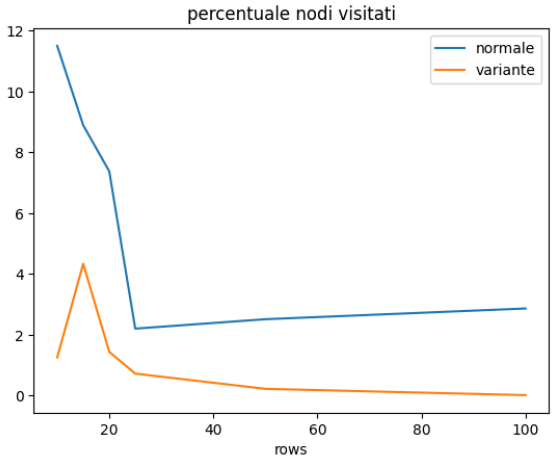

Figura 5.11: Percentuali nodi visitati

5.3 Test sul numero di agenti

In questa fase si è utilizzata sempre una griglia di dimensioni 50x50 ed è stato fatto variare il numero di agenti presenti al suo interno 5.12; partendo da 0 agenti, quindi griglia libera con solo gli ostacoli, fino a 50.

rows	cols	traversability_ratio	obstacle_agglomeration_ratio	num_agents	maximum_time
50	50	0.8	0.1	0	75
50	50	0.8	0.1	5	75
50	50	0.8	0.1	10	75
50	50	0.8	0.1	15	75
50	50	0.8	0.1	20	75
50	50	0.8	0.1	25	75
50	50	0.8	0.1	30	75
50	50	0.8	0.1	35	75
50	50	0.8	0.1	40	75
50	50	0.8	0.1	45	75
50	50	0.8	0.1	50	75

Figura 5.12: configurazioni test sul numero di agenti

Guardando al tempo di ricerca 5.13 si può vedere come si ha una situazione simile a quella precedente: la variante risulta impiegare meno tempo per arrivare alla soluzione. Però come ci si può aspettare per entrambe il tempo aumenta all'aumentare degli agenti.

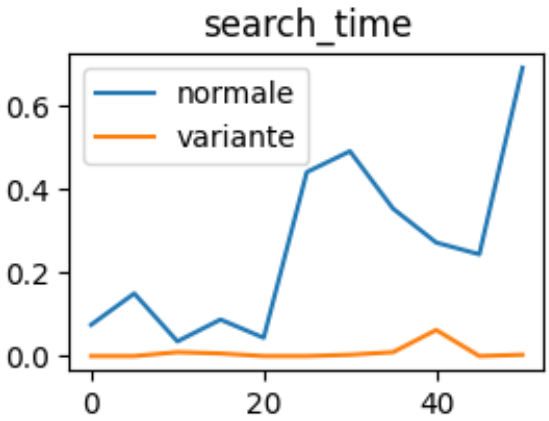

Figura 5.13: Tempo ricerca soluzione

per la percentuale di nodi visitati 5.14 si possono fare considerazioni simili a quelle precedenti: anche qui la variante risulta più efficiente, ma entrambi all'aumentare degli agenti peggiorano.

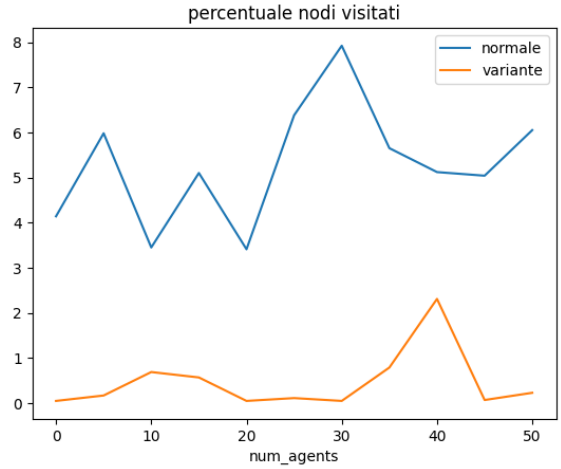

Figura 5.14: Percentuale nodi visitati

5.4 Test sul numero di ostacoli

Per questo test, come nel precedente, è stata mantenuta costante la dimensione della griglia, ma è stata fatta variare l' attraversabilità della griglia 5.15: partendo da 100%, quindi da una griglia senza ostacoli fino al massimo ad una del 40%. Pure in questo caso la variante risulta essere in vantaggio rispetto alla normale.

rows	cols	traversability_ratio	obstacle_agglomeration_ratio	num_agents	maximum_time
50	50	1	0.1	25	50
50	50	0.95	0.1	25	50
50	50	0.9	0.1	25	50
50	50	0.85	0.1	25	50
50	50	0.8	0.1	25	50
50	50	0.75	0.1	25	50
50	50	0.7	0.1	25	50
50	50	0.65	0.1	25	50
50	50	0.6	0.1	25	50
50	50	0.55	0.1	25	50
50	50	0.5	0.1	25	50
50	50	0.45	0.1	25	50
50	50	0.4	0.1	25	50

Figura 5.15: configurazioni test sul numero di ostacoli

Come in precedenza anche qui (ma in maniera inversa) diminuendo l' attraversabilità della griglia aumenta il tempo per trovare la soluzione 5.16.

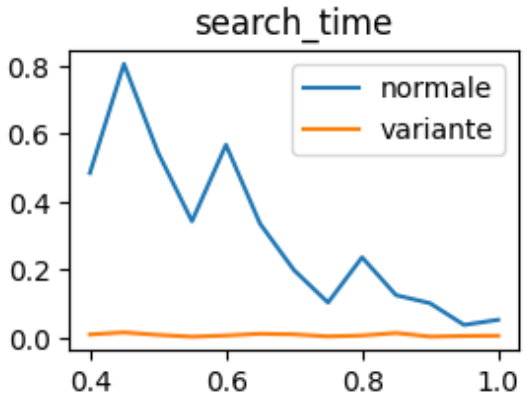

Figura 5.16: Tempo ricerca soluzione

Anche per quanto riguarda lo spazio occupato dalla lista closed 5.17, come prima, per entrambe la memoria aumenta al diminuire dell'attraversabilità. Pure in questo caso la variante risulta più efficiente.

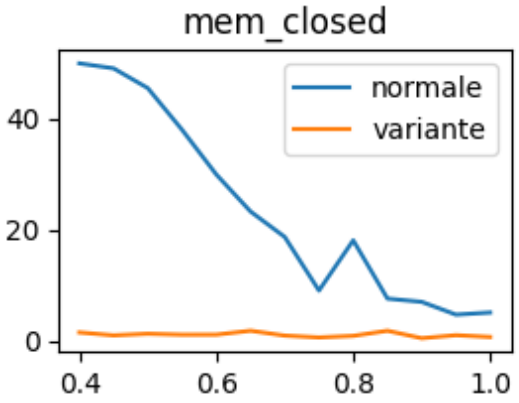

Figura 5.17: Memoria occupata da closed

Per quanto riguarda la percentuale di nodi visitati 5.18 al diminuire dell'attraversabilità aumenta la percentuale di nodi visitati, come in precedenza si può vedere la variante comportarsi in maniera più efficiente esplorando meno della griglia.

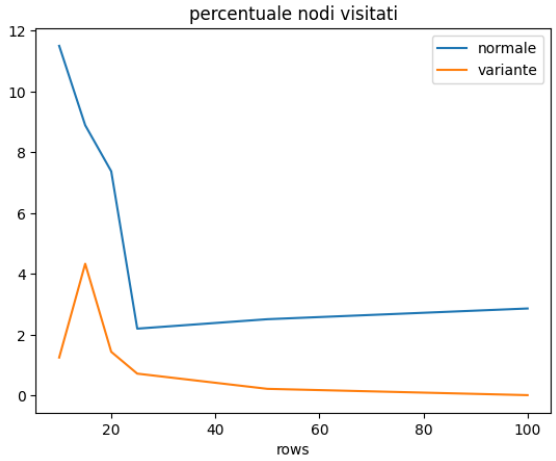

Figura 5.18: percentuale nodi visitati

5.5 Test sull'agglomerazione degli ostacoli

In fine per quanto riguarda l'agglomerazione degli ostacoli 5.19 si trovano risultati ancora concordi con quello che è stato visto in precedenza. Inoltre si può notare come; aumentando l'agglomerazione degli ostacoli aumenti anche il tempo per la generazione del problema5.20.

rows	cols	traversability_ratio	obstacle_agglomeration_ratio	num_agents	maximum_time
50	50	0.65	0.1	25	50
50	50	0.65	0.15	25	50
50	50	0.65	0.2	25	50
50	50	0.65	0.25	25	50
50	50	0.65	0.3	25	50
50	50	0.65	0.45	25	50
50	50	0.65	0.5	25	50
50	50	0.65	0.65	25	50
50	50	0.65	0.7	25	50
50	50	0.65	0.75	25	50
50	50	0.65	0.8	25	50
50	50	0.65	0.85	25	50
50	50	0.65	0.9	25	50
50	50	0.65	0.95	25	50
50	50	0.65	1	25	50

Figura 5.19: configurazioni test sull'agglomerazione

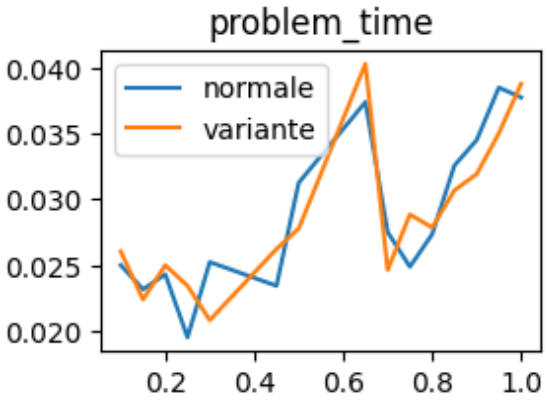

Figura 5.20: Tempo generazione problema

Oltre a quello possiamo anche vedere come a crescere sia anche la percentuale di nodi visitati5.21, probabilmente al fatto che è più probabile che all'agente sia richiesto di "girare attorno agli ostacoli", non potendo prendere magari una strada più diretta.

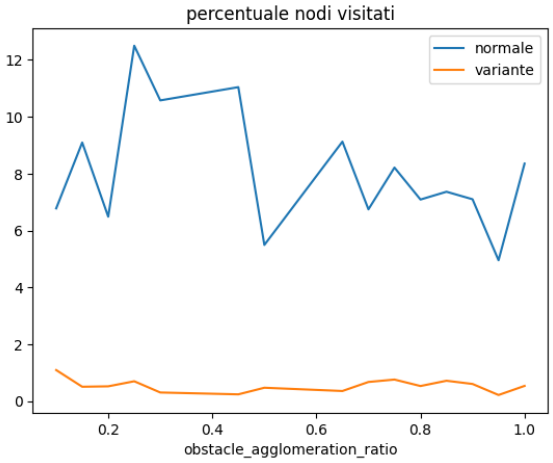

Figura 5.21: percentuale nodi visitati

Capitolo 6

Conclusioni

Come ipotizzato, i risultati sperimentali mostrano che la variante di ReachGoal converge più velocemente alla soluzione rispetto a ReachGoal standard. Il compromesso è che necessita di un tempo aggiuntivo per pre-calcolare e memorizzare i valori dell'euristica. La versione standard, invece, non richiede tempo o spazio aggiuntivi per l'euristica, ma di contro impiega più tempo per la ricerca della soluzione, esplorando una porzione più ampia della griglia e di conseguenza un aumentando la dimensione del closed set. In conclusione, possiamo affermare che la versione ottimizzata è una scelta preferibile in buona parte dei casi considerati.