PROGETTO CARBUROBOT

Consegna

CarbuRobot. Un robot si può muovere (move) in una sequenza di stanze, in numero di stanze qualsiasi. Nelle stanze possono trovarsi oggetti che possono essere afferrati (get) o rilasciati a terra (put) dal robot. Il robot puo' afferrare e tenere un solo un oggetto alla volta. Gli oggetti hanno un peso espresso in unità intere nonnegative. Quando il robot si muove senza oggetti consuma per ogni move 1 litro di carburante, quando si muove con un oggetto consuma per la move tanti litri di carburante quanto il peso dell'oggetto più 1. Quando afferra o rilascia l'oggetto get/put consuma tanti litri quanto il peso dell'oggetto manipolato. Es. se afferra un oggetto di 5kg consuma 5 litri. Nelle stanze possono trovarsi dei distributori di carburante, se il robot si trova in una stanza con ristributore può esreguire l'azione rifornisci (refill) che ricarica, incrementando di una quantità fissa es.+10, il carburante contenuto nel serbatotio del robot. Naturalmente se non ha sufficiente carburante non può eseguire una certa azione. Esempio Problema : dato uno stato iniziale di carburante nel robot e oggetti di pesi diversi sparsi in giro e distributori si richiede uno stato finale con posizioni finali di oggetti.

Implementazione

```
class robotLecture(Problem):
    def __init__(self, initial, goal):
        self.goal = goal
        Problem.__init__(self, initial, goal)
        self.corridor_size = 5

        self.objects = {
            "shoes": {"weight": 1},
            "desk": {"weight": 1},
            "candle": {"weight": 1},
            "bag": {"weight": 2},
            "station": {"weight": 0}
        }
```

La prima parte di codice, definisce la classe robotLecture, che è una sottoclasse della classe Problem. La classe Problem è stata definita precedentemente e contiene attributi e metodi che verranno ereditati dalla classe robotLecture.
Verrà ora fatta una panoramica su ciascun componente all'interno del costruttore della classe robotLecture:
self.goal = goal: questa riga assegna il valore del parametro goal alla variabile di istanza self.goal. Il parametro goal rappresenta lo stato finale che si desidera raggiungere nel problema. Questo stato finale è fornito come argomento quando si crea un'istanza della classe robotLecture.
Problem.__init__(self, initial, goal): Questa riga chiama il costruttore della classe genitore Problem, passando l'istanza corrente (self), lo stato iniziale (initial), e lo stato obiettivo (goal) come parametri. In questo modo, la classe robotLecture eredita i metodi e gli attributi dalla classe Problem e li espande per adattarli al problema specifico, con nuovi metodi e/o attributi.
self.corridor_size = 5: Questa riga imposta il valore di self.corridor_size a 5. Questa variabile rappresenta la dimensione del corridoio in cui il robot si muove; la dimensione del corridoio influenzerà il movimento del robot all'interno dell'ambiente e i due estremi verranno utilizzati come riferimenti negli spostamenti.
self.objects = { <lista degli oggetti> }: questa parte del codice definisce un dizionario denominato objects; esso contiene informazioni sugli oggetti presenti nell'ambiente del robot. Ciascuna chiave nel dizionario rappresenta il nome di un oggetto (ad esempio, "shoes") e il valore associato a ciascuna chiave è un altro

dizionario con una singola chiave "weight" che rappresenta il peso dell'oggetto. Il peso degli oggetti è importante poiché influisce sulle azioni che il robot può eseguire, come la get/put e il trasporto di oggetti.

```
def actions(self, state):
        possible_actions = []

        current_room = state["rooms"][state["robot"]]
        # Azione get
        if state["hand"] is None and current_room != "null":
           for object_name, object_info in self.objects.items():
               object_weight = object_info["weight"]
               if object_name == current_room and object_weight <=
state["fuel"]:
                   possible_actions.append("get_" + current_room)

        # Azione put
        if state["hand"] is not None and current_room == "null":
           object_name_in_hand = state["hand"]
           object_weight_in_hand =
self.objects[object_name_in_hand]["weight"]
           if object_weight_in_hand <= state["fuel"]:
              possible_actions.append("put_" + object_name_in_hand)

        # Azione refill
        if "station" in current_room and state["hand"] is None:
            possible_actions.append("refill")

        # Azione per spostarsi a sinistra
        if state["robot"] > 0 and state["fuel"] >= 1:
           if state["hand"] is None or (state["hand"] in self.objects and
self.objects[state["hand"]]["weight"] + 1 <= state["fuel"]):
              possible_actions.append("left")

        # Azione per spostarsi a destra
        if state["robot"] < self.corridor_size - 1 and state["fuel"] >= 1:
           if state["hand"] is None or (state["hand"] in self.objects and
self.objects[state["hand"]]["weight"] + 1 <= state["fuel"]):
              possible_actions.append("right")

        return possible_actions
```

In questa parte di codice viene definito il metodo actions(self, state). Questo metodo genera e restituisce la lista delle azioni possibili che il robot può eseguire a partire dallo stato corrente. Ogni azione rappresenta un possibile movimento o interazione che il robot può compiere nell'ambiente, portando conseguenze come vedremo poi nei nuovi stati.
In questo metodo, quindi, vengono specificate le azioni possibili e le relative condizioni:
Azione "get" (Raccogliere oggetto):
consente al robot di raccogliere un oggetto dalla stanza corrente e tenerlo nella “hand”.
Per poter eseguire questa azione, il robot deve avere le seguenti condizioni:

Non deve avere alcun oggetto in mano (la variabile hand nel suo stato deve essere None).
La stanza corrente (current_room) non deve essere "null" (cioè deve contenere un oggetto).
L'oggetto presente nella stanza corrente deve avere un peso (object_weight) che sia inferiore o uguale alla quantità di carburante disponibile (state["fuel"]).
Se queste condizioni sono soddisfatte, viene aggiunta un'azione "get_" (seguita dal nome della stanza corrente "current_room") alla lista delle possibili azioni. Questa azione rappresenta il tentativo di raccogliere l'oggetto presente nella stanza corrente.
Azione "put" (Posare oggetto):
Questa azione consente al robot di rilasciare un oggetto che tiene in mano nella stanza corrente.
Per poter eseguire questa azione, il robot deve avere le seguenti condizioni:
deve tenere un oggetto in mano (la variabile hand nel suo stato deve contenere il nome dell'oggetto).
la stanza corrente (current_room) deve essere "null" (vuota).
Il peso dell'oggetto tenuto in mano (object_weight_in_hand) deve essere inferiore o uguale alla quantità di carburante disponibile (state["fuel"]).
Se queste condizioni sono soddisfatte, viene aggiunta un'azione "put_" seguita dal nome dell'oggetto tenuto in mano (object_name_in_hand) alla lista delle possibili azioni. Questa azione rappresenta il tentativo di depositare l'oggetto nella stanza corrente.
Azione "refill" (Rifornimento di carburante):
Questa azione consente al robot di rifornire il suo carburante se c'è una stazione nella stanza corrente.
Per poter eseguire questa azione, il robot deve avere le seguenti condizioni:
non deve avere alcun oggetto in mano (la variabile hand nel suo stato deve essere None) e
deve esserci una stazione nella stanza corrente ("station" in current_room).
Se queste condizioni sono soddisfatte, viene aggiunta l'azione "refill" alla lista delle possibili azioni, la quale rappresenta la possibilità di rifornimento di carburante del robot nella stanza corrente.
Azione "left" (Spostamento a sinistra):
Questa azione consente al robot di spostarsi a sinistra nel corridoio.
Per poter eseguire questa azione, il robot deve avere le seguenti condizioni:
la posizione attuale del robot (state["robot"]) deve essere maggiore di 0 (non può essere all'estremo sinistro del corridoio).
Inoltre, deve avere almeno 1 unità di carburante disponibile (state["fuel"] >= 1).
Se il robot tiene un oggetto in mano, il suo peso più 1 (object_weight + 1) deve essere inferiore o uguale alla quantità di carburante disponibile.
Se queste condizioni sono soddisfatte, viene aggiunta l'azione "left" alla lista delle possibili azioni.
Azione "right" (Spostamento a destra):
Questa azione consente al robot di spostarsi a destra nel corridoio.
Le condizioni necessarie per eseguire questa azione sono simili a quelle dell'azione "left", ma in questo caso il robot si sposta verso destra se possibile, cioè se non si trova nell'ultima stanza a destra ma in una delle altre (self.corridor_size – 1).
Le azioni vengono aggiunte a una lista denominata possible_actions, che verrà quindi restituita come risultato del metodo. Questa lista rappresenta tutte le azioni possibili a partire dallo stato corrente del robot.

```
def result(self, state, action):
        new_state = copy.deepcopy(state)

        action = action.split("_")

        if action[0] == "left":
          new_state["robot"] -= 1
          new_state["fuel"] -= 1
          if state["hand"] is not None:
            object_weight = self.objects[state["hand"]]["weight"]+1
            new_state["fuel"] -= object_weight
```

```
        if action[0] == "right":
          new_state["robot"] += 1
          new_state["fuel"] -= 1
          if state["hand"] is not None:
            object_weight = self.objects[state["hand"]]["weight"]+1
            new_state["fuel"] -= object_weight
        if action[0] == "get":
          object_name = action[1]
          object_weight = self.objects[object_name]["weight"]
          if state["fuel"] >= object_weight:
             new_state["hand"] = object_name
             new_state["rooms"][state["robot"]] = "null"
             new_state["fuel"] -= object_weight
        if action[0] == "put":
          object_name = action[1]
          object_weight = self.objects[object_name]["weight"]
          if state["fuel"] >= object_weight:
            new_state["hand"] = None
            new_state["rooms"][state["robot"]] = object_name
            new_state["fuel"] -= object_weight
        if action[0] == "refill":
          new_state["fuel"] += 10

        return new_state
```

Questo codice implementa il metodo result (self, state, action) che calcola e restituisce il nuovo stato del robot dopo aver eseguito un'azione specifica.
La funzione riceve due argomenti:
state: lo stato corrente del robot.
action: l'azione da eseguire.
Lo schema di questo metodo può essere sintetizzato come (state,action)->(new state).
Le azioni, portano a nuovi stati con le seguenti caratteristiche:
Azione "left" (Spostamento a sinistra):
Se l'azione è "left", il robot viene spostato di una posizione a sinistra nel corridoio (new_state["robot"] -= 1) e la sua quantità di carburante viene ridotta di 1 (new_state["fuel"] -= 1).
Inoltre, se il robot tiene un oggetto in mano (state["hand"] is not None), il suo peso più 1 (object_weight + 1) viene sottratto dalla quantità di carburante del nuovo stato.
Azione "right" (Spostamento a destra):
Se l'azione è "right", il robot viene spostato di una posizione a destra nel corridoio (new_state["robot"] += 1) e la sua quantità di carburante viene ridotta di 1 (new_state["fuel"] -= 1).
Anche in questo caso, se il robot tiene un oggetto in mano (state["hand"] is not None), il suo peso più 1 (object_weight + 1) viene sottratto dalla quantità di carburante del nuovo stato.
Azione "get" (Raccogliere oggetto):
Se l'azione inizia con "get", viene estratto il nome dell'oggetto da raccogliere (object_name) dall'azione.
Il peso dell'oggetto selezionato (object_weight) viene utilizzato per verificare se il robot ha abbastanza carburante per raccoglierlo. Se la quantità di carburante nel suo stato è sufficiente (state["fuel"] >= object_weight), il robot può raccogliere l'oggetto. In questo caso, il nome dell'oggetto viene assegnato alla variabile hand del nuovo stato, e l'oggetto viene rimosso dalla stanza corrente e sostituito con "null". La quantità di carburante viene ovviamente aggiornata.
Azione "put" (Depositare oggetto):

Se l'azione inizia con "put", viene estratto il nome dell'oggetto da posare (object_name) dall'azione, allo stesso modo di sopra.
Il peso dell'oggetto da posare (object_weight) viene utilizzato per verificare se il robot ha abbastanza carburante per posare l'oggetto. Se la quantità di carburante nel suo stato è sufficiente (state["fuel"] >= object_weight), il robot può posare l'oggetto. In questo caso, la variabile hand nel nuovo stato viene impostata su None, e l'oggetto viene aggiunto alla stanza corrente al posto di "null". La quantità di carburante viene aggiornata di conseguenza.
Azione "refill" (Rifornimento di carburante):
Se l'azione è "refill", il robot si trova in una stanza con una stazione di rifornimento. La quantità di carburante del nuovo stato viene incrementata di 10 unità (new_state["fuel"] += 10).
Infine, il metodo restituisce il nuovo stato del robot (new_state), che rappresenta lo stato risultante dopo l'esecuzione dell'azione specificata.

```
def goal_test(self, state):
        return (
            state["robot"] == self.goal["robot"] and
            state["hand"] == self.goal["hand"] and
            state["fuel"] >= self.goal["fuel"] and
            state["rooms"] == self.goal["rooms"]
        )
```

Questa parte di codice definisce il metodo goal_test(self, state). Questo metodo verifica se lo stato corrente del robot corrisponde al goal state specificato.
Il metodo restituisce True se le seguenti condizioni sono verificate:
la posizione attuale del robot (state["robot"]) è uguale alla posizione del robot nel goal state (self.goal["robot"]);
l'oggetto attualmente in mano al robot (state["hand"]) è uguale all'oggetto in mano nel goal state (self.goal["hand"]);
la quantità di carburante disponibile al robot (state["fuel"]) è maggiore o uguale alla quantità di carburante richiesta nel goal state (self.goal["fuel"]);
la lista delle stanze con i loro oggetti (state["rooms"]) è uguale alla lista delle stanze con i loro oggetti nel goal state (self.goal["rooms"]).
In sostanza, il metodo verifica se il robot si trova nella posizione desiderata, tiene in hand l'oggetto desiderato, ha abbastanza carburante e le stanze con i loro oggetti corrispondono esattamente al goal state. Se tutte queste condizioni sono verificate, il metodo restituisce True, indicando che il robot ha raggiunto lo stato di goal. Altrimenti, restituirà False, indicando che il robot non ha ancora raggiunto lo stato desiderato. Questo metodo, costituisce il passo fondamentale per capire se effettivamente la soluzione è stata trovata o c'è ancora bisogno di andare avanti; influenza, quindi, anche il costo della ricerca.

```
def h1(self, node):
        current_state = node.state
        goal_state = self.goal

        misplaced_objects = 0
        for current_room, goal_room in zip(current_state["rooms"],
goal_state["rooms"]):
            if current_room != "null" and set(current_room) !=
set(goal_room):
                misplaced_objects += 1
```

```
        return misplaced_objects

    def h2(self, node):
        current_state = node.state
        goal_state = self.goal

        misplaced_rooms = sum(1 for current_room, goal_room in
zip(current_state["rooms"], goal_state["rooms"]) if set(current_room) !=
set(goal_room))

        return misplaced_rooms
```

h1 (Euristica del numero di oggetti fuori posto):
Descrizione: Questa euristica valuta quanti oggetti sono posizionati in modo errato rispetto allo stato obiettivo. Conta il numero di oggetti presenti nelle stanze nello stato corrente che non coincidono con gli oggetti nelle stanze dello stato obiettivo.
Questa euristica, quindi, permette di stimare quanti oggetti devono essere spostati o scambiati di posizione per far coincidere il tuo stato corrente con lo stato obiettivo.
h2 (Euristica del numero di stanze fuori posto):
Descrizione: Questa euristica valuta quanti spazi o stanze sono posizionati in modo errato rispetto allo stato obiettivo. Conta il numero di stanze che contengono oggetti diversi rispetto alle stanze dello stato obiettivo.
L'utilità è quella di riuscire a stimare quanti spazi o stanze devono essere "riordinate" o "ripiene" per raggiungere lo stato obiettivo.

```
# actions()

actions = problem.actions(initial_state)

# Stampare le azioni disponibili
print("Possible actions for initial state:")
for action in actions:
    print(action)

# result()

action = "right"

print("Apply "+action+" ...")
# Calcola il nuovo stato
new_state = problem.result(initial_state, action)

# Stampa il nuovo stato
```

```
print("New state after action: "+action , new_state)
```

Questa parte di codice è stata utilizzata come test per i metodi result() e action(), al fine di andare a verificare la bontà della loro implementazione, monitorando la gestione dei costi delle azioni e la loro effettiva riuscita.

```
import time

# iterative deepening search
start_time = time.time()
s3 = iterative_deepening_search(problem, display=True)
end_time = time.time()
print("Iterative Deepening Search Solution:", s3.solution())
print("Iterative Deepening Search Time:", end_time - start_time,
"seconds\n\n")

# breadth-first tree search
start_time = time.time()
s3 = breadth_first_tree_search(problem, display=True)
end_time = time.time()
print("Breadth-First Tree Search Solution:", s3.solution())
print("Breadth-First Tree Search Time:", end_time - start_time,
"seconds\n\n")

# uniform cost search
start_time = time.time()
sol2 = uniform_cost_search(problem, display=True)
end_time = time.time()
print("Uniform Cost Search Solution:", sol2.solution())
print("Uniform Cost Search Time:", end_time - start_time, "seconds\n\n")

# heuristic search with h1
start_time = time.time()
s3 = astar_search(problem, problem.h1, display=True)
end_time = time.time()
print("Heuristic Search with h1 Solution:", s3.solution())
print("Heuristic Search with h1 Time:", end_time - start_time,
"seconds\n\n")

# heuristic search with h2
start_time = time.time()
s3 = astar_search(problem, problem.h2, display=True)
end_time = time.time()
print("Heuristic Search with h2 Solution:", s3.solution())
print("Heuristic Search with h2 Time:", end_time - start_time,
"seconds\n\n")
```

Iterative Deepening Search (IDS):

è un algoritmo di ricerca iterativa che combina le caratteristiche di una ricerca in profondità con la capacità di trovare soluzioni ottimali.
IDS è chiamato con l'oggetto problem come argomento, che rappresenta il problema specifico da risolvere.
display=True indica che l'algoritmo dovrebbe mostrare l'output durante l'esecuzione.
s3 conterrà le informazioni sulla soluzione trovata.
s3.solution() restituirà la sequenza di azioni che costituiscono la soluzione.
s3.state conterrà lo stato finale raggiunto dalla ricerca.

Breadth-First Tree Search:

è una strategia di ricerca in ampiezza che esplora tutti i successori di un nodo prima di passare ai successori dei suoi successori.
BFS è chiamato con l'oggetto problem come argomento, che rappresenta il problema specifico da risolvere.
display=True indica che l'algoritmo dovrebbe mostrare l'output durante l'esecuzione.
s3 conterrà le informazioni sulla soluzione trovata.
s3.solution() restituirà la sequenza di azioni che costituiscono la soluzione.
s3.state conterrà lo stato finale raggiunto dalla ricerca.

Uniform Cost Search:

è un algoritmo basato su costo che esplora lo spazio degli stati minimizzando il costo totale.
Uniform_cost_search è chiamato con l'oggetto problem come argomento, che rappresenta il problema specifico da risolvere.
display=True indica che l'algoritmo dovrebbe mostrare l'output durante l'esecuzione.
sol2 conterrà le informazioni sulla soluzione trovata.
sol2.solution() restituirà la sequenza di azioni che costituiscono la soluzione.
sol2.state conterrà lo stato finale raggiunto dalla ricerca.

A Search con Euristiche:

è un algoritmo basato su euristiche che cerca di minimizzare il costo totale stimato.
Astar_search è chiamato con l'oggetto problem come argomento, che rappresenta il problema specifico da risolvere, e una specifica euristica (problem.h1 o problem.h2) per guidare la ricerca.
display=True indica che l'algoritmo dovrebbe mostrare l'output durante l'esecuzione.
s3 conterrà le informazioni sulla soluzione trovata.
s3.solution() restituirà la sequenza di azioni che costituiscono la soluzione.
s3.state conterrà lo stato finale raggiunto dalla ricerca.
Inoltre, il codice utilizza il modulo time per calcolare il tempo di esecuzione di ciascun algoritmo di ricerca.
La logica del tempo di esecuzione è semplice: prima dell'avvio della ricerca viene dato lo start() al contatore del tempo, poi viene eseguito l'algoritmo di ricerca e all' uscita da questo (soluzione trovata) viene stoppato il contatore (stop()).
In questo caso sono stati presentati gli algoritmi utilizzati nel primo problema, ma anche negli altri due si sono valutate le soluzioni con gli stessi, ma cambiando solo il problema passato sulla base dei diversi initial/goal state definiti (spiegati successivamente…)

Problemi specifici e valutazione risultati

Dopo aver descritto le parti implementate, veniamo ora alla discussione dei risultati e dei 3 diversi problemi implementati, in ordine di difficoltà, oltre che del segmento mdi codice utilizzato per verificare la correttezza dei metodi actions() e result().

Controllo implementazioni actions() e result()

```
get_station
refill
right
```

```
Apply right ...
New state after action: right {'robot': 1, 'hand': None, 'fuel': 14,
'rooms': ['station', 'candle', 'null', 'null', 'null']}
```

Come spiegato nella parte precedente, è stato utilizzato del codice di prova, per valutare l'implementazione delle due funzioni principali.
In questo caso, i risultati evidenziati in giallo sono le azioni possibili partendo dallo stato iniziale (tramite la funzione actions, output di "actions = problem.actions(initial_state)".
Nella parte evidenziata di rosso, invece, è stata applicata manualmente una delle possibili azioni (in questo caso right) allo stato iniziale, permettendo di vedere e valutare il new state nel quale si arriva.
Come si può vedere, il robot è passato dalla posizione 0 alla posizione 1, ha ancora la hand vuota (nessuna get fatta) e ha consumato 1 unità di fuel (giusto in quanto il robot si è spostato con la hand vuota).
In questo caso, la buona riuscita dei due metodi è stata verificata.

Problema 1

```
problem = robotLecture(initial_state, goal_state)
```

```
initial_state = {
    "robot": 0,
    "hand": None,
    "fuel": 15,
    "rooms": ['station', 'candle', 'null', 'null','null']
}

goal_state = {
    "robot": 2,
    "hand": None,
    "fuel": 5,
    "rooms": ['station', 'null', 'candle', 'null', 'null']
}
```

Questo primo problema, consiste nel passare dallo stato iniziale allo stato finale spostando l'oggetto candle da una stanza alla successiva.
Considerando gli indici, in questo caso lo spostamento sarà dalla room n.1 alla room n.2 (numerazione stanze da 0 a 4).

Soluzioni e risultati

Applicando gli algoritmi visti in precedenza, quelle di seguito sono le soluzioni ottenute.
Si può concludere che la ricerca euristica con la funzione h2, è la migliore in quanto fornisce una soluzione esaminando 32 nodi e espandendo 15 cammini, con anche il tempo di esecuzione minore tra gli algoritmi proposti.
In generale, la funzione euristica h2, guida la ricerca migliorando le prestazioni, eliminando nodi superflui o cammini con poca probabilità di portare alla soluzione.
In conclusione, la ricerca euristica con la funzione h2 è da preferirsi alla ricerca iterativa in questo problema specifico.
Al contrario, in questo caso, la soluzione con costo più elevato è fornita dall' Uniform Cost Search.

```
Examined  64 Nodes
```

```
Iterative Deepening Search Solution: ['right', 'get_candle', 'right',
'put_candle']
Iterative Deepening Search Time: 0.002848386764526367 seconds


Examined 72 nodes
Breadth-First Tree Search Solution: ['right', 'get_candle', 'right',
'put_candle']
Breadth-First Tree Search Time: 0.006258726119995117 seconds


Examined 109 nodes
43 paths have been expanded and 40 paths remain in the frontier
Uniform Cost Search Solution: ['right', 'get_candle', 'right', 'put_candle']
Uniform Cost Search Time: 0.009505748748779297 seconds


Examined 74 nodes
29 paths have been expanded and 31 paths remain in the frontier
Heuristic Search with h1 Solution: ['right', 'get_candle', 'right',
'put_candle']
Heuristic Search with h1 Time: 0.00725555419921875 seconds


Examined 32 nodes
12 paths have been expanded and 15 paths remain in the frontier
Heuristic Search with h2 Solution: ['right', 'get_candle', 'right',
'put_candle']
Heuristic Search with h2 Time: 0.001172780990600586 seconds → BEST SOLUTION
```

Problema 2

```
problem1 = robotLecture(initial_state1, goal_state1)
```

```
initial_state1 = {
    "robot": 0,
    "hand": None,
    "fuel": 15,
    "rooms": ['station', 'candle', 'bag', 'null','null']
}

goal_state1 = {
    "robot": 3,
    "hand": None,
    "fuel": 5,
    "rooms": ['station', 'null', 'candle', 'bag', 'null']
}
```

Questo secondo problema, invece, consiste nel passare dallo stato iniziale allo stato finale spostando 2 oggetti: “candle” dalla room n.1 alla room n.2 e “bag” dalla room n.2 alla room n.3.
La soluzione, inoltre, richiede di passare dalla room n.0 alla room n.3.

Il problema ha una difficoltà superiore rispetto al precedente, in quanto richiede lo spostamento di 2 oggetti e una necessaria azione di refill, in quanto il carburante iniziale non è sufficiente per completare tutte le azioni.
Considerando gli indici, in questo caso lo spostamento sarà dalla room n.1 alla room n.2 (numerazione stanze da 0 a 4).

Soluzioni e risultati

Applicando gli stessi algoritmi visti in precedenza, notiamo che la ricerca euristica con la funzione h2, è la migliore anche in questo caso, sia come tempi che come nodi visitati.
In questo caso, fornisce una soluzione esaminando 1395 nodi e espandendo 270 cammini, con tempo di esecuzione di circa 0.15 secondi.
In generale, la funzione euristica h2, guida la ricerca anche per questo tipo di problema, migliorando le prestazioni, eliminando nodi superflui o cammini con poca probabilità di portare alla soluzione.
In conclusione, la ricerca euristica con la funzione h2 è da preferirsi alla ricerca iterativa e alle altre proposte anche in questo problema specifico.
Al contrario, in questo caso, la soluzione con costo più elevato è fornita dall' BFS.

```
Examined  97306 Nodes
Iterative Deepening Search Solution: ['refill', 'right', 'right', 'get_bag',
'right', 'put_bag', 'left', 'left', 'get_candle', 'right', 'put_candle',
'right']
Iterative Deepening Search Time: 2.552614212036133 seconds
```

```
Examined 118239 nodes
Breadth-First Tree Search Solution: ['refill', 'right', 'right', 'get_bag',
'right', 'put_bag', 'left', 'left', 'get_candle', 'right', 'put_candle',
'right']
Breadth-First Tree Search Time: 5.324701547622681 seconds
```

```
Examined 3307 nodes
1462 paths have been expanded and 498 paths remain in the frontier
Uniform Cost Search Solution: ['refill', 'right', 'right', 'get_bag',
'right', 'put_bag', 'left', 'left', 'get_candle', 'right', 'put_candle',
'right']
Uniform Cost Search Time: 0.5799834728240967 seconds
```

```
Examined 2033 nodes
899 paths have been expanded and 363 paths remain in the frontier
Heuristic Search with h1 Solution: ['refill', 'right', 'right', 'get_bag',
'right', 'put_bag', 'left', 'left', 'get_candle', 'right', 'put_candle',
'right']
Heuristic Search with h1 Time: 0.2741544246673584 seconds
```

```
Examined 1395 nodes
612 paths have been expanded and 270 paths remain in the frontier
Heuristic Search with h2 Solution: ['refill', 'right', 'right', 'get_bag',
'right', 'put_bag', 'left', 'left', 'get_candle', 'right', 'put_candle',
'right']
Heuristic Search with h2 Time: 0.1483011245727539 seconds → BEST SOLUTION
```

Problema 3

```
problem2 = robotLecture(initial_state2, goal_state2)
```

```
# Esempio di utilizzo 3
initial_state2 = {
    "robot": 0,
    "hand": "bag",
    "fuel": 5,
    "rooms": ['null', 'station', 'desk', 'candle','null']
}

goal_state2 = {
    "robot": 4,
    "hand": "desk",
    "fuel": 5,
    "rooms": ['bag', 'station', 'null', 'null', 'candle']
}
```

In questo ultimo problema, consideriamo un caso più complesso:
la room di arrivo sarà la n.5 e il robot avrà in mano nello stato di partenza un oggetto “bag”; nello stato finale, invece, il robot avrà come vincolo quello di avere nella hand un oggetto “desk”(non più “bag”), che invece verrà rilasciato nella stanza 0 e insieme a tutti gli altri spostamenti di oggetti tra stanze contribuirà a raggiungere la disposizione definita nel goal state.
Inoltre, sarà necessario effettuare azioni di refill durante il passaggio da initial state a goal state.

Soluzioni e risultati

Applicando ancora gli algoritmi visti in precedenza, notiamo che la ricerca euristica con la funzione h2, è ancora la migliore soluzione, sia come tempi che come nodi visitati.
In questo caso, fornisce una soluzione esaminando 1146 nodi e espandendo 213 cammini, con tempo di esecuzione di circa 0.10 secondi.
In generale, la funzione euristica h2, guida la ricerca anche per questo tipo di problema, migliorando ancora le prestazioni, eliminando nodi superflui o cammini con poca probabilità di portare alla soluzione.
In conclusione, la ricerca euristica con la funzione h2 è da preferirsi alla ricerca iterativa e alle altre proposte anche in questo problema specifico.
In questo caso, ancor più rispetto alle altre casistiche, si nota la differenza di nodi toccati tra la ricerca iterativa e quella euristica.
Al contrario, in questo caso, la soluzione con costo più elevato è fornita dall’ BFS, che si dimostra quindi sempre la più costosa per problemi complessi.

```
Examined  120228 Nodes
Iterative Deepening Search Solution: ['put_bag', 'right', 'refill',
'refill', 'right', 'right', 'get_candle', 'right', 'put_candle', 'left',
'left', 'get_desk', 'right', 'right']
Iterative Deepening Search Time: 4.5215277671813965 seconds
```

```
Examined 160326 nodes
Breadth-First Tree Search Solution: ['put_bag', 'right', 'refill', 'refill',
'right', 'right', 'get_candle', 'right', 'put_candle', 'left', 'left',
'get_desk', 'right', 'right']
Breadth-First Tree Search Time: 7.659287929534912 seconds


Examined 2350 nodes
1069 paths have been expanded and 313 paths remain in the frontier
Uniform Cost Search Solution: ['put_bag', 'right', 'refill', 'refill',
'right', 'right', 'get_candle', 'right', 'put_candle', 'left', 'left',
'get_desk', 'right', 'right']
Uniform Cost Search Time: 0.28685855865478516 seconds


Examined 1547 nodes
715 paths have been expanded and 225 paths remain in the frontier
Heuristic Search with h1 Solution: ['put_bag', 'right', 'refill', 'refill',
'right', 'right', 'get_candle', 'right', 'put_candle', 'left', 'left',
'get_desk', 'right', 'right']
Heuristic Search with h1 Time: 0.15833115577697754 seconds


Examined 1146 nodes
518 paths have been expanded and 213 paths remain in the frontier
Heuristic Search with h2 Solution: ['put_bag', 'right', 'refill', 'refill',
'right', 'right', 'get_candle', 'right', 'put_candle', 'left', 'left',
'get_desk', 'right', 'right']
Heuristic Search with h2 Time: 0.10745000839233398 seconds → BEST SOLUTION


I


                                                    Esonero
                                                    Marco Amici
```