

Introduzione all'Intelligenza Artificiale

Supporto per le Esercitazioni

Dipartimento di Informatica – Università di Pisa

Corso di Laurea in Informatica

Anno Accademico 2016 – 2017

e successivi

Credits:

Claudio Gallicchio, Ph.D.
Computational Intelligence and Machine Learning Group
Dipartimento di Informatica - Università di Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy

web: www.di.unipi.it/~gallicch

email: gallicch@di.unipi.it

Docenti del corso
Prof. Alessio Micheli
Prof. Maria Simi

Pagina web del corso:
Moodle anno del corso

Table of Contents

Indice delle Tabelle.....	4
Informazioni su questo documento	5
1. Note sull'Installazione e sull'Utilizzo di Python.....	6
Installazione di Python	6
Eseguire un Programma in Python	6
Ambienti di Sviluppo Integrato.....	6
Ulteriori Risorse	7
2. Strutture di Dati per Algoritmi di Ricerca	8
Rappresentazione di un Problema: la classe Problem	8
Rappresentazione di un problema specifico	9
Rappresentazione di un nodo nell'albero di ricerca: la classe Node.....	10
Rappresentazione della frontiera mediante una coda: la classe Queue.....	12
Coda FIFO.....	13
Coda LIFO.....	13
Coda con Priorità	13
3. Algoritmi di Ricerca.....	15
Breath-first Search: Ricerca-grafo in ampiezza	15
Uniform-cost Search: Ricerca-grafo a costo uniforme	15
Ricerca A*	16
Depth-first Search: Ricerca in profondità.....	16
Depth-first Search Tree: Ricerca-albero in profondità	16
Depth-first Search Graph: Ricerca-grafo in profondità	17
Recursive Depth-first Search: Ricerca ricorsiva in profondità.....	17
Limited Depth-first Search Tree: Ricerca-albero a profondità limitata.....	18
Limited Recursive Depth-first Search: Ricerca ricorsiva a profondità limitata.....	18
Algoritmi di Ricerca Locale	19
Ricerca Hill-climbing	19
Ricerca Simulated annealing	19
4. Ricerca con Avversari.....	21
Rappresentazione di un Gioco: la classe Game.....	21
L'algoritmo minimax.....	23
L'algoritmo di Potatura Alfa-Beta.....	23

Indice delle Tabelle

TABELLA 1 LA CLASSE PROBLEM.	9
TABELLA 2 LA CLASSE TOYPROBLEM1.	10
TABELLA 3 LA CLASSE NODE.	11
TABELLA 4 LA CLASSE QUEUE.	12
TABELLA 5 LA CLASSE FIFOQUEUE.	13
TABELLA 6 LA CLASSE LIFOQUEUE.	13
TABELLA 7 LA CLASSE PRIORITYQUEUE.	14
TABELLA 8 RICERCA-GRAFO IN AMPIEZZA.	15
TABELLA 9 RICERCA-GRAFO A COSTO UNIFORME.	16
TABELLA 10 RICERCA A*	16
TABELLA 11 RICERCA-ALBERO IN PROFONDITÀ.	17
TABELLA 12 RICERCA-GRAFO IN PROFONDITÀ.	17
TABELLA 13 RICERCA RICORSIVA IN PROFONDITÀ.	17
TABELLA 14 RICERCA-ALBERO A PROFONDITÀ LIMITATA.	18
TABELLA 15 RICERCA RICORSIVA A PROFONDITÀ LIMITATA.	18
TABELLA 16 HILL-CLIMBING.	19
TABELLA 17 SIMULATED ANNEALING ED ESEMPIO DI FUNZIONE SCHEDULE.	20
TABELLA 18 LA CLASSE GAME.	21
TABELLA 19 LA CLASSE TOYGAME12.	22
TABELLA 20 MINIMAX.	23
TABELLA 21 ALPHA-BETA PRUNING.	24

Informazioni su questo documento

Questo documento contiene una guida all'utilizzo delle funzionalità fornite nell'implementazione in Python degli algoritmi e modelli di supporto ai temi dell'Intelligenza Artificiale e sue applicazioni, nell'ambito del corso "Introduzione all'Intelligenza Artificiale", corso di Laurea in Informatica, anno accademico 2016/2017.

Il codice descritto in questo documento rappresenta un'implementazione delle funzioni riportate nel testo di riferimento del corso

S. Russell, P. Norvig, "Artificial Intelligence: a modern approach", Pearson, Third Edition, 2010.

Ulteriori risorse (incluse implementazione degli algoritmi anche in altri linguaggi) possono essere reperite on-line all'indirizzo:

<https://github.com/aimacode/>

1. Note sull'Installazione e sull'Utilizzo di Python

Di seguito vengono riportate alcune note preliminari sull'installazione e set up di Python.

Installazione di Python

Il sito ufficiale di Python è <https://www.python.org/>, da cui è possibile effettuare il download per Windows, Mac OS ed altri sistemi operativi (incluso Linux, in cui Python è comunque solitamente presente nel setup di base).

Python è attualmente disponibile in due release: Python3 (la versione più recente) e Python2 (ancora disponibile principalmente per motivi retrocompatibilità). Tra queste versioni è generalmente consigliabile l'installazione della versione più recente disponibile. Il codice descritto in questo documento è stato scritto per Python3.

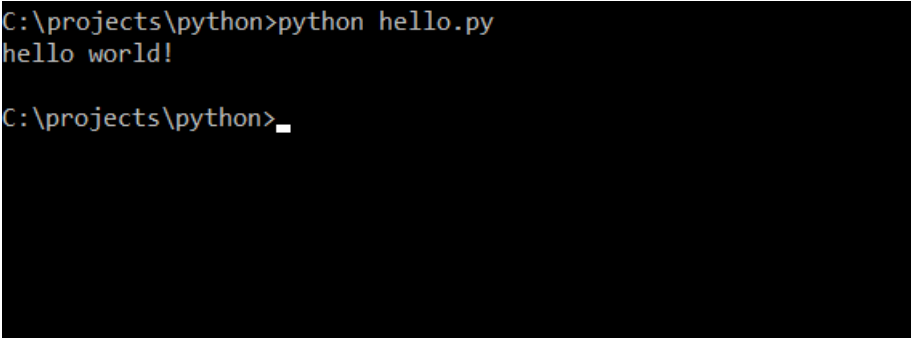
Eseguire un Programma in Python

Per eseguire l'interprete Python:

- Su Windows: dal prompt dei comandi digitare python
- Su Mac/Linux: dalla riga di comando python3

Una volta avviato l'interprete Python è possibile eseguire comandi Python da riga di comando. Per uscire dall'interprete usare i comandi `exit()` oppure `quit()`.

Per eseguire un programma in Python indicare il nome del programma (solitamente con estensione .py) come parametro quando si invoca l'interprete. Ad esempio, su Windows, per eseguire il programma `hello.py` digitare `python hello.py`, come mostrato in Figura 1.



```
C:\projects\python>python hello.py
hello world!

C:\projects\python>
```

Figura 1 Eseguire un programma in Python

Ambienti di Sviluppo Integrato

Un'alternativa all'esecuzione dei programmi Python da riga di comando consiste nell'utilizzo di un Integrated Development Environment (IDE), che fornisce un editor di programmi Python ed è solitamente collegato automaticamente all'interprete.

Esempi di Python IDE disponibili online:

- Thonny (a Python IDE for beginners) <http://thonny.org/>
- Eric Python IDE <http://eric-ide.python-projects.org/>
- PyCharm (software commerciale) <https://www.jetbrains.com/pycharm/>

Ulteriori Risorse

- Website ufficiale di Python (per download, documentazione, ecc.):
<https://www.python.org/>
- Python Succintly, una guida introduttiva a Python:
<https://www.syncfusion.com/resources/techportal/details/ebooks/python>
- Documentazione online di Python
<https://docs.python.org/3/>

2. Strutture di Dati per Algoritmi di Ricerca

Di seguito vengono descritte le strutture di dati utilizzate per rappresentare un problema, la ricerca nello spazio degli stati e le code.

Rappresentazione di un Problema: la classe Problem

La classe Problem implementa le funzioni necessarie per rappresentare un problema:

- **actions:** dato uno stato, restituisce l'insieme delle azioni possibili in quello stato;
- **result:** dato uno stato ed un'azione, restituisce lo stato risultante (quindi implementa la funzione successore, secondo il modello di transizione del problema);
- **goal_test:** dato uno stato restituisce True se lo stato e' uno stato obiettivo, False altrimenti;
- **step_cost:** data una azione, uno stato A ed uno stato B, restituisce il costo dell'esecuzione dell'azione che porta dallo stato A allo stato B.
- **path_cost:** dato il costo del parziale del cammino fino alla penultima azione, un'azione, uno stato A ed uno stato B, restituisce il costo aggiornato dell'intero cammino (assumendo che la funzione costo sia additiva e utilizzando la funzione step_cost)

Il codice della classe Problem e' riportato in Tabella 1, ed e' fornito nel file Problem.py.

```
#class Problem

class Problem:
    """
    Questa classe impementa l'astrazione di un problema.
    Per rappresentare un problema specializzare questa classe implementando (almeno) i metodi
    actions e result
    """

    def __init__(self, initial_state, goal_state = None):
        """ Costruttore. Specifica lo stato iniziale e lo stato (o la lista di stati)
        obiettivo."""
        self.initial_state = initial_state # lo stato iniziale del problema
        self.goal_state = goal_state # lo stato obiettivo, o la lista di stati obiettivo

    def actions(self, state):
        """ Dato lo stato state, restituisce una lista di azioni che possono essere eseguite.
        Questa funzione deve essere implementata nella sottoclasse che specializza Problem."""
        raise NotImplementedError

    def result(self, state, action):
        """ Dato lo stato state e l'azione action, questa funzione restituisce lo stato risultante
        in base al modello di transizione del problema.
        Questa funzione deve essere implementata nella sottoclasse che specializza Problem."""
        raise NotImplementedError

    def goal_test(self, state):
        """ Dato lo stato state, restituisce True se state e' uno stato obiettivo e
        False altrimenti.
        Questa funzione implementa il test obiettivo del problema. """
        # L'implementazione di default per questa funzione restituisce:
        # True: se state e' nella lista degli stati obiettivo
        # False: altrimenti

        if isinstance(self.goal_state, list):
            # in questo caso self.goal_state e' una lista (quindi ci sono piu' stati obiettivo)
            try:
                # cerca state tra gli stati obiettivo
                # un'eccezione e' lanciata nel caso in cui state non
                # sia presente in self.goal_state
                index = self.goal_state.index(state)
                # in questo caso state e' stato trovato tra gli stati obiettivo
                found = True
            except:
                # in questo caso state non e' stato trovato tra gli stati obiettivo
                index = -1
                found = False
```



```

        return found
    else:
        # in questo caso self.goal_state e' un solo stato (quindi c'e' un solo stato
        obiettivo)
        return (state==self.goal_state)

    def step_cost(self, action, stateA = None, stateB = None):
        """ Data l'azione action, lo stato A e lo stato B,
            restituisce il costo dell'esecuzione dell'azione action che porti
            dallo stato A allo stato B."""
        # L'implementazione di default per questa funzione restituisce
        # 1 come costo di ogni azione
        return 1

    def path_cost(self, partial_cost, action, stateA = None, stateB=None):
        """ Dato il costo partial_cost del cammino fino al precedente nodo, l'azione action,
            lo stato A e lo stato B, restituisce il costo aggiornato del cammino. """
        # L'implementazione di default di questa funzione assume che i costi siano additivi
        # e usa la funzione self.step_cost per calcolare il costo dell'ultima azione

        return partial_cost + self.step_cost(action,stateA,stateB)

```

Tabella 1 la classe Problem.

Rappresentazione di un problema specifico

Per rappresentare il problema desiderato, occorre specializzare la classe Problem, implementando almeno le funzioni actions e result. Un semplice esempio pratico, la cui mappa degli stati è raffigurata in Figura 2, è rappresentato dalla classe ToyProblem1, il cui codice è riportato in Tabella 2 e nel file ToyProblems.py.

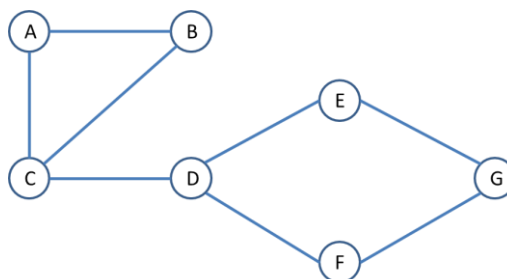


Figura 2 Mappa degli stati relativa al problema nella classe ToyProblem1

```

class ToyProblem1(Problem):
    """ Questa classe specializza la classe Problem
        implementando le funzioni actions e result.
        Gli stati e le azioni possibili sono definiti come stringhe:
        Gli stati possibili sono:
        'A', 'B', 'C', 'D', 'E', 'F', 'G'
        Le azioni possibili sono:
        'A->B', 'A->C', 'B->A', 'B->C', 'C->B', 'C->D', 'D->E', 'D->F',
        'E->D', 'E->G', 'F->D', 'F->G', 'G->E', 'G->F'
        dove e' usato il delimitatore '->' tra lo stato e il suo successore."""

    # Usa il costruttore di default della classe Problem

    def actions(self, state):
        # Per ogni stato restituisce l'insieme delle possibili azioni
        possible_actions = []; #inizializza la lista delle psosibili azioni

        #possibili azioni per ogni stato valido:
        if state == 'A':
            possible_actions = ['A->B', 'A->C']
        elif state == 'B':
            possible_actions = ['B->A', 'B->C']
        elif state == 'C':
            possible_actions = ['C->B', 'C->D']

```

```

        elif state == 'D':
            possible_actions = ['D->E', 'D->F']
        elif state == 'E':
            possible_actions = ['E->D', 'E->G']
        elif state == 'F':
            possible_actions = ['F->D', 'F->G']
        elif state == 'G':
            possible_actions = ['G->E', 'G->F']
        else:
            #default case
            possible_actions = []

        return possible_actions

    def result(self, state, action):
        # Dato lo stato state e l'azione action,
        # restituisce il nuovo stato

        new_state = '' # inizializza il nuovo stato
        # in questo caso il nuovo stato e' semplicemente l'ultimo carattere
        # della stringa che definisce l'azione
        new_state = action[-1:]

        return new_state

```

Tabella 2 La classe `ToyProblem1`.

Ulteriori esempi di problemi, tra cui la mappa della Romania e il labirinto di Teseo, sono presenti all'interno del file `ToyProblems.py`.

Rappresentazione di un nodo nell'albero di ricerca: la classe `Node`

La classe `Node` implementa funzionalità necessarie a rappresentare un nodo di un albero di ricerca.

Per ogni nodo sono definiti i seguenti attributi:

- `state`: lo stato nello spazio degli stati che corrisponde al nodo
- `parent`: il nodo padre
- `action`: l'azione che è stata applicata al nodo padre per generare il nodo
- `path_cost`: il costo del cammino dallo stato iniziale al nodo.

Per ogni nodo sono implementate le seguenti funzioni:

- `child_node`: dato un problema ed un'azione, restituisce il nodo corrispondente allo stato risultante
- `path`: restituisce la lista di nodi nel cammino dalla radice dell'albero fino al nodo
- `solution`: restituisce la soluzione corrispondente al nodo in questione, rappresentata per mezzo di:
 - la sequenza delle azioni richieste per andare dalla radice dell'albero di ricerca fino al nodo,
 - la sequenza degli stati lungo il cammino dalla radice fino al nodo,
 - il costo del cammino dalla radice al nodo,
 - l'insieme degli stati esplorati.

Il codice per la classe `Node` è fornito nella file `Node.py`, ed è riportato in Tabella 3.

```

#class Node

class Node:
    """
    Implementa l'astrazione di un nodo in un albero di ricerca.
    """

    def __init__(self, state, parent = None, action = None, path_cost = 0, depth = 0):
        """Costruttore. Specifica lo stato corrispondente al nodo, il nodo padre,
        l'azione che ha generato il nodo in questione a partire dal nodo padre
        e il costo del cammino dalla radice al nodo.
        I valori di default per parent, action e path cost si riferiscono

```

```

        alla radice dell'albero."""

self.state = state # lo stato associato al nodo
self.parent = parent # il nodo padre (dev'essere un altro oggetto di tipo Node)
self.action = action # l'azione che e' stata eseguita nello stato
                    # self.parent.state per ottenere lo stato self.state
self.path_cost = path_cost # il costo del cammino dalla radice fino a questo nodo
self.depth = depth # la profondita' del nodo

def __repr__(self):
    """ Specifica la stringa da stampare
        per rappresentare il nodo."""

    description = 'state: {} - path cost {}'.format(self.state,self.path_cost);
    return description;

def child_node(self, problem, action):
    """ Restituisce il nodo (nello spazio di ricerca del problema in input)
        ottenuto eseguendo l'azione action quando
        lo stato attuale e' self.state."""

    new_state = problem.result(self.state,action); #il nuovo stato

    # il nuovo nodo
    new_node = Node(new_state,self,action,problem.path_cost(self.path_cost,
        action,self.state),self.depth+1);
    return new_node

def path(self):
    """ Restituisce la lista di nodi nel cammino dalla radice dell'albero fino al nodo
self."""

    # costruisce iterativamente il cammino fino a self
    # aggiungendo in testa ad una lista tutti i nodi che
    # si incontrano seguendo a ritroso i puntatori in parent

    node = self # il nodo di arrivo
    path_back = [] # lista di nodi dalla radice al nodo in questione
    while node is not None: # continua finche' ci sono nodi nel cammino
        path_back.insert(0,node) #inserisce il padre del nodo all'inizio della lista
        node = node.parent
    return path_back

def solution(self, explored_set = None):
    """ Restituisce la soluzione corrispondente al nodo self.
        La soluzione e' rappresentata per mezzo di una lista che contiene
        i seguenti elementi:
        - in posizione 0: la lista delle azioni da eseguire per andare dalla radice
dell'albero
                                di ricerca fino al nodo self
        - in posizione 1: la lista degli stati lungo il cammino dalla radice fino al nodo self
        - in posizione 2: il costo del cammino dalla radice fino al nodo self.
        - in posizione 3: l'insieme degli stati esplorati (con default None per tree
search)."""

    node_path = self.path() # lista di nodi nel cammino dalla radice fino a self
    action_list = [] # lista (ordinata) delle azioni da eseguire per andare dalla radice
                    # fino al nodo self
    state_list = [] # lista (ordinata) degli stati nello spazio di ricerca
                    # dallo stato iniziale fino a self.state
    for node in node_path:
        if node.action is not None: # se non si e' raggiunta la radice
            action_list.append(node.action) # aggiungi l'azione
            # in ogni caso (anche se si e' raggiunta la radice)
            # aggiungi lo stato
            state_list.append(node.state)

    return [action_list, state_list, self.path_cost,explored_set]

```

Tabella 3 La classe Node.

Rappresentazione della frontiera mediante una coda: la classe Queue

La coda è una struttura di dati adatta alla rappresentazione della frontiera di un albero di ricerca. Le code possono essere implementate in Python come liste, sfruttando un'opportuna gestione degli elementi in esse contenute. Le operazioni di base per una coda sono le seguenti:

- `insert`: inserisce un nuovo elemento nella coda
- `pop`: estrae un elemento dalla coda
- `isempty`: restituisce `True` se non ci sono più elementi nella coda.

Oltre a queste operazioni di base è anche utile considerare la funzione `contains` che restituisca `True` se l'elemento specificato come argomento è presente nella coda, e `False` altrimenti.

L'astrazione della coda è implementata nella classe `Queue`, riportata in Tabella 4e nel file `Queue.py`. Le funzioni

```
#class Queue

class Queue:
    """
    Implementa l'astrazione di una coda.
    """

    def __init__(self):
        """ Costruttore. """
        #inizializza la lista degli elementi con una lista vuota
        self.elements = []

    def isempty(self):
        """ Restituisce True se la coda e' vuota,
            False altrimenti. """
        return (len(self.elements)==0)

    def insert(self, element):
        """ Inserisce l'elemento E nella coda. """
        #Questo metodo deve essere necessariamente implementato nella sottoclasse
        raise NotImplementedError

    def pop(self):
        """ Estrae e restituisce un elemento dalla coda. """
        #Questo metodo deve essere necessariamente implementato nella sottoclasse
        raise NotImplementedError

    def __repr__(self):
        """ Specifica la stringa da stampare
            per rappresentare la coda. """
        return 'Gli elementi nella coda sono: ' + str(self.elements);

    def contains(self, element):
        """ Restituisce True se la coda contiene element,
            False altrimenti. """
        return element in self.elements
```

Tabella 4 La classe Queue.

La strategia in base alla quale sono eseguite le operazioni di `insert` e `pop` caratterizza il tipo di coda. In questo senso, le tre varianti principali sono le seguenti:

- First-in first-out (FIFO): ogni operazione di `pop` restituisce l'elemento da più tempo nella coda
- Last-in first-out (LIFO): ogni operazione di `pop` restituisce l'elemento più recentemente inserito nella coda
- Coda con priorità: ogni operazione di `pop` restituisce l'elemento con più alta priorità in base ad una funzione di ordinamento.

Queste tre varianti sono implementate specializzando la classe `Queue`.

Coda FIFO

La classe `FIFOQueue` implementa le funzionalità di una coda FIFO, ed è fornita nel file `Queue.py` e in Tabella 5.

```
#class FIFOQueue - il primo elemento inserito e' il primo ad essere rimosso
class FIFOQueue(Queue):
    """
    Implementa una coda FIFO.
    """
    def insert(self,E):
        """ Inserisce l'elemento E nella coda. """
        # l'elemento e' inserito come primo elemento della coda
        self.elements.insert(0,E)

    def pop(self):
        """ Estrae e restituisce il primo elemento della coda. """
        # estrae e restituisce l'elemento piu' vecchio nella coda
        # (quello corrispondente all'ultimo elemento nella lista)
        return self.elements.pop();
```

Tabella 5 La classe `FIFOQueue`.

Coda LIFO

La classe `LIFOQueue` implementa le funzionalità di una coda LIFO, ed è fornita nel file `Queue.py` e in Tabella 6.

```
#class LIFOQueue - l'ultimo elemento ad essere inserito
#                  e' il primo ad essere rimosso
class LIFOQueue(Queue):
    """
    Implementa una coda LIFO.
    """

    def insert(self,E):
        """ Inserisce l'elemento E nella coda. """
        # ogni nuovo elemento e' inserito alla fine della lista
        self.elements.append(E)

    def pop(self):
        """ Estrae e restituisce il primo elemento della coda. """
        # estrae e restituisce l'elemento piu' recentemente inserito nella coda
        # (quello corrispondente all'ultimo elemento nella lista)

        return self.elements.pop()
```

Tabella 6 La classe `LIFOQueue`.

Coda con Priorità

Una coda con priorità può essere implementata mantenendo gli elementi nella coda ordinati secondo uno specifico ordinamento. Assumendo che gli elementi siano ordinati in modo crescente, ogni operazione di `pop` estrae l'elemento di posizione 0.

Nel nostro caso, con lo scopo di rappresentare la frontiera di un albero di ricerca, l'implementazione fornita utilizza come default un ordinamento crescente sulla base del valore del costo del cammino associato a ciascun nodo. Inoltre, sono aggiunte funzionalità specifiche per cercare nodi con uno specifico stato associato e per la rimozione (senza estrazione) di uno specifico nodo.

La classe `Priorityqueue` implementa le funzionalità di una coda con priorità, è fornita nel file `Queue.py` e in Tabella 7.

```

#class PriorityQueue - gli elementi nella coda sono ordinati
#                               in base ad una funzione di ordinamento
class PriorityQueue(Queue):

    def __init__(self, f = lambda x:x):
        """Costruttore.
        L'argomento f specifica la funzione da usare per calcolare la priorita'
        associata a ciascun elemento nella coda."""

        #inizializza la lista degli elementi come una lista vuota
        self.elements = []
        self.f = f

    def insert(self,element):
        """ Inserisce l'elemento element nella coda. """
        # per gestire la priorita' di ciascun elemento nella lista
        # per ogni inserimento si aggiunge anche la sua priorita'
        E = [element, self.f(element)]
        self.elements.append(E)
        # ri-ordina gli elementi nella lista in base alla funzione di ordinamento
        self.elements.sort(key = lambda x:x[1]) # x[1] e' il secondo elemento
                                                # cioe' il valore in base al quale ordinare

    def pop(self):
        """ Estrae e restituisce il primo elemento della coda in base all'ordinamento definito."""
        return self.elements.pop(0)[0];

    def __repr__(self):
        """ Specifica la stringa da stampare
        per rappresentare la coda."""
        return 'Gli elementi nella coda sono (in ordine crescente): '+ str(self.elements)

    def contains(self, element):
        """ Restituisce True se la coda contiene element,
        False altrimenti. """
        return element in self.elements

# --- ulteriori funzionalita' utili per gestire la frontiera ---
def index_state(self, state):
    """ Cerca nella coda un nodo con stato specificato.
    Se lo stato viene trovato restituisce l'indice nella lista corrispondente,
    altrimenti restituisce -1. """
    found = -1
    for index in range(len(self.elements)):
        # cerca lo stato tra gli elementi della coda
        if self.elements[index][0].state == state:
            return index
    return found

def contains_state(self, state):
    """ Restituisce True se nella coda c'e' un nodo con stato state,
    False altrimenti. """
    return self.index_state(state)>-1

def remove(self, index):
    """ Rimuove dalla coda l'elemento di indice specificato da index. """
    del self.elements[index]

def get_node(self, index):
    """ Restituisce il nodo nella coda di indice specificato. """
    if len(self.elements)> index:
        # se ci sono almeno index elementi restituisci l'elemento
        # della lista di posizione index
        return self.elements[index][0]
    else:
        # altrimenti restituisce None
        return None

```

Tabella 7 La classe PriorityQueue.

3. Algoritmi di Ricerca

In questa sezione vengono riportate le funzioni che implementano gli algoritmi di ricerca trattati. Tutte le funzioni descritte qui sono fornite nel file `SearchingAlgorithms.py`.

Nel file `SearchingAlgorithms.py`, è contenente anche una funzione ausiliare `print_solution` (vedi pacchetto del codice), che stampa a video una descrizione testuale di una soluzione del problema (vedi Sezione 2).

Breadth-first Search: Ricerca-grafo in ampiezza

La funzione che implementa la ricerca grafo in ampiezza ha come argomento il problema (vedi Sezione 2). Il codice è riportato in Tabella 8.

```
def breadth_first_search(problem):
    """Ricerca-grafo in ampiezza"""
    explored = [] # insieme (implementato come una lista) degli stati già visitati
    node = Node(problem.initial_state) # il costo del cammino è inizializzato
                                         # nel costruttore del nodo
    # controlla se lo stato iniziale è uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution(explored_set = explored)
    frontier = FIFOQueue() # la frontiera è una coda FIFO
    frontier.insert(node)
    while not frontier.isempty():
        # seleziona il nodo per l'espansione
        node = frontier.pop()
        explored.append(node.state) # inserisce il nodo nell'insieme dei nodi esplorati
        for action in problem.actions(node.state):
            child_node = node.child_node(problem, action)
            # controlla se lo stato del nodo figlio non è nell'insieme dei nodi esplorati
            # e non è nella frontiera
            if (child_node.state not in explored) and \
                (not frontier.contains_state(child_node.state)):
                # controlla se lo stato del nodo figlio è uno stato obiettivo
                if problem.goal_test(child_node.state):
                    return child_node.solution(explored_set = explored)
                # se lo stato non è uno stato obiettivo allora inserisci il nodo nella frontiera
                frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento
```

Tabella 8 Ricerca-grafo in ampiezza.

Uniform-cost Search: Ricerca-grafo a costo uniforme

La funzione che implementa la ricerca grafo a costo uniforme in ampiezza ha come argomento il problema (vedi Sezione 2). Il codice è riportato in Tabella 9.

```
def uniform_cost_search(problem):
    """Ricerca-grafo UC"""
    explored = [] # insieme (implementato come una lista) degli stati già visitati
    node = Node(problem.initial_state) # il costo del cammino è inizializzato
                                         # nel costruttore del nodo
    # la frontiera è una coda con priorità
    frontier = PriorityQueue(f = lambda x:x.path_cost) # lambda serve a definire
                                                         # una funzione anonima a runtime
    frontier.insert(node)
    while not frontier.isempty():
        # seleziona il nodo per l'espansione
        node = frontier.pop() # estrae il nodo con costo minore
        # controlla se lo stato del nodo è uno stato obiettivo
        if problem.goal_test(node.state):
            return node.solution(explored_set = explored)
        else:
            # se non lo è inserisci lo stato nell'insieme degli esplorati
            explored.append(node.state)
        for action in problem.actions(node.state):
            child_node = node.child_node(problem, action)
            # controlla se lo stato del nodo figlio non è nell'insieme dei nodi esplorati
            # e non è nella frontiera
            if (child_node.state not in explored) and \
                (not frontier.contains_state(child_node.state)):
                frontier.insert(child_node)
```

```

        # se lo stato del nodo figlio e' gia' nella frontiera, ma con un costo piu' alto
        # allora sostituisci il nodo nella frontiera con il nodo figlio
        elif frontier.contains_state(child_node.state) and \
(frontier.get_node(frontier.index_state(child_node.state)).path_cost >
                                     child_node.path_cost):
            frontier.remove(frontier.index_state(child_node.state))
            frontier.insert(child_node)

    return None # in questo caso ritorna con fallimento

```

Tabella 9 Ricerca-grafo a costo uniforme.

Ricerca A*

La funzione che implementa la ricerca con euristica mediante algoritmo A* ha come argomento il problema (vedi Sezione 2). Il codice è riportato in Tabella 10.

```

def astar_search(problem):
    """ Ricerca A*. """
    explored = [] # insieme (implementato come una lista) degli stati gia' visitati
    node = Node(problem.initial_state) # il costo del cammino e' inizializzato
                                         # nel costruttore del nodo

    #lambda serve a definire una funzione anonima a runtime
    frontier = PriorityQueue(f = lambda x: (x.path_cost) + problem.h(x))
    frontier.insert(node)
    while not frontier.isempty():
        # seleziona un nodo per l'espansione
        node = frontier.pop() # seleziona il nodo con costo del cammino piu' basso
        # controlla se lo stato del nodo e' uno stato obiettivo
        if problem.goal_test(node.state):
            return node.solution(explored_set = explored)
        else:
            # se lo stato non e' uno stato obiettivo aggiungilo all'insieme degli esplorati
            explored.append(node.state)
            for action in problem.actions(node.state):
                child_node = node.child_node(problem, action)
                # controlla se lo stato del nodo figlio non e' nell'insieme dei nodi esplorati
                # e non e' nella frontiera
                if (child_node.state not in explored) and \
                    (not frontier.contains_state(child_node.state)):
                    frontier.insert(child_node)
                # se lo stato del nodo figlio e' gia' nella frontiera, ma con un costo piu' alto
                # allora sostituisci il nodo nella frontiera con il nodo figlio
                elif frontier.contains_state(child_node.state) and \
                    (frontier.get_node(frontier.index_state(child_node.state)).path_cost >
                     child_node.path_cost):
                    frontier.remove(frontier.index_state(child_node.state))
                    frontier.insert(child_node)

    return None #in questo caso ritorna con fallimento

```

Tabella 10 Ricerca A*.

Depth-first Search: Ricerca in profondità

Per la ricerca in profondità sono riportate le varianti riguardanti le metodologie di ricerca-albero (in cui è possibile tornare in uno stato già visitato) e ricerca-grafo (in ogni stato può essere visitato al più una volta). Sono inoltre riportate anche una implementazione ricorsiva della ricerca in profondità, e le varianti della ricerca-albero a profondità limitata.

Depth-first Search Tree: Ricerca-albero in profondità

La funzione che implementa la ricerca-albero in profondità ha come argomento il problema (vedi Sezione 2). Il codice è riportato in Tabella 11.

```

def depth_first_search_tree(problem):
    """Ricerca-albero in profondita' """
    node = Node(problem.initial_state) # il costo del cammino e' inizializzato
                                         # nel costruttore del nodo

    # controlla se lo stato iniziale e' uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()

```



```

frontier = LIFOQueue() # la frontiera e' una coda LIFO
frontier.insert(node)
while not frontier.isempty():
    node = frontier.pop() # estrae il nodo dalla frontiera
    # controlla se lo stato del nodo e' uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    # espandi la frontiera
    for action in problem.actions(node.state):
        child_node = node.child_node(problem,action)
        #if (child_node.state not in explored) and (not frontier.contains(child_node)):
        frontier.insert(child_node)
return None # in questo caso ritorna con fallimento

```

Tabella 11 Ricerca-albero in profondità.

Depth-first Search Graph: Ricerca-grafo in profondità

La funzione che implementa la ricerca-grafo in profondità ha come argomento il problema (vedi Sezione 2).

Il codice è riportato in Tabella 12.

```

def depth_first_search_graph(problem):
    """Ricerca-grafo in profondita' """
    explored = [] # insieme (implementato come una lista) degli stati gia' visitati
    node = Node(problem.initial_state) # il costo del cammino e' inizializzato
    # nel costruttore del nodo
    # controlla se lo stato iniziale e' uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    frontier = LIFOQueue() #la frontiera e' una coda LIFO
    frontier.insert(node)
    while not frontier.isempty():
        node = frontier.pop() #estrae il nodo dalla frontiera
        # controlla se lo stato del nodo e' uno stato obiettivo
        if problem.goal_test(node.state):
            return node.solution(explored_set = explored)
        else:
            # se lo stato non e' uno stato obiettivo aggiungilo all'insieme degli esplorati
            explored.append(node.state)
        # espandi la frontiera
        for action in problem.actions(node.state):
            child_node = node.child_node(problem,action)
            if (child_node.state not in explored) and (not frontier.contains(child_node.state)):
                frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento

```

Tabella 12 Ricerca-grafo in profondità.

Recursive Depth-first Search: Ricerca ricorsiva in profondità

La funzione che implementa la versione ricorsiva della ricerca in profondità ha come argomenti il problema e il nodo di partenza da cui iniziare la ricerca (vedi Sezione 2).

Il codice della funzione è riportato in Tabella 13.

```

def recursive_depth_first_search(problem, node):
    """Ricerca in profondita' ricorsiva """
    # controlla se lo stato del nodo e' uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    # in caso contrario continua con la ricerca
    for action in problem.actions(node.state):
        child_node = node.child_node(problem, action)
        result = recursive_depth_first_search(problem, child_node)
        if result is not None:
            return result
    return None

```

Tabella 13 Ricerca ricorsiva in profondità.

Il nodo di partenza va inizializzato mediante il costruttore della classe Node, indicando come stato lo stato iniziale del problema, ad esempio:

```

initial_node = Node(p.initial_state)
recursive_depth_first_search(p, initial_node)

```

dove p indica la variabile contenente l'istanza del problema.

Limited Depth-first Search Tree: Ricerca-albero a profondità limitata

La funzione che implementa la ricerca-albero a profondità limitata ha come argomento il problema (vedi Sezione 2) e la profondità massima. Il codice è riportato in Tabella 14.

```
def limited_depth_first_search_tree(problem, depth_limit):
    """Ricerca-albero in profondita' con depth_limit"""
    node = Node(problem.initial_state) # il costo del cammino e'
                                     # inizializzato nel costruttore del nodo
    # controlla se lo stato iniziale e' uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    frontier = LIFOQueue() # la frontiera e' una coda LIFO
    frontier.insert(node)
    while not frontier.isempty():
        node = frontier.pop() #estrae il nodo dalla frontiera
        if node.depth > depth_limit:
            continue
        # controlla se lo stato del nodo e' uno stato obiettivo
        if problem.goal_test(node.state):
            return node.solution()
        # espandi la frontiera
        for action in problem.actions(node.state):
            child_node = node.child_node(problem, action)
            frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento
```

Tabella 14 Ricerca-albero a profondità limitata.

Limited Recursive Depth-first Search: Ricerca ricorsiva a profondità limitata

La funzione che implementa la versione ricorsiva della ricerca a profondità limitata ha come argomento il problema, il nodo di partenza da cui iniziare la ricerca (vedi Sezione 2) e la profondità massima. Il codice è riportato in Tabella 15.

```
def limited_recursive_depth_first_search(problem, node, depth_limit):
    """Ricerca in profondita' ricorsiva con depth_limit"""

    if depth_limit < 0:
        return None
    # controlla se lo stato del nodo e' uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    # in caso contrario continua con la ricerca
    for action in problem.actions(node.state):
        child_node = node.child_node(problem, action)
        result = limited_recursive_depth_first_search(problem, child_node, depth_limit-1)
        if result is not None:
            return result
    return None
```

Tabella 15 Ricerca ricorsiva a profondità limitata.

Il nodo di partenza va inizializzato mediante il costruttore della classe Node, indicando come stato lo stato iniziale del problema, ad esempio:

```
initial_node = Node(p.initial_state)
depth_limit = 10
limited_recursive_depth_first_search(p, initial_node, depth_limit)
```

dove p indica la variabile contenente l'istanza del problema.

Algoritmi di Ricerca Locale

In questa sezione sono descritte le funzioni che implementano gli algoritmi di ricerca locale Hill-climbing e Simulated annealing. L'output di queste funzioni consiste nel nodo ottenuto alla fine della ricerca, ed è quindi differente rispetto a quello ottenuto mediante le funzioni che implementano gli algoritmi di ricerca in ampiezza, a costo uniforme, A* e in profondità.

L'implementazione fornita per gli algoritmi di ricerca in profondità richiede che nell'oggetto che descrive il problema sia definita una funzione value, che abbia come argomento un nodo. Questo si può realizzare specializzando la classe Problem, o una sua sottoclasse, ad esempio:

```
class Problema_per_Ricerca_Locale(Problem):
    """ ... """
    def value(self, node):
        # questa funzione calcola il valore della funzione da massimizzare
        # tramite l'algoritmo di ricerca locale

    """ ... """
```

Ricerca Hill-climbing

La funzione che implementa l'algoritmo di ricerca locale Hill-climbing ha come argomento il problema (vedi Sezione 2). Il codice è riportato in Tabella 16.

```
def hill_climbing(problem):
    """ Ricerca locale - Hill-climbing. """
    current = Node(problem.initial_state)
    while True:
        neighbors = [current.child_node(problem, action) for action in
                     problem.actions(current.state)]
        # se current non ha successori esci e restituisci current
        if not neighbors:
            break
        # scegli il vicino con valore piu' alto
        neighbor = (sorted(neighbors, key = lambda x: problem.value(x), reverse = True))[0]
        if problem.value(neighbor) <= problem.value(current):
            break
        else:
            current = neighbor
    return current
```

Tabella 16 Hill-climbing.

Ricerca Simulated annealing

La funzione che implementa l'algoritmo di ricerca locale Simulated annealing ha come argomento il problema (vedi Sezione 2) e una funzione schedule, che descrive l'andamento della temperatura in funzione del tempo. Il codice dell'algoritmo di ricerca e un esempio di implementazione di funzione di schedule sono riportati in Tabella 17.

```
def schedule_ex(k=20, lam=0.005, limit=100):
    """ Una possibile funzione di scheduling per
        l'algoritmo di simulated annealing. """
    return lambda t: (k * math.exp(-lam * t) if t < limit else 0)

def simulated_annealing(problem, schedule=schedule_ex()):
    """ Ricerca locale - Simulated Annealing. """
    current = Node(problem.initial_state)
    for t in range(sys.maxsize): # sys.maxsize e' il massimo numero intero
        T = schedule(t)
        if T == 0:
            return current
        neighbors = [current.child_node(problem, action) for action in
```

```
        problem.actions(current.state)]  
# se current non ha successori esci e restituisci current  
if not neighbors:  
    return current  
next = random.choice(neighbors)  
delta_e = problem.value(next) - problem.value(current)  
if delta_e > 0 or (random.random() < (math.exp(delta_e / T))):  
    current = next
```

Tabella 17 Simulated annealing ed esempio di funzione schedule.

4. Ricerca con Avversari

In questa sezione sono introdotte la struttura di dati usata per rappresentare un gioco e le funzioni considerate per la ricerca con avversari. Tutte le funzioni descritte in questa sezione sono riportate nel file `AdversarialSearch.py`.

Rappresentazione di un Gioco: la classe `Game`

La classe `Game` implementa le funzioni necessarie per rappresentare un gioco:

- `actions`: dato uno stato, restituisce una lista di mosse possibili in quello stato;
- `result`: dato uno stato e un'azione, restituisce lo stato risultante (implementa il modello di transizione);
- `terminal_test`: dato uno stato restituisce `True` se il gioco è terminato, `False` altrimenti
- `utility`: dato uno stato restituisce il valore numerico finale (il punteggio ottenuto) se il gioco è terminato nello stato indicato.

Il codice della classe `Game` è riportato in Tabella 18, ed è fornito nel file `Game.py`.

```
class Game:
    """
    Questa classe impementa l'astrazione di un gioco.
    Per rappresentare un gioco specializzare questa classe implementandone i metodi.
    """

    def actions(self, state):
        """Restituisce la lista delle possibili mosse nello state state."""
        raise NotImplementedError

    def result(self, state, action):
        """Restituisce lo stato che risulta dalla mossa action nello stato state."""
        raise NotImplementedError

    def terminal_test(self, state):
        """Restituisce True se state e' uno stato finale (se il gioco e' terminato),
        False altrimenti."""
        return (len(self.actions)==0)

    def utility(self, state):
        """restituisce il valore di utilita' dello state state per il giocatore."""
        raise NotImplementedError
```

Tabella 18 La classe `Game`.

Per rappresentare il gioco desiderato, occorre specializzare la classe `Game`, implementando almeno le funzioni `actions`, `result` e `utility`. Un semplice esempio pratico, il cui albero è raffigurato in Figura 3, è rappresentato dalla classe `ToyGame1`, il cui codice è riportato in Tabella 19 e nel file `ToyGames.py`.

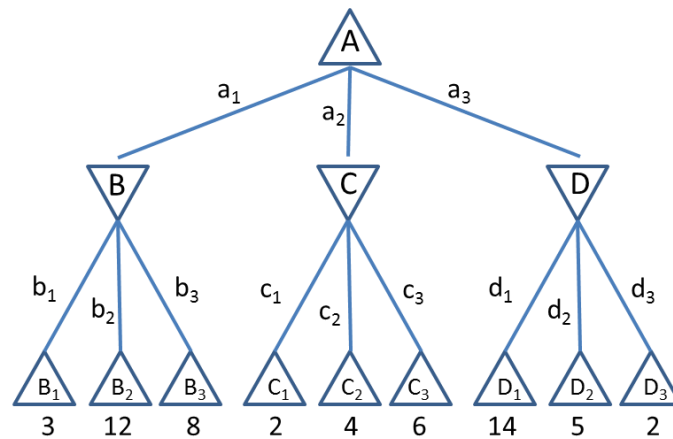


Figura 3 Albero relativo al gioco nella classe ToyGame1; sotto ciascuno stato finale è indicata la corrispondente utilità.

```
class ToyGame1(Game):
    """Rappresentazione di un gioco di esempio."""
    risultato = {'B1':3, 'B2':12, 'B3':8, 'C1':2, 'C2':4, 'C3':6, 'D1':14, 'D2':5, 'D3':2}

    def actions(self, state):
        if state == 'A':
            return ['a1','a2','a3']
        if state == 'B':
            return ['b1','b2','b3']
        if state == 'C':
            return ['c1','c2','c3']
        if state == 'D':
            return ['d1','d2','d3']
        return []

    def result(self, state, move):
        if move == 'a1':
            return 'B'
        if move == 'a2':
            return 'C'
        if move == 'a3':
            return 'D'
        #
        if move == 'b1':
            return 'B1'
        if move == 'b2':
            return 'B2'
        if move == 'b3':
            return 'B3'
        #
        if move == 'c1':
            return 'C1'
        if move == 'c2':
            return 'C2'
        if move == 'c3':
            return 'C3'
        #
        if move == 'd1':
            return 'D1'
        if move == 'd2':
            return 'D2'
        if move == 'd3':
            return 'D3'

    def utility(self, state):
        return self.risultato[state]

    def terminal_test(self, state):
        return state not in ('A', 'B', 'C', 'D')
```

Tabella 19 La classe ToyGame1.

L'algoritmo minimax

L'algoritmo minimax è implementato nella funzione `minimax_decision`. Questa funzione ha come argomenti il gioco e lo stato in cui si trova il giocatore, e restituisce la mossa migliore. Il codice della funzione `minimax_decision` è riportato in Tabella 20. Nota che il codice stampa, per tutti i nodi interni, lo stato e il valore di utilità di minimax; infine viene stampata l'azione selezionata.

```
def minimax_decision(game, state):
    def max_value(state):
        if game.terminal_test(state):
            # se in state il gioco e' concluso restituisci il risultato
            return game.utility(state)
        v = - float('inf') # v e' inizializzato a - infinito
        for a in game.actions(state):
            v = max(v, min_value(game.result(state,a)))
        print('MAX: stato {} - utilita\' {}'.format(state,v))
        return v

    def min_value(state):
        if game.terminal_test(state):
            # se in state il gioco e' concluso restituisci il risultato
            return game.utility(state)
        v = float('inf') # v e' inizializzato a + infinito
        for a in game.actions(state):
            v = min(v,max_value(game.result(state,a)))
        print('MIN: stato {} - utilita\' {}'.format(state,v))
        return v

    best_action = max(game.actions(state), key = lambda x:min_value(game.result(state,x)))
    # best_action e' l'argomento (l'azione) che massimizza l'output di min_value
    print("L'azione selezionata e' {}".format(best_action))
    return best_action
```

Tabella 20 Minimax.

L'algoritmo di Potatura Alfa-Beta

L'algoritmo di ricerca con potatura alfa-beta è implementato nella funzione `alpha_beta`. Questa funzione ha come argomenti il gioco e lo stato in cui si trova il giocatore, e restituisce la mossa migliore, calcolata utilizzando l'algoritmo di potatura alfa-beta. Il codice della funzione `alpha_beta` è riportato in Tabella 21. Nota che il codice stampa, per tutti i nodi interni, lo stato, il valore di utilità di minimax, alfa e beta, segnalando anche dove si verificano le condizioni per il taglio alfa o beta; infine viene stampata l'azione selezionata.

```
def alpha_beta(game,state):
    def max_value(state, alpha, beta):
        if game.terminal_test(state):
            # se in state il gioco e' concluso restituisci il risultato
            return game.utility(state)
        v = - float('inf') # v e' inizializzato a - infinito
        for a in game.actions(state):
            v = max(v, min_value(game.result(state,a), alpha, beta))
            if v >= beta: # taglio beta
                print('MAX: stato {} - utilita\' {} - '+
                    'TAGLIO BETA (alpha = {}, beta = {})'.format(state,v,alpha,beta))
                return v
            alpha = max(alpha,v) # aggiorna il migliore per MAX
        print('MIN: stato {} - utilita\' {} - (alpha = {}, beta = {})'.format(state,v,alpha,beta))
        return v

    def min_value(state, alpha, beta):
        if game.terminal_test(state):
            # se in state il gioco e' concluso restituisci il risultato
            return game.utility(state)
        v = float('inf') # v e' inizializzato a + infinito
        for a in game.actions(state):
            v = min(v, max_value(game.result(state,a), alpha, beta))
            if v <= alpha: # taglio alpha
                print('MIN: stato {} - utilita\' {} - '+
                    'TAGLIO ALPHA (alpha = {}, beta = {})'.format(state,v,alpha,beta))
                return v
            beta = min(beta,v) # aggiorna il migliore per MIN
        print('MAX: stato {} - utilita\' {} - (alpha = {}, beta = {})'.format(state,v,alpha,beta))
        return v

    best_action = max(game.actions(state), key = lambda x:alpha_beta(game,result(state,x)))
    print("L'azione selezionata e' {}".format(best_action))
    return best_action
```

```

        'TAGLIO ALPHA (alpha = {}, beta = {})'
        .format(state,v,alpha,beta))
    return v
    beta = min(beta,v) # aggiorna il migliore per MIN
    print('MIN: stato {} - utilita\' {} - (alpha = {}, beta = {})'
        .format(state,v,alpha,beta))
    return v

#inizializza alpha e beta
alpha = - float('inf')
beta = float('inf')

best_action = None
# esegue un ciclo esterno di max_value "controllato"
# memorizzando la mossa migliore
for a in game.actions(state):
    v = min_value(game.result(state,a), alpha, beta)
    if v > alpha:
        # se lo score della mossa a e' il migliore fino a qui
        # allora memorizza la mossa a
        best_action = a
        alpha = v
    print("L'azione selezionata e' {}".format(best_action))
return best_action

```

Tabella 21 Alpha-Beta pruning.