

**Università degli Studi di Verona**  
Dipartimento di Informatica  
CdLM in Ingegneria e scienze informatiche

# **Fondamenti di intelligenza artificiale**

Riassunto del corso

Creato da: **Davide Zampieri**

**Anno Accademico 2020-2021**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Cos'è l'intelligenza artificiale? . . . . .	1
1.1.1	Agire razionalmente . . . . .	1
1.2	I fondamenti dell'intelligenza artificiale . . . . .	1
1.3	La storia dell'intelligenza artificiale . . . . .	2
<b>2</b>	<b>Agenti intelligenti</b>	<b>3</b>
2.1	Agenti e ambienti . . . . .	3
2.1.1	Razionalità . . . . .	3
2.2	La natura degli ambienti . . . . .	3
2.2.1	Specificare un ambiente . . . . .	3
2.2.2	Proprietà degli ambienti . . . . .	4
2.3	La struttura degli agenti . . . . .	4
2.3.1	Programmi agente . . . . .	4
2.3.2	Tipi di agenti . . . . .	4
<b>3</b>	<b>Risolvere i problemi con la ricerca</b>	<b>6</b>
3.1	Agenti risolutori di problemi . . . . .	6
3.1.1	La formulazione dei problemi . . . . .	6
3.2	Cercare soluzioni . . . . .	6
3.2.1	Strutture dati per algoritmi di ricerca . . . . .	7
3.2.2	Misurare le prestazioni nella risoluzione di problemi . . . . .	7
3.3	Strategie di ricerca non informata . . . . .	8
3.3.1	Ricerca in ampiezza . . . . .	8
3.3.2	Ricerca a costo uniforme . . . . .	8
3.3.3	Ricerca in profondità . . . . .	8
3.3.4	Ricerca ad approfondimento iterativo . . . . .	8
3.3.5	Confronto tra le strategie di ricerca non informata . . . . .	9
3.4	Strategie di ricerca informata o euristica . . . . .	10
3.4.1	Ricerca greedy best-first . . . . .	10
3.4.2	Ricerca $A^*$ . . . . .	10
3.4.3	Funzioni euristiche . . . . .	10
<b>4</b>	<b>Oltre la ricerca classica</b>	<b>11</b>
4.1	Algoritmi di ricerca locale e problemi di ottimizzazione . . . . .	11
4.1.1	Ricerca hill climbing . . . . .	11

4.1.2	Simulated annealing e algoritmi genetici . . . . .	11
4.2	Ricerca locale in spazi continui . . . . .	12
<b>5</b>	<b>Problemi di soddisfacimento di vincoli</b>	<b>13</b>
5.1	Definizione dei problemi di soddisfacimento di vincoli . . . . .	13
5.1.1	Reti a vincoli . . . . .	13
5.1.2	Un problema di esempio: colorazione di una mappa . . . . .	14
5.2	Tecniche risolutive per le reti a vincoli . . . . .	15
5.2.1	Ricerca con backtracking . . . . .	15
5.2.2	Inferenza e propagazione dei vincoli . . . . .	15
5.2.3	Metodi di consistenza . . . . .	15
5.3	Strategie di ricerca per la propagazione dei vincoli . . . . .	16
5.3.1	Backtracking search . . . . .	17
5.3.2	Ordinamento di variabili . . . . .	17
5.3.3	Ordinamento di valori . . . . .	18
5.3.4	Alternanza di ricerca e inferenza . . . . .	18
5.3.5	Mantaining Arc Consistency . . . . .	19
5.4	La struttura dei problemi . . . . .	19
5.4.1	Scomposizione ad albero . . . . .	20
5.4.2	Concetto di ipergrafo . . . . .	20
5.4.3	Rappresentazione di una rete a vincoli . . . . .	20
5.4.4	Reti acicliche . . . . .	20
5.4.5	Risolvere una rete aciclica . . . . .	20
5.4.6	Clustering . . . . .	21
5.5	Constraint optimisation problems . . . . .	22
5.5.1	Metodi risolutivi per le reti di costo . . . . .	22
5.5.2	Branch and Bound . . . . .	22
5.5.3	Bucket Elimination . . . . .	22
<b>6</b>	<b>Incertezza</b>	<b>24</b>
6.1	Agire in condizioni di incertezza . . . . .	24
6.2	Notazione base della teoria della probabilità . . . . .	24
6.2.1	Probabilità a priori . . . . .	24
6.2.2	Probabilità condizionate . . . . .	24
6.3	Inferenza basata su distribuzioni congiunte complete . . . . .	25
6.4	Indipendenza . . . . .	25
6.4.1	La regola di Bayes e il suo utilizzo . . . . .	25
6.4.2	Indipendenza condizionale . . . . .	25
6.5	Il mondo del wumpus . . . . .	26
<b>7</b>	<b>Problemi di decisione sequenziali</b>	<b>27</b>
7.1	Decisioni complesse . . . . .	27
7.1.1	Markov Decision Processes . . . . .	27
7.1.2	La necessità delle politiche . . . . .	28
7.2	Iterazione dei valori . . . . .	28
7.3	Iterazione delle politiche . . . . .	29
7.4	MDP parzialmente osservabili . . . . .	29

<b>8 Apprendimento per rinforzo</b>	<b>30</b>
8.1 Introduzione . . . . .	30
8.1.1 Progetto globale . . . . .	30
8.2 Metodi model-based . . . . .	31
8.3 Metodi model-free . . . . .	31
8.3.1 Q-Learning . . . . .	31
8.3.2 Sfruttamento ed esplorazione . . . . .	31
8.3.3 SARSA . . . . .	32
8.4 Deep reinforcement learning . . . . .	32
8.4.1 Deep Neural Networks . . . . .	32
8.4.2 Deep Q-Network . . . . .	33
<b>A Codici dei programmi del laboratorio</b>	<b>34</b>
<b>Credits</b>	<b>69</b>

# **Lista dei codici**

<b>Sessione 1 - Ricerca non informata</b>	<b>34</b>
<b>Sessione 2 - Ricerca informata</b>	<b>41</b>
<b>Sessione 3 - Markov Decision Processes</b>	<b>49</b>
<b>Sessione 4 - Reinforcement learning</b>	<b>57</b>
<b>Sessione 5 - Deep reinforcement learning</b>	<b>65</b>

# Capitolo 1

## Introduzione

### 1.1 Cos'è l'intelligenza artificiale?

Quando ci si avvicina per la prima volta all'IA è necessario porsi *due domande*:

1. Interessa di più il pensiero o il comportamento?
2. Si vuole usare come modello gli esseri umani o fare riferimento ad uno standard ideale?

Con le *risposte* a queste domande possiamo costruire la seguente tabella:

Systems that think like humans	Systems that think rationally
Systems that act like humans	Systems that act rationally

#### 1.1.1 Agire razionalmente

Da questo momento in poi adottiamo l'idea che l'intelligenza riguardi principalmente l'*azione razionale*. Idealmente, un *agente intelligente* intraprende in ogni situazione la migliore azione possibile. Bisognerà quindi trovare il modo di costruire agenti intelligenti secondo questa particolare accezione.

### 1.2 I fondamenti dell'intelligenza artificiale

I *filosofi*, a partire dal 400 a.C., hanno reso concepibile lo sviluppo dell'IA proponendo che la mente fosse per certi aspetti simile ad una macchina.

I *matematici* hanno poi fornito gli strumenti per manipolare gli enunciati logici (sia in condizioni di certezza che di incertezza), e hanno anche sviluppato la teoria della computazione e dell'analisi di algoritmi.

Inoltre, gli *economisti* hanno formalizzato il problema di prendere decisioni che massimizzino i risultati attesi, mentre gli *studiosi di neuroscienze* hanno scoperto alcuni fatti sul funzionamento del cervello e sui relativi punti di somiglianza e di differenza rispetto ai computer.

Infine, gli *psicologi* e i *linguisti* hanno adottato l'idea che gli esseri umani possano essere considerati macchine che elaborano informazioni, mentre gli *ingegneri informatici* hanno creato macchine sempre più potenti per rendere possibili le applicazioni dell'IA.

### 1.3 La storia dell'intelligenza artificiale

Attualmente la *teoria del controllo* si sta avvicinando all'IA, in quanto si occupa della progettazione di dispositivi che agiscono in maniera ottimale basandosi sull'ambiente.

Ma la *storia* dell'IA è stata anche segnata da cicli in cui al successo ha fatto seguito un eccessivo ottimismo che ha portato a cadute d'entusiasmo e a tagli di fondi.

Ora, grazie all'introduzione di nuovi approcci creativi e all'uso diffuso del metodo scientifico, i *diversi settori* dell'IA sono arrivati ad un'integrazione e, con l'aumento delle capacità di calcolo dei sistemi, l'IA stessa sta trovando un terreno comune con altre discipline.

# Capitolo 2

## Agenti intelligenti

### 2.1 Agenti e ambienti

Un *agente* è qualcosa che percepisce e agisce all'interno di un *ambiente*. La sua *funzione agente* specifica l'azione intrapresa in risposta a qualsiasi sequenza di percezioni. Formalmente, la funzione agente è definita nel seguente modo:

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

dove  $\mathcal{P}^*$  è lo storico delle percezioni e  $A$  è un insieme di azioni.

Inoltre, si fa notare che se un agente ha  $|\mathcal{P}|$  percezioni possibili in ingresso, dopo  $T$  unità di tempo la sua funzione agente avrà un numero di entry pari a  $\sum_{t=1}^T |\mathcal{P}|^t$ .

#### 2.1.1 Razionalità

La *misura di prestazione* valuta il comportamento dell'agente nell'ambiente in cui opera. Un agente *razionale* dovrà quindi agire in modo da massimizzare il valore atteso della misura di prestazione (data la sequenza percettiva fino a quel momento).

### 2.2 La natura degli ambienti

#### 2.2.1 Specificare un ambiente

La specifica di un task environment, o *ambiente operativo*, include:

- La misura di prestazione.
- L'ambiente esterno.
- Gli attuatori.
- I sensori.

Nella *progettazione* di un agente razionale il primo passo deve sempre consistere nella specifica più dettagliata possibile dell'ambiente operativo.

## 2.2.2 Proprietà degli ambienti

Gli ambienti operativi possono essere classificati in base a molte *proprietà* significative. Basandoci su alcune di esse potremo stabilire se gli agenti sono:

- Completamente o parzialmente osservabili.
- Deterministici o stocastici.
- Episodici o sequenziali.
- Statici o dinamici.
- Discreti o continui.
- A singolo agente o multi-agente.

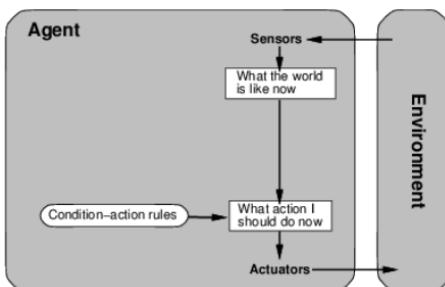
## 2.3 La struttura degli agenti

### 2.3.1 Programmi agente

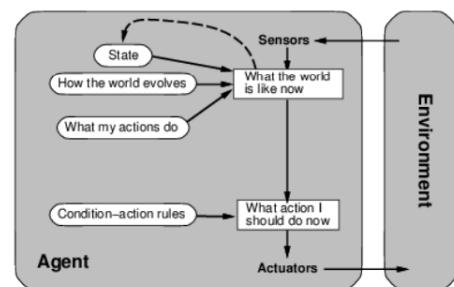
Il *programma agente* implementa la funzione agente, ovvero prende in input la percezione attuale e ritorna in output la prossima azione da svolgere.

Esisteranno quindi diversi *schemi* base per tali programmi, che rifletteranno il tipo di informazione esplicitata e utilizzata nel processo decisionale. Siccome gli schemi possono variare in efficienza, compattezza e flessibilità, lo schema più appropriato per un dato programma agente dipenderà dalla natura dell'ambiente stesso.

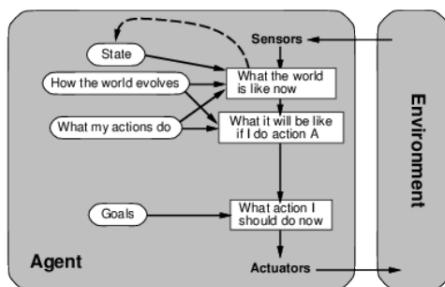
### 2.3.2 Tipi di agenti



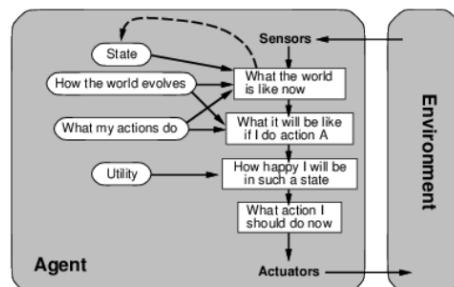
(a) Simple reflex agents



(b) Model-based agents



(c) Goal-based agents



(d) Utility-based agents

Esistono principalmente quattro *tipi* di agenti:

- (a) Gli *agenti reattivi semplici*, che rispondono direttamente alle percezioni.
- (b) Gli *agenti reattivi basati su modello*, che mantengono uno stato interno per tener traccia degli aspetti del mondo che non sono visibili nelle percezioni correnti.
- (c) Gli *agenti basati su obiettivi*, le cui azioni sono condizionate dall'obiettivo prefissato.
- (d) Gli *agenti basati sull'utilità*, che cercano di massimizzare un valore atteso.

Tutte le tipologie di agenti possono migliorare le loro prestazioni mediante l'*apprendimento*.

# Capitolo 3

## Risolvere i problemi con la ricerca

### 3.1 Agenti risolutori di problemi

I *problem*i possono essere:

1. Deterministici e completamente osservabili (a stato singolo).
2. Non osservabili.
3. Non deterministici e/o parzialmente osservabili.

Nel primo caso, ovvero all'interno di ambienti deterministici, osservabili, statici e completamente noti, l'*agente* può costruire sequenze di azioni che raggiungono il suo obiettivo; questo processo si chiama *ricerca*.

#### 3.1.1 La formulazione dei problemi

Prima di cominciare a cercare le soluzioni, è necessario identificare un *obiettivo* e formulare un *problema* ben definito, il quale è composto da quattro parti:

1. Lo *stato iniziale*.
2. Un *modello di transizione* (successor function) che descrive i risultati di un insieme di *azioni*.
3. Una funzione *test obiettivo*.
4. Una funzione *costo di cammino*.

L'ambiente del problema è rappresentato invece dallo *spazio degli stati*. Una *soluzione* è quindi un cammino attraverso lo spazio degli stati che va da uno stato iniziale a uno stato di goal (obiettivo).

### 3.2 Cercare soluzioni

In generale possiamo avere due tipi di algoritmi di ricerca:

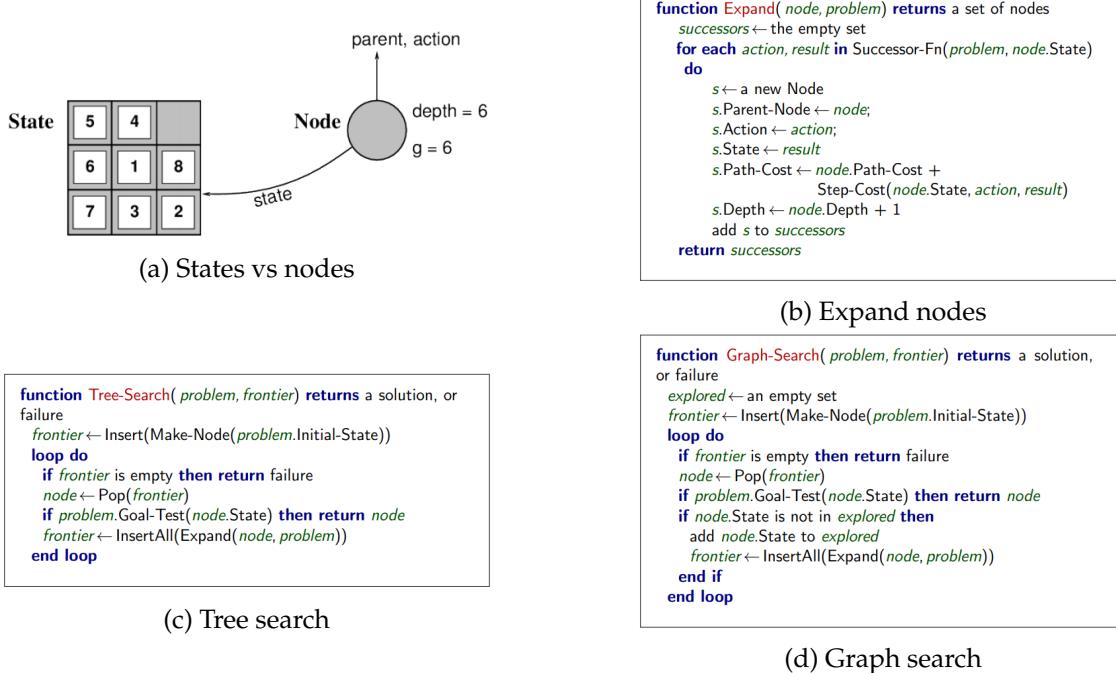
- Algoritmi *Tree-Search*, in cui si considerano tutti i possibili cammini.
- Algoritmi *Graph-Search*, in cui i cammini ridondanti non vengono presi in considerazione.

### 3.2.1 Strutture dati per algoritmi di ricerca

Le *strutture dati* utilizzate negli algoritmi di ricerca sono:

- Una coda FIFO *frontier* contenente i nodi foglia disponibili.
- Un insieme di nodi *explored* contenente i nodi della frontiera che sono stati espansi in passi precedenti.

Va ricordato inoltre che il *nodo* è una struttura dati che può contenere vari campi, mentre lo *stato* è una rappresentazione fisica della configurazione di un ambiente e quindi non ha campi.



### 3.2.2 Misurare le prestazioni nella risoluzione di problemi

Gli algoritmi di ricerca sono valutati sulla base delle seguenti *dimensioni*:

- *Completezza*, ovvero se l'algoritmo trova sempre una soluzione quando essa esiste.
- *Complessità temporale*, ovvero il numero di nodi generati/espansi.
- *Complessità spaziale*, ovvero il massimo numero di nodi da mantenere in memoria.
- *Ottimalità*, ovvero se l'algoritmo trova sempre la soluzione a costo minore.

La *complessità* dipende da:

- $b$ , cioè il massimo fattore di ramificazione (*branching factor*) dello spazio degli stati.
- $d$ , cioè la profondità (*depth*) della soluzione a costo minore (quella più vicina alla radice dell'albero).
- $m$ , cioè la massima profondità dello spazio degli stati (può essere  $\infty$ ).

### 3.3 Strategie di ricerca non informata

I metodi di *ricerca non informata* hanno accesso soltanto alla definizione del problema. Di seguito vengono presentati i vari algoritmi di base.

#### 3.3.1 Ricerca in ampiezza

L'algoritmo di *ricerca in ampiezza* (Breadth-first search, BFS) espande per primi i nodi più vicini alla radice. La *frontiera* sarà quindi una coda FIFO. Per quanto riguarda le proprietà, la ricerca in ampiezza è *completa* e, quando i passi hanno costo unitario, anche *ottima* (in generale non lo è). Ma il vero problema di questo algoritmo risiede nel fatto che ha una *complessità spaziale* esponenziale dell'ordine di  $O(b^d)$ ; infatti, siccome deve tenere traccia di ogni nodo, più è grande l'albero più spazio verrà occupato in memoria.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier  $\leftarrow$  a FIFO queue with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier  $\leftarrow$  INSERT(child, frontier)
```

#### 3.3.2 Ricerca a costo uniforme

L'algoritmo di *ricerca a costo uniforme* (Uniform cost search, UCS) espande sempre il nodo che ha il minor costo di cammino. La *frontiera* sarà quindi una coda ordinata per costo in modo crescente. Se tutti i costi sono uguali, questo algoritmo è equivalente alla ricerca in ampiezza. Per quanto riguarda le proprietà, la ricerca a costo uniforme è *completa* e *ottima* per costi di passo qualsiasi.

#### 3.3.3 Ricerca in profondità

L'algoritmo di *ricerca in profondità* (Depth-first search, DFS) espande prima il nodo più profondo non ancora espanso. La *frontiera* sarà quindi una coda LIFO. Per quanto riguarda le proprietà, la ricerca in profondità non è né *completa* né *ottima*. Tuttavia, la complessità spaziale è lineare in  $O(bm)$ , che sarebbe ideale se non per il fatto che l'algoritmo fallisce se sono presenti cicli che determinano cammini infiniti (incompletezza).

#### 3.3.4 Ricerca ad approfondimento iterativo

Per gli algoritmi di *ricerca ad approfondimento iterativo* (Iterative deepening search, IDS) ci si serve di un'implementazione ricorsiva di un algoritmo di *ricerca a profondità limitata* (Depth-limited search, DLS), il quale prevede di mettere un limite alla profondità (risolvendo il problema dei cammini infiniti).

La ricerca ad approfondimento iterativo eseguirà quindi una serie di ricerche a profondità limitata, estendendo progressivamente il limite di profondità finché non trova una soluzione. Per quanto riguarda le proprietà, la ricerca ad approfondimento iterativo è *completa* e, quando i passi hanno costo unitario, anche *ottima*. Inoltre, ha una complessità temporale pari a  $O(b^d)$  (comparabile alla ricerca in ampiezza) e una complessità spaziale lineare in  $O(bd)$ .

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true
      else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure

function ITERATIVE-DEEPENING(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

```

### 3.3.5 Confronto tra le strategie di ricerca non informata

Confrontiamo ora le strategie di ricerca secondo i quattro *criteri di valutazione* (completezza, ottimalità, complessità temporale e spaziale). Tale confronto riguarda le versioni di *ricerca su alberi*. Nelle *ricerche su grafi* la differenza principale è che le complessità spaziali e temporali risultano più efficienti in quanto limitate dalla dimensione dello spazio degli stati.

Criterio	BFS	UCS	DFS	DLS	IDS
Completa?	Sì <sup>a</sup>	Sì <sup>a,b</sup>	No	Sì <sup>a,d</sup>	Sì <sup>a</sup>
Tempo	$O(b^d)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Spazio	$O(b^d)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Ottima?	Sì <sup>c</sup>	Sì	No	Sì <sup>c,d</sup>	Sì <sup>c</sup>

<sup>a</sup> completa se il fattore di ramificazione  $b$  è finito.

<sup>b</sup> completa se i costi dei passi sono  $\geq$  di un  $\epsilon$  positivo.

<sup>c</sup> ottima se i costi dei passi sono tutti identici.

<sup>d</sup> se il livello del goal si trova entro il limite.

## 3.4 Strategie di ricerca informata o euristica

Le strategie di *ricerca informata* possono utilizzare conoscenze specifiche riguardanti il problema (oltre alla definizione dello stesso) e pertanto sono più efficienti. In particolare, i metodi di ricerca informata possono avere accesso ad una funzione *euristica*  $h(n)$  che stima il costo di una soluzione da  $n$ . Gli approcci generali sono di due tipi:

- Ricerca *greedy best-first*.
- Ricerca  $A^*$ .

### 3.4.1 Ricerca greedy best-first

Un algoritmo di *ricerca best-first* generico seleziona un nodo per l'espansione in base ad una *funzione di valutazione* (detta euristica). In particolare, l'approccio di *ricerca greedy best-first* espande i nodi con  $h(n)$  minima, ovvero cerca di espandere il nodo che sembra essere il più vicino al goal. Per quanto riguarda le proprietà, la ricerca greedy best-first non è né completa (può fallire in caso di cicli) né ottima. Inoltre, ha una complessità temporale pari a  $O(b^m)$  (migliorabile utilizzando euristiche migliori) e una complessità spaziale a sua volta pari a  $O(b^m)$ , in quanto è necessario tenere in memoria tutti i nodi.

### 3.4.2 Ricerca $A^*$

La ricerca  $A^*$  espande i nodi con  $f(n) = g(n) + h(n)$  minima. L'idea è quella di evitare di espandersi percorsi che risultano già essere troppo costosi. Vediamo in dettaglio le componenti della *funzione di valutazione*:

- $g(n)$  è il costo per raggiungere  $n$ .
- $h(n)$  è il costo stimato per raggiungere il goal a partire da  $n$ .
- $f(n)$  è il costo stimato totale del percorso che raggiunge il goal attraverso  $n$ .

La ricerca  $A^*$  usa un'euristica *ammissibile*, ovvero un'euristica in cui  $h(n) \leq h^*(n)$  dove  $h^*(n)$  è il vero costo per raggiungere  $n$ ; viene anche richiesto  $h(n) \geq 0$ , in modo da avere  $h(G) = 0$  per ogni goal  $G$ . Per studiare le proprietà della ricerca  $A^*$  consideriamo due casi:

- $A^*$  nella versione *Tree-Search* è completa e ottima, purché  $h(n)$  sia ammissibile.
- $A^*$  nella versione *Graph-Search* è completa e ottima, purché  $h(n)$  sia consistente.

Si ricorda inoltre che *consistenza*  $\rightarrow$  *ammissibilità* ma *ammissibilità*  $\not\rightarrow$  *consistenza*. Infine, si fa notare che la complessità spaziale di  $A^*$  è ancora proibitiva, in quanto deve comunque tenere tutti i nodi in memoria.

### 3.4.3 Funzioni euristiche

Un'euristica è *consistente* se  $h(n) \leq c(n, a, n') + h(n')$  dove  $c(n, a, n')$  è il costo per raggiungere  $n'$  da  $n$ . Inoltre, un'euristica  $h_2$  *domina* un'altra euristica  $h_1$  se  $\forall n. h_2(n) \geq h_1(n)$ . L'euristica dominante sarà quindi sempre migliore dal punto di vista prestazionale. Infine, è bene sapere che talvolta si possono costruire buone euristiche *rilassando* la definizione del problema.

# Capitolo 4

## Oltre la ricerca classica

### 4.1 Algoritmi di ricerca locale e problemi di ottimizzazione

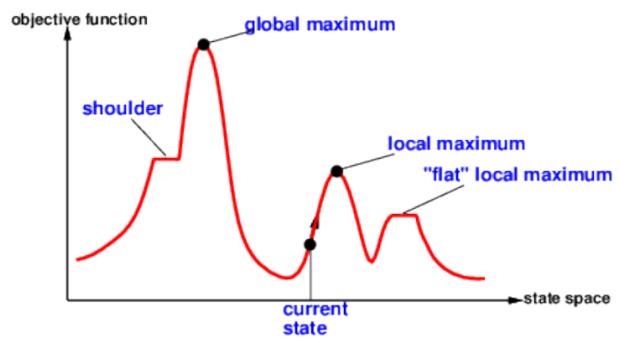
Prenderemo ora in esame algoritmi di ricerca per problemi che vanno oltre il caso "classico" di trovare il cammino più breve per arrivare ad un obiettivo in un ambiente osservabile, deterministico e discreto.

#### 4.1.1 Ricerca hill climbing

La ricerca *hill climbing* è un metodo di *ricerca locale* che lavora su formulazioni del problema a stato completo. Si tratta di un semplice ciclo che si muove continuamente verso l'alto, cioè nella direzione dei valori crescenti, e termina quando raggiunge un picco che non ha vicini di valore più alto (*massimo globale*). L'algoritmo mantiene in memoria solo un piccolo numero di nodi.

```
function Hill-Climbing(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                neighbor, a node
  current  $\leftarrow$  Make-Node(problem.Initial-State)
  loop do
    neighbor  $\leftarrow$  a highest-valued successor of current
    if neighbor.Value  $\leq$  current.Value then return
      current.State
    end if
    current  $\leftarrow$  neighbor
  end
```

(a) Algoritmo di ricerca hill climbing



(b) Spazio degli stati monodimensionale

#### 4.1.2 Simulated annealing e algoritmi genetici

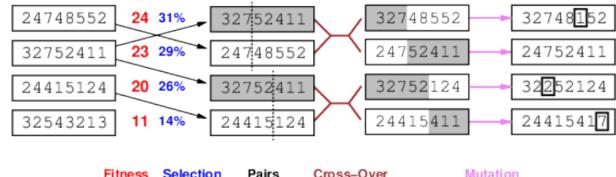
Il *simulated annealing* è un algoritmo stocastico che fornisce soluzioni ottime quando viene impostata una velocità di raffreddamento appropriata (*schedule*). Gli *algoritmi genetici*, invece, sono un particolare tipo di ricerca hill climbing stocastica in cui viene tenuta in memoria una grande popolazione di stati e in cui i nuovi stati vengono generati attraverso la *mutazione* e il *crossover*, il quale genera nuovi individui mediante la combinazione di coppie di stati della popolazione.

```

function Simulated-Annealing(problem, schedule) returns a
solution state
  inputs: problem, a problem
           schedule, a mapping from time to "temperature"
  local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of
downward steps
  current  $\leftarrow$  Make-Node(problem.Initial-State)
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  schedule(t)
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  next.Value - current.Value
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 

```

(a) Algoritmo simulated annealing



(b) Algoritmo genetico sul problema delle 8 regine

## 4.2 Ricerca locale in spazi continui

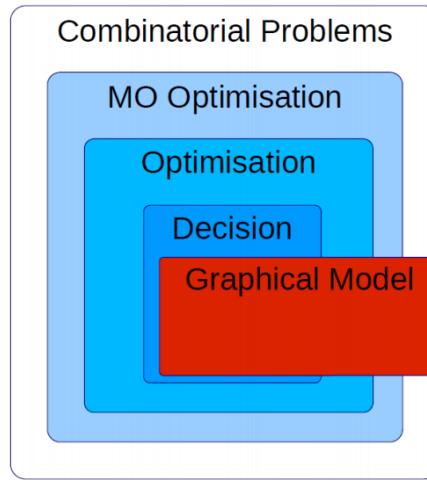
Si possono usare diversi metodi di ricerca locale anche per risolvere problemi negli *spazi continui*. In particolare, i problemi di programmazione lineare e di ottimizzazione convessa obbediscono a determinati vincoli sulla forma dello spazio degli stati e sulla natura della *funzione obiettivo*, e ammettono algoritmi con complessità temporale polinomiale che spesso sono estremamente efficienti nella pratica.

# Capitolo 5

## Problemi di soddisfacimento di vincoli

### 5.1 Definizione dei problemi di soddisfacimento di vincoli

Consideriamo come principale categoria di problemi risolvibili tramite reti a vincoli i *problem combinatori*, ovvero quei problemi in cui si deve scegliere la soluzione migliore tra le tante possibili. I problemi combinatori includono i problemi di *decisione* e di *ottimizzazione*.



#### 5.1.1 Reti a vincoli

Una *rete a vincoli* è costituita da tre componenti:

- Un insieme di *variabili*  $X = \{x_1, \dots, x_n\}$ .
- Un insieme di *domini*  $D = \{D_1, \dots, D_n\}$  (uno per ogni variabile).
- Un insieme di *vincoli*  $C = \{(S_1, R_1), \dots, (S_m, R_m)\}$  che specificano le combinazioni di valori ammesse.

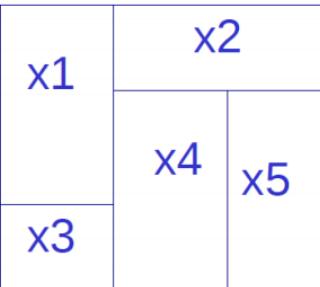
Si noti che un *vincolo* è una coppia  $(S_i, R_i)$  dove  $S_i$  è una o più variabili ( $S_i \subseteq X$ ) e  $R_i$  è un sottoinsieme del prodotto cartesiano delle variabili in  $S_i$ .

La *soluzione* di una rete a vincoli consiste nell'assegnamento dei valori ad alcune o a tutte le variabili in modo che i vincoli vengano soddisfatti (assegnamento *consistente*). Un assegnamento è *completo* se a tutte le variabili viene assegnato un valore, mentre è *parziale* se menziona solo alcune delle variabili (un assegnamento parziale e consistente potrebbe non fare parte della soluzione).

### 5.1.2 Un problema di esempio: colorazione di una mappa

Il problema del *map colouring* consiste nel fare in modo che in una mappa ogni regione abbia un colore diverso dalle regioni adiacenti. Supponendo di avere 5 regioni ( $x_1, \dots, x_5$ ), il grafo dei vincoli si costruisce collegando tra loro con un arco le regioni che non devono avere lo stesso colore. Per sapere quali sono bisogna guardare ai *vincoli*, i quali avranno una forma del tipo  $C_k = (\{x_i, x_j\}, x_i \neq x_j)$ . Il grafo dei vincoli può essere rappresentato anche tramite:

- *Grafo primale*, il quale ha come nodi le variabili e come archi i vincoli.
- *Grafo duale*, il quale ha come nodi i vincoli e come archi le variabili comuni ai vincoli collegati.

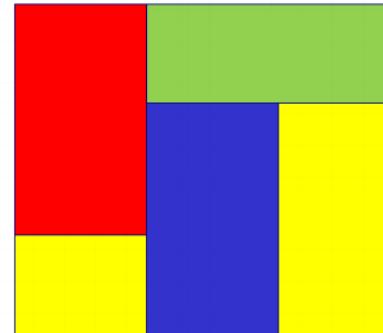


$$C1 = (\{x_1, x_2\}, x_1 \neq x_2)$$

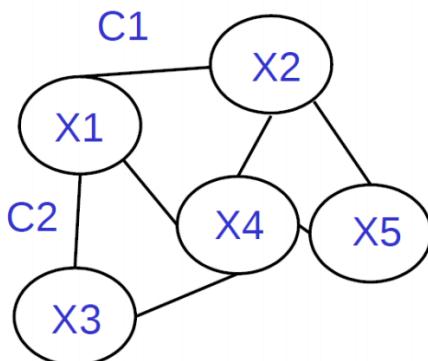
$$C2 = (\{x_1, x_3\}, x_1 \neq x_3)$$

...

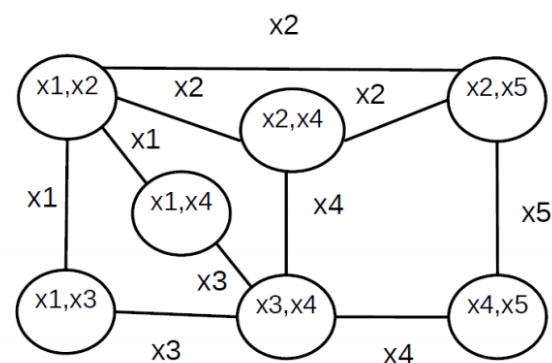
(a) Grafo dei vincoli



Solution



(c) Grafo primale



(d) Grafo duale

## 5.2 Tecniche risolutive per le reti a vincoli

Le tecniche risolutive per le reti a vincoli più utilizzate sono quelle basate su:

- *Inferenza*, cioè quelle che generano nuovi vincoli a partire da quelli già esistenti.
- *Ricerca*, cioè quelle che cercano la soluzione per tentativi.

### 5.2.1 Ricerca con backtracking

La *ricerca con backtracking* è una forma particolare di ricerca in profondità in cui l'idea è quella di scegliere una variabile e aggiungere un nuovo vincolo su quella variabile per tentare di risolvere il resto del problema. Tuttavia, l'assegnamento parziale di una variabile può violare altri vincoli del problema e quindi è proprio qui che si fa *backtracking* per tornare alla situazione precedente e procedere per un'altra via.

### 5.2.2 Inferenza e propagazione dei vincoli

Molte tecniche di *inferenza* utilizzano i vincoli per inferire quali coppie valore/variabile sono consistenti e quali no allo scopo di generare reti più forti con uno spazio di ricerca più piccolo e quindi di migliorare prestazionalmente la ricerca della soluzione. Infatti, grazie alla *propagazione* di nuovi vincoli è possibile ridurre lo spazio dei valori accettabili da una variabile, che a sua volta ridurrà lo spazio di un'altra variabile e così via.

### 5.2.3 Metodi di consistenza

Un'approssimazione dell'inferenza si può ottenere utilizzando i metodi di *consistenza*. Gli approcci possibili sono: consistenza di *nodo*, consistenza di *arco*, consistenza di *cammino* e *i*-consistenza.

**Consistenza di nodo.** Una variabile  $x_i$  del dominio  $D_i$  è *node consistent* se ogni suo valore all'interno del dominio soddisfa ogni vincolo unario.

$$\forall v \in D_i. \forall C = \{< x_i >, R_{x_i}\}. v \in R_{x_i}$$

In generale, se una variabile non è *node consistent*, si possono rimuovere tutti i suoi valori nel dominio che non soddisfano tutti i vincoli unari per quella variabile. Il nuovo dominio della variabile diventerà un dominio che conterrà solo valori che soddisfano tutti i vincoli unari, e quindi quelli esclusi non potranno far parte della soluzione.

$$D'_i = D_i \setminus \{v \mid \exists C = \{< x_i >, R_{x_i}\} \wedge v \notin R_{x_i}\}$$

**Consistenza di arco.** Date due variabili  $x_i$  e  $x_j$ ,  $x_i$  è *arc consistent* rispetto a  $x_j$  se vale che:

$$\forall a_i \in D_i. \exists a_j \in D_j. (a_i, a_j) \in R_{x_i, x_j}$$

Da questo si conclude che  $R_{x_i, x_j}$  è *arc consistent* se  $x_i$  è *arc consistent* rispetto a  $x_j$  e viceversa. Il nuovo dominio conterrà solo valori che soddisfano i vincoli binari, e quindi quelli esclusi non potranno far parte della soluzione.

Per verificare l'*arc consistency* si possono utilizzare gli algoritmi *AC-1* e *AC-3*. Tali algoritmi fanno uso della procedura di *Revise*, ovvero della rimozione dei valori in un dominio per i quali non vale l'*arc consistency* rispetto ad un secondo dominio. L'algoritmo *AC-1* ha complessità  $O(nek^3)$  dove  $n$  sono i nodi,  $e$  sono gli archi e  $k$  è il massimo numero di valori in un dominio. Inoltre, *AC-1* termina sempre e mantiene l'*arc consistency*, tranne se la rete è vuota in quanto non esisterà alcuna soluzione. L'algoritmo *AC-3*, invece, è un miglioramento di *AC-1* con complessità  $O(ek^3)$ .

---

**Require:**  $\mathcal{R} = \langle X, D, C \rangle$   
**Ensure:**  $\mathcal{R}'$  the loosest arc consistent network for  $\mathcal{R}$

```

repeat
    for all Pairs  $x_i, x_j$  that participate in a constraint do
        Revise $((x_i), x_j)$ ;
        Revise $((x_j), x_i)$ ;
    end for
until no domain is changed

```

---

(a) Algoritmo AC-1

---

**Require:**  $\mathcal{R} = \langle X, D, C \rangle$   
**Ensure:**  $\mathcal{R}'$  the loosest arc consistent network for  $\mathcal{R}$

```

for all pairs  $(x_i, x_j)$  that participate in a constraint  $R_{x_i, x_j} \in \mathcal{R}$  do
     $Q \leftarrow Q \cup \{(x_i, x_j), (x_j, x_i)\}$ 
end for
while  $Q \neq \{\}$  do
    pop  $(x_i, x_j)$  from  $Q$ 
    REVISE $((x_i), x_j)$ 
    if  $D_i$  changed then
         $Q \leftarrow Q \cup \{(x_k, x_i), k \neq i, k \neq j\}$ 
    end if
end while

```

---

(b) Algoritmo AC-3

Abbiamo detto che se un dominio è vuoto, allora si ha un problema inconsistente (non esiste alcuna soluzione). Al contrario, se tutti i domini sono non vuoti *non implica* che il problema è consistente. Infatti, il principale punto debole dell'*arc consistency* è che lavora solamente su vincoli binari e su vincoli con un singolo dominio.

**Consistenza di cammino.** È un'evoluzione della consistenza di arco che restringe i vincoli binari utilizzando vincoli impliciti che sono inferiti considerando *triplette di variabili*.

**i-consistenza.** La *i-consistenza* è un'estensione del concetto della consistenza di arco (e della consistenza di cammino) a reti di  $i - 1$  variabili. Diciamo che una rete è *i-consistent* se, per ogni insieme di  $i - 1$  variabili e ogni loro assegnamento consistente, è sempre possibile assegnare un valore consistente a ogni  $i$ -esima variabile. Inoltre, diciamo che una rete è *fortemente i-consistente* se è anche  $j$ -consistente per ogni  $j \leq i$ .

### 5.3 Strategie di ricerca per la propagazione dei vincoli

Una strategia di ricerca *standard* può consistere nell'applicare la ricerca a profondità limitata. In questo caso, uno stato sarebbe un assegnamento parziale e un'azione sarebbe l'aggiunta di *variabile = valore* all'assegnamento. Tuttavia, per un problema di soddisfacimento di vincoli (*CSP*) con  $n$  variabili con dominio di dimensione  $d$  si può notare che il fattore di ramificazione al livello  $l$  è pari a  $b = (n - l)d$ , e quindi l'albero generato avrebbe  $n! \cdot d^n$  foglie.

La formulazione del problema precedente è apparentemente ragionevole, ma in realtà è *ingenua* in quanto ha ignorato una proprietà fondamentale comune a tutti i *CSP*, ovvero la commutatività. Un problema è *commutativo* se l'ordine di applicazione di un qualsiasi insieme di azioni non ha effetto sul risultato finale.

### 5.3.1 Backtracking search

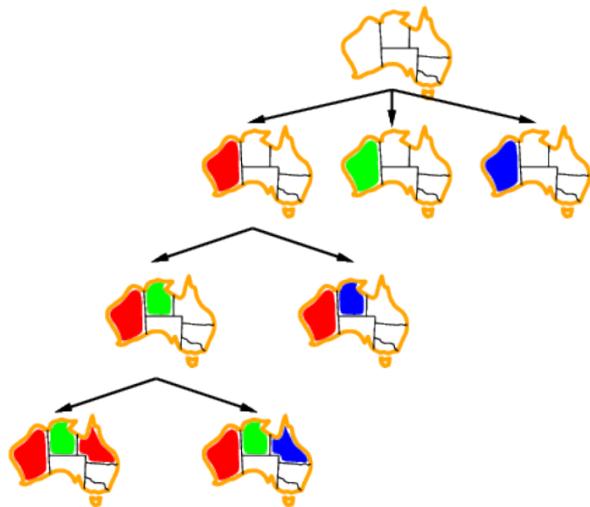
I CSP sono commutativi perché assegnando valori alle variabili si ottiene sempre lo stesso assegnamento parziale indipendentemente dall'ordine degli assegnamenti. Di conseguenza ci basta considerare *una sola* variabile in ogni livello dell'albero di ricerca. Con questa restrizione il numero di foglie risulterebbe pari a  $d^n$ . Indicheremo quindi con il termine *backtracking search* una ricerca in profondità che assegna valori a una variabile per volta e torna indietro quando non ci sono più valori legali da assegnare.

```

function Backtracking-Search(csp) returns solution or failure
    return Backtrack({ }, csp)
function Backtrack(assignment, csp) returns solution or failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← Inferences(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result ← Backtracking(assignment, csp)
                if result ≠ failure then
                    return result
                endif
            endif
        endif
    endfor
    remove {var = value} and inferences from assignment
return failure

```

(a) Algoritmo backtracking search



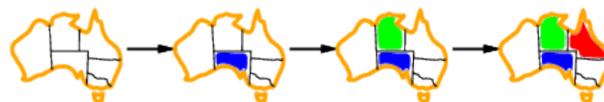
(b) Albero di ricerca (parziale)

### 5.3.2 Ordinamento di variabili

Per migliorare la *backtracking search* possiamo cambiare la strategia per scegliere la variabile da assegnare nel passo successivo. La strategia più semplice è quella di scegliere la prima variabile non assegnata, ma tale ordinamento statico raramente dà come risultato la ricerca più efficiente. L'idea è quindi quella di scegliere la variabile con il minor numero di valori legali. Tale euristica è chiamata *MRV* (Minimum Remaining Values) perché sceglie appunto la variabile per cui è maggiore la probabilità di arrivare presto ad un fallimento (variabile più vincolata). Tuttavia, l'euristica *MRV* non è di alcun aiuto nella scelta della prima regione da colorare, perché all'inizio ognuna di esse ha esattamente tre colori legali. In questo caso viene in aiuto l'euristica *DH* (Degree Heuristic), la quale cerca di ridurre il fattore di ramificazione delle scelte future scegliendo la variabile coinvolta nel maggior numero di vincoli con le altre variabili non assegnate.



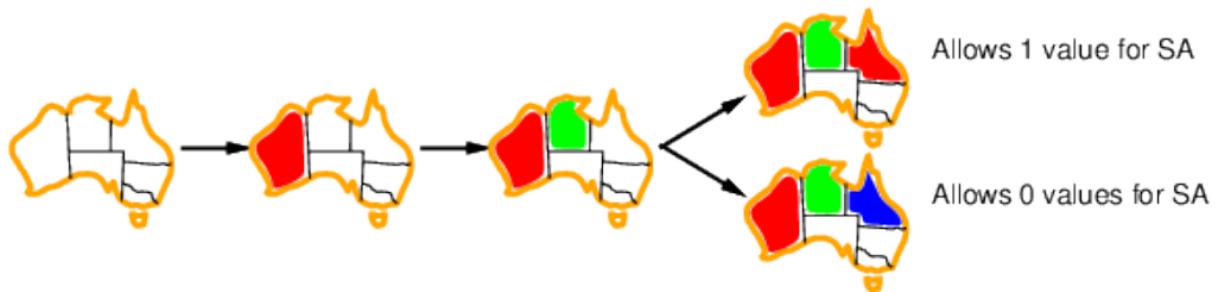
(a) Euristica MRV



(b) Euristica DH

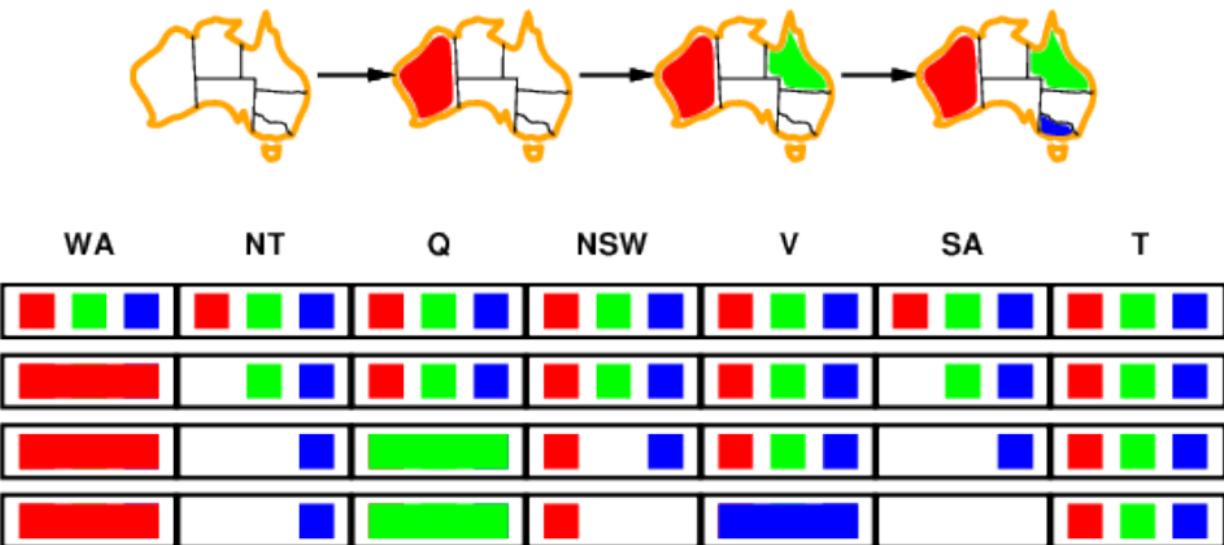
### 5.3.3 Ordinamento di valori

Una volta scelta una variabile, l'algoritmo deve anche decidere l'ordine con cui esaminare i suoi possibili valori. Sempre per migliorare la *backtracking search*, in alcuni casi può essere efficace l'euristica del *valore meno vincolante*, la quale predilige il valore che lascia più libertà alle variabili adiacenti sul grafo dei vincoli. In generale, l'euristica cerca sempre di lasciare la massima flessibilità ai successivi assegnamenti di variabili.



### 5.3.4 Alternanza di ricerca e inferenza

Finora abbiamo visto come inferire riduzioni del dominio di variabili *prima* di iniziare la ricerca, ma l'inferenza può risultare ancora più potente nel corso di una ricerca. Infatti, ogni volta che scegliamo un valore per una variabile, abbiamo un'opportunità del tutto nuova di inferire nuove riduzioni di dominio sulle variabili adiacenti. Una delle forme di inferenza più semplici è la cosiddetta verifica in avanti (*forward checking*) in cui, ogni volta che una variabile  $X$  viene assegnata, si stabilisce la *consistenza di arco* per essa, ovvero per ogni variabile non assegnata  $Y$  collegata a  $X$  da un vincolo si cancella dal dominio di  $Y$  ogni valore non consistente con quello scelto per  $X$ . La verifica in avanti rileva molte inconsistenze, ma non tutte. Il problema è che rende la variabile corrente *arc consistent*, ma non guarda avanti per rendere *arc consistent* tutte le altre.



### 5.3.5 Maintaining Arc Consistency

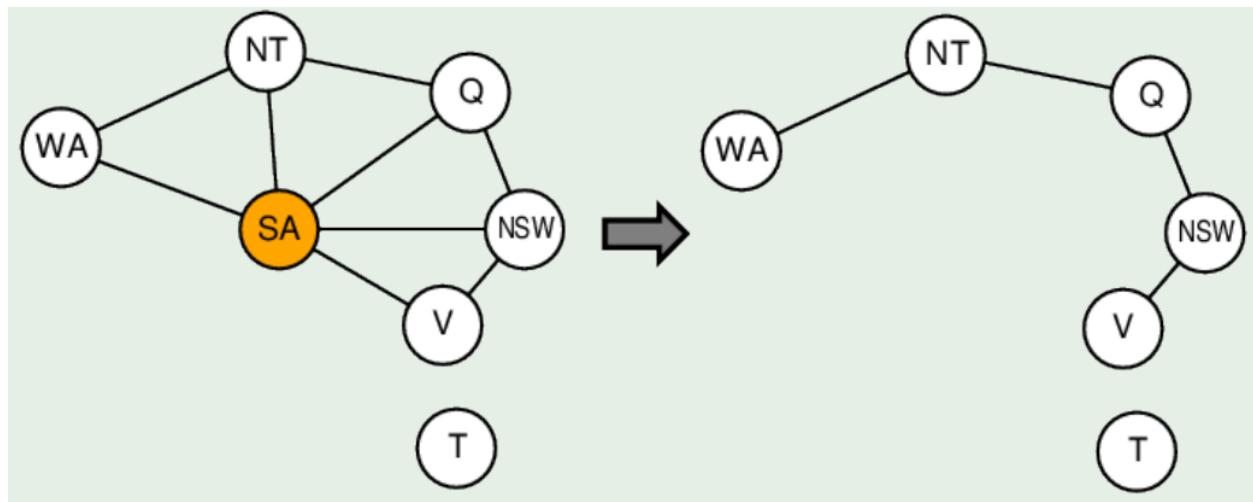
L'algoritmo *MAC* (Maintaining Arc Consistency) rileva le inconsistenze che la verifica in avanti non riesce a rilevare. Infatti, *MAC* è strettamente più potente della verifica in avanti perché quest'ultima procede come *MAC* sugli archi iniziali nella coda di *MAC*, ma non propaga ricorsivamente i vincoli quando si apportano modifiche ai domini delle variabili.

## 5.4 La struttura dei problemi

Per riuscire a trovare più rapidamente una soluzione, si può sfruttare la *struttura* stessa del problema (rappresentata dal grafo dei vincoli). L'unico modo in cui possiamo sperare di gestire la complessità del mondo reale è scomporlo in molti sotto-problemi. Infatti, se riuscissimo a scomporre il grafo dei vincoli fino a farlo diventare un albero, potremmo risolvere il *CSP* in un tempo che cresce linearmente con il numero delle variabili. Un grafo dei vincoli è un *albero* quando due variabili qualsiasi sono collegate da un solo cammino. Per *ridurre* un grafo dei vincoli ad un albero si può utilizzare un approccio, basato sulla rimozione dei nodi, che prevede che si assegnino dei valori ad alcune variabili in modo che quelle rimanenti formino un albero. L'algoritmo generale di questo approccio è il seguente:

1. Scegliere un sottoinsieme  $S$  delle variabili del *CSP* tale che il grafo dei vincoli diventi un albero dopo la rimozione di  $S$  (*cycle cutset*).
2. Per ogni possibile assegnamento delle variabili in  $S$  che soddisfa tutti i vincoli su  $S$ :
  - (a) Rimuovere dal dominio delle variabili rimanenti tutti i valori non consistenti con gli assegnamenti in  $S$ .
  - (b) Se il *CSP* risultante ha una soluzione, restituirla insieme all'assegnamento per  $S$ .

Se il *cycle cutset*  $S$  ha dimensione  $c$ , il tempo di esecuzione totale è  $O(d^c \cdot (n - c)d^2)$ . Se il grafo è quasi un albero  $c$  sarà piccolo, e il risparmio di tempo rispetto al backtracking risulterà enorme.



### 5.4.1 Scomposizione ad albero

Un processo alternativo per ridurre un grafo dei vincoli ad un albero è quello basato sulla costruzione di una scomposizione ad albero (*tree decomposition*) del grafo dei vincoli in un insieme di sotto-problemi collegati. Secondo questo approccio, poi, ogni sotto-problema viene risolto indipendentemente e le soluzioni vengono combinate insieme. Il processo di tree decomposition viene quindi applicato quando si vuole passare da reti cicliche a *reti acicliche*, in quanto l'inferenza su queste ultime è molto più efficiente.

### 5.4.2 Concetto di ipergrafo

Un *ipergrafo* è una struttura  $H = (V, S)$  dove  $V$  è un insieme di nodi e  $S$  è un insieme di iperarchi ossia un insieme  $S = \{S_1, \dots, S_k\}$  tale che  $S_i \subseteq V$ . Il *grafo primale* di un ipergrafo sarà quindi un grafo in cui: i nodi sono i vertici; due nodi vengono collegati da un arco solo se appaiono nello stesso iperarco. Il *grafo duale* di un ipergrafo, invece, sarà un grafo in cui: i nodi sono gli iperarchi; due nodi vengono collegati da un arco solo se condividono almeno un vertice (gli archi verranno quindi etichettati con i vertici condivisi tra i nodi).

### 5.4.3 Rappresentazione di una rete a vincoli

Ogni rete a vincoli può essere rappresentata tramite un ipergrafo. Infatti, per la rete a vincoli  $\mathcal{R} = \{X, D, C\}$  con  $C = \{R_{S_1}, \dots, R_{S_r}\}$  il relativo ipergrafo è  $\mathcal{H}_{\mathcal{R}} = (X, H)$  con  $H = \{S_1, \dots, S_r\}$ . Inoltre, il grafo duale è  $\mathcal{H}_{\mathcal{R}}^d = (H, E)$  con  $\langle S_i, S_j \rangle \in E \iff S_i \cap S_j \neq \emptyset$ .

### 5.4.4 Reti acicliche

I concetti principali relativi alle reti acicliche sono i seguenti.

- *Sottografo*: si definisce sottografo di un grafo  $G = (V, E)$  il grafo  $G' = (V, E')$  tale per cui  $E' \subseteq E$ .
- *Running intersection property*: dato il grafo duale  $G$  di un ipergrafo, il suo sottografo  $G'$  soddisfa la running intersection property se, tra due nodi qualsiasi di  $G'$  che condividono una variabile, esiste un percorso formato da archi etichettati almeno con la variabile condivisa.
- *Join-graph*: si definisce join-graph un sottografo  $G'$ , di un grafo duale  $G$  di un ipergrafo, che soddisfa la running intersection property.
- *Join-tree*: si definisce join-tree un join-graph aciclico.
- *Iperalbero*: si definisce iperalbero un ipergrafo il cui grafo duale ha un join-tree.

Una *rete aciclica* viene quindi definita come una rete il cui ipergrafo è un iperalbero.

### 5.4.5 Risolvere una rete aciclica

Per risolvere una rete aciclica è sufficiente risolvere l'albero mediante l'*algoritmo Tree Solver*. Tuttavia, è necessario prima sapere come fare a determinare se una rete è aciclica. I principali metodi per eseguire questa verifica sono la dual-based e la primal-based recognition.

L'idea alla base della *dual-based recognition* è che se un ipergrafo ammette un join-tree, allora ogni maximum spanning tree del suo grafo duale è un join-tree. La *procedura* da applicare è quindi la seguente:

- Costruire il grafo duale dell'ipergrafo.
- Determinare un maximum spanning tree usando come peso il numero delle variabili condivise.
- Controllare se l'iperalbero è un join-tree.

L'idea alla base della *primal-based recognition*, invece, è che un ipergrafo ammette un join-tree se valgono le proprietà di:

- *Cordalità*, per cui un grafo primale si dice cordale se ogni ciclo di lunghezza almeno pari a 4 ha una corda (ovvero un arco che collega due vertici non adiacenti in un ciclo).
- *Conformalità*, per cui un grafo primale si dice conforme ad un ipergrafo se esiste una corrispondenza uno a uno tra clique massimali e scope dei vincoli.

La verifica di queste due proprietà viene eseguita in maniera efficiente se viene usato un *ordine di massima cardinalità*. La *procedura* da applicare è quindi la seguente:

- Costruire un ordine di massima cardinalità.
- Testare se il grafo è cordale, ovvero se nell'ordine di massima cardinalità ogni vertice e i suoi predecessori formano una clique.
- Testare se il grafo è conforme estraendo la clique massimale.

```
Require: An Acyclic Constraint Network  $\mathcal{R}$ . A join-tree  $T$  of  $\mathcal{R}$ 
Ensure: Determine consistency and generate a solution
 $d = \{R_1, \dots, R_r\}$  order induced by  $T$  (from root to leaves)
for all  $j = r$  to 1 and for all edges  $< j, k >$  in the  $T$  with  $k < j$  do
   $R_k \leftarrow \pi_{R_k}(R_k \bowtie R_j)$ 
  if we find the empty relation then
    EXIT and state the problem has NO SOLUTION
  end if
end for
Select a tuple in  $R_1$ 
for all  $i = 2$  to  $r$  do
  Select a tuple that is consistent with all previous assigned tuples
   $R_1, \dots, R_{i-1}$ 
end for
return The problem is CONSISTENT return the selected SOLUTION
```

(a) Algoritmo Tree Solver

```
Require: A hypergraph  $\mathcal{H}_{\mathcal{R}} = (X, S)$  of a constraint network  $\mathcal{R} = (X, D, C)$ 
Ensure: A join tree  $T = (S, E)$  of  $\mathcal{H}_{\mathcal{R}}$  if  $\mathcal{R}$  is acyclic
 $T = (S, E) \leftarrow$  generate a maximum spanning tree of the weighted dual constraint graph of  $\mathcal{R}$ 
for all couples  $u, v$  where  $u, v \in S$  do
  if the unique path connecting them in  $T$  does not satisfy the running intersection property then
    EXIT ( $\mathcal{R}$  is not acyclic)
  end if
end for
return  $\mathcal{R}$  is acyclic and  $T$  is a join tree
```

(b) Algoritmo DualAcyclicity

```
Require: A constraint network  $\mathcal{R} = (X, D, C)$  and its primal graph  $G$ 
Ensure: A join tree  $T = (S, E)$  of  $\mathcal{H}_{\mathcal{R}}$  if  $\mathcal{R}$  is acyclic
Build  $d^{\text{max}} = \{x_1, \dots, x_r\}$  max-cardinality order
Test Chordality using  $d^{\text{max}}$ 
for all  $i = r$  to 1 do
  if the ancestors of  $x_i$  are not all connected then
    EXIT ( $\mathcal{R}$  is not acyclic)
  end if
end for
Test Conformality using  $d^{\text{max}}$ : Let  $\{C_1, \dots, C_r\}$  be the maximal cliques (a node and all its ancestors)
for all  $i = r$  to 1 do
  if  $C_i$  corresponds to scope of one constraints  $C$  then
    else
      EXIT ( $\mathcal{R}$  is not acyclic)
    end if
  end for
Create a join tree of the cliques (e.g., create a maximum spanning tree were weights are number of shared variables)
return  $\mathcal{R}$  is acyclic and  $T$  is a join tree
```

(c) Algoritmo PrimalAcyclicity

## 5.4.6 Clustering

La procedura del *clustering* consiste nell'accoppare vincoli per ottenere una struttura ad albero e cercare quindi di risolvere in maniera più efficiente la rete. L'approccio più utilizzato è il cosiddetto *join-tree clustering*, in cui per generare un join-tree si usa il seguente metodo:

1. Si sceglie un ordine delle variabili.
2. Si crea un grafo indotto per il quale valga la running intersection property:
  - Calcolando gli antenati di ogni variabile (processandole dall'ultima alla prima).
  - Verificando che ogni variabile formi una clique.

3. Si crea un join-tree:

- Identificando tutte le clique massimali  $C_1, \dots, C_t$  nel grafo indotto.
  - Creando una struttura ad albero con le clique (ovvero un maximum spanning tree dove i pesi rappresentano il numero delle variabili condivise).
4. Si allocano i vincoli ad ogni clique che contiene il relativo scope (sotto-problema  $P_i$  associato a  $C_i$ ).
5. Si risolve ogni sotto-problema  $P_i$  ricavando il relativo insieme di soluzioni  $R'_i$ .
6. Si determina la soluzione globale  $C' = \{R'_1, \dots, R'_t\}$  applicando l'algoritmo Tree Solver.

La *complessità* del join-tree clustering è dominata dal calcolo dell'insieme di soluzioni di ogni sotto-problema, il quale è esponenziale nella dimensione della clique. L'ordine delle variabili usato per calcolare le clique è quindi fondamentale, ma trovare l'ordine migliore è estremamente difficile.

## 5.5 Constraint optimisation problems

Per risolvere un COP (constraint optimisation problem) si può ricorrere alle reti di costo. Si parla di reti di costo (*cost network*) quando si ha a che fare con massimizzazioni o minimizzazioni di funzioni di costo. Infatti, una rete di costo è una rete a vincoli con una *funzione di costo* applicata su di essa e definita come  $F(\bar{a}) = \sum_{j=1}^l F_j(\bar{a})$  dove  $\bar{a} = \{a_1, \dots, a_n\}$  sono gli assegnamenti delle variabili  $X = \{x_1, \dots, x_n\}$ . Lo scopo sarà quindi quello di trovare un'allocazione  $\bar{a}^*$  di tali variabili che massimizzi o minimizzi la funzione di costo, ovvero  $\bar{a}^* = \max_{\bar{a}} F(\bar{a})$  oppure  $\bar{a}^* = \min_{\bar{a}} F(\bar{a})$ .

### 5.5.1 Metodi risolutivi per le reti di costo

Le reti di costo si possono risolvere tramite:

- *Branch and Bound*, ovvero tramite un approccio simile alla ricerca con backtracking.
- *Programmazione dinamica*, ovvero tramite approcci di inferenza.

### 5.5.2 Branch and Bound

Si ricorda che l'idea di base del *backtracking* è quella di trovare progressivamente la soluzione migliore sulla base di una soluzione parziale, e di sfruttare tale soluzione parziale per il taglio (pruning) dell'albero di ricerca. Se per risolvere un COP dobbiamo, ad esempio, massimizzare una funzione di costo allora dovremo mantenere un *lower bound* per riuscire a tagliare l'albero di ricerca. Infatti, vogliamo che tutte le soluzioni minori del lower bound siano scartate, in quanto non ha senso proseguire lungo un valore che è minore di quello massimo trovato finora.

### 5.5.3 Bucket Elimination

L'idea di base della *programmazione dinamica* è quella di costruire la soluzione di un problema in modo incrementale partendo dalle soluzioni dei suoi sotto-problemi (si sfrutta la struttura del problema). L'algoritmo di *Bucket Elimination* è una tecnica specifica di programmazione dinamica usata per risolvere i COP.

I concetti alla base dell'algoritmo sono i seguenti passaggi.

1. *Bucket partition*: si assegnano i vincoli al bucket, il quale non è altro che un insieme di vincoli che fanno riferimento ad una variabile.
2. *Bucket processing*: si processa il bucket dall'ultima variabile alla prima secondo un certo ordinamento delle variabili.
3. *Value propagation*: si calcola la tupla ottimale propagando i valori dalla prima variabile all'ultima.

La *complessità* dell'algoritmo di Bucket Elimination è influenzata dalla scelta dell'ordinamento delle variabili; infatti, essa è esponenziale nella dimensione del bucket più grande.

- Consider the general cost network  $\mathcal{CN}$
- Consider the order  $d = \{a, c, b, f, d, g\}$
- We want to compute  $\max_{a,c,b,f,d,g} F_0(a) + F_1(a, b) + F_2(a, c) + F_3(b, c, f) + F_4(a, d, b) + F_5(f, g)$
- We can manipulate the formula to push maximisation where needed

(a) Bucket Elimination su una rete di costo

- Process last buckets:  $B_g = \{F_5(f, g)\}$
- $H^g(f) = \max_g F_5(f, g)$ , place  $H^g(f)$  in bucket  $B_f$
- Process bucket:  $B_d = \{F_4(d, b, a)\}$ ,  
 $H^d(b, a) = \max_d F_4(d, b, a)$ , place  $H^d(b, a)$  in  $B_b$
- Process bucket:  $B_f = \{F_3(f, b, c), H^g(f)\}$ ,  
 $H^f(b, c) = \max_f (F_3(f, b, c) + H^g(f))$ , place  $H^f(b, c)$  in  $B_b$
- ...
- Process bucket:  $B_a = \{F_0(a), H^c(a)\}$ ,  
 $M = \max_a (F_0(a) + H^c(a))$

(c) Bucket processing

- Consider the general cost network  $\mathcal{CN}$
- Consider the order  $d = \{a, c, b, f, d, g\}$
- Buckets:  $\{B_a, B_c, B_b, B_f, B_d, B_g\}$
- Partition:  $B_g = \{F_5(f, g)\}$ ,  $B_d = \{F_4(a, d, b)\}$ ,  $B_f = \{F_3(b, c, f)\}$ ,  $B_b = \{F_1(b, a)\}$ ,  $B_c = \{F_2(c, a)\}$ ,  $B_a = \{F_0(a)\}$

(b) Bucket partition

- Compute  $\bar{a}_1^* = \{a^*\}$   $a^* = \arg \max_a (F_0(a) + H^c(a))$
- Compute  $\bar{a}_2^* = \{a^*, c^*\}$   
 $c^* = \arg \max_c (F_2(a^*, c) + H^b(a^*, c))$
- ...
- Compute  $\bar{a}_6^* = \{a^*, c^*, b^*, f^*, d^*, g^*\}$   
 $g^* = \arg \max_g (F_5(f^*, g))$

(d) Value propagation

# Capitolo 6

## Incertezza

### 6.1 Agire in condizioni di incertezza

La teoria della probabilità è il modo migliore di ragionare in condizioni di incertezza. Tale *incertezza* può sorgere a causa di pigrizia e ignoranza, ed è infatti una caratteristica inevitabile dei mondi complessi, dinamici o non del tutto accessibili. La presenza di incertezza impedisce di effettuare molte delle semplificazioni possibili grazie all'inferenza deduttiva.

### 6.2 Notazione base della teoria della probabilità

Le *probabilità* esprimono l'incapacità dell'agente di arrivare ad una decisione definitiva sul valore di verità di una formula e servono a riassumere le sue credenze. Gli enunciati base includono le probabilità a priori e le probabilità condizionate di proposizioni semplici e complesse.

#### 6.2.1 Probabilità a priori

La *probabilità a priori* (o non condizionata) associata con una proposizione  $a$  è il grado di credenza che le viene accordato in assenza di ogni altra informazione; viene indicata con  $P(a)$ . Una *distribuzione di probabilità* viene definita quando si considerano le probabilità di tutti i possibili valori di una variabile casuale. Se invece vogliamo denotare le probabilità di tutte le combinazioni di valori di un insieme di variabili casuali, allora definiremo una distribuzione di probabilità *congiunta*.

#### 6.2.2 Probabilità condizionate

Una volta che l'agente ha ottenuto delle prove precedentemente sconosciute riguardo alle variabili casuali del dominio, le probabilità a priori non sono più applicabili. Al posto di queste ultime si devono infatti usare le *probabilità condizionate* (o a posteriori). La notazione è  $P(a|b)$  (con  $a$  e  $b$  proposizioni qualsiasi) e indica la probabilità di  $a$  posto che tutto quello che sappiamo è  $b$ . Infine, le probabilità condizionate possono essere definite partendo da quelle non condizionate:

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

con  $P(b) \neq 0$ .

### 6.3 Inferenza basata su distribuzioni congiunte complete

La distribuzione di probabilità *congiunta completa* specifica la probabilità di ogni assegnamento completo di valori alle variabili casuali. Quando tale distribuzione è disponibile, si può rispondere ad un'interrogazione semplicemente sommando le probabilità degli eventi atomici che corrispondono alle proposizioni della query. Solitamente però la distribuzione congiunta completa risulta troppo grande per essere creata o utilizzata in forma esplicita.

Start with the joint distribution:

	toothache	$\neg$ toothache		
	catch	$\neg$ catch	catch	
cavity	.108	.012	.072	.008
$\neg$ cavity	.016	.064	.144	.576

For any proposition  $\phi$ , sum the atomic events where it is true:

$$P(\phi) = \sum_{\omega: \omega \models \phi} P(\omega)$$

$$P(\text{toothache}) = 0.108 + 0.012 + 0.016 + 0.064 = 0.2$$

## 6.4 Indipendenza

Le proposizioni  $a$  e  $b$  sono *indipendenti* se e solo se  $P(a|b) = P(a)$  oppure  $P(b|a) = P(b)$  oppure  $P(a \wedge b) = P(a)P(b)$ . L'*indipendenza assoluta* tra sottoinsiemi di variabili casuali può consentire di fattorizzare la distribuzione congiunta completa in distribuzioni congiunte più piccole. Questo può ridurre drasticamente la complessità, ma raramente si verifica nella pratica.

### 6.4.1 La regola di Bayes e il suo utilizzo

La *regola di Bayes* permette di calcolare probabilità sconosciute partendo da probabilità condizionate note, solitamente in direzione casuale. In presenza di molti elementi di prova, l'applicazione della regola di Bayes avrà in generale gli stessi problemi di scalabilità dell'uso diretto della distribuzione congiunta completa. L'*equazione* della regola di Bayes è:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

### 6.4.2 Indipendenza condizionale

L'*indipendenza condizionale* dovuta a relazioni casuali dirette nel dominio può permettere di fattorizzare la distribuzione congiunta completa in distribuzioni condizionate più piccole. Il *modello di Bayes ingenuo* assume come ipotesi l'indipendenza condizionale di tutte le variabili effetto data una singola variabile causa e cresce linearmente con il numero di effetti. In tal caso, quindi, la distribuzione congiunta completa può essere scritta come:

$$P(Causa, Effetto_1, \dots, Effetto_n) = P(Causa) \prod_i P(Effetto_i | Causa)$$

## 6.5 Il mondo del wumpus

Un agente del *mondo del wumpus* può calcolare le probabilità degli aspetti del mondo non osservati e avvalersene per prendere decisioni migliori di quelle di un agente puramente logico.

Performance measure

gold +1000, death -1000

-1 per step, -10 for using the arrow

Environment

Squares adjacent to wumpus are smelly

Squares adjacent to pit are breezy

Glitter iff gold is in the same square

Shooting kills wumpus if you are facing it

Shooting uses up the only arrow

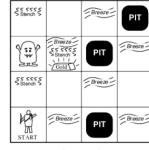
Grabbing picks up gold if in same square

Releasing drops the gold in same square

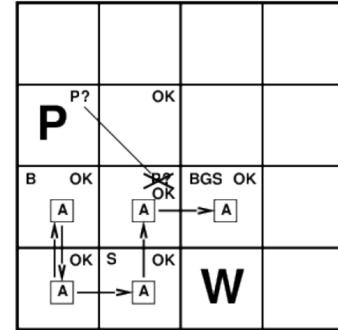
Actuators Left turn, Right turn,

Forward, Grab, Release, Shoot

Sensors Breeze, Glitter, Smell

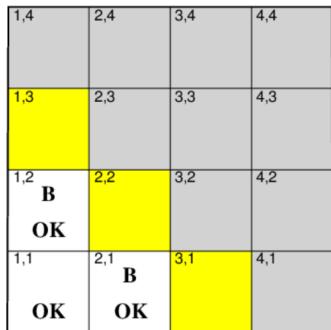


(a) Descrizione del mondo del wumpus

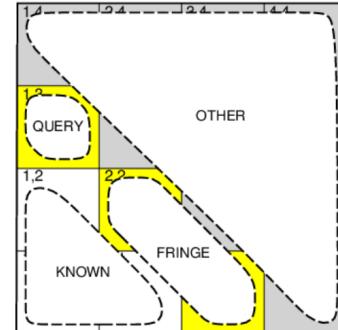


(b) Esempio di esplorazione

Nella figura (a) sottostante si può notare che, dopo aver rilevato uno spostamento d'aria sia in [1, 2] che in [2, 1], l'agente si blocca in quanto non c'è più alcuna stanza sicura da esplorare. L'agente dovrà quindi fare un'interrogazione, ad esempio per [1, 3], suddividendo le stanze come mostrato nella figura (b) sottostante.



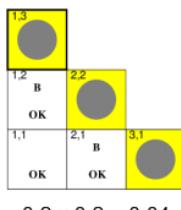
(a)



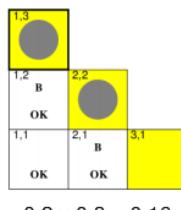
(b)

A seguito dell'interrogazione si otterranno i seguenti *modelli* consistenti per le variabili di frontiera  $P_{2,2}$  e  $P_{3,1}$  (con l'indicazione di  $P$ (frontiera) sotto ad ogni modello):

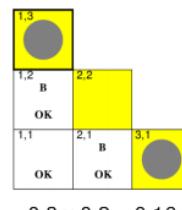
- (a) Tre modelli con  $P_{1,3} = \text{true}$  e la presenza di due o tre pozzi.
- (b) Due modelli con  $P_{1,3} = \text{false}$  e la presenza di uno o due pozzi.



$$0.2 \times 0.2 = 0.04$$

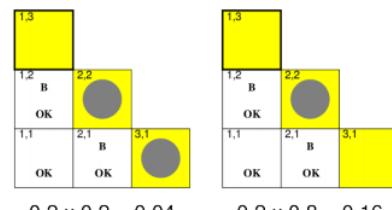


$$0.2 \times 0.8 = 0.16$$

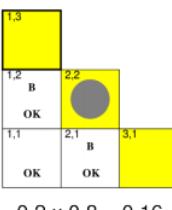


$$0.8 \times 0.2 = 0.16$$

(a)



$$0.2 \times 0.2 = 0.04$$



$$0.2 \times 0.8 = 0.16$$

(b)

# Capitolo 7

## Problemi di decisione sequenziali

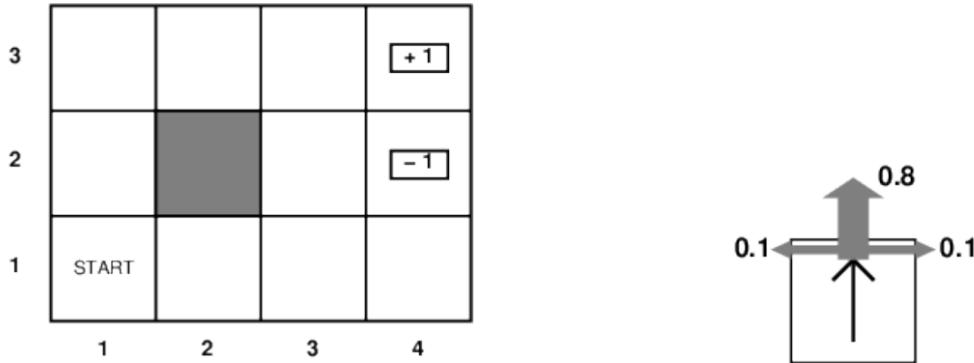
### 7.1 Decisioni complesse

Quando dobbiamo prendere *decisioni complesse*, occorre sfruttare la nostra conoscenza del mondo per prendere tali decisioni anche quando le loro conseguenze sono incerte, e le eventuali ricompense potrebbero non essere disponibili finché non sono state compiute diverse altre azioni.

#### 7.1.1 Markov Decision Processes

I problemi di decisione sequenziali in ambienti incerti, chiamati anche *processi decisionali di Markov* (MDP), sono definiti da:

- Un *modello di transizione*, che specifica gli esiti probabilistici delle azioni.
- Una *funzione di ricompensa*, che specifica la ricompensa associata ad ogni stato.



States  $s \in S$ , actions  $a \in A$

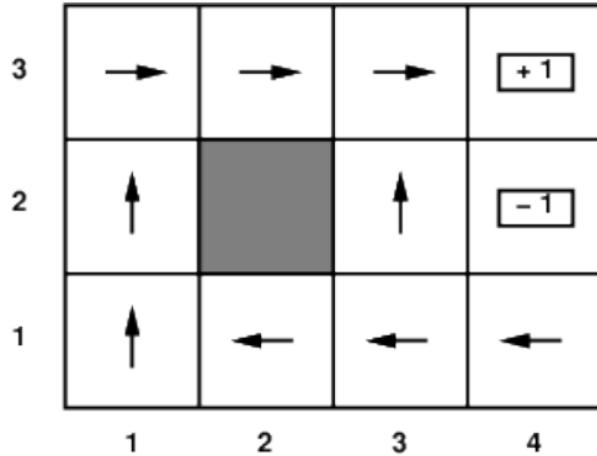
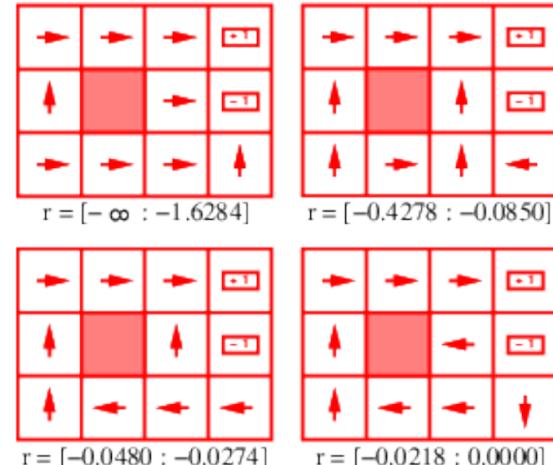
Model  $T(s, a, s') \equiv P(s'|s, a)$  = probability that  $a$  in  $s$  leads to  $s'$

Reward function  $R(s)$  (or  $R(s, a)$ ,  $R(s, a, s')$ )

$$= \begin{cases} -0.04 & \text{(small penalty) for nonterminal states} \\ \pm 1 & \text{for terminal states} \end{cases}$$

### 7.1.2 La necessità delle politiche

L'utilità di una sequenza di stati è pari alla somma di tutte le ricompense degli stati che contiene, con un possibile sconto dipendente dal tempo. La soluzione di un MDP sarà quindi una *politica* (policy) che associa una decisione ad ogni stato raggiungibile dall'agente. Una politica ottima massimizza l'utilità delle sequenze di stati attraversati durante la sua esecuzione, ovvero trova la migliore azione per ogni possibile stato.

(a) Policy ottima con  $R(s) = -0,04$ (b) Policy ottime per quattro diversi intervalli di  $R(s)$ 

## 7.2 Iterazione dei valori

L'utilità di uno stato corrisponde all'utilità attesa delle sequenze di stati attraversati eseguendo una politica ottima partendo da esso. L'algoritmo di *iterazione dei valori* per la soluzione degli MDP funziona risolvendo iterativamente le equazioni che collegano l'utilità di ogni stato a quella dei suoi vicini.

Initialize array  $v$  arbitrarily (e.g.,  $v(s) = 0$  for all  $s \in \mathcal{S}^+$ )

Repeat

$\Delta \leftarrow 0$

For each  $s \in \mathcal{S}$ :

$temp \leftarrow v(s)$

$v(s) \leftarrow \max_a \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')]$

$\Delta \leftarrow \max(\Delta, |temp - v(s)|)$

until  $\Delta < \theta$  (a small positive number)

Output a deterministic policy,  $\pi$ , such that

$$\pi(s) = \arg \max_a \sum_{s'} p(s'|s, a) [r(s, a, s') + \gamma v(s')]$$

(a) Algoritmo di iterazione dei valori

3	0.812	0.868	0.912	+1
2	0.762		0.660	-1
1	0.705	0.655	0.611	0.388
	1	2	3	4

(b) Utilità degli stati e policy

### 7.3 Iterazione delle politiche

L'iterazione delle politiche alterna il calcolo delle utilità degli stati rispetto alla politica corrente (policy evaluation) con il miglioramento della politica in base alle utilità correnti (policy improvement). L'algoritmo ripete policy evaluation e policy improvement fino a quando non è più possibile migliorare, garantendo quindi l'ottimalità della politica.

1. Initialization  
 $v(s) \in \mathbb{R}$  and  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$
2. Policy Evaluation  
 Repeat
 

```

 $\Delta \leftarrow 0$ 
For each  $s \in \mathcal{S}$ :
 $temp \leftarrow v(s)$ 
 $v(s) \leftarrow \sum_{s'} p(s'|s, \pi(s)) [r(s, \pi(s), s') + \gamma v(s')]$ 
 $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$ 
until  $\Delta < \theta$  (a small positive number)
    
```
3. Policy Improvement  
 $policy\text{-stable} \leftarrow true$   
 For each  $s \in \mathcal{S}$ :
 

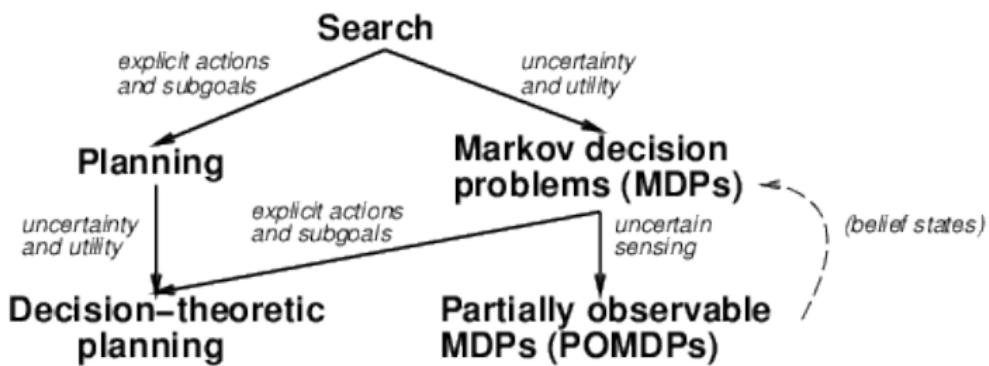
```

 $temp \leftarrow \pi(s)$ 
 $\pi(s) \leftarrow \arg \max_a \sum_{s'} p(s'|s, a) [r(s, a, s') + \gamma v(s')]$ 
If  $temp \neq \pi(s)$ , then  $policy\text{-stable} \leftarrow false$ 
    
```

 If  $policy\text{-stable}$ , then stop and return  $v$  and  $\pi$ ; else go to 2

### 7.4 MDP parzialmente osservabili

Gli MDP parzialmente osservabili (POMDP) sono molto più difficili da risolvere degli MDP. Infatti, la policy ottima in un POMDP è una funzione  $\pi(b)$  dove  $b$  è lo stato-credenza, in quanto l'agente non sa in quale stato si trova. È quindi possibile convertire un POMDP in un MDP nello spazio continuo degli stati-credenza. Il comportamento ottimo in un POMDP include la *raccolta di informazioni* per ridurre l'incertezza e prendere decisioni migliori.



# Capitolo 8

## Apprendimento per rinforzo

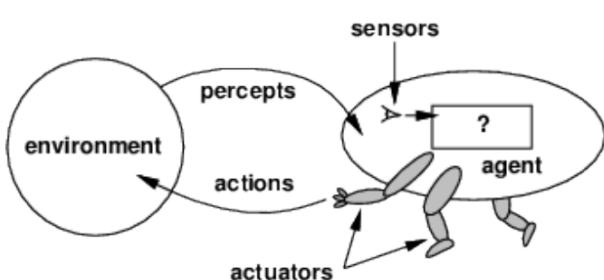
### 8.1 Introduzione

Il problema dell'*apprendimento per rinforzo* riguarda come un agente può imparare a comportarsi efficacemente in un ambiente sconosciuto, date solo le sue percezioni e occasionali ricompense. L'apprendimento per rinforzo può essere considerato un microcosmo per l'intero problema dell'IA, ma per facilitare i progressi viene studiato in diverse configurazioni semplificate.

#### 8.1.1 Progetto globale

Il *progetto globale* dell'agente determina il tipo di informazione che deve essere appresa. I progetti principali sono quelli:

- *Basati su modello* (model-based), che utilizzano un modello e una funzione di utilità.
- *Privi di modello* (model-free), che utilizzano una funzione azione-valore.



(a) Reinforcement learning (idea di base)

- ◊ Model Based vs. Model Free approaches
- ◊ GOAL: compute expected age for this class.
- ◊ Given probability distribution of ages:  $\mathbb{E}[A] = \sum_a P(a) \cdot a$
- **Model Based:** estimate  $\hat{P}(a)$
- $\hat{P}(a) = \frac{\text{num}(a)}{N}$
- $\mathbb{E}[A] \approx \sum_a \hat{P}(a) \cdot a$
- where  $\text{num}(a)$  is the number of students that have age  $a$
- works because we learn the right model
- **Model Free:** no estimate
- $\mathbb{E}[A] \approx \frac{1}{N} \sum_i a_i$
- where  $a_i$  is the age value of person  $i$
- works because samples appear with right frequency

(b) Model-based vs model-free

## 8.2 Metodi model-based

I metodi *model-based* apprendono un modello e una funzione ricompensa dalle osservazioni e quindi utilizzano l'iterazione dei valori o delle politiche per ottenere le utilità o una politica ottima.

---

### Algorithm 1 Model Based approach to RL

---

**Require:**  $A, S, S_0$

**Ensure:**  $\hat{T}, \hat{R}, \hat{\pi}$

Initialize  $\hat{T}, \hat{R}, \hat{\pi}$

**repeat**

    Execute  $\hat{\pi}$  for a learning episode

    Acquire a sequence of tuples  $\langle(s, a, s', r)\rangle$

    Update  $\hat{T}$  and  $\hat{R}$  according to tuples  $\langle(s, a, s', r)\rangle$

    Given current dynamics compute a policy (e.g., VI or PI)

**until** termination condition is met

---

## 8.3 Metodi model-free

I metodi *model-free* aggiornano le stime di utilità in modo che si accordino con quelle degli stati successori. Usare un modello appreso per generare pseudo-esperienze, comunque, può velocizzare l'apprendimento.

### 8.3.1 Q-Learning

Il *Q-Learning* non richiede alcun modello, né per l'apprendimento né in fase di selezione delle azioni. Questo semplifica il problema ma potenzialmente riduce la capacità di apprendere in ambienti complessi, perché l'agente non può simulare i risultati di possibili corsi d'azione.

### 8.3.2 Sfruttamento ed esplorazione

Q-Learning è un algoritmo *off-policy*, ovvero impara la policy ottima senza seguirla ma arriva a risultati sub-ottimi, in quanto il modello appreso non coincide con l'ambiente vero (ciò che è ottimo nel modello potrebbe non esserlo nell'ambiente). Infatti, siccome l'agente non conosce l'ambiente reale, non può riuscire a calcolare l'azione realmente ottima. Quello che l'agente non considera è che le azioni non si limitano a fornire ricompense in base al modello appreso corrente, ma contribuiscono anche ad apprendere il modello vero influenzando le percezioni ricevute. Un agente deve quindi gestire il compromesso tra lo *sfruttamento* (exploitation), che massimizza la ricompensa calcolata in base alle stime correnti di utilità, e l'*esplorazione* (exploration), che massimizza il benessere a lungo termine. In generale, non è possibile trovare una soluzione esatta al problema dell'esplorazione, ma esistono semplici euristiche che permettono di ottenere risultati accettabili (ad esempio *e-greedy* e *soft-max*).

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Repeat (for each step of episode):
        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
        Take action  $A$ , observe  $R, S'$ 
         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
         $S \leftarrow S'$ ;
    until  $S$  is terminal

```

(a) Algoritmo Q-Learning

◇ Key point: to guarantee convergence to optimal we need to explore every state-action pairs sufficiently often in the long run.

◇ Main methods used in practice:

- $\epsilon$ -greedy:
  - choose greedily most of the time (probability  $1-\epsilon$ ) and choose randomly with probability  $\epsilon$
- soft-max (or Boltzmann)
  - choose action  $a$  with probability  $p(a) = \frac{e^{Q(s,a)/T}}{\sum_{a'} e^{Q(s,a')/T}}$
  - $T$  is a parameter (often called temperature)
  - high  $T \rightarrow$  all actions are equiprobable (we explore more)
  - low  $T \rightarrow$  greater difference in selection probability towards actions with highest  $Q$  (we exploit more)

(b) Exploration vs exploitation

### 8.3.3 SARSA

*SARSA* è un algoritmo model-free alternativo a Q-Learning caratterizzato dal fatto che calcola la prossima azione basandosi sulla policy (algoritmo *on-policy*). Confrontiamo brevemente i due algoritmi:

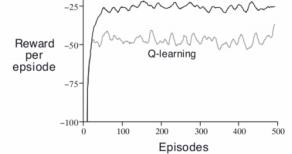
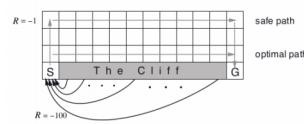
- *Q-Learning* impara la policy ottima ma a volte fallisce a causa della selezione dell'azione con  $\epsilon$ -greedy.
- *SARSA*, essendo on-policy, ha migliori performance on-line.

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Repeat (for each step of episode):
        Take action  $A$ , observe  $R, S'$ 
        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$ 
         $S \leftarrow S'; A \leftarrow A'$ ;
    until  $S$  is terminal

```

(a) Algoritmo SARSA



(b) Q-Learning vs SARSA

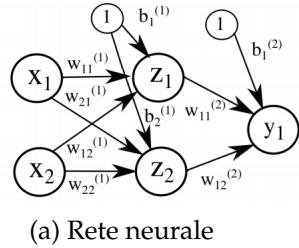
## 8.4 Deep reinforcement learning

Il *deep reinforcement learning* è una delle aree di ricerca più attive nel campo dell'IA. In particolare, le applicazioni alla robotica sono quelle che promettono meglio; in questo campo infatti si devono gestire ambienti continui, a molte dimensioni e parzialmente osservabili in cui i comportamenti di successo possono essere composti anche da migliaia di azioni primitive. Quindi, nel momento in cui gli spazi degli stati diventano molto grandi, gli algoritmi di apprendimento per rinforzo devono ricorrere ad una rappresentazione approssimata della funzione  $Q(s, a)$  (*Q-Function*) per poter generalizzare. Una rappresentazione possibile sono le *reti neurali*.

### 8.4.1 Deep Neural Networks

Le DNN (Deep Neural Networks) possono approssimare arbitrariamente bene un ampio insieme di stati. Le DNN usano *gradient descent* e *back propagation* per imparare il modello.

- ◊ hidden units:  $z_j = h_1(\mathbf{w}_j^{(1)} \mathbf{x} + b_j^{(1)})$
- ◊ output units:  $y_k = h_2(\mathbf{w}_k^{(2)} \mathbf{z} + b_k^{(2)})$
- ◊ overall  $y_k = h_2(\sum_j w_{kj}^{(2)} h_1(\sum_i w_{ji}^{(1)} \mathbf{x}_i + b_j^{(1)}) + b_k^{(2)})$
- $\mathbf{w}$ : weights,  $\mathbf{x}$ : inputs to node,  $b$ : bias
- $h$ : **activation function**, usually non-linear



◊ Key idea to optimize the weights: minimize the error with respect to the output (Loss)

$$E(\mathbf{W}) = \sum_n E_n(\mathbf{W})^2 = \sum_n |f(\mathbf{x}_n; \mathbf{W}) - \mathbf{y}_n|_2^2$$

◊ Non convex optimization problem: can train by using gradient descent

- Given sample  $(\mathbf{x}_n, \mathbf{y}_n)$  update weights as follows:

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E_n}{\partial w_{ji}}$$

- Backpropagation algorithm to compute the gradient

(b) Gradient descent e back propagation

#### 8.4.2 Deep Q-Network

*Deep Q-Network* è un potente approccio usato per eseguire deep reinforcement learning in spazi di azione discreti. Deep Q-Network approssima  $Q(s, a)$  con una DNN; in particolare, usa *experience replay* (mini-batch) e due reti differenti (Q-network e target network) per mitigare la divergenza dell'algoritmo *Gradient Q-Learning* su cui esso si basa.

---

##### Algorithm 1 Gradient Q-Learning

---

- 1: Initialize weights  $\mathbf{w}$  randomly in  $[-1, 1]$
  - 2: Initialize  $s$  {observe current state}
  - 3: **loop**
  - 4:   Select and execute action  $a$
  - 5:   Observe new state  $s'$  receive immediate reward  $r$
  - 6:    $\frac{\partial Err(\mathbf{w})}{\partial \mathbf{w}} = (Q_{\mathbf{w}}(s, a) - r - \gamma \max_{a'} Q_{\mathbf{w}}(s', a')) \frac{\partial Q_{\mathbf{w}}(s, a)}{\partial \mathbf{w}}$
  - 7:   update weights  $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial Err(\mathbf{w})}{\partial \mathbf{w}}$
  - 8:   update state  $s \leftarrow s'$
  - 9: **end loop**
- 

(a) Gradient Q-Learning

---

##### Algorithm 2 DQN

---

- 1: Initialize weights  $\mathbf{w}$  and  $\bar{\mathbf{w}}$  randomly in  $[-1, 1]$
  - 2: Initialize  $s$  {observe current state}
  - 3: **loop**
  - 4:   Select and execute action  $a$
  - 5:   Observe new state  $s'$  receive immediate reward  $r$
  - 6:   Add  $(s, a, s', r)$  to experience buffer
  - 7:   Sample mini-batch  $MB$  of experiences from buffer
  - 8:   **for**  $(\hat{s}, \hat{a}, \hat{s}', \hat{r}) \in MB$  **do**
  - 9:      $\frac{\partial Err(\mathbf{w})}{\partial \mathbf{w}} = (Q_{\mathbf{w}}(\hat{s}, \hat{a}) - \hat{r} - \gamma \max_{\hat{a}'} Q_{\mathbf{w}}(\hat{s}', \hat{a}')) \frac{\partial Q_{\mathbf{w}}(\hat{s}, \hat{a})}{\partial \mathbf{w}}$
  - 10:    update weights  $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial Err(\mathbf{w})}{\partial \mathbf{w}}$
  - 11:   **end for**
  - 12:   update state  $s \leftarrow s'$
  - 13:   every  $c$  steps, update target:  $\bar{\mathbf{w}} \leftarrow \mathbf{w}$
  - 14: **end loop**
- 

(b) Deep Q-Network

## Appendice A

# Codici dei programmi del laboratorio

```
Sessione 1 - Ricerca non informata
[1]: import os, sys, time, math

module_path = os.path.abspath(os.path.join('../tools'))
if module_path not in sys.path:
    sys.path.append(module_path)

from utils.ai_lab_functions import *
import gym, envs

# ***Assignment 1: Breadth-First Search (BFS)***

def BFS_TreeSearch(problem):
    """
    Tree Search BFS

    Args:
        problem: OpenAI Gym environment

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """

    node = Node(problem.startstate, None)
    time_cost = 0
    space_cost = 0

    if node.state == problem.goalstate:
        return build_path(node), time_cost, space_cost
    frontier = NodeQueue()
    frontier.add(node)
```

```

space_cost += 1

while True:
    if frontier.is_empty():
        return "failure", time_cost, space_cost
    node = frontier.remove()

    space_cost -= 1 #space_cost è la dimensione massima di frontier
    for action in range(problem.action_space.n):
        child = Node(problem.sample(node.state, action), node)
        #if child.state not in explored and child.state not in frontier: ↴
        →#posso visitare più volte lo stesso stato!
        if child.state == problem.goalstate:
            return build_path(child), time_cost, space_cost
        frontier.add(child)
    space_cost += 1
    time_cost += 1 #time_cost++ ogni volta che creo un figlio

def BFS_GraphSearch(problem):
    """
    Graph Search BFS

    Args:
        problem: OpenAI Gym environment

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """
    node = Node(problem.startstate, None)

    time_cost = 0
    space_cost = 0

    if node.state == problem.goalstate:
        return build_path(node), time_cost, space_cost
    frontier = NodeQueue()
    frontier.add(node)
    explored = set()
    while True:
        if frontier.is_empty():
            return "failure", time_cost, space_cost
        node = frontier.remove()
        explored.add(node.state)
        space_cost += 1 #space_cost è la dimensione massima di explored
        for action in range(problem.action_space.n):
            child = Node(problem.sample(node.state, action), node)

```

```

        if child.state not in explored and child.state not in frontier:
            if child.state == problem.goalstate:
                return build_path(child), time_cost, space_cost
            frontier.add(child)
            time_cost += 1 #time_cost++ ogni volta che creo un figlio

# The following code calls your tree search and graph search version of BFS and ↴prints the results:

envname = "SmallMaze-v0"
environment = gym.make(envname)

solution_ts, time_ts, memory_ts = BFS_TreeSearch(environment)
solution_gs, time_gs, memory_gs = BFS_GraphSearch(environment)

print("\n-----")
print("\tBFS TREE SEARCH PROBLEM: ")
print("-----")
print("Solution: {}".format(solution_2_string(solution_ts, environment)))
print("Nº of nodes explored: {}".format(time_ts))
print("Max nº of nodes in memory: {}".format(memory_ts))

print("\n-----")
print("\tBFS GRAPH SEARCH PROBLEM: ")
print("-----")
print("Solution: {}".format(solution_2_string(solution_gs, environment)))
print("Nº of nodes explored: {}".format(time_gs))
print("Max nº of nodes in memory: {}".format(memory_gs))

# ***Assignment 2: Depth-Limited Search (DLS) and Iterative Deepening ↴depth-first Search (IDS)***

def DLS(problem, limit, RDLS_Function):
    """
    DLS

    Args:
        problem: OpenAI Gym environment
        limit: depth limit for the exploration, negative number means 'no limit'

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """
    node = Node(problem.startstate, None)

```

```

    return RDLS_Function(node, problem, limit, set())

def Recursive_DLS_GraphSearch(node, problem, limit, closed):
    """
    Recursive DLS

    Args:
        node: node to explore
        problem: OpenAI Gym environment
        limit: depth limit for the exploration, negative number means 'no limit'
        closed: completely explored nodes

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """
    space_cost = node.pathcost
    time_cost = 1

    if node.state == problem.goalstate:
        return build_path(node), time_cost, space_cost #ultimo passo: space_cost
    ↪non cresce più e time_cost = 1
    elif limit == 0:
        return "cut_off", time_cost, space_cost
    else:
        closed.add(node.state)
        cutoff_occurred = False
        for action in range(problem.action_space.n):
            child = Node(problem.sample(node.state, action), node, node.pathcost,
    ↪+ 1) #il terzo parametro diventerà space_cost
            if child.state in closed: #non posso visitare più volte lo stesso
    ↪stato!
                continue
            result = Recursive_DLS_GraphSearch(child, problem, limit - 1, closed)

            space_cost = result[2] #space_cost è la lunghezza del cammino che
    ↪porta al goal (ultimo passo)
            time_cost += result[1] #aggiungo al time_cost di ogni passo tutti i
    ↪nodi visitati durante la ricorsione

            if result[0] == "cut_off":
                cutoff_occurred = True
            elif result[0] != "failure":
                return result[0], time_cost, space_cost
        if cutoff_occurred:
            return "cut_off", time_cost, space_cost
    else:

```

```

        return "failure", time_cost, space_cost

def Recursive_DLS_TreeSearch(node, problem, limit, closed=None):
    """
    DLS (Tree Search Version)

    Args:
        node: node to explore
        problem: OpenAI Gym environment
        limit: depth limit for the exploration, negative number means 'no limit'

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """
    space_cost = node.pathcost
    time_cost = 1

    if node.state == problem.goalstate:
        return build_path(node), time_cost, space_cost #ultimo passo: space_cost
    ↪non cresce più e time_cost = 1
    elif limit == 0:
        return "cut_off", time_cost, space_cost
    else:
        cutoff_occurred = False
        for action in range(problem.action_space.n):
            child = Node(problem.sample(node.state, action), node, node.pathcost
    ↪+ 1) #il terzo parametro diventerà space_cost
            result = Recursive_DLS_TreeSearch(child, problem, limit - 1)

            space_cost = result[2] #space_cost è la lunghezza del cammino che
    ↪porta al goal (ultimo passo)
            time_cost += result[1] #aggiungo al time_cost di ogni passo tutti i
    ↪nodi visitati durante la ricorsione

            if result[0] == "cut_off":
                cutoff_occurred = True
            elif result[0] != "failure":
                return result[0], time_cost, space_cost
        if cutoff_occurred:
            return "cut_off", time_cost, space_cost
        else:
            return "failure", time_cost, space_cost

def IDS(problem, DLS_Function):
    """
    Iterative_DLS DLS

```

```

Args:
    problem: OpenAI Gym environment

Returns:
    (path, time_cost, space_cost): solution as a path and stats.
"""

total_time_cost = 0
total_space_cost = 0

for i in zero_to_infinity():
    result = DLS(problem, i, DLS_Function)

    total_time_cost += result[1] #sommo i time_cost di ogni livello

    if result[0] != "cut_off":
        return result[0], total_time_cost, result[2], i #solution_dls, ↴
    →total_time_cost, total_space_cost, i

# The following code calls your version of IDS and prints the results:

envname = "SmallMaze-v0"
environment = gym.make(envname)

solution_ts, time_ts, memory_ts, iterations_ts = IDS(environment, ↴
    →Recursive_DLS_TreeSearch)
solution_gs, time_gs, memory_gs, iterations_gs = IDS(environment, ↴
    →Recursive_DLS_GraphSearch)

print("\n-----")
print("\tIDS TREE SEARCH PROBLEM: ")
print("-----")
print("Necessary Iterations: {}".format(iterations_ts))
print("Solution: {}".format(solution_2_string(solution_ts, environment)))
print("Nº of nodes explored: {}".format(time_ts))
print("Max nº of nodes in memory: {}".format(memory_ts))

print("\n-----")
print("\tIDS GRAPH SEARCH PROBLEM: ")
print("-----")
print("Necessary Iterations: {}".format(iterations_gs))
print("Solution: {}".format(solution_2_string(solution_gs, environment)))
print("Nº of nodes explored: {}".format(time_gs))
print("Max nº of nodes in memory: {}".format(memory_gs))

```

---

---

BFS TREE SEARCH PROBLEM:

---

Solution: [(0, 1), (0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 103721

Max n° of nodes in memory: 77791

---

---

BFS GRAPH SEARCH PROBLEM:

---

Solution: [(0, 1), (0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 57

Max n° of nodes in memory: 15

---

---

IDS TREE SEARCH PROBLEM:

---

Necessary Iterations: 9

Solution: [(0, 1), (0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 138298

Max n° of nodes in memory: 9

---

---

IDS GRAPH SEARCH PROBLEM:

---

Necessary Iterations: 11

Solution: [(0, 1), (0, 0), (1, 0), (1, 1), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 132

Max n° of nodes in memory: 11

## Sessione 2 - Ricerca informata

```
[1]: import os
import sys
module_path = os.path.abspath(os.path.join('..../tools'))
if module_path not in sys.path:
    sys.path.append(module_path)

from utils.ai_lab_functions import *

# ***Uniform-Cost Search (UCS)***

def present_with_higher_cost(queue, node):
    if (node.state in queue):
        if(queue[node.state].value > node.value): return True
    return False

import gym
import envs

def ucs(environment):
    """
    Uniform-cost search

    Args:
        environment: OpenAI Gym environment

    Returns:
        path: solution as a path
    """
    queue = PriorityQueue()
    queue.add(Node(environment.startstate))

    explored = set()
    time_cost = 0
    space_cost = 1

    while True:
        if queue.is_empty(): return None

        node = queue.remove() # Retrieve node from the queue
        if node.state == environment.goalstate: return build_path(node), time_cost, space_cost
        explored.add(node.state)

        for action in range(environment.action_space.n): # Look around
```

```

# Child node where value and pathcost are both the pathcost of ↵
→parent + 1
    child = Node(environment.sample(node.state, action), node, node.
→pathcost + 1, node.pathcost + 1)
    time_cost += 1
    if(child.state not in queue and child.state not in explored):
        #if child.state == environment.goalstate: return ↵
→build_path(child), time_cost, space_cost
        queue.add(child)
    elif present_with_higher_cost(queue, child):
        queue.replace(child)
    space_cost = max(space_cost, len(queue) + len(explored))

# The following code calls the implementation in graph search version of UCS and ↵
→prints the results:

env = gym.make("SmallMaze-v0")
solution, time, memory = ucs(env)

print("\n-----")
print("\tUNIFORM GRAPH SEARCH PROBLEM: ")
print("-----")
print("Solution: {}".format(solution_2_string(solution, env)))
print("Nº of nodes explored: {}".format(time))
print("Max n° of nodes in memory: {}".format(memory))

# ***Assignment 1: Greedy Best-First Search***

def greedy_tree_search(environment, timeout=10000):
    """
    Greedy-best-first Tree search

    Args:
        problem: OpenAI Gym environment

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """

    goalpos = environment.state_to_pos(environment.goalstate)
    queue = PriorityQueue()

    queue.add(Node(environment.startstate))

    time_cost = 0
    space_cost = 1

```

```

while True:
    if time_cost >= timeout:
        return ("time-out", time_cost, space_cost) # timeout check

    if queue.is_empty(): return None

    node = queue.remove() # Retrieve node from the queue
    if node.state == environment.goalstate: return build_path(node), ↴
    ↪time_cost, space_cost

    for action in range(environment.action_space.n): # Look around
        # Child node where pathcost is the pathcost of parent + 1
        child = Node(environment.sample(node.state, action), node, node. ↪
        ↪pathcost + 1)

        # Child node where value is the distance heuristic
        child.value = Heu.l1_norm(environment.state_to_pos(child.state), ↪
        ↪goalpos)

        time_cost += 1
        queue.add(child)

    space_cost = max(space_cost, len(queue))

def greedy_graph_search(environment):
    """
    Greedy-best-first Graph search

    Args:
        problem: OpenAI Gym environment

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """

    goalpos = environment.state_to_pos(environment.goalstate)
    queue = PriorityQueue()
    queue.add(Node(environment.startstate))

    explored = set()
    time_cost = 0
    space_cost = 1

    while True:
        if queue.is_empty(): return None

```

```

        node = queue.remove()  # Retrieve node from the queue
        if node.state == environment.goalstate: return build_path(node), □
→time_cost, space_cost
        explored.add(node.state)

        for action in range(environment.action_space.n):  # Look around
            # Child node where pathcost is the pathcost of parent + 1
            child = Node(environment.sample(node.state, action), node, node.
→pathcost + 1)

            # Child node where value is the distance heuristic
            child.value = Heu.l1_norm(environment.state_to_pos(child.state), □
→goalpos)

            time_cost += 1
            if(child.state not in queue and child.state not in explored):
                queue.add(child)
            elif present_with_higher_cost(queue, child):
                queue.replace(child)

            space_cost = max(space_cost, len(queue) + len(explored))

def greedy(environment, search_type):
    """
    Greedy-best-first search

    Args:
        problem: OpenAI Gym environment
        search_type: type of search - greedy_tree_search or greedy_graph_search
→(function pointer)

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """
    path, time_cost, space_cost = search_type(environment)
    return path, time_cost, space_cost

# The following code calls your tree search and graph search version of □
→Greedy-best-first search and prints the results:

envname = "SmallMaze-v0"
environment = gym.make(envname)

solution_ts, time_ts, memory_ts = greedy(environment, greedy_tree_search)
solution_gs, time_gs, memory_gs = greedy(environment, greedy_graph_search)

```

```

print("\n-----")
print("\tGREEDY BEST FIRST TREE SEARCH PROBLEM: ")
print("-----")
print("Solution: {}".format(solution_2_string(solution_ts, environment)))
print("Nº of nodes explored: {}".format(time_ts))
print("Max n° of nodes in memory: {}".format(memory_ts))

print("\n-----")
print("\tGREEDY BEST FIRST GRAPH SEARCH PROBLEM: ")
print("-----")
print("Solution: {}".format(solution_2_string(solution_gs, environment)))
print("Nº of nodes explored: {}".format(time_gs))
print("Max n° of nodes in memory: {}".format(memory_gs))

# ***Assignment 2: A* Search***

def astar_tree_search(environment):
    """
    A* Tree search

    Args:
        problem: OpenAI Gym environment

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """
    goalpos = environment.state_to_pos(environment.goalstate)
    queue = PriorityQueue()

    queue.add(Node(environment.startstate))

    time_cost = 0
    space_cost = 1

    while True:
        if queue.is_empty(): return None

        node = queue.remove() # Retrieve node from the queue
        if node.state == environment.goalstate: return build_path(node), time_cost, space_cost

        for action in range(environment.action_space.n): # Look around
            # Child node where pathcost is the pathcost of parent + 1
            child = Node(environment.sample(node.state, action), node, node.
                         pathcost + 1)

```

```

# Child node where value is pathcost + distance heuristic: g(n) + h(n)
child.value = child.pathcost + Heu.l1_norm(environment.
→state_to_pos(child.state), goalpos)

time_cost += 1
queue.add(child)

space_cost = max(space_cost, len(queue))

def astar_graph_search(environment):
    """
    A* Graph search

    Args:
        problem: OpenAI Gym environment

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """

goalpos = environment.state_to_pos(environment.goalstate)
queue = PriorityQueue()
queue.add(Node(environment.startstate))

explored = set()
time_cost = 0
space_cost = 1

while True:
    if queue.is_empty(): return None

    node = queue.remove() # Retrieve node from the queue
    if node.state == environment.goalstate: return build_path(node),
→time_cost, space_cost
    explored.add(node.state)

    for action in range(environment.action_space.n): # Look around
        # Child node where pathcost is the pathcost of parent + 1
        child = Node(environment.sample(node.state, action), node, node.
→pathcost + 1)

        # Child node where value is pathcost + distance heuristic: g(n) + h(n)
        child.value = child.pathcost + Heu.l1_norm(environment.
→state_to_pos(child.state), goalpos)

```

```

        time_cost += 1
        if(child.state not in queue and child.state not in explored):
            queue.add(child)
        elif present_with_higher_cost(queue, child):
            queue.replace(child)

    space_cost = max(space_cost, len(queue) + len(explored))

def astar(environment, search_type):
    """
    A* search

    Args:
        environment: OpenAI Gym environment
        search_type: type of search - astar_tree_search or astar_graph_search
                      (function pointer)

    Returns:
        (path, time_cost, space_cost): solution as a path and stats.
    """
    path, time_cost, space_cost = search_type(environment)
    return path, time_cost, space_cost

# The following code calls your tree search and graph search version of A* search and prints the results:

envname = "SmallMaze-v0"
environment = gym.make(envname)

solution_ts, time_ts, memory_ts = astar(environment, astar_tree_search)
solution_gs, time_gs, memory_gs = astar(environment, astar_graph_search)

print("\n-----")
print("\tA* TREE SEARCH PROBLEM: ")
print("-----")
print("Solution: {}".format(solution_2_string(solution_ts, environment)))
print("Nº of nodes explored: {}".format(time_ts))
print("Max n° of nodes in memory: {}".format(memory_ts))

print("\n-----")
print("\tA* GRAPH SEARCH PROBLEM: ")
print("-----")
print("Solution: {}".format(solution_2_string(solution_gs, environment)))
print("Nº of nodes explored: {}".format(time_gs))
print("Max n° of nodes in memory: {}".format(memory_gs))

```

---

---

UNIFORM GRAPH SEARCH PROBLEM:

---

---

Solution: [(0, 1), (0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 60

Max n° of nodes in memory: 16

---

---

GREEDY BEST FIRST TREE SEARCH PROBLEM:

---

---

Solution: time-out

N° of nodes explored: 10000

Max n° of nodes in memory: 7501

---

---

GREEDY BEST FIRST GRAPH SEARCH PROBLEM:

---

---

Solution: [(0, 3), (1, 3), (2, 3), (2, 2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 44

Max n° of nodes in memory: 15

---

---

A\* TREE SEARCH PROBLEM:

---

---

Solution: [(0, 1), (1, 1), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 8360

Max n° of nodes in memory: 6271

---

---

A\* GRAPH SEARCH PROBLEM:

---

---

Solution: [(0, 1), (1, 1), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3)]

N° of nodes explored: 60

Max n° of nodes in memory: 16

## Sessione 3 - Markov Decision Processes

```
[1]: import os, sys
module_path = os.path.abspath(os.path.join('../tools'))
if module_path not in sys.path:
    sys.path.append(module_path)

import gym, envs
from utils.ai_lab_functions import *
from timeit import default_timer as timer
from tqdm import tqdm as tqdm

# ***Assignment 1: Value Iteration Algorithm***

def value_iteration(environment, maxiters=300, discount=0.9, max_error=1e-3):
    """
    Performs the value iteration algorithm for a specific environment

    Args:
        environment: OpenAI Gym environment
        maxiters: timeout for the iterations
        discount: gamma value, the discount factor for the Bellman equation
        max_error: the maximum error allowed in the utility of any state

    Returns:
        policy: 1-d dimensional array of action identifiers where index `i` corresponds to state id `i`
    """
    U_1 = [0 for _ in range(environment.observation_space.n)] # vector of utilities for states S

    delta = 1
    iters = 0

    # until(delta < max_error*((1-discount)/discount)) è uguale a while(delta >= max_error*((1-discount)/discount))
    while delta >= max_error * ((1 - discount) / discount) and iters < maxiters:
        delta = 0 # maximum change in the utility of any state in an iteration
        U = U_1.copy()

        for s in range(environment.observation_space.n):
            sommatorie = []
            for a in range(environment.action_space.n):
                somma = 0
                for s_1 in range(environment.observation_space.n):
                    somma += environment.T[s, a, s_1] * U[s_1]
```

```

    sommatorie.append(somma)

    U_1[s] = environment.RS[s] + discount * max(sommatorie)

    if abs(U_1[s] - U[s]) > delta:
        delta = abs(U_1[s] - U[s])

    iters += 1

    return values_to_policy(np.asarray(U), environment) # automatically convert
→the value matrix  $U$  to a policy

```

*# The following code executes Value Iteration and prints the resulting policy:*

```

envnames = ["LavaFloor-v0", "NiceLavaFloor-v0", "VeryBadLavaFloor-v0"]

# in alcuni stati ogni azione va bene (tipo quelli vicino al muro)
for envname in envnames:

    print("\n-----")
    print("\tEnvironment: {} \n\tValue Iteration".format(envname))
    print("-----")

    env = gym.make(envname)
    print("\nRENDER:")
    env.render()

    t = timer()
    policy = value_iteration(env)

    print("\nTIME: \n{}".format(round(timer() - t, 4)))
    print("\nPOLICY:")
    print(np.vectorize(env.actions.get)(policy.reshape(env.rows, env.cols)))

```

```

# ***Assignment 2: Policy Iteration Algorithm***

def policy_evaluation(policy, U, environment, discount, maxviter):
    for i in range(maxviter): # va ripetuta per un certo numero di volte
        for s in range(environment.observation_space.n):
            somma = 0
            for s_1 in range(environment.observation_space.n):
                somma += environment.T[s, policy[s], s_1] * U[s_1]

            U[s] = environment.RS[s] + discount * somma

    return U

def policy_iteration(environment, maxiters=300, discount=0.9, maxviter=10):
    """
    Performs the policy iteration algorithm for a specific environment

    Args:
        environment: OpenAI Gym environment
        maxiters: timeout for the iterations
        discount: gamma value, the discount factor for the Bellman equation

    Returns:
        policy: 1-d dimensional array of action identifiers where index `i` ↴
        corresponds to state id `i`
    """

    policy = [0 for _ in range(environment.observation_space.n)] #initial policy
    U = [0 for _ in range(environment.observation_space.n)] #utility array

    iters = 0
    unchanged = False

    # until(unchanged) è uguale a while(not unchanged)
    while not unchanged and iters < maxiters:
        # Step (1): Policy Evaluation
        U = policy_evaluation(policy, U, environment, discount, maxviter)

        # Step (2): Policy Improvement
        unchanged = True
        for s in range(environment.observation_space.n):
            # tutte le azioni
            left = []
            for a in range(environment.action_space.n):
                somma = 0
                for s_1 in range(environment.observation_space.n):
                    somma += environment.T[s, a, s_1] * U[s_1]

                if somma > left[-1]:
                    left.append(somma)

            if len(left) == 1:
                policy[s] = 0
            else:
                policy[s] = left.index(max(left))

            if policy[s] != s:
                unchanged = False

        iters += 1

```

```

    left.append(somma)

    # azione della policy
    right = 0
    for s_1 in range(environment.observation_space.n):
        right += environment.T[s, policy[s], s_1] * U[s_1]

    if max(left) > right:
        policy[s] = np.argmax(left)
        unchanged = False

    iters += 1

return np.asarray(policy)

# The following code executes Policy Iteration and prints the resulting policy:

envnames = ["LavaFloor-v0", "NiceLavaFloor-v0", "VeryBadLavaFloor-v0"]

# in alcuni stati ogni azione va bene (tipo quelli vicino al muro)
for envname in envnames:

    print("\n-----")
    print("\tEnvironment: {} \n\tPolicy Iteration".format(envname))
    print("-----")

    env = gym.make(envname)
    print("\nRENDER:")
    env.render()

    t = timer()
    policy = policy_iteration(env)

    print("\nTIME: \n{}".format(round(timer() - t, 4)))
    print("\nPOLICY:")
    print(np.vectorize(env.actions.get)(policy.reshape(env.rows, env.cols)))

# ***Comparison***

envnames = ["LavaFloor-v0", "HugeLavaFloor-v0"]

for envname in envnames:

    maxiters = 50

```

```

env = gym.make(envname)

series = [] # Series of learning rates to plot
liters = np.arange(maxiters + 1) # Learning iteration values
liters[0] = 1
elimit = 100 # Limit of steps per episode
rep = 10 # Number of repetitions per iteration value
virewards = np.zeros(len(liters)) # Rewards array
c = 0

t = timer()

# Value iteration
for i in tqdm(liters, desc="Value Iteration", leave=True):
    reprew = 0
    policy = value_iteration(env, maxiters=i) # Compute policy
    # Repeat multiple times and compute mean reward
    for _ in range(rep):
        reprew += run_episode(env, policy, elimit) # Execute policy
    virewards[c] = reprew / rep
    c += 1
    series.append({"x": liters, "y": virewards, "ls": "-", "label": "Value\u2192Iteration"})

vmaxiters = 5 # Max number of iterations to perform while evaluating a\u2192policy
pirewards = np.zeros(len(liters)) # Rewards array
c = 0

# Policy iteration
for i in tqdm(liters, desc="Policy Iteration", leave=True):
    reprew = 0
    policy = policy_iteration(env, maxiters=i) # Compute policy
    # Repeat multiple times and compute mean reward
    for _ in range(rep):
        reprew += run_episode(env, policy, elimit) # Execute policy
    pirewards[c] = reprew / rep
    c += 1
    series.append({"x": liters, "y": pirewards, "ls": "-", "label": "Policy\u2192Iteration"})

print("Execution time: {}s".format(round(timer() - t, 4)))
np.set_printoptions(linewidth=10000)

plot(series, "Learning Rate", "Iterations", "Reward")

```

```
-----  
Environment: LavaFloor-v0  
Value Iteration  
-----
```

RENDER:

```
[['S' 'L' 'L' 'L']  
 ['L' 'W' 'L' 'P']  
 ['L' 'L' 'L' 'G']]
```

TIME:

0.0428

POLICY:

```
[['D' 'L' 'L' 'U']  
 ['D' 'L' 'L' 'L']  
 ['R' 'R' 'R' 'L']]
```

```
-----  
Environment: NiceLavaFloor-v0  
Value Iteration  
-----
```

RENDER:

```
[['S' 'L' 'L' 'L']  
 ['L' 'W' 'L' 'P']  
 ['L' 'L' 'L' 'G']]
```

TIME:

0.0459

POLICY:

```
[['D' 'L' 'D' 'U']  
 ['U' 'L' 'L' 'L']  
 ['R' 'L' 'L' 'L']]
```

```
-----  
Environment: VeryBadLavaFloor-v0  
Value Iteration  
-----
```

RENDER:

```
[['S' 'L' 'L' 'L']  
 ['L' 'W' 'L' 'P']  
 ['L' 'L' 'L' 'G']]
```

TIME:

0.0396

POLICY:

```
[[['D' 'R' 'D' 'L']]  
[['D' 'L' 'D' 'L']]  
[['R' 'R' 'R' 'L']]]
```

---

Environment: LavaFloor-v0  
Policy Iteration

---

RENDER:

```
[[['S' 'L' 'L' 'L']]  
[['L' 'W' 'L' 'P']]  
[['L' 'L' 'L' 'G']]]
```

TIME:

0.0106

POLICY:

```
[[['D' 'L' 'L' 'U']]  
[['D' 'L' 'L' 'L']]  
[['R' 'R' 'R' 'L']]]
```

---

Environment: NiceLavaFloor-v0  
Policy Iteration

---

RENDER:

```
[[['S' 'L' 'L' 'L']]  
[['L' 'W' 'L' 'P']]  
[['L' 'L' 'L' 'G']]]
```

TIME:

0.036

POLICY:

```
[[['D' 'L' 'D' 'U']]  
[['D' 'L' 'L' 'L']]  
[['R' 'R' 'L' 'L']]]
```

---

Environment: VeryBadLavaFloor-v0  
Policy Iteration

---

RENDER:

```
[['S' 'L' 'L' 'L'],
 ['L' 'W' 'L' 'P'],
 ['L' 'L' 'L' 'G']]
```

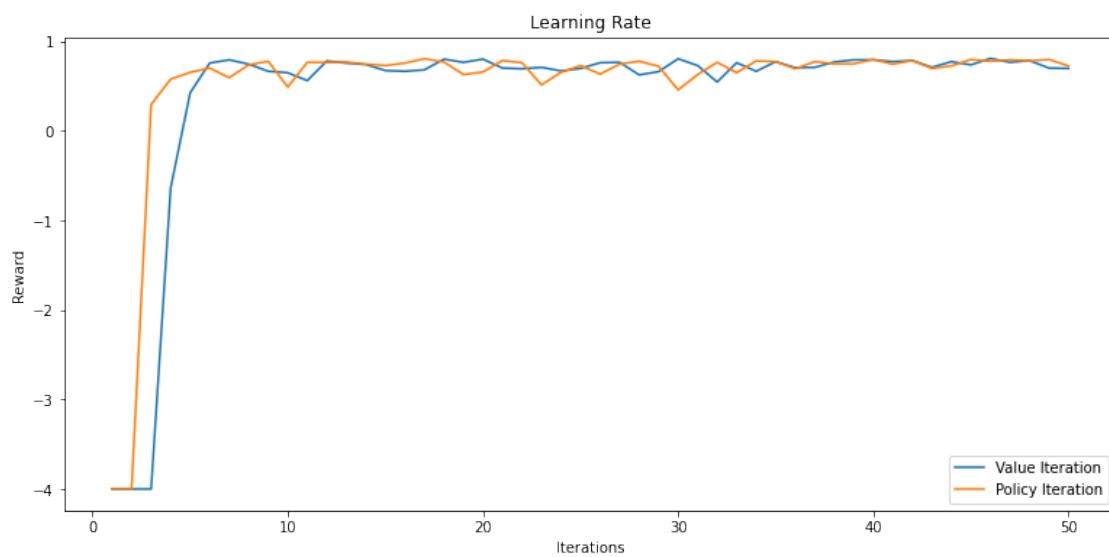
TIME:

0.0063

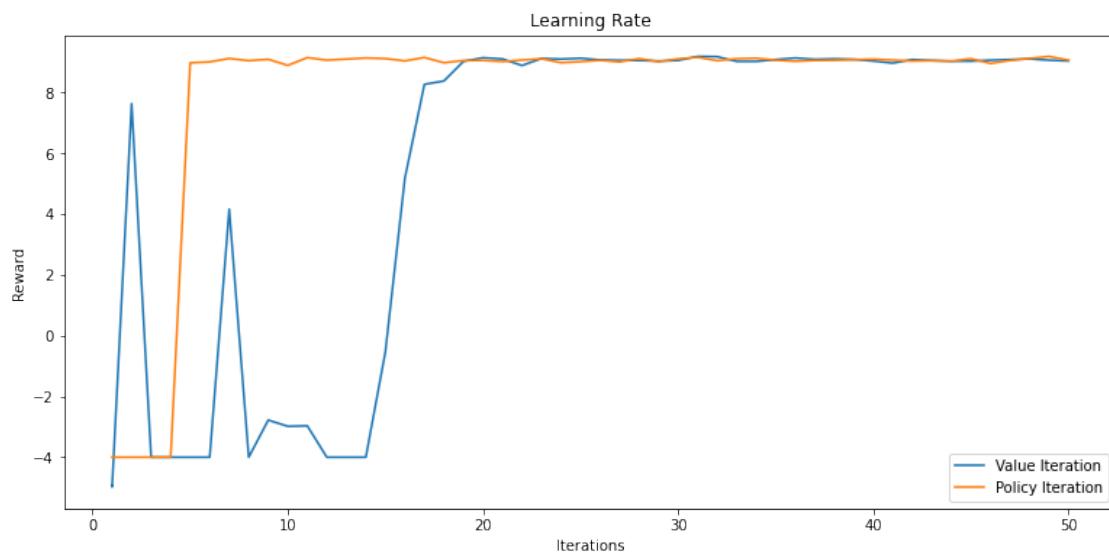
POLICY:

```
[['D' 'R' 'D' 'L'],
 ['D' 'L' 'D' 'L'],
 ['R' 'R' 'R' 'L']]
```

Execution time: 1.6407s



Execution time: 66.5544s



## Sessione 4 - Reinforcement learning

```
[1]: import os, sys
module_path = os.path.abspath(os.path.join('..../tools'))
if module_path not in sys.path:
    sys.path.append(module_path)

import gym, envs
from utils.ai_lab_functions import *
from timeit import default_timer as timer
from tqdm import tqdm as tqdm

# The following policy functions could be used in the algorithms:

def epsilon_greedy(q, state, epsilon):
    """
    Epsilon-greedy action selection function

    Args:
        q: q table
        state: agent's current state
        epsilon: epsilon parameter

    Returns:
        action id
    """
    an = q.shape[1]
    probs = np.full(an, epsilon / an)
    probs[q[state].argmax()] += 1 - epsilon
    return np.random.choice(an, p=probs)

def softmax(q, state, temp):
    """
    Softmax action selection function

    Args:
        q: q table
        state: agent's current state
        temp: temperature parameter

    Returns:
        action id
    """
    e = np.exp(q[state] / temp)
    return np.random.choice(q.shape[1], p=e / e.sum())
```

```

# ***Assignment 1: Q-Learning***

def q_learning(environment, episodes, alpha, gamma, expl_func, expl_param):
    """
    Performs the Q-Learning algorithm for a specific environment

    Args:
        environment: OpenAI Gym environment
        episodes: number of episodes for training
        alpha: alpha parameter
        gamma: gamma parameter
        expl_func: exploration function (epsilon_greedy, softmax)
        expl_param: exploration parameter (epsilon, T)

    Returns:
        (policy, rewards, lengths): final policy, rewards for each episode ↴ [array], length of each episode [array]
    """
    q = np.zeros((environment.observation_space.n, environment.action_space.n)) ↴
    # Q(s, a)
    rews = np.zeros(episodes)
    lengths = np.zeros(episodes)

    for i in range(0, episodes):
        s = env.reset() # inizializzo lo stato riportando l'agente nel punto ↴ iniziale

        # preparo i contatori che serviranno per aggiornare rews e lengths
        tot_r = 0
        tot_l = 0

        done = False
        while done != True:
            a = expl_func(q, s, expl_param) # imparo la policy ottima senza ↴ seguirla (off-policy)
            s_1, r, done, _ = env.step(a) # eseguo l'azione e osservo lo stato ↴ raggiunto e il reward ottenuto

            # costruisco un array che contiene i valori di q[s_1] per ogni ↴ azione a_1 e ne calcolo il massimo
            arr = []
            for a_1 in range(environment.action_space.n):
                arr.append(q[s_1][a_1])
            m = max(arr)

```

```

q[s][a] = q[s][a] + alpha * (r + gamma * m - q[s][a])

# aggiorno lo stato e i contatori
s = s_1
tot_r += r
tot_l += 1

# aggiorno rews e lengths alla fine di ogni episodio
rews[i] = tot_r
lengths[i] = tot_l

policy = q.argmax(axis=1) # q.argmax(axis=1) automatically extract the
→policy from the q table
return policy, rews, lengths

# The following code calls your Q-Learning implementation and prints the results:

envname = "Cliff-v0"

print("\n-----")
print("\tEnvironment: {} \n\tQ-Learning".format(envname))
print("-----\n")

env = gym.make(envname)
env.render()
print()

# Learning parameters
episodes = 500
alpha = .3
gamma = .9
epsilon = .1

t = timer()

# Q-Learning epsilon greedy
policy, rewards, lengths = q_learning(env, episodes, alpha, gamma,
→epsilon_greedy, epsilon)
print("Execution time: {}s\nPolicy:\n{}\n".format(round(timer() - t, 4), np.
→vectorize(env.actions.get)(policy.reshape(env.shape))))
_ = run_episode(env, policy, 20)

```

```

# ***Assignment 2: SARSA***

def sarsa(environment, episodes, alpha, gamma, expl_func, expl_param):
    """
    Performs the SARSA algorithm for a specific environment

    Args:
        environment: OpenAI gym environment
        episodes: number of episodes for training
        alpha: alpha parameter
        gamma: gamma parameter
        expl_func: exploration function (epsilon_greedy, softmax)
        expl_param: exploration parameter (epsilon, T)

    Returns:
        (policy, rewards, lengths): final policy, rewards for each episode
        → [array], length of each episode [array]
    """

    q = np.zeros((environment.observation_space.n, environment.action_space.n))
    # Q(s, a)
    rews = np.zeros(episodes)
    lengths = np.zeros(episodes)

    for i in range(0, episodes):
        s = env.reset() # inizializzo lo stato riportando l'agente nel punto
        → iniziale
        a = expl_func(q, s, expl_param) # calcolo a perché devo sempre calcolare
        → la prossima azione basandomi sulla policy

        # preparo i contatori che serviranno per aggiornare rews e lengths
        tot_r = 0
        tot_l = 0

        done = False
        while done != True:
            s_1, r, done, _ = env.step(a) # eseguo eseguo l'azione e osservo lo
            → stato raggiunto e il reward ottenuto
            a_1 = expl_func(q, s_1, expl_param) # calcolo la prossima azione
            → (non faccio il massimo tra tutte le azioni)

            q[s][a] = q[s][a] + alpha * (r + gamma * q[s_1][a_1] - q[s][a])

            # aggiorno stato, azione (on-policy) e contatori
            s = s_1
            a = a_1
            tot_r += r

```

```

    tot_l += 1

    # aggiorno rews e lengths alla fine di ogni episodio
    rews[i] = tot_r
    lengths[i] = tot_l

policy = q.argmax(axis=1) # q.argmax(axis=1) automatically extract the
→policy from the q table
return policy, rews, lengths

# The following code calls your SARSA implementation and prints the results:

envname = "Cliff-v0"

print("\n-----")
print("\tEnvironment: {} \n\tSARSA".format(envname))
print("-----\n")

env = gym.make(envname)
env.render()
print()

# Learning parameters
episodes = 500
alpha = .3
gamma = .9
epsilon = .1

t = timer()

# SARSA epsilon greedy
policy, rews, lengths = sarsa(env, episodes, alpha, gamma, epsilon_greedy,
→epsilon)
print("Execution time: {}s\nPolicy:\n{}\n".format(round(timer() - t, 4), np.
→vectorize(env.actions.get)(policy.reshape(env.shape))))
_ = run_episode(env, policy, 20)

```

```

# ***Comparison***

#The following code performs a comparison between the 2 reinforcement learning algorithms:
→algorithms:

envname = "Cliff-v0"

print("\n-----")
print("\tEnvironment: ", envname)
print("-----\n")

env = gym.make(envname)
env.render()

# Learning parameters
episodes = 500
ep_limit = 50
vmaxiters = 50
alpha = .3
gamma = .9
epsilon = .1
delta = 1e-3

rewser = []
lenser = []

litres = np.arange(1, episodes + 1) # Learning iteration values
window = 10 # Rolling window
mrew = np.zeros(episodes)
mlen = np.zeros(episodes)

t = timer()

# Q-Learning
_, rews, lengths = q_learning(env, episodes, alpha, gamma, epsilon_greedy,
→epsilon)
rews = rolling(rews, window)
rewser.append({"x": np.arange(1, len(rews) + 1), "y": rews, "ls": "-", "label": "Q-Learning"})
lengths = rolling(lengths, window)
lenser.append({"x": np.arange(1, len(lengths) + 1), "y": lengths, "ls": "-", "label": "Q-Learning"})

# SARSA
_, rews, lengths = sarsa(env, episodes, alpha, gamma, epsilon_greedy, epsilon)
rews = rolling(rews, window)
rewser.append({"x": np.arange(1, len(rews) + 1), "y": rews, "label": "SARSA"})

```

```

lengths = rolling(lengths, window)
lenser.append({"x": np.arange(1, len(lengths) + 1), "y": lengths, "label": "SARSA"})

print("Execution time: {}s".format(round(timer() - t, 4)))

plot(rewser, "Rewards", "Episodes", "Rewards")
plot(lenser, "Lengths", "Episodes", "Lengths")

```

-----  
Environment: Cliff-v0  
Q-Learning  
-----

```

o   o   o   o   o   o   o   o   o   o   o
o   o   o   o   o   o   o   o   o   o   o
o   o   o   o   o   o   o   o   o   o   o
x   C   C   C   C   C   C   C   C   C   T

```

Execution time: 0.7814s

Policy:

```

[[['L' 'D' 'R' 'R' 'R' 'D' 'R' 'R' 'D' 'R' 'R' 'D'],
 ['U' 'R' 'R' 'R' 'R' 'D' 'R' 'D' 'R' 'R' 'D' 'D'],
 ['R' 'R' 'D'],
 ['U' 'U' 'U']]

```

-----  
Environment: Cliff-v0  
SARSA  
-----

```

o   o   o   o   o   o   o   o   o   o   o
o   o   o   o   o   o   o   o   o   o   o
o   o   o   o   o   o   o   o   o   o   o
x   C   C   C   C   C   C   C   C   C   T

```

Execution time: 0.7075s

Policy:

```

[[['R' 'R' 'D'],
 ['U' 'U' 'U' 'U' 'U' 'R' 'U' 'U' 'R' 'L' 'R' 'D'],
 ['U' 'U' 'U' 'U' 'L' 'R' 'U' 'U' 'R' 'R' 'R' 'D'],
 ['U' 'U' 'U']]

```

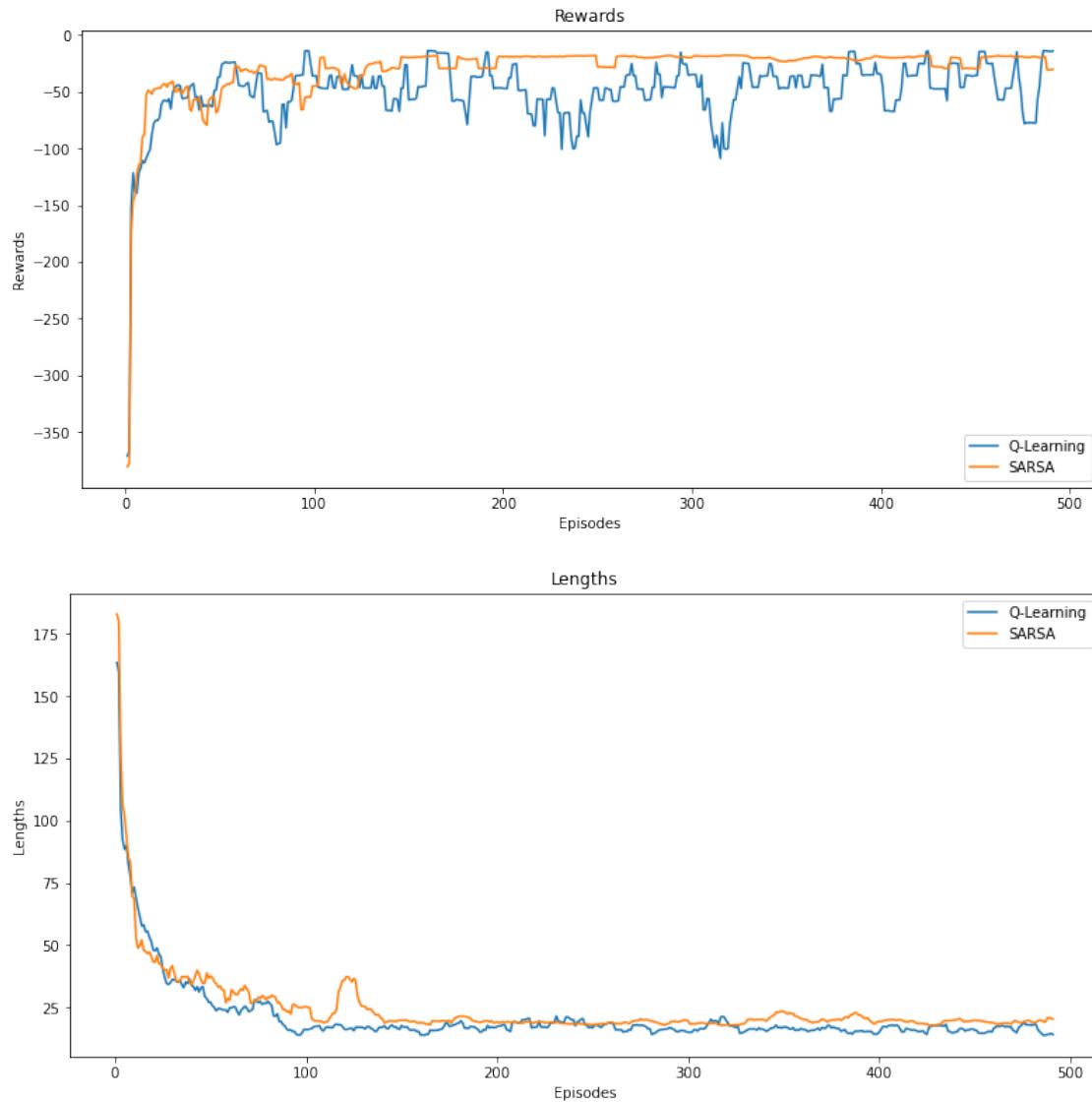
---

Environment: Cliff-v0

---

```
o   o   o   o   o   o   o   o   o   o   o   o  
o   o   o   o   o   o   o   o   o   o   o   o  
o   o   o   o   o   o   o   o   o   o   o   o  
x   C   C   C   C   C   C   C   C   C   C   T
```

Execution time: 1.3729s



## Sessione 5 - Deep reinforcement learning

```
[1]: import os, sys, keras, random, numpy
module_path = os.path.abspath(os.path.join('..../tools'))
if module_path not in sys.path:
    sys.path.append(module_path)

import gym, envs
from utils.ai_lab_functions import *
from timeit import default_timer as timer
from tqdm import tqdm as tqdm
from collections import deque
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

# ***Assignment: Q-Learning***

def create_model(input_size, output_size, hidden_layer_size,
→hidden_layer_number):
    """
    Create the neural network model with the given parameters

    Args:
        input_size: the number of nodes for the input layer
        output_size: the number of nodes for the output layer
        hidden_layer_size: the number of nodes for each hidden layer
        hidden_layer_number: the number of hidden layers

    Returns:
        model: the corresponding neural network
    """

model = Sequential()

model.add(Dense(hidden_layer_size, input_dim=input_size, activation="relu"))
→#input layer + hidden layer #1

for i in range(hidden_layer_number-1):
    model.add(Dense(hidden_layer_size, activation="relu")) #hidden layer
→#2-#hidden_layer_number

model.add(Dense(output_size, activation="linear")) #output layer

model.compile(loss="mean_squared_error", optimizer='adam') #loss function
→and optimizer definition
```

```

    return model

def train_model(model, memory, batch_size, gamma=0.99):
    """
    Performs the value iteration algorithm for a specific environment

    Args:
        model: the neural network model to train
        memory: the memory array on which perform the training
        batch_size: the size of the batch sampled from the memory
        gamma: gamma value, the discount factor for the Bellman equation
    """
    if len(memory) > batch_size:
        mb = random.sample(memory, batch_size)

        for exp in mb: # exp = (s, a, s_1, r, done)
            target = model.predict(exp[0])[0]
            if exp[4]:
                target[exp[1]] = exp[3]
            else:
                max_q = max(model.predict(exp[2])[0])
                target[exp[1]] = exp[3] + (max_q * gamma)
            model.fit(exp[0], np.array([target])) # sposta fuori dal for per
            →avere tempi di computazione minori

    return model

def DQN(environment, neural_network, trials, goal_score, batch_size,
→epsilon_decay=0.9995):
    """
    Performs the Q-Learning algorithm for a specific environment on a specific
    →neural network model

    Args:
        environment: OpenAI Gym environment
        neural_network: the neural network to train
        trials: the number of iterations for the training phase
        goal_score: the minimum score to consider 'solved' the problem
        batch_size: the size of the batch sampled from the memory
        epsilon_decay: the decay value of epsilon for the eps-greedy exploration

    Returns:
        score_queue: 1-d dimensional array of the reward obtained at each trial
    """

```

```

epsilon = 1.0; epsilon_min = 0.01
score = 0; score_queue = []

exp_buffer = deque(maxlen=10000)

for trial in range(trials):
    s = environment.reset()
    s = s.reshape(1, environment.observation_space.shape[0]) # ogni stato è
    →una tupla di 4 valori

    score = 0 # initialize reward obtained at this trial step

    done = False
    while done != True:
        # se probabilità < epsilon, allora scelgo casualmente (all'inizio
        →sempre casuale perché epsilon=1)
        if random.randrange(0, environment.action_space.n) < epsilon:
            a = random.randrange(0, environment.action_space.n)
        else:
            a = np.argmax(neural_network.predict(s)[0])

        # epsilon decade finché non raggiunge un minimo
        epsilon *= epsilon_decay
        if epsilon < epsilon_min:
            epsilon = epsilon_min

        s_1, r, done, _ = environment.step(a)
        s_1 = s_1.reshape(1, environment.observation_space.shape[0])

        exp_buffer.append((s, a, s_1, r, done)) # update exp_buff

        neural_network = train_model(neural_network, exp_buffer, batch_size)

        score += r # update reward obtained at this trial step

        s = s_1

        score_queue.append(score) # update score_queue

        print("Episode: {:.7.0f}, Score: {:.3.0f}, EPS: {:.3.2f}".format(trial,
        →score_queue[-1], epsilon))
        if(score > goal_score): break

return neural_network, score_queue

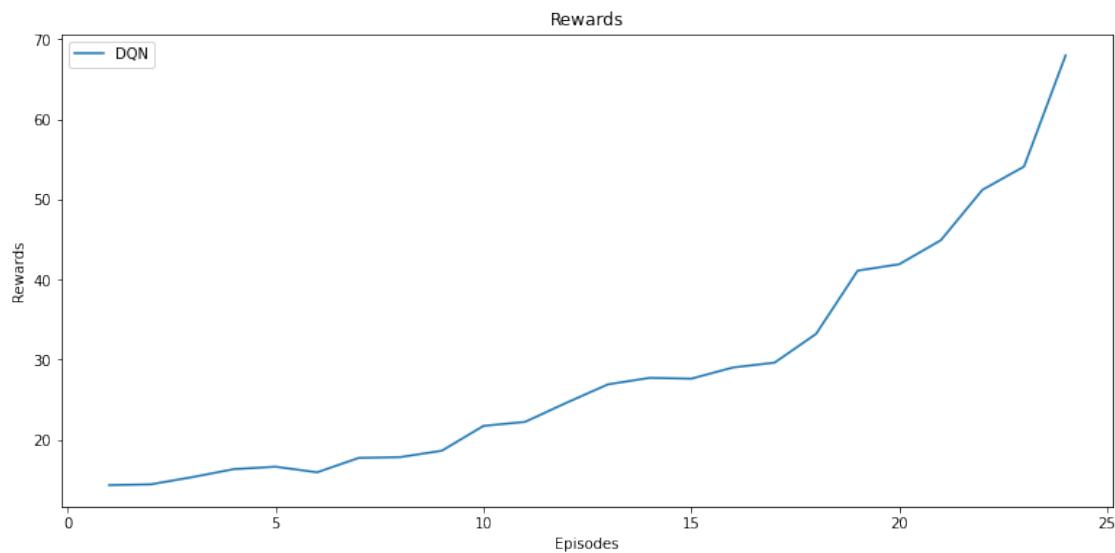
```

```
# The following code executes the DQN and plots the reward function:

env = gym.make("CartPole-v1")
neural_network = create_model(4, 2, 32, 2)
neural_network, score = DQN(env, neural_network, trials=1000, goal_score=130, batch_size=32)

rewser = []
window = 10

score = rolling(np.array(score), window)
rewser.append({"x": np.arange(1, len(score) + 1), "y": score, "ls": "-", "label": "DQN"})
plot(rewser, "Rewards", "Episodes", "Rewards")
```



# Credits

Basato sulle slide fornite dal prof. Alessandro Farinelli

Basato sui capitoli 1, 2, 3, 4, 6, 13, 17 e 21 del libro “*Intelligenza artificiale. Un approccio moderno*” (Vol. 1 e 2) (*Italiano*)

Basato sulla dispensa disponibile in: <https://github.com/davbianchi/dispense-info-univr/tree/master/magistrale/intelligenza-artificiale>

Repository GitHub personale: <https://github.com/zampierida98/UniVR-informatica>

Indirizzo e-mail personale: [zampieri.davide@outlook.com](mailto:zampieri.davide@outlook.com)