Roberto Bombieri 882818
Giorgio Mecca 880847

Relazione IALab – CLIPS

Obiettivo

L'obiettivo del progetto è quello di sviluppare due strategie diverse per risolvere il gioco *Mastermind*.

Definizione Gioco

Il gioco consiste nello scoprire un codice segreto, composto da quattro cifre, con al più 10 tentativi. Nel nostro caso, occorre implementare una variante del gioco definita in questo modo:

- il codice segreto è composto da quattro colori tutti diversi tra loro
- i colori ammessi per comporre il codice segreto sono 8 e sono:
  blue, green, red, yellow, orange, white, black, purple
- il giocatore ha al più 10 tentativi, se non indovina il codice in questi dieci passi, ha perso
- ad ogni tentativo il giocatore riceve una risposta dal sistema che indica quanti colori sono stati indovinati nella corretta posizione e quanti sono stati indovinati ma in una posizione errata
- il tentativo non può contenere colori ripetuti, se ciò accade il giocatore non riceve alcuna risposta, ma la mossa viene comunque contata

Strategia 1

La prima strategia è quella che tenta di avvicinarsi di più ad un comportamento umano e si basa sul concetto delle permutazioni.

Sapendo di avere a disposizione 8 colori, che ogni combinazione è composta da 4 colori e che ogni colore in una combinazione non può essere ripetuto, per ogni *right placed* verranno mantenuti un numero n di *right placed* dalla combinazione precedente, mentre se sono presenti dei *miss placed* verrà eseguita una permutazione della combinazione fino a che non si otterranno solo *right placed.*

L'idea della strategia è quella di dare priorità ai *right placed* e poi controllare i *miss placed* cercando di ottenere la combinazione ideale X-0 con X *right placed* e 0 *miss placed.*

Struttura

Un codice è formato da 4 colori.
Esistono due tipologie di codici indicati dai template *Code CodeS*.
Per le posizioni vuote in *Code* e *CodeS* viene utilizzato il codice *blank.*

```
(deftemplate code
 (slot p1) (slot p2) (slot p3) (slot p4)
 (slot rp) (slot mp)
 (slot id)
)
(deftemplate codeS
 (slot p1) (slot p2) (slot p3) (slot p4)
 (slot id)
)
```

Code indica la combinazione (1, 2, 3 colori) che l'utente reputa corretta, composta dai soli rp.
IDEA: "dato un risultato ricordo tutte le possibili combinazioni di rp"

```
Esempio:
       codice giocato (a b c d)
       risultato ottenuto 1 rp - 1 mp
       in questo caso l'utente considera che 1 lettera nel posto giusto,
       perciò ha 4 Code
       (a blank blank blank (rp 1) (mp 1) )
       (blank b blank blank (rp 1) (mp 1) )
       e così via…
```

Il template *Code* contiene il risultato ottenuto quando è stato asserito.
Questo viene riproposto con gli slot *mp rp.*
IDEA: "I rp e mp di ogni code sono valori da battere per trovare un codice migliore, più probabile"

Ad ogni *Code* è associato tramite un identificativo un *CodeS,* analogamente questo indica gli mp, colori che l'utente vuole permutare per trovare la giusta combinazione; i *CodeS* sono composti da un template analogo con 4 colori (oppure *blank*).

IDEA: I colori di *CodeS* dovranno essere permutati per permettere al giocatore di individuare la loro posizione corretta.

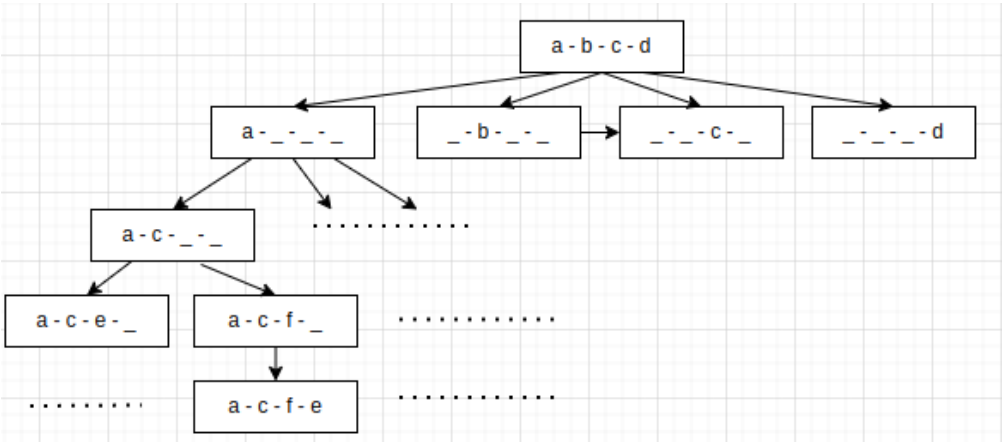
```
Esempio:
```

```
codice giocato (a b c d)
risultato ottenuto 1 rp - 1 mp
in questo caso l'utente considera che 1 lettera sia nel posto giusto,
perciò ha 4 Code e 4 codeS
(Code a blank blank blank)
(CodeS blank b c d)
e così via…
```

Cercando il codice

Possiamo immaginare la ricerca del codice vincente come l'espansione di un albero partendo dalla radice e arrivando alla giusta foglia.

L'idea è di partire dalla radice (salvare questa in memoria / asserire un fatto) ed espanderla in profondità ottenendo i nodi figlio (questo comporta l'asserire un fatto per ogni nodo e cancellare la radice).
A questo punto l'idea è quella di testare un nodo figlio. Se otteniamo un risultato migliore, questo verrà espanso (verranno asseriti i suoi nodi figlio) e verrà cancellato il resto dell'albero. Se otteniamo un risultato peggiore, verrà cancellato dallo stack questo nodo (retract) e si passerà ad analizzare un nodo fratello.



Fasi di Gioco

E' possibile dividere la strategia in 2 fasi.
La prima fase sarà l'esame della mossa che è appena avvenuta basata sul risultato ottenuto ovvero gli *answer* restituiti dal codice, la seconda fase sarà la scelta della mossa successiva in base ai risultati ottenuti.

Scelta della mossa

La scelta della Mossa avviene tramite i codici asseriti. Viene selezionato il codice con id minore (l'id utilizzato per collegare un *Code e CodeS* ma anche per creare un ordinamento tra codici). Scelto un codice questo verrà completato dal suo *CodeS* permutando ciclicamente i suoi colori.

```
Esempio:
       tra i codici asseriti abbiamo:
       Code a blank blank blank
       CodeS blank b c d.

       codice giocato: a c d b
```

Quando un CodeS viene permutato questo comporta una modifica del suo *fact*.

Viene distinto un caso per ogni possibile combinazione di codici (4 diversi casi per 1 rp, 6 per 2 rp e 4 per 3 rp)

```
(defrule computer-stepN-2G-POS2 (declare (salience -10))
 (status (step ?n) (mode computer))
 (code (p1 ?c1&:(neq ?c1 blank)) (p2 blank) (p3 ?c3&:(neq ?c3 blank)) (p4 blank)
(id ?idC))
 (not (code (id ?idC2&:(< ?idC2 ?idC))))
 ?cS <- (codeS (p1 blank) (p2 ?s2&:(neq ?s2 blank)) (p3 blank) (p4 ?s4&:(neq ?s4
blank)))
 =>
 (assert (guess (step ?n) (g ?c1 ?s4 ?c3 ?s2) ))
 (printout t "La tua giocata allo step: " ?n " -> " ?c1 " " ?s4 " " ?c3 " " ?s2
crlf)
 (modify ?cS (p1 blank) (p2 ?s4) (p3 blank) (p4 ?s2))
 (pop-focus)
)
```

Allo step 0, il gioco inizia con l'inserimento statico di una combinazione.

```
(assert (guess (step 0) (g blue red yellow green) ))
(assert (code (p1 blue) (p2 red) (p3 yellow) (p4 green) (rp 0) (mp 0) (id 0)))
(assert (codeS (p1 blank) (p2 blank) (p3 blank) (p4 blank) (id 0) ))
```

Per ogni risultato ottenuto viene eseguita la fase di "Esame della mossa" antecedente alla scelta della nuova mossa.

In questa fase vengono definite varie casistiche dividendo il risultato ottenuto creando 4 casi descritti come segue.

0 right placed – 0 miss placed

```
(defrule step0N-0-0 (declare (salience -7))
 (answer (step ?n) (right-placed ?rp&:(= ?rp 0)) (miss-placed ?mp&:(= ?mp 0)))
 (guess (step ?n) (g  ?c1 ?c2 ?c3 ?c4) )

 =>
 (printout t "-----------------------------------------------------------U--"
"Right placed " ?rp " missplaced "?mp crlf)
  (do-for-all-facts ((?var code)) TRUE (retract ?var))
 (do-for-all-facts ((?var codeS)) TRUE (retract ?var))

 (assert (code (p1 blank) (p2 blank) (p3 blank) (p4 blank) (rp 0) (mp 4) (id ?n)))
 (do-for-fact ((?s1 color) (?s2 color) (?s3 color) (?s4 color))
   (and
     (neq ?s1:name ?c1) (neq ?s1:name ?c2) (neq ?s1:name ?c3) (neq ?s1:name ?c4)
     (neq ?s2:name ?c1) (neq ?s2:name ?c2) (neq ?s2:name ?c3) (neq ?s2:name ?c4)
(neq ?s2:name ?s1:name)
     (neq ?s3:name ?c1) (neq ?s3:name ?c2) (neq ?s3:name ?c3) (neq ?s3:name ?c4)
(neq ?s3:name ?s2:name) (neq ?s3:name ?s1:name)
     (neq ?s4:name ?c1) (neq ?s4:name ?c2) (neq ?s4:name ?c3) (neq ?s4:name ?c4)
(neq ?s4:name ?s3:name) (neq ?s4:name ?s2:name) (neq ?s4:name ?s1:name)
   )
   (assert (codeS (p1 ?s1:name) (p2 ?s2:name) (p3 ?s3:name) (p4 ?s4:name) (id ?n)))
 )

 (printout t
"-------------------------------------------------------------------------------
-------------" crlf)
)
```

Nel caso in cui, dopo un qualsiasi step, si avrà come risultato *rp = 0* e *mp = 0* vuol dire che sicuramente nessuno dei 4 colori inseriti nella combinazione è presente nel *secret code.*

Avendo questa informazione, verrà fatta la *retract* del codice precedente perché non contiene nessun colore che può essere utile per trovare la combinazione finale e verrà asserito un nuovo codice che non conterrà nessuno dei 4 colori scelti in precedenza.

0 right placed – Y miss placed

```
;SE MIGLIORE
(defrule 0-Y-Migliore (declare (salience -7))
 (answer (step ?n) (right-placed ?rp&:(= ?rp 0)) (miss-placed ?mp&:(> ?mp 0)))
 (guess (step ?n) (g  ?g1 ?g2 ?g3 ?g4) )

 (code (p1 ?c1) (p2 ?c2) (p3 ?c3) (p4 ?c4) (rp ?rp0) (mp ?mp0))
```

```
 (codeS (p1 ?s1) (p2 ?s2) (p3 ?s3) (p4 ?s4))
 (test (> (+ (* ?rp 4) ?mp) (+ (* ?rpO 4) ?mpO)))
 =>
 (printout t "----------" "Siamo in migliore Right placed " ?rp " missplaced " ?mp
crlf)
 ;-------cancello tutto
 (do-for-all-facts ((?var code)) TRUE (retract ?var))
 (do-for-all-facts ((?var codeS)) TRUE (retract ?var))

 ;-------assert questi code
 (assert (code (p1 blank) (p2 blank) (p3 blank) (p4 blank) (rp 0) (mp ?mp) (id
?n)))
 (assert (codeS (p1 ?g1) (p2 ?g2) (p3 ?g3) (p4 ?g4) (id ?n) ))
 (printout t "-----------------------------------------" crlf)
)
```

Nel caso in cui, dopo un qualsiasi step, si avrà come risultato *rp = 0* e *mp = Y* dove Y è un numero compreso tra 0 e 3 (perchè se Y = 4 significa che basterebbe eseguire delle permutazioni per trovare il *secret code*), si potranno distinguere 3 casi.

- *0 rp - Y mp* è peggiore del risultato ottenuto nello step precedente.
  Nella logica umana, in questo caso vuol dire che l'ipotesi fatta non ha portato a risultati migliori rispetto alle ipotesi fatte in precedenza, per questo motivo non ha senso procedere per questa strada ma è conveniente passare ad osservare un altro codice asserito.
  Nel codice CLIPS questa fase corrisponde ad eseguire una *retract* del codice appena testato (il gioco poi continuerà andando a scegliere uno degli altri codici precedentemente asseriti).
- *0 rp - Y mp* è migliore del risultato ottenuto nello step precedente.
  Nella logica umana, in questo caso vuol dire che la nuova ipotesi appena fatta porta a risultati migliori rispetto a quella fatta in precedenza, quindi in questo caso, tutte le possibili strade alternative percorribili possono essere rimosse per concentrarsi nello sviluppare questa perchè tanto è la più promettente.
  In CLIPS questa fase corrisponde ad eseguire una *retract* di tutti i codici precedentemente asseriti per poi eseguire un *assert* dei nuovi code e codeS (come spiegato della **struttura**).
- *0 rp - Y mp* è uguale al risultato ottenuto nello step precedente.
  In questo caso significa che i colori sono presenti ma non sono in posizione giusta, quindi viene fatta una permutazione del codice per verificare allo step successivo se i

colori del nuovo codice sono inseriti in posizione giusta (questa parte avverrà nella fase della scelta della mossa).

X right placed – Y miss placed

```
(defrule X-Y-Peggiore (declare (salience -8))
 (answer (step ?n) (right-placed ?rp&:(> ?rp 0)) (miss-placed ?mp&:(> ?mp 0)))
 (guess (step ?n) (g  ?g1 ?g2 ?g3 ?g4) )
 =>
  ;(test (< (+ (* ?rp 4) ?mp) (+ (* ?rpO 4) ?mpO)))
 (do-for-fact ((?var code))
   (< (+ (* ?rp 4) ?mp) (+ (* ?var:rp 4) ?var:mp))
   (and
     (printout t "----------" "Siamo in peggiore Right placed " ?rp " missplaced "
?mp crlf)
     (do-for-fact ((?varS codeS)) (eq ?varS:id ?var:id) (retract ?varS))
     (retract ?var)
     (printout t "-------------------------------------------" crlf)
   )
 )
  (do-for-fact ((?var code))
   (= (+ (* ?rp 4) ?mp) (+ (* ?var:rp 4) ?var:mp))
   (and
     (printout t "----------" "Siamo in UGUALE Right placed " ?rp " missplaced "
?mp crlf)
     (printout t "-------------------------------------------" crlf)
   )
 )

)
```

Nel caso in cui, dopo un qualsiasi step, si avrà come risultato *rp = X* e *mp = Y*, si potranno distinguere 2 casi.

- Come nel caso *0 rp e Y mp*, se *rp = X* e *mp = Y* è peggiore del risultato precedente vuol dire che l'ipotesi appena scelta non ha portato a miglioramenti rispetto all'ipotesi fatta in precedenza, quindi viene fatta una *retract* di questo codice per permettere al programma di scegliere successivamente un'altro codice già asserito in precedenza.
- Se *rp = X* e *mp = Y* è migliore del risultato precedente, come nel caso *0 rp - Y mp* viene fatta la retract di tutti i codici precedentemente asseriti perché le nuove ipotesi sono migliori delle precedenti per poi asserire i nuovi code e codeS.

X right placed – 0 miss placed

Nel caso in cui, dopo un qualsiasi step, si avrà come risultato *rp = X* e *mp = 0*, si potranno distinguere 2 casi.

- Se *rp = X* e *mp = 0* è peggiore del risultato precedente, come nei casi appena visti viene fatta la *retract* del codice appena scelto per permettere al programma di scegliere successivamente un altro codice già asserito in precedenza.
- Se *rp = X* e *mp = 0* è migliore del risultato precedente, anche in questo caso verrà fatta la *retract* di tutti i codici precedentemente asseriti per poi asserire i nuovi code e codeS.

Strategia 2

Questa seconda strategia adotta un approccio probabilistico basato sulla randomicità dei codici.

Idealmente viene scelto randomicamente un codice e per ogni coppia *<posizione,colore>* viene dato un punteggio in base a quanti rp e quanti mp vengono ottenuti come *answer.* Andando avanti con il gioco questo punteggio andrà a indicare quanto è probabile che un colore stia in una determinata posizione nel codice segreto.
“Il valore indica in modo numerico l’affidabilità di un colore-posizione per un giocatore”

Struttura

Il template *cp* indica le varie coppie *<posizione,colore>*, ognuna è indicata da una posizione nel codice (1, 2, 3 o 4), un colore e un valore che verrà modificato nel corso del gioco

```
(deftemplate cp ; colore-posizione
 (slot posizione) ;indica la posizione nella guess (1 2 3 o 4)
 (slot colore) ; contiene la stringa del colore
 (slot valore) ; valore associato a ciascuna lettera-posizione
)

;(colors blue green red yellow orange white black purple)
(deffacts colore-posizione
 (cp (posizione 1) (colore blue)   (valore 0) )
 (cp (posizione 1) (colore green)  (valore 0) )
 (cp (posizione 1) (colore red)    (valore 0) )
 (cp (posizione 1) (colore yellow) (valore 0) )
 (cp (posizione 1) (colore orange) (valore 0) )
 (cp (posizione 1) (colore white)  (valore 0) )
 (cp (posizione 1) (colore black)  (valore 0) )
```

```
 (cp (posizione 1) (colore purple) (valore 0) )
 ;-------------------------------------------
 (cp (posizione 2) (colore blue)   (valore 0) )
 (cp (posizione 2) (colore green)  (valore 0) )
 (cp (posizione 2) (colore red)    (valore 0) )
 (cp (posizione 2) (colore yellow) (valore 0) )
 (cp (posizione 2) (colore orange) (valore 0) )
 (cp (posizione 2) (colore white)  (valore 0) )
 (cp (posizione 2) (colore black)  (valore 0) )
 (cp (posizione 2) (colore purple) (valore 0) )
 ;-------------------------------------------
 (cp (posizione 3) (colore blue)   (valore 0) )
 (cp (posizione 3) (colore green)  (valore 0) )
 (cp (posizione 3) (colore red)    (valore 0) )
 (cp (posizione 3) (colore yellow) (valore 0) )
 (cp (posizione 3) (colore orange) (valore 0) )
 (cp (posizione 3) (colore white)  (valore 0) )
 (cp (posizione 3) (colore black)  (valore 0) )
 (cp (posizione 3) (colore purple) (valore 0) )
 ;-------------------------------------------
 (cp (posizione 4) (colore blue)   (valore 0) )
 (cp (posizione 4) (colore green)  (valore 0) )
 (cp (posizione 4) (colore red)    (valore 0) )
 (cp (posizione 4) (colore yellow) (valore 0) )
 (cp (posizione 4) (colore orange) (valore 0) )
 (cp (posizione 4) (colore white)  (valore 0) )
 (cp (posizione 4) (colore black)  (valore 0) )
 (cp (posizione 4) (colore purple) (valore 0) )
)
```

Iniziamo definendo tutti i fact riferiti a tutti i colori e le posizioni

Fasi di Gioco

Il gioco si divide in 2 fasi:

- Assegnazione del valore (dal turno 0 al penultimo turno)
- Ricerca del codice (ultimo turno)

Assegnazione del valore

In questa fase, allo step 0 e allo step 1 vengono inseriti staticamente tutti i colori (4 allo step 0 e 4 allo step 1) per essere sicuri che tutti i colori vengano esplorati almeno una volta all’interno di una partita.

Una volta eseguiti i primi due step, per i restanti turni (tranne l'ultimo) vengono generate randomicamente delle combinazioni di codici e per ogni coppia *<posizione,colore>* viene assegnato un valore in base a quanti *rp* e *mp* si ottengono come risposta.

Per evitare di giocare una posizione-colore di cui siamo sicuri non essere nel codice segreto, ad ogni turno quando si effettua la scelta vengono selezionate solo le coppie *cp* con valore positivo.

L'assegnamento del valore avviene nel seguente modo:

- *0 rp e 0 mp*: in questo caso nessun colore è presente nella combinazione finale, quindi per ogni coppia *cp* che contiene quel determinato colore in tutte le posizione associamo un valore molto negativo (-100) così che queste coppie non vengano mai più scelte.

```
(defrule aggiorna-pesi-0-0 (declare (salience -7))
 (answer (step ?s) (right-placed ?rp&:(= ?rp 0)) (miss-placed ?mp&:(= ?mp 0)))
 (guess (step ?s) (g  ?c1 ?c2 ?c3 ?c4) )
 =>
 (printout t "Right placed " ?rp " missplaced " ?mp crlf)

 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c1) (modify ?var (valore (-
?var:valore 100))) )
 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c2) (modify ?var (valore (-
?var:valore 100))) )
 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c3) (modify ?var (valore (-
?var:valore 100))) )
 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c4) (modify ?var (valore (-
?var:valore 100))) )
)
```

- *0 rp e X mp*: in questo caso siamo sicuri che i colori che sono presenti sicuramente non sono nella posizione in cui sono stati inseriti ma aumenta la possibilità che gli stessi colori siano in altre posizioni. Alle coppie *cp* presenti nel codice viene assegnato un valore negativo, mentre a tutte le coppie che hanno i colori del codice ma in diverse posizioni viene assegnato un valore piccolo valore positivo (0.5) per aumentare la probabilità di essere scelti all'ultimo turno.

```
 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c1) (modify ?var (valore (+
?var:valore 0.5))) )
 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c2) (modify ?var (valore (+
```

```
?var:valore 0.5))) )
 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c3) (modify ?var (valore (+
?var:valore 0.5))) )
 (delayed-do-for-all-facts ((?var cp)) (eq ?var:colore ?c4) (modify ?var (valore (+
?var:valore 0.5))) )
```

- *X rp e 0 mp*: in questo caso siamo sicuri che i colori che sono presenti saranno nella posizione corretta e che gli stessi colori non saranno in altre posizioni diverse da quelle giocate.
  Per questo motivo alle coppie *cp* presenti nel codice verrà assegnato un incremento di valore (+1), mentre alle coppie contenenti i colori del codice ma in posizioni diverse da quelle giocate verrà assegnato il valore -100 perchè sicuramente sono sbagliate.
- *X rp e X mp con rp > mp*: in questo caso non si può escludere nessun valore perchè qualsiasi coppia *cp* presente nel codice potrebbe essere corretta, quindi per ognuna di queste coppie viene assegnato un leggero incremento di valore di (+1.5)
- *X rp e X mp con rp < mp*: in questo caso non si può escludere nessun valore perchè qualsiasi coppia *cp* presente nel codice potrebbe essere corretta, quindi per ognuna di queste coppie viene assegnato un leggero incremento di valore di (+0.5)
  In questo caso il valore è più piccolo rispetto al caso in cui *rp > mp* per dare più importanza agli *rp* rispetto agli *mp* (porta più informazione sapere che un colore potrebbe essere in posizione giusta piuttosto che sapere che un colore è presente nel *secret code*)

```
(defrule aggiorna-pesi-X-X-mp (declare (salience -7))
 (answer (step ?s) (right-placed ?rp&:(> ?rp 0)) (miss-placed ?mp&:(> ?mp 0)&:(<=
?rp ?mp)))
 (guess (step ?s) (g  ?c1 ?c2 ?c3 ?c4) )

 =>
 (printout t "Right placed " ?rp " missplaced " ?mp crlf)

 (bind ?cp1 (nth$ 1 (find-fact ((?var cp)) (and (= ?var:posizione 1) (eq
?var:colore ?c1)) ) ))
 (bind ?cp2 (nth$ 1 (find-fact ((?var cp)) (and (= ?var:posizione 2) (eq
?var:colore ?c2)) ) ))
 (bind ?cp3 (nth$ 1 (find-fact ((?var cp)) (and (= ?var:posizione 3) (eq
?var:colore ?c3)) ) ))
 (bind ?cp4 (nth$ 1 (find-fact ((?var cp)) (and (= ?var:posizione 4) (eq
?var:colore ?c4)) ) ))
```

```
 (bind ?v1  (fact-slot-value ?cp1 valore))
 (bind ?v2  (fact-slot-value ?cp2 valore))
 (bind ?v3  (fact-slot-value ?cp3 valore))
 (bind ?v4  (fact-slot-value ?cp4 valore))

 (modify ?cp1 (valore (+ ?v1 0.5)) )
 (modify ?cp2 (valore (+ ?v2 0.5)) )
 (modify ?cp3 (valore (+ ?v3 0.5)) )
 (modify ?cp4 (valore (+ ?v4 0.5)) )
)
```

- *0 rp e 4 mp*: in questo caso particolare siamo sicuri che ci siano esattamente quei 4 colori ma che non sono in quelle posizioni. Partendo da questa informazione viene dato un valore molto negativo (-100) alle coppie *cp* contenenti quei 4 colori in quelle 4 posizioni perchè sicuramente non sono corrette, ma viene inserito un valore di -100 anche a tutte le coppie contenenti gli altri colori in tutte le posizioni perché anche questi sono sicuramente sbagliati.

Ricerca del codice

Nell'ultimo turno di gioco vengono valutate tutte le coppie cp, prese le quattro con maggiore valore, una per ogni posizione e costruito un codice.

Problematiche riscontrate

Sebbene riteniamo che questa strategia probabilistica sia efficiente (soprattutto all'aumentare del numero degli step) non abbiamo ottenuto i risultati sperati. Analizzando la problematica ci siamo resi conto che la funzione *random* implementata in clips non genera i valori in modo totalmente randomico. Essendo una strategia che si basa sul testare i colori in tutte le varie posizioni, avere una random che selezionava sempre gli stessi codici ci ha portato ad avere meno possibilità di individuare il codice corretto.

```
CLIPS> (run)
La tua giocata allo step: 0 -> blue green red yellow
Right placed 0 missplaced 2
La tua giocata allo step: 1 -> black blue green white
Numero possibilità eliminate: 4
Right placed 1 missplaced 1
La tua giocata allo step: 2 -> purple black orange blue
Numero possibilità eliminate: 4
Right placed 1 missplaced 0
La tua giocata allo step: 3 -> yellow black white green
Numero possibilità eliminate: 15
Right placed 0 missplaced 3
```

```
La tua giocata allo step: 4 -> green white yellow red
Numero possibilità eliminate: 19
Right placed 1 missplaced 2
La tua giocata allo step: 5 -> purple white orange blue
Numero possibilità eliminate: 19
Right placed 1 missplaced 1
La tua giocata allo step: 6 -> purple yellow orange blue
Numero possibilità eliminate: 19
Right placed 2 missplaced 0
La tua giocata allo step: 7 -> purple yellow green white
Numero possibilità eliminate: 20
Right placed 2 missplaced 1
La tua giocata allo step: 8 -> purple white green blue
Numero possibilità eliminate: 20
Right placed 0 missplaced 2
Ultimo GIRO: 10
La tua giocata allo step: 9 -> green yellow orange white
You have discovered the secrete code!
```

Strategia 3
(Caso studio di una strategia quasi perfetta)

Questa strategia è una versione modificata dell'algoritmo di Knuth. Nella versione originale dell'algoritmo è possibile avere colori ripetuti, mentre in questa versione no.

L'algoritmo di Knuth modificato funziona nel seguente modo:

- Vengono generate (e asserite) tutte le combinazioni possibili senza ripetizioni
- Viene scelta una combinazione casuale tra quelle possibili generate
- Verrà ricevuta una risposta dal programma che conterrà un certo numero di *rp* e di *mp*
- Nel caso in cui la soluzione sia quella corretta l'algoritmo termina
- Se la soluzione non è corretta vengono eliminati dalle combinazioni generate tutti quei codici che non darebbero lo stesso numero di *rp* e di *mp* se la combinazione appena testata fosse il *secret code*
- Si riparte dal secondo punto

Struttura

La defrule *genera_combinazioni* fa un assert di tutte le possibili combinazioni di colori che possono essere presenti nel codice.

```
(deffacts colori
```

```
 (colore blue)
 (colore green)
 (colore red)
 (colore yellow)
 (colore orange)
 (colore white)
 (colore black)
 (colore purple)
)

(defrule genera_combinazioni
 (colore ?color_1)
 (colore ?color_2&:(neq ?color_2 ?color_1))
 (colore ?color_3&:(neq ?color_3 ?color_2)&:(neq ?color_3 ?color_1))
 (colore ?color_4&:(neq ?color_4 ?color_3)&:(neq ?color_4 ?color_2)&:(neq ?color_4
?color_1))
 =>
 (printout ?color_1 crlf)
 (assert (code (p1 ?color_1) (p2 ?color_2) (p3 ?color_3) (p4 ?color_4)) )
)
```

I template *rp* e *mp* servono a capire quanti right placed e quanti miss placed sono presenti dopo aver testato un codice per capire successivamente come eliminare le combinazioni in base al risultato ottenuto.

```
(deftemplate rp
 (slot valore)
 (slot step)
)
(deftemplate mp
 (slot valore)
 (slot step)
)

(deftemplate code
 (slot p1) (slot p2) (slot p3) (slot p4)
)
```

Fasi di Gioco

Dopo aver generato tutte le possibili combinazioni, è possibile dividere la strategia in 3 fasi. La prima fase sarà quella di scelta del codice da giocare in modo randomico tra quelli asseriti, la seconda fase in cui verranno aggiornati i valori di *rp* e *mp* nei due template per capire cosa eliminare dalla lista delle combinazioni e la terza fase si occuperà di eliminare tutte le combinazioni che non rispettano i vincoli dell'algoritmo.

Scelta della mossa

```
(defrule computer-player-step-n (declare (salience -10))
 (status (step ?n) (mode computer))
 =>
 (bind ?l (length (find-all-facts ((?var code)) TRUE)))
 (printout t "mosse rimanenti  " ?l crlf)

 (bind ?i (random 0 ?l) )
 (bind ?g (nth ?i (find-all-facts ((?var code)) TRUE) ) )

 (bind ?c1  (fact-slot-value ?g p1))
 (bind ?c2  (fact-slot-value ?g p2))
 (bind ?c3  (fact-slot-value ?g p3))
 (bind ?c4  (fact-slot-value ?g p4))

 (assert (guess (step ?n) (g ?c1 ?c2 ?c3 ?c4) ))
 (printout t "La tua giocata allo step: " ?n " -> " ?c1 " " ?c2 " " ?c3 " " ?c4
crlf)

 (pop-focus)
)
```

In questa fase, inizialmente viene generato il numero totale di combinazioni ancora presenti nella lista di quelle asserite. Partendo da questo numero viene generato un indice randomico (compreso tra 0 e il numero totale di combinazioni presenti).
A questo punto verrà selezionato il codice presente nella lista totale di tutte le combinazioni in posizione i-esima (ovvero la posizione calcolata tramite la funzione random).

Il codice selezionato sarà quello che verrà giocato nello step corrente.

Per selezionare il codice è stata utilizzata una funzione randomica per evitare che venissero scelte le combinazioni in ordine e rendere l'algoritmo più casuale possibile.

Aggiornamento pesi

```
(defrule aggiorna-pesi (declare (salience -7))
 (answer (step ?s) (right-placed ?rp) (miss-placed ?mp))
 (guess (step ?s) (g  ?c1 ?c2 ?c3 ?c4) )
 =>
 (assert (rp (valore ?rp) (step ?s)))
 (assert (mp (valore ?mp) (step ?s)))

 (printout t "Right placed " ?rp " missplaced " ?mp crlf)

)
```

In questa fase vengono semplicemente aggiornati i valori di right placed e di miss placed per sapere cosa sarà possibile cancellare dalla lista delle combinazioni nella fase successiva.

Eliminazione combinazioni

In quest'ultima fase vengono effettivamente cancellate le combinazioni che sicuramente non saranno il *secret code* dalla lista totale delle combinazioni.

E' possibile dividere questa fase in due casistiche, quella in cui vengono eliminati i codici in base agli *rp* e quella in cui vengono eliminati i codici in base agli *mp*.

Nel caso degli *rp*:

- se *rp = 0* verranno eliminate tutte le combinazioni che hanno quei colori in quelle esatte posizioni
- se *rp = 1* verranno eliminate tutte le combinazioni che non hanno esattamente un colore in quella esatta posizione
- se *rp = 2* verranno eliminate tutte le combinazioni che non hanno esattamente due colori in quelle esatte posizioni
- se *rp = 3* verranno eliminate tutte le combinazioni che non hanno esattamente tre colori in quelle esatte posizioni

Nel caso degli *mp*:

- se *mp = 0* verranno eliminate tutte le combinazioni che hanno quei colori NON in quelle posizioni
- se *mp = 1* verranno eliminate tutte le combinazioni che non hanno esattamente uno dei colori in posizione diversa da quella giocata nello step
- se *mp = 2* verranno eliminate tutte le combinazioni che non hanno esattamente i due colori in posizioni diverse da quelle giocate nello step
- se *mp = 3* verranno eliminate tutte le combinazioni che non hanno esattamente i tre colori in posizioni diverse da quelle giocate nello step
- se *mp = 4* verranno eliminate tutte le combinazioni che non hanno esattamente i quattro colori in posizioni diverse da quelle giocate nello step

Considerazioni e test finali

Dalle strategie precedenti sono stati ottenuti i seguenti risultati:

- Su 10 step:

	Media	Minimo	Massimo
Strategia 1	6.5	5	9
Strategia 2	7.8	3	9
Strategia 3	5.6	3	9

- Su 20 step:

	Media	Minimo	Massimo
Strategia 1	10,34	2	18
Strategia 2	10.34	6	19
Strategia 3	5.6	3	9

I test sono stati divisi in due fasi, una standard in cui abbiamo giocato su 10 step e una in cui abbiamo giocato su 20.

Le giocate su 20 step sono state fatte perché la strategia umana fatica a trovare una soluzione in sole 10 mosse, in più aumentando il numero di step è molto più evidente quanto la terza strategia (Knuth) sia molto più efficiente delle altre due perchè aumentando il numero di round la media, il valore minimo e il massimo non cambiano, mentre nelle altre si.

Nonostante i risultati ottenuti, ci aspettavamo che la strategia 1 fosse la peggiore perché è quella più simile al ragionamento umano. La strategia 3 doveva essere la migliore strategia implementata delle tre ed effettivamente i risultati sono stati quelli sperati.
Per quanto riguarda la strategia 2, pensiamo che possa essere molto più efficiente se la funzione random di CLIPS funzionasse nel modo corretto, quindi i risultati non sono quelli sperati.

La Strategia 2 presenta un valore minimo inaspettato, in quanto la strategia si concentra sul creare una combinazione possibile solo all'ultimo turno. Questo probabilmente è dovuto alla

scelta delle possibili combinazioni da provare perchè non andiamo a considerare i valori negativi che possono andare ad eliminare un numero elevato di combinazioni.