Intelligenza Artificiale e Laboratorio

Relazione progetto Clips anno 2019/2020

Studenti:

Sanfilippo Paolo

Giuseppe Biondi

Antonio Surdo

Il gioco

Il progetto prevede la costruzione di un sistema esperto che giochi ad una versione semplificata della famigerata Battaglia Navale.

Il gioco è nella versione "in-solitario", dove il nostro sistema esperto in qualità di giocatore deve indovinare la posizione di una flotta di navi distribuite su una griglia 10x10.

Le navi da individuare sono le seguenti:

- 1 corazzata da 4 caselle

- 2 incrociatori da 3 caselle ciascuno

- 3 cacciatorpedinieri da 2 caselle ciascuno

- 4 sottomarini da 1 casella ciascuno

Le navi saranno posizionate in verticale o orizzontale e saranno circondate da uno strato di acqua, in altre parole vi dev'essere almeno una casella water di distanza tra due celle di tipo nave.

Inoltre, in corrispondenza di ciascuna riga e colonna sarà indicato il numero di celle che contengono navi e sarà possibile fornire all'inizio fino ad un massimo di quattro celle conosciute di tipo nave.

Le azioni eseguibili dal nostro sistema esperto saranno:

- Fire x y: permette all'agente di svelare il contenuto della cella con coordinate x e y.
- Guess x y: permette di segnare la casella di coordinate x e y con una "bandierina" che indicherà la cella come possibile tipo nave.
- Unguess x y: permette di rimuovere la bandierina in posizione x e y.
- Solve: invocato al termine per calcolare lo "scoring" finale sulle prestazioni dell'agente.

Ulteriori vincoli sono che l'agente ha un numero limitato di operazioni guess e fire, rispettivamente 20 e 5, e che non potrà eseguire più azioni nella stessa azione. In fine il gioco termina in automatico se si raggiunge il limite di 100 azioni eseguibili.

L'idea generale

Abbiamo deciso di implementare il nostro sistema esperto AGENT come un unico modulo, il quale dopo ogni azione (fire, guess, unguess o solve) restituirà il controllo al modulo MAIN

L'idea generale è di partire dall'osservazione del mondo, cercando di estrapolare quanta più informazione possibile dai fatti iniziali, quindi sarà presenta una serie di regole con priorità più alta finalizzate all'aumento della conoscenza a cui seguiranno altre regole con priorità più bassa per la scelta delle fire da eseguire.

Per memorizzare tale conoscenza abbiamo implementato tre tipi di celle:

- My-k-cell: celle di cui conosciamo esattamente il contenuto: water o la parte specifica della barca (es. Se la barca è orizzontale può essere left, middle o right).
- F-cell: celle che sappiamo per certo siano di tipo nave ma non sappiamo quale parte di essa.
- B-cell: celle in cui abbiamo una possibilità che sia di tipo nave ma non la certezza.

Per ogni nuova cella scoperta avremo un set di informazioni su quelle adiacenti che saranno memorizzate con i tipi appena accennati, lo scopo è quello di scoprire quante più celle possibile per ridurre le probabilità di eseguire azioni di tipo Fire su caselle water.

Inoltre, sfrutteremo la semplificazione di avere un contatore del numero di barche presenti per ogni riga e colonna. La nostra soluzione prevede di decrementare tale contatore ad ogni casella di tipo barca trovata così da inferire una nuova conoscenza:

- se questo raggiunge il valore 0 significa che per quella riga (o colonna) le caselle sconosciute restanti, se vi sono, son di tipo water.
- se il numero di caselle sconosciute è esattamente quello del contatore allora sapremo che sono di tipo nave.

Infine, abbiamo costruito due agenti in cui la parte iniziale di analisi delle informazioni sarà uguale ma useranno diverse euristiche per la scelta delle azioni.

Fatti iniziali

Abbiamo costruito un fatto iniziale per memorizzare la conoscenza legata al numero di barche e il tipo di ognuna:

```
(deffacts numerobarche
      (barca (tipo 4)(num 1))
      (barca (tipo 3)(num 2))
      (barca (tipo 2)(num 3))
      (barca (tipo 1)(num 4)))
```

Template

- **My-k-cell:** Rappresenta celle di cui conosciamo il contenuto. Questo template è reso necessario perché durante l'esecuzione dell'agente, se asserissimo la conoscenza utilizzando le K-cell dell'ambiente queste attiverebbero altre regole indesiderate

```
(deftemplate my-k-cell
      (slot x)
      (slot y)
      (slot content))
```

- **F-cell:** Rappresenta celle di tipo nave ma di cui non conosciamo il contenuto.

```
(deftemplate f-cell
      (slot x)
      (slot y))
```

- **B-cell:** Rappresenta celle con possibilità di essere di tipo nave ma di cui non abbiamo la certezza.

```
(deftemplate b-cell
      (slot x)
      (slot y))
```

- **Visto:** Rappresenta le celle visitate.

```
(deftemplate visto
      (slot x)
      (slot y))
```

- **Barca:** Rappresenta il tipo di barca e il numero di barche esistenti per quel tipo.

```
(deftemplate barca
      (slot tipo)
      (slot num))
```

- **Crea-my-k-cell-water**

```
(deftemplate crea-my-k-cell-water
      (slot x)
      (slot y)
      (slot c))
```

- **Crea-f-cell**

```
(deftemplate crea-f-cell
      (slot x)
      (slot y))
```

- **Crea-b-cell**

```
(deftemplate b-cell
      (slot x)
      (slot y))
```

Le regole

Regole per osservazione iniziale.

Per differenziare la priorità delle diverse regole le abbiamo suddivise in un range di salience compreso tra [-100, 100] dove avremo con priorità positiva quelle con lo scopo di analizzare la conoscenza e con priorità negativa quelle legate alle azioni di fire.

In particolare, avremo una priorità molto alta per due regole specifiche:

- **convert-k-to-my-k:**
  Per iniziare è necessario leggere le conoscenze iniziali e creare le nostre my-k-cell. Questa regola asserisce per ogni K-cell proveniente dall'ambiente una My-k-Cell

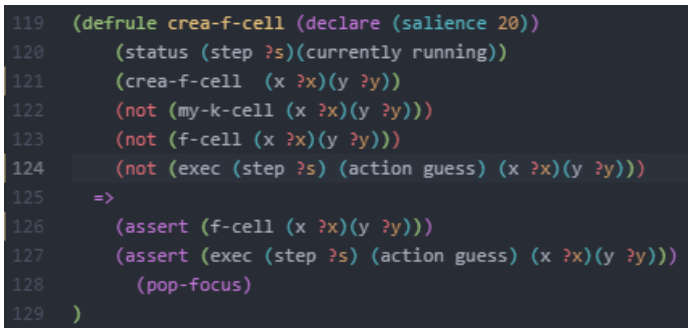
- **decrement-k-row-col-battleship:**
  Questa regola serve per aggiornare i contatori che tengono traccia delle celle di tipo nave da scoprire per ogni riga e colonna. Ad ogni nuova My-k-cell o F-cell verranno decrementati i contatori corrispondenti. Questa regola sfrutta il template "Visto" per assicurarci un corretto decremento.

```
105  (defrule decrement-k-row-col-battleship (declare (salience 100))
106      (or (my-k-cell (x ?x) (y ?y) (content ?c&:(neq ?c water)))
107          (f-cell (x ?x)(y ?y)))
108      ?kpr <- (k-per-row (row ?x) (num ?num-row))
109      ?kpc <-(k-per-col (col ?y) (num ?num-col))
110      (not (visto(x ?x)(y ?y)))
111    =>
112      (modify ?kpr (num (- ?num-row 1)))
113      (modify ?kpc (num (- ?num-col 1)))
114      (assert (visto (x ?x) (y ?y)))
115  )
```

Seguiranno una serie di regole di osservazione ed estrazione delle informazioni con una salience in un range tra 10 e 40, avremo una salience maggiore per quelle regole che ci permetteranno di ottenere più informazioni e preferiamo siano eseguite per prime.

Esse saranno suddivise nei seguenti sottogruppi:

- **Regole per la creazione di celle nuove:**
  La scelta di utilizzare delle regole a parte per la creazione di nuove celle invece di asserirle direttamente nelle regole che le attivano è principalmente dovuta per una pulizia di codice.
  - *crea-f-cell*:
    Questa regola si attiva quando viene asserito il fatto
    `(crea-f-cell (x ?x)(y ?y))`
    Oltre ad asserire la cella di tipo F esegue l'azione di guess su X e Y e restituisce il controllo al modulo MAIN



```
(defrule crea-f-cell (declare (salience 20))
    (status (step ?s)(currently running))
    (crea-f-cell  (x ?x)(y ?y))
    (not (my-k-cell (x ?x)(y ?y)))
    (not (f-cell (x ?x)(y ?y)))
    (not (exec (step ?s) (action guess) (x ?x)(y ?y)))
  =>
    (assert (f-cell (x ?x)(y ?y)))
    (assert (exec (step ?s) (action guess) (x ?x)(y ?y)))
      (pop-focus)
)
```

- *crea-b-cell:*
  In questo caso sono necessari più controlli tra le precondizioni. Per poter asserire una B-cell dobbiamo essere sicuri che non sia presenta già una My-k-cell o una F-cell, che la posizione scelta sia interna alla mappa e che in tale posizione ci sia la possibilità di esserci una barca (se i contatori della riga/colonna scendono a 0 significa che abbiamo già scoperto tutte le barche in quella riga/colonna)

- *my-k-cell-creator-water:*
  Per ogni fatto "*crea-my-k-cell-water*" asserisce una nuova My-k-cell di contenuto water solo se non sono già presenti altre celle (my-k-cell o f-cell) e se la posizione scelta è interna alla mappa.

- *create-my-k-cell-water-in-diagonal:*
  Asserisce quattro fatti *"crea-my-k-cell-water"* nelle diagonali di ogni f-cell e my-k-cell di tipo non water.
  Dal momento che le navi devono distare almeno una casella e non possiamo avere navi in diagonale, possiamo considerare le caselle adiacenti in diagonale di ogni cella di tipo nave come water

- **Regole My-k-cell:**
  Questo insieme di regole si attivano quando l'agente conosce nuove caselle di tipo My-k-cell, ovvero delle quali conosce esattamente il contenuto.
  - *my-k-cell-sub:*
    Per ogni cella di contenuto "sub" creiamo delle My-k-cell di tipo water intorno e asseriamo un fatto "*decrement-sub-counter*" che servirà per diminuire il contatore delle barche di tipo sub.

  - *my-k-cell-left:*
    Per ogni cella di contenuto "left", creiamo My-k-cell water sopra, sotto e a sinistra di essa. Asseriamo un fatto "*crea-f-cell"* nella cella adiacente a destra e "*crea-b-cell"* nella successiva.
    Per ogni cella di tipo left conosciuta sappiamo con certezza che quella adiacente a destra sia parte di barca ma non sappiamo se "middle" o "right". Inoltre, vogliamo memorizzare la possibilità che anche la successiva sia di tipo barca ma non avendone la certezza viene memorizzata come b-cell.

```
(defrule my-k-cell-left (declare (salience 10))
    (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c left)))
  =>
    ; creo my-k-cell water sopra,sotto e a sinistra
    (assert (crea-my-k-cell-water (x (+ ?x 1)) (y ?y) (c water)))
    (assert (crea-my-k-cell-water (x (- ?x 1)) (y ?y) (c water)))
    (assert (crea-my-k-cell-water (x ?x) (y (- ?y 1)) (c water)))

    (assert (crea-f-cell (x ?x) (y (+ ?y 1))))
    (assert (crea-b-cell (x ?x)(y (+ ?y 2))))
    (printout t "Ho trovato una my-k-cell di tipo left in x: " ?x " y: " ?y  crlf)
)
```

- *my-k-cell-right:*
  Ci aspettiamo una cella di tipo “right”, creiamo My-k-cell water sopra, sotto e a destra. Asseriamo “*crea-f-cell”* nella cella a sinistra e “*crea-b-cell”* nella successiva.

- *my-k-cell-top:*
  Ci aspettiamo una cella di tipo “top”, creiamo My-k-cell water a destra, sinistra e sopra. Asseriamo “*crea-f-cell”* nella cella sotto e “*crea-b-cell”* nella successiva.

- *my-k-cell-bot:*
  Ci aspettiamo una cella di tipo “bot”, creiamo My-k-cell water a destra, sinistra e sotto. Asseriamo “*crea-f-cell”* nella cella sopra e “*crea-b-cell”* nella successiva.

Nel caso delle My-k-cell di tipo “middle” la situazione si complica leggermente perché dobbiamo considerare le celle adiacenti per capire se si tratta di una barca orizzontale o verticale.

- *my-k-cell-middle-near-left-or-middle:*
  Se abbiamo una cella di tipo barca a sinistra della nostra middle sappiamo che vi sarà un proseguimento a destra, quindi sicuramente la prossima è una cella di tipo nave e non abbiamo conoscenza per la cella successiva

```
(defrule my-k-cell-middle-near-right-or-middle (declare (salience 10))
    (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c middle)))
    (or (my-k-cell (x ?x) (y =(+ 1 ?y)) (content ?c1&:(neq ?c1 water)))
        (f-cell (x ?x) (y =(+ 1 ?y))))
  =>
    (assert (crea-f-cell  (x ?x)(y (- ?y 1))))
    (assert (crea-b-cell (x ?x)(y (- ?y 2))))
)
```

- *my-k-cell-middle-near-right-or-middle:*
  Asserisce una f-cell nella cella a sinistra e una b-cell nella successiva quando trova una cella di tipo “middle” con una cella di tipo nave a destra.

- *my-k-cell-middle-near-top-or-middle:*
  Asserisce una f-cell nella cella sotto e una b-cell nella successiva quando trova una cella di tipo “middle” con una cella di tipo nave sopra.

- *my-k-cell-middle-near-bot-or-middle:*
  Asserisce una f-cell nella cella sopra e una b-cell nella successiva quando trova una cella di tipo “middle” con una cella di tipo nave sotto.

- *my-k-cell-middle-ultimaspiaggia:*
  Questa regola si attiva solo se abbiamo una cella “middle” circondata da caselle sconosciute, non avendo abbastanza informazioni asseriamo le quattro celle adiacenti sopra, sotto, destra e sinistra come b-cell.

```
(defrule my-k-cell-middle-ultimaspiaggia (declare (salience 5))
    (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c middle)))
    (not (my-k-cell (x =(- ?x 1))(y ?y))) (not (f-cell (x =(- ?x 1))(y ?y)))
    (not (my-k-cell (x =(+ 1 ?x))(y ?y))) (not (f-cell (x =(+ 1 ?x))(y ?y)))
    (not (my-k-cell (x ?x)(y =(- ?y 1)))) (not (f-cell (x ?x)(y =(- ?y 1))))
    (not (my-k-cell (x ?x)(y =(+ 1 ?y)))) (not (f-cell (x ?x)(y =(+ 1 ?y))))
  =>
    ;creao b cell sopra-sotto-destra-sinistra
    (assert (crea-b-cell  (x (- ?x 1))(y ?y)))
    (assert (crea-b-cell  (x (+ ?x 1))(y ?y)))
    (assert (crea-b-cell  (x ?x)(y (- ?y 1))))
    (assert (crea-b-cell  (x ?x)(y (+ ?y 1))))
)
```

Son stati poi ideate due regole più specifiche nel caso fosse incontrata una cella di tipo middle adiacente ad un bordo della mappa. Questa regola ha una priorità leggermente maggiore perché ci assicura di conoscere l’andamento della barca e quindi asserire direttamente due f-cell:

- *my-k-cell-middle-near-vertical-border:*
  Nel caso la cella middle è adiacente ad uno dei bordi verticali avremo una f-cell sopra e sotto seguite entrambe da una b-cell.

```
(defrule my-k-cell-middle-near-vertical-border (declare (salience 30))
    (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c middle)))
    (or (test(eq ?y 0)) (test(eq ?y 9)))
  =>
    (assert (crea-f-cell  (x (- ?x 1))(y ?y)))
    (assert (crea-b-cell (x (- ?x 2))(y ?y)))
    (assert (crea-f-cell  (x (+ ?x 1))(y ?y)))
    (assert (crea-b-cell (x (+ ?x 2))(y ?y)))
)
```

- *my-k-cell-middle-near-horizontal-border:*
  Nel caso la cella middle è adiacente ad uno dei bordi orizzontali, creeremo le f-cell e b-cell a destra e sinistra.

Inoltre, nel caso avessimo due celle di tipo middle adiacente, essendo le barche al massimo lungo quattro caselle, possiamo concludere che siamo in presenza di una barca da quattro. Decidiamo quindi di asserire due caselle water che delimitano i bordi della barca per facilitare l’attivazione delle regole in F-cell che si occuperanno della loro conversione in My-k-cell.

- *kmid-near-kmid-ver:*
  Se abbiamo due celle middle verticali, creiamo celle water due caselle prima e tre caselle dopo della cella middle più in alto.

```
(defrule kmid-near-kmid-ver (declare (salience 20))
    (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c middle)))
    (my-k-cell (x =(+ ?x 1)) (y ?y) (content ?c&:(eq ?c middle)))
  =>
    (assert (crea-my-k-cell-water (x (+ ?x 3)) (y ?y) (c water)))
    (assert (crea-my-k-cell-water (x (- ?x 2)) (y ?y) (c water)))
)
```

- *kmid-near-kmid-hor:*
  Se abbiamo due celle middle orizzontali, creiamo celle water due caselle prima e tre caselle dopo della cella middle più a sinistra.

- **Regole F-cell:**
  Questo gruppo contiene delle regole che han lo scopo di riconoscere la parte di barca delle nostre f-cell che come accennato rappresentano celle di tipo nave ma di cui non sappiamo il contenuto specifico.

  - *convert-f-to-my-k-cell-if-right:*
    Converte una f-cell in my-k-cell di tipo “right” se è presente una cella di tipo nave a sinistra e, o c’è una casella di tipo water a destra o è adiacente al bordo destro della mappa.

```
(defrule convert-f-to-my-k-cell-if-right (declare (salience 20))
    ?fcell <- (f-cell (x ?x)(y ?y))
    (or (f-cell (x ?x) (y =(- ?y 1)))
      (my-k-cell (x ?x) (y =(- ?y 1)) (content ?c1&:(neq ?c1 water))))
    (or (my-k-cell (x ?x)(y =(+ 1 ?y))(content ?c&:(eq ?c water)))
        (test(>= ?y 9)))
  =>
    (assert (my-k-cell (x ?x) (y ?y) (content right)))
    (retract ?fcell)
    (printout t "trasformo la fcell in kcell right: " ?x " " ?y crlf)
)
```

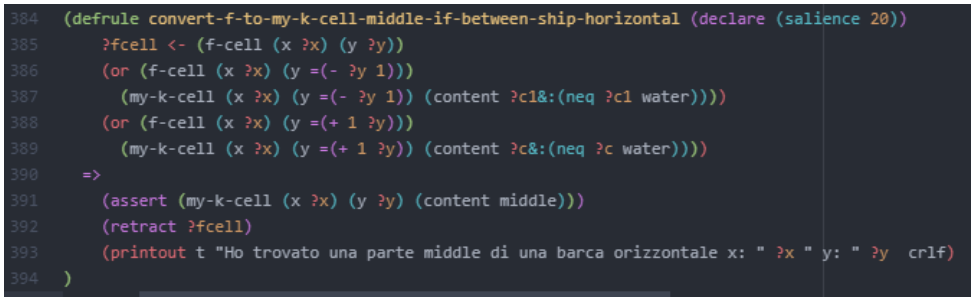
  - *convert-f-to-my-k-cell-if-left.*
  - *convert-f-to-my-k-cell-if-top.*
  - *convert-f-to-my-k-cell-if-bot.*

  - *convert-f-to-my-k-cell-sub-with-waterframe:*
    Converte una f-cell in my-k-cell “sub” se ad ogni suo lato son presenti caselle di tipo water o il bordo della mappa.

```
(defrule convert-f-to-my-k-cell-sub-with-waterframe (declare (salience 20))
    ?fcell <- (f-cell (x ?x)(y ?y))
    (or (my-k-cell (x =(- ?x 1)) (y ?y) (content water)) (test(<= ?x 0))) ;sopra
    (or (my-k-cell (x =(+ ?x 1)) (y ?y) (content water)) (test(>= ?x 9))) ;sotto
    (or (my-k-cell (x ?x) (y =(+ ?y 1)) (content water)) (test(>= ?y 9))) ;destra
    (or (my-k-cell (x ?x) (y =(- ?y 1)) (content water)) (test(<= ?y 0)));sinistra
  =>
    (retract ?fcell)
    (assert (my-k-cell (x ?x) (y ?y) (content sub)))
)
```

Dal momento che sappiamo per certo che le f-cell son parti di nave, se troviamo una f-cell compresa tra altre due parti di nave possiamo asserire che sia di tipo “middle” e quindi convertirla.

- *convert-f-to-my-k-cell-middle-if-between-ship-horizontal:*



```
(defrule convert-f-to-my-k-cell-middle-if-between-ship-horizontal (declare (salience 20))
    ?fcell <- (f-cell (x ?x) (y ?y))
    (or (f-cell (x ?x) (y =(- ?y 1)))
      (my-k-cell (x ?x) (y =(- ?y 1)) (content ?c1&:(neq ?c1 water))))
    (or (f-cell (x ?x) (y =(+ 1 ?y)))
      (my-k-cell (x ?x) (y =(+ 1 ?y)) (content ?c&:(neq ?c water))))
  =>
    (assert (my-k-cell (x ?x) (y ?y) (content middle)))
    (retract ?fcell)
    (printout t "Ho trovato una parte middle di una barca orizzontale x: " ?x " y: " ?y  crlf)
)
```

- *convert-f-to-my-k-cell-middle-if-between-ship-vertical.*

Inoltre, come vedremo nella parti successive relative ai fatti sul numero di barche trovate, se son state trovate tutte le barche di tipo due e abbiamo una f-cell vicina ad una cella estremo (left, right, top o bot) possiamo escludere la possibilità che si tratti di una barca da due e quindi convertire tale cella in una “middle”.

- *convert-f-to-k-middle-when-battleship-type-two-already-been-found:*

```
(defrule convert-f-to-k-middle-when-battleship-type-two-already-been-found (declare (salience 20))
    (barca (tipo 2)(num ?t&:(eq ?t 0)))
    ?fcell <-(f-cell (x ?x) (y ?y))
    (or
      (my-k-cell (x =(+ ?x 1)) (y ?y) (content ?c&:(eq ?c bot)))
      (my-k-cell (x =(- ?x 1)) (y ?y) (content ?c1&:(eq ?c1 top)))
      (my-k-cell (x ?x) (y  =(+ ?y 1)) (content ?c2&:(eq ?c2 right)))
      (my-k-cell (x ?x) (y  =(- ?y 1)) (content ?c3&:(eq ?c3 left)))
    )
  =>
    (retract ?fcell)
    (assert (my-k-cell (x ?x) (y ?y) (content middle)))
)
```

- **Regole B-cell:**
  Il seguente set di regole prevede delle situazioni in cui siamo sicuri che non è possibile avere una parte di barca nella posizione della b-cell e provvediamo alla sua eliminazione.

  - *delete-b-cell-if-my-k-cell:*
    Elimina la b-cell se esiste una my-k-cell nella stessa posizione.

  - *delete-b-cell-where-col-is-zero:*
    Elimina la b-cell se il contatore per la colonna raggiunge lo zero.

  - *delete-b-cell-where-row-is-zero:*
    Elimina la b-cell se il contatore per la riga raggiunge lo zero.

- **Regole legate ai contatori di riga e colonna:**
  Son state implementate quattro regole legate oltre a quella già accennata con priorità alta per decrementare tali valori.
  Le prime due prevedono che quando un contatore arriva a 0, quindi non vi sono ulteriori parti di navi da cercare in quella riga o colonna, asserisce un fatto "*crea-my-k-cell-water"* per ogni posizione cella di quella riga o colonna.
  - *create-my-k-cell-water-when-row-value-is-zero:*

```
(defrule create-my-k-cell-water-when-row-value-is-zero (declare (salience 30))
    (k-per-row (row ?row) (num ?num-row&:(eq ?num-row 0)))
  =>
    (loop-for-count (?i 0 9) do
      (assert (crea-my-k-cell-water (x ?row) (y ?i) (c water))))
)
```

  - *create-my-k-cell-water-when-col-value-is-zero.*

  Al contrario, quando il contatore avrà un numero maggiore di 0, se il numero di caselle restanti sconosciute per tale riga o colonna sarà uguale al contatore potremo infierire che si tratta di celle di tipo nave.

  - *create-f-cell-in-row-where-i-know-all-water-cells:*
    Utilizziamo "find-all-facts" per cercare tutte le celle che già conosciamo, consideriamo anche le f-cell perché contribuiscono al decremento del valore del contatore.
    Sommiamo il numero di caselle trovate e quindi conosciute al nostro contatore e se il risultato è 10, il numero esatto di caselle per tale riga o colonna, possiamo asserire un "crea-f-cell" per ogni posizione che soddisfa i nostri criteri.

```
(defrule create-f-cell-in-row-where-i-know-all-water-cells (declare (salience 20))
    (k-per-row (row ?x) (num ?num-row&:(> ?num-row 0)))
    (status (step ?s)(currently running))
    (test (eq (+  (+ (length$ (find-all-facts ((?f my-k-cell)) (eq ?f:x ?x)))
                     (length$ (find-all-facts ((?f1 f-cell)) (eq ?f1:x ?x)))) ?num-row) 10))
    (k-per-col (col ?y)(num ?num-col&:(> ?num-col 0)))
    (not (my-k-cell (x ?x) (y ?y)))
    (not (f-cell (x ?x) (y ?y)))
  =>
    (assert (crea-f-cell (x ?x)(y ?y)))
)
```

  - *create-f-cell-in-col-where-i-know-all-water-cells.*

- **Regole barche trovate:**
  In questa sezione troviamo due sottogruppi di regole: il primo è finalizzato alla ricerca delle barche, quando troveremo una serie di my-k-cell comprese tra due estremi il sistema decrementerà il contatore relativo.

  - *found-battleship-type-four:*
    Asserisce il fatto `(decrement-fourth-counter)` se trova una my-k-cell left (o top) seguita da due middle e una right(o bot).

```
(defrule found-battleship-type-four(declare (salience 40))
  (or
    (and  (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c left)))
          (my-k-cell (x ?x) (y =(+ 1 ?y)) (content ?c1&:(eq ?c1 middle)))
          (my-k-cell (x ?x) (y =(+ 2 ?y)) (content ?c2&:(eq ?c2 middle)))
          (my-k-cell (x ?x) (y =(+ 3 ?y)) (content ?c3&:(eq ?c3 right))))
    (and  (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c top)))
          (my-k-cell (x =(+ 1 ?x)) (y ?y) (content ?c1&:(eq ?c1 middle)))
          (my-k-cell (x =(+ 2 ?x)) (y ?y) (content ?c2&:(eq ?c2 middle)))
          (my-k-cell (x =(+ 3 ?x)) (y ?y) (content ?c3&:(eq ?c3 bot))))
  )
  =>
    (assert (decrement-fourth-counter))
)
```

- *found-battleship-type-three:*
  Asserisce il fatto `(decrement- triple-counter)` se trova una my-k-cell left (o top) seguita da una middle e una right (o bot).

- found-battleship-type-two:
  Asserisce il fatto `(decrement- double-counter)` se trova una my-k-cell left (o top) seguita da una right (o bot).

- *decrement-battleship-type-four-counter:*
  Decrementa il contatore relative alle barche di tipo quattro.

```
(defrule decrement-battleship-type-four-counter(declare (salience 40))
    ?dsc <- (decrement-fourth-counter)
    ?nb <- (barca (tipo 4)(num ?t))
  =>
    (modify ?nb (num (- ?t 1)))
    (retract ?dsc)
    (printout t "-Trovata barca da quattro-"  crlf)
)
```

- *decrement-battleship-type-three-counter:*
  Decrementa il contatore relative alle barche di tipo tre.

- *decrement-battleship-type-two-counter:*
  Decrementa il contatore relative alle barche di tipo due.

- *decrement-battleship-sub-counter:*
  Decrementa il contatore relative alle barche di tipo sub.

Il secondo sottogruppo delle regole sulle barche ha come obiettivo la realizzazione di esse, in base al tipo di navi ancora da trovare potremo asserire delle caselle di tipo water per limitare i bordi e attivare così le regole di conversione viste in precedenza.

La scelta di asserire delle caselle water per sfruttare le regole precedenti è dettata dal problema che non sappiamo quale delle celle della nostra barca bisogna convertire, sarebbero stato necessario un numero maggiore di regole per considerare la posizione di ogni f-cell.

Inoltre, non avendo implementato una regola per la creazione di my-k-cell, asserendole direttamente da queste regole avrebbe potuto ricreare il fatto e di conseguenza riattivare regole già eseguite. Al contrario la creazione di celle water ha una regola con dei controlli specifici:

- *add-water-if-battleship-type-three-horizontal:*
  Se abbiamo già trovato la nave di lunghezza quattro, se abbiamo tre celle di tipo nave, quindi my-k-cell o f-cell, vicine possiamo asserire che si tratta di una nave completa.

```
(defrule add-water-if-battleship-type-three-horizontal (declare (salience 20))
      (barca (tipo 4)(num ?t&:(eq ?t 0)))
      (my-k-cell (x ?x) (y ?y) (content ?c&:(eq ?c middle)))
      (or (my-k-cell (x ?x) (y =(- ?y 1)) (content ?c&:(neq ?c water)))
          (f-cell (x ?x) (y =(- ?y 1))))
      (or (my-k-cell (x ?x) (y =(+ ?y 1)) (content ?c1&:(neq ?c1 water)))
          (f-cell (x ?x) (y =(+ ?y 1))))
    =>
      (assert (crea-my-k-cell-water (x ?x) (y (- ?y 2)) (c water)))
      (assert (crea-my-k-cell-water (x ?x) (y (+ ?y 2)) (c water)))
)
```

- *add-water-if-battleship-type-three-vertical.*

- *add-water-if-battleship-type-two-horizontal:*
  Date due celle di tipo nave adiacenti ed entrambi i contatori per le navi da tre e quattro pari a zero, asseriamo due celle water rispettivamente prima e dopo la nostra nave.

- *add-water-if-battleship-type-two-vertical.*

- *add-water-if-battleship-type-one:*
  Quando i contatori delle barche di tipo due, tre e quattro scendono a zero, le restanti f-cell sono navi di tipo sub e possiamo di conseguenza asserire celle water intorno.

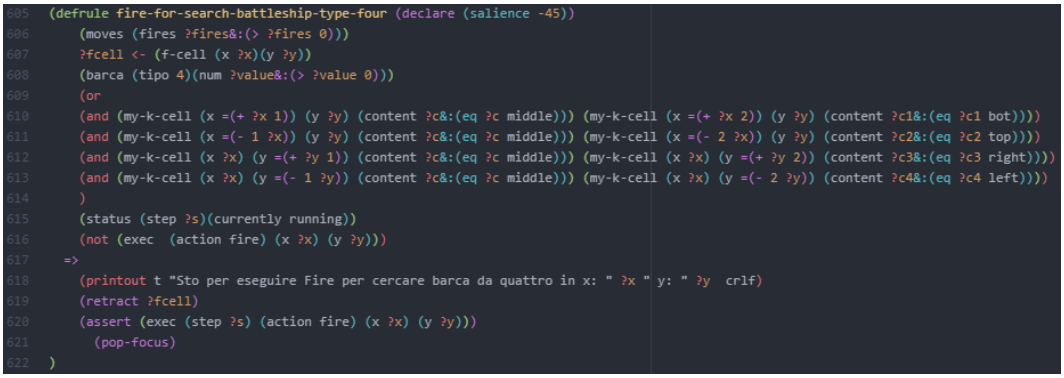
Regole Fire.

Come accennato in precedenza son stati implementati due agenti che sfruttano euristiche diverse per la ricerca di nuove parti di nave e quindi l'utilizzo delle azioni Fire.

Tuttavia sono state implementate delle regole comuni che prevedono situazioni particolari nelle quali preferiamo approfondire la ricerca in quanto le informazioni che ne deriverebbero ci aumenterebbero le probabilità di non sprecare le successive Fire.

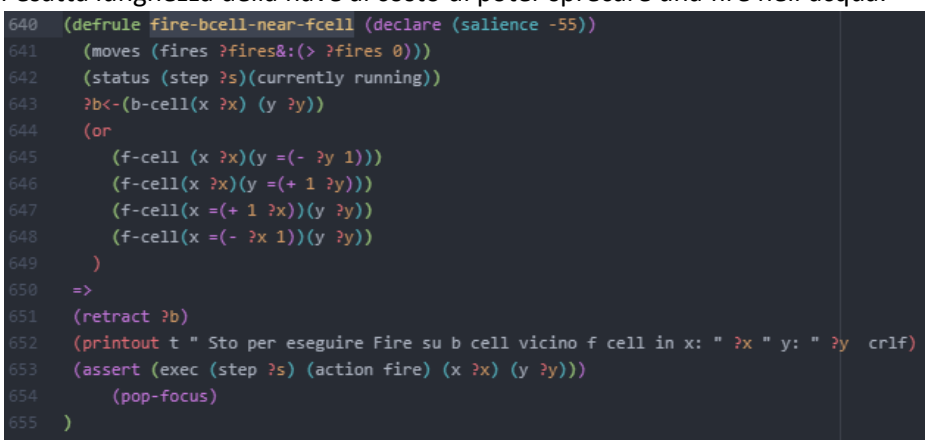
Dal momento che si tratta di un unico modulo per entrambi gli agenti queste due fire che affronteremo avranno una salience maggiore rispetto a quella sull'euristica.

- *fire-for-search-battleship-type-four:*
  Esegue l'azione Fire su una f-cell che segue una my-k-cell di tipo middle che a sua volta segue una my-k-cell estremo (left, right, top o bot). Questa fire si attiva soltanto se la barca di tipo quattro non è ancora stata scoperta.



```
(defrule fire-for-search-battleship-type-four (declare (salience -45))
    (moves (fires ?fires&:(> ?fires 0)))
    ?fcell <- (f-cell (x ?x)(y ?y))
    (barca (tipo 4)(num ?value&:(> ?value 0)))
    (or
    (and (my-k-cell (x =(+ ?x 1)) (y ?y) (content ?c&:(eq ?c middle))) (my-k-cell (x =(+ ?x 2)) (y ?y) (content ?c1&:(eq ?c1 bot))))
    (and (my-k-cell (x =(- 1 ?x)) (y ?y) (content ?c&:(eq ?c middle))) (my-k-cell (x =(- 2 ?x)) (y ?y) (content ?c2&:(eq ?c2 top))))
    (and (my-k-cell (x ?x) (y =(+ ?y 1)) (content ?c&:(eq ?c middle))) (my-k-cell (x ?x) (y =(+ ?y 2)) (content ?c3&:(eq ?c3 right))))
    (and (my-k-cell (x ?x) (y =(- 1 ?y)) (content ?c&:(eq ?c middle))) (my-k-cell (x ?x) (y =(- 2 ?y)) (content ?c4&:(eq ?c4 left))))
    )
    (status (step ?s)(currently running))
    (not (exec  (action fire) (x ?x) (y ?y)))
  =>
    (printout t "Sto per eseguire Fire per cercare barca da quattro in x: " ?x " y: " ?y  crlf)
    (retract ?fcell)
    (assert (exec (step ?s) (action fire) (x ?x) (y ?y)))
      (pop-focus)
)
```

- *fire-bcell-near-fcell:*
  Esegue l'azione Fire su una b-cell se vicina ad una f-cell. L'idea è quella di massimizzare la ricerca sulle barche incomplete in maniera tale da poter sfruttare il più possibile le regole legate al numero di barche trovate.
  Per questa regola vi era la possibilità di eseguire la fire sulla f-cell anzi che sulla b-cell portando ad uno scoring finale maggiore in quanto avremmo colpito sicuramente una nave.
  Tuttavia, spesso la nave restava ancora incompleta. Si pensi all'esempio di una left vicino una f-cell, la scoperta di una middle non ci dava abbastanza informazioni per determinare se fosse una nave da tre o da quattro. Al contrario colpendo la b-cell in qualunque caso ci permetteva di scoprire l'esatta lunghezza della nave al costo di poter sprecare una fire nell'acqua.



```
(defrule fire-bcell-near-fcell (declare (salience -55))
  (moves (fires ?fires&:(> ?fires 0)))
  (status (step ?s)(currently running))
  ?b<-(b-cell(x ?x) (y ?y))
  (or
     (f-cell (x ?x)(y =(- ?y 1)))
     (f-cell(x ?x)(y =(+ 1 ?y)))
     (f-cell(x =(+ 1 ?x))(y ?y))
     (f-cell(x =(- ?x 1))(y ?y))
   )
 =>
  (retract ?b)
  (printout t " Sto per eseguire Fire su b cell vicino f cell in x: " ?x " y: " ?y  crlf)
  (assert (exec (step ?s) (action fire) (x ?x) (y ?y)))
      (pop-focus)
)
```

Inoltre, è stata implementata una regola che nel caso la fire non avesse successo e quindi colpisse l'acqua vorremmo che venga creata la my-k-cell water.

- *add-my-k-cell-water-if-fire-fail:*
  Questa regola controlla che siamo in uno stato successive alla fire eseguita e che non esista nessuna cella conosciuta in tale posizione.

```
(defrule add-my-k-cell-water-if-fire-fail (declare (salience 20))
    (exec (step ?s) (action fire) (x ?x) (y ?y))
    (status (step ?s1&:(> ?s1 ?s))(currently running))
    (not (my-k-cell (x ?x) (y ?y)))
  =>
    (assert(my-k-cell(x ?x) (y ?y)(content water)))
)
```

L'agente incrocio

L'euristica di questo agente consiste nel mirare nelle caselle dove abbiamo i contatori della riga e colonna più alti e di conseguenza ci aspettiamo una più probabile presenza di navi nell'incrocio

Questa euristica viene utilizzata sia per le azioni di tipo fire che per le azioni di tipo guess quando non abbiamo più altre conoscenze da estrapolare e quindi cerchiamo di utilizzare tutte le bandierine restanti per coprire quante più zone possibili

- *fire-where-krow-kcol-have-max-value:*
  Per trovare la cella su cui eseguire la nostra azione controlliamo che non esistano celle con valori dei contatori maggiori e per le quali non sia già stata effettuata l'azione fire.

```
(defrule fire-where-krow-kcol-have-max-value (declare (salience -65))
    (moves (fires ?fires&:(> ?fires 0)))
    (k-per-row (row ?x) (num ?num-row))
    (k-per-col (col ?y) (num ?num-col))
    (not (and
      (k-per-row (row ?x2) (num ?num-row2&:(> ?num-row2 ?num-row)))
      (k-per-col (col ?y2) (num ?num-col2&:(> ?num-col2 ?num-col)))
      (not (exec  (action fire) (x ?x2) (y ?y2)))
    ))
    (status (step ?s)(currently running))
    (not (exec  (action fire) (x ?x) (y ?y)))
    (not (my-k-cell (x ?x) (y ?y)))
  =>
    (printout t " FIRE incrocio in x: " ?x " y: " ?y  crlf)
    (assert (exec (step ?s) (action fire) (x ?x) (y ?y)))
    (pop-focus)
)
```

Lo stesso procedimento si applica anche ai guess finali.

In questo caso abbiamo affrontato una scelta legata ai contatori della riga e colonna: le opzioni erano di eseguire la guess lasciando inalterati i valori o di creare una f-cell, che come abbiamo visto oltre a creare l'azione di guess decrementa anche i valori dei contatori.

In base ai test eseguiti abbiamo ottenuto risultati più positivi nella prima scelta, infatti il decremento dei contatori nel caso in cui avessimo effettivamente colpito la nave dava ottimi

risultati evitando di usare le guess finali che avrebbero sicuramente colpito l'acqua. Tuttavia nel caso la guess non colpisse una nave non solo avremmo la guess fallita ma ci priveremmo anche la possibilità di colpire la cella di quella riga (o colonna) con le guess successive.

Questa differenza si è notata principalmente in questo agente in quanto utilizza un'euristica meno efficace.

- *guess-where-krow-kcol-have-max-value:*
  Esattamente come la fire precedente con l'unica differenza che controlliamo che non ci siano più fire da eseguire.

```
660  (defrule guess-where-krow-kcol-have-max-value (declare (salience -65))
661      (status (step ?s)(currently running))
662      (moves (fires 0) (guesses ?g&:(> ?g 0)))
663      (k-per-row (row ?x) (num ?num-row))
664      (k-per-col (col ?y) (num ?num-col))
665      (not (exec  (action guess) (x ?x)(y ?y)))
666      (not (my-k-cell (x ?x) (y ?y)))
667      (not (and
668        (k-per-row (row ?x1)(num ?num-row2&:(> ?num-row2 ?num-row)))
669        (k-per-col (col ?y1)(num ?num-col2&:(> ?num-col2 ?num-col)))
670        (not (my-k-cell (x ?x1) (y ?y1)))
671        (not (exec  (action guess) (x ?x1)(y ?y1)))
672      ))
673    =>
674      (printout t " guess finali in x: " ?x " y: " ?y  crlf)
675      (assert (exec (step ?s) (action guess) (x ?x)(y ?y)))
676        (pop-focus)
677  )
```

L'agente probabilità

Potremmo considerarlo un'evoluzione del precedente, l'idea è sempre di sfruttare i contatori per determinare le celle con probabilità di essere di tipo nave maggiore.

In questo caso determiniamo una probabilità associata a tutte le celle candidate, ovvero quelle sconosciute, questa è ottenuta tramite un calcolo matematico scritto in funzioni.

Calcoliamo il rapporto tra il valore del contatore della riga (o della colonna) e il numero di caselle sconosciute su tale riga (o colonna). Successivamente otteniamo la probabilità associata ad una cella come il prodotto dei due rapporti ottenuti rispettivamente usando riga e colonna.

In termini matematici:

$$Probabilità(X,Y) = \frac{krow(X)}{csr(X)} * \frac{kcol(Y)}{\text{csc }(Y)}$$

$$krow, kcol = contatori\ riga, colonna$$

$$csr, csc = caselle\ sconosciute\ riga, colonna$$

Vediamo ora le funzioni scritte in clips:

- *get-known-cell-for-row:*
  Calcola la somma di tutte le celle my-k-cell e f-cell conosciute sulla riga "row"

```
(deffunction get-known-cell-for-row (?row)
  (+
    (length$ (find-all-facts ((?f my-k-cell)) (eq ?f:x ?row)))
    (length$ (find-all-facts ((?f1 f-cell)) (eq ?f1:x ?row)))
  )
)
```

- *get-known-cell-for-col.*

- *calculate-prob-x-y:*
  Calcola la probabilità sulla cella X,Y dati i valori dei contatori della riga (krow) e della colonna (kcol).

```
(deffunction calculate-prob-x-y (?num-row ?num-col ?x ?y)
  (*
    (/ ?num-row (- 10 (get-known-cell-for-row ?x)))
    (/ ?num-col (- 10 (get-known-cell-for-col ?y)))
  )
)
```

- *my-predicate:*
  Questo predicato viene utilizzato nella funzione *find-max* per confrontare le probabilità su due celle.
  Per semplificare e non dover passare un numero esagerato di parametri in questo caso le X e Y, così come i valori dei contatori li estraiamo utilizzando *fact-slot-value.* Quindi i krow e kcol ricevuti in input sono i fatti interi rispettivamente per la prima e la seconda casella candidata

```
(deffunction my-predicate (?krow1 ?kcol1 ?krow2 ?kcol2)
  (> (calculate-prob-x-y (fact-slot-value ?krow1 num) (fact-slot-value ?kcol1 num)
      (fact-slot-value ?krow1 row) (fact-slot-value ?kcol1 col))
     (calculate-prob-x-y (fact-slot-value ?krow2 num) (fact-slot-value ?kcol2 num)
      (fact-slot-value ?krow2 row) (fact-slot-value ?kcol2 col)))
)
```

- *find-max:*
  Restituisce la cella con probabilità più alta che non sia una my-k-cell e per cui non sia già stata eseguita una guess, in quanto useremo questa funzione sia per le fire che per i guess finali.
  Il controllo sulle fire eseguite non è necessario in quanto abbiamo visto la regola che per ogni fire fallita crea una cella water, di conseguenza è sufficiente controllare le my-k-cell.

```
(deffunction find-max ()
   (bind ?max FALSE)
   (do-for-all-facts
     ((?kr k-per-row) (?kc k-per-col))
     (and (> ?kr:num 0) (> ?kc:num 0)
         (not (any-factp ((?mk my-k-cell)) (and (eq ?mk:x ?kr:row) (eq ?mk:y ?kc:col))))
         (not (any-factp ((?ex exec)) (and (eq ?ex:action guess) (eq ?ex:x ?kr:row) (eq ?ex:y ?kc:col)))))
     (if (or (not ?max) (my-predicate ?kr ?kc (nth$ 1 ?max) (nth$ 2 ?max)))
        then
        (bind ?max (create$ ?kr ?kc))))
    (return ?max)
)
```

A questo punto le regole di fire e guess finali saranno molto semplici in quanto dovranno semplicemente invocare la funzione find-max che restituirà un array contenente i fatti di krow e kcol dai quali basterà estrarre il valore della riga e della colonna per avere le nostre X e Y su cui eseguire l'azione.

- *fire-probability:*
  Eseguiamo un test nelle precondizioni per assicurarci che sia stato trovato un massimo, nel caso in cui tutti i contatori scendessero a zero, e quindi tutte le barche son state scoperte, la funzione find-max ritornerebbe "false".
  Tramite "nth$ 1" otteniamo il primo elemento dell'array restituito da find-max, ovvero il fatto krow dal quale estraiamo il valore di row, cioè la nostra X.

```
(defrule fire-probability (declare (salience -65))
    (moves (fires ?fires&:(> ?fires 0)))
    (status (step ?s)(currently running))
    (test (find-max))
  =>
    (printout t " FIRE sulla probabilità in x: " (fact-slot-value (nth$ 1 (find-max)) row)
                                          " y: " (fact-slot-value (nth$ 2 (find-max)) col)  crlf)
    (assert (exec (step ?s) (action fire) (x (fact-slot-value (nth$ 1 (find-max)) row))
                                          (y (fact-slot-value (nth$ 2 (find-max)) col))))
      (pop-focus)
)
```

- *guess-probability:*
  Come per l'agente precedente abbiamo preferito l'eseguire l'azione guess lasciando inalterati i valori dei contatori. A differenza della fire cambia il controllo sul numero di fire e guess.

```
(defrule guess-probability (declare (salience -65))
    (moves (fires 0) (guesses ?g&:(> ?g 0)))
    (status (step ?s)(currently running))
    (test (find-max))
  =>
    (printout t " guess sulla probabilità in x: " (fact-slot-value (nth$ 1 (find-max)) row)
                                          " y: " (fact-slot-value (nth$ 2 (find-max)) col)  crlf)

    (assert (exec (step ?s) (action guess) (x (fact-slot-value (nth$ 1 (find-max)) row))
                                          (y (fact-slot-value (nth$ 2 (find-max)) col))))
      (pop-focus))
```

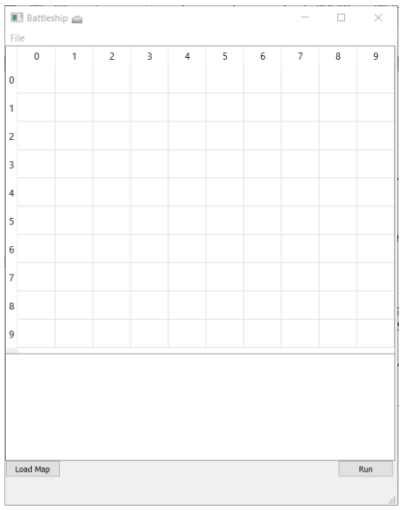
In fine avremo la regola *solve* con priorità minima che asserirà l'azione di 'solve' per concludere.

App

Per concludere il lavoro sviluppato si è creata un'applicazione fornita di GUI in python che permette di eseguire il lavoro.

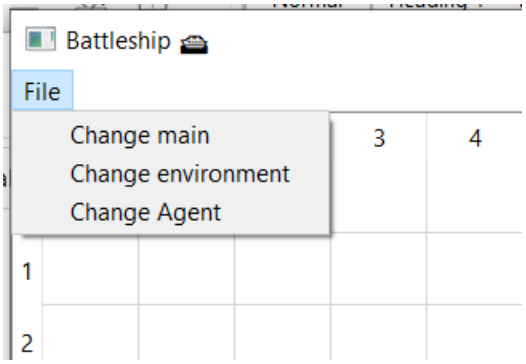
Le librerie più importanti utilizzate per questa applicazione sono:

- PyQt5 utilizzata per creare l'interfaccia gui del nostro programma in python;



- Clipspy invece permette di eseguire il nostro sistema esperto in CLIPS, offrendo degli strumenti sia per l'esecuzione vera e propria sostituendo così l'operato di "CLIPSIDE" sia per gestire i risultati ottenuti dall'esecuzione.

- altre librerie secondarie utilizzate per facilitare il lavoro sono: sys, os, numpy.

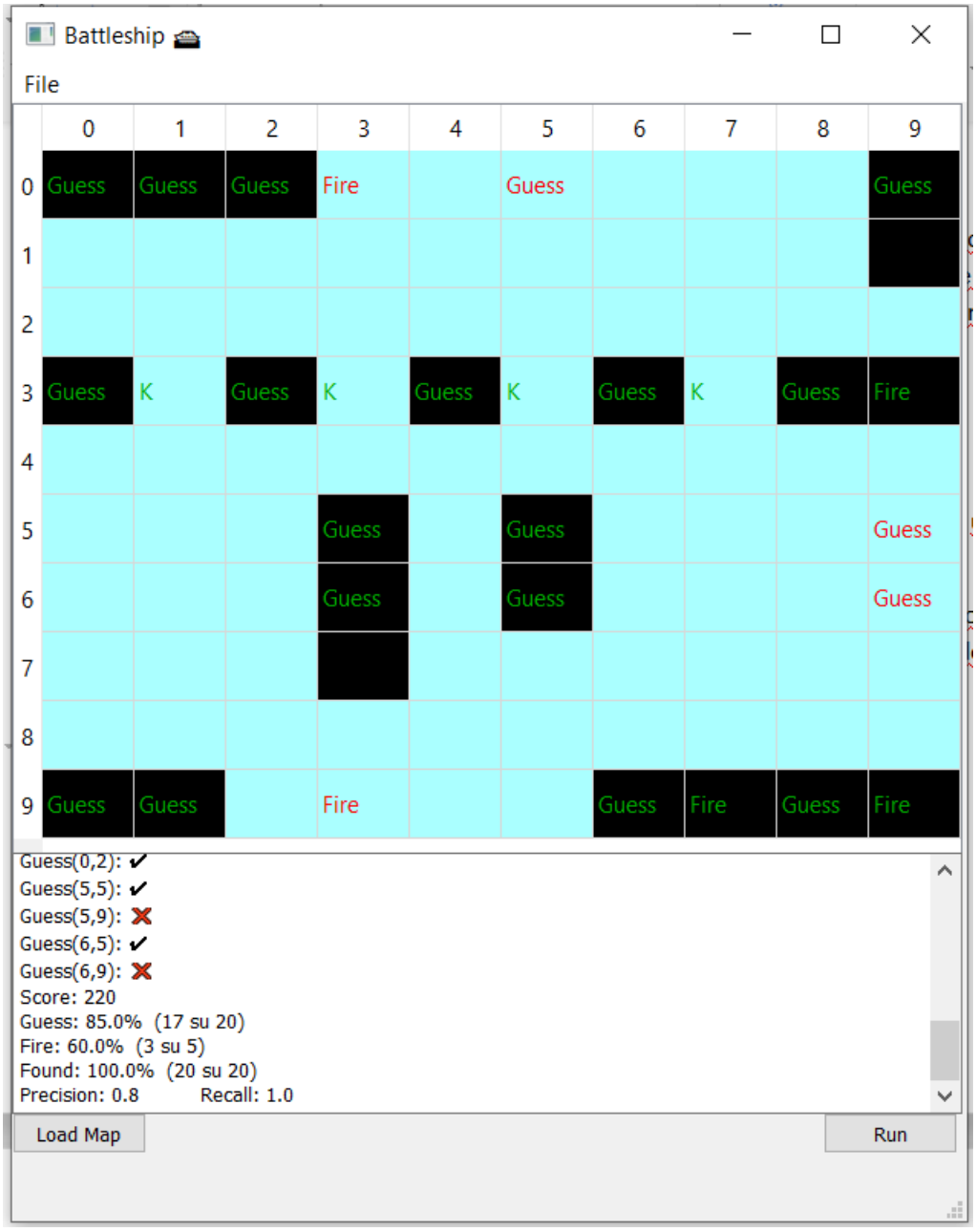


Nell'applicazione sarà possibile cambiare il settaggio di default avendo la possibilità di scegliere l'agente (nel nostro caso incrocio o probabilistico) oppure cambiare il Main o l'Environment.

Le due schermate principali sono la "griglia" dove verrà caricata la mappa ed una finestra di dialogo dove verranno stampate le operazioni e i risultati finali.

Per caricare la mappa in basso a sinistra è presente un bottone "Load Map" che permette appunto di selezionare la mappa sulla quale dovrà "giocare" l'agente, invece in basso a destra sarà presente un bottone run per avviare l'applicativo.

Qui rappresentato uno screen di esecuzione della nostra GUI in python:



Le celle che hanno al loro interno una “K” sono le k-cell cioè le celle le quali il contenuto è conosciuto nella knowledge base iniziale.

Mappe e risultati

Abbiamo raccolto in queste due tabelle i risultati dei due agenti.

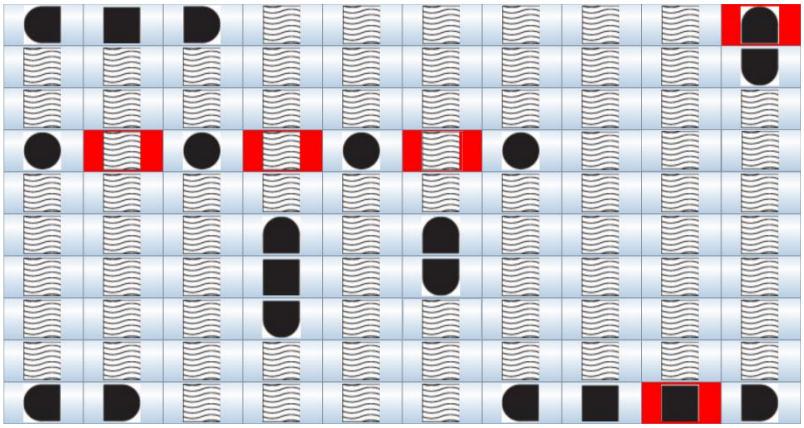
Per come son stati costruiti i due agenti ci aspettavamo risultati leggermente migliori sull'agente probabilistico in quanto effettua un calcolo più accurato rispetto a quello ad incrocio, questo ci dà una maggiore accuratezza ma rende un po' più pesante il lavoro in termini computazionali e spaziali in quanto deve fare più calcoli. Infatti, l'agente probabilistico ha una media di 196 punti, quello ad incrocio ha una media di 151 punti.

Dai risultati e dalle mappe abbiamo notato che l'agente ad incrocio si muove bene quando le barche son meno sparse e tendo a raggrupparsi sulle stesse righe e colonne, questo permette di avere una forte informazione usando le sole krow e kcol maggiori. L'agente probabilistico invece è più accurato e riesce a muoversi anche nelle situazioni più complesse, un esempio è la mappa 11 dove oltre a non avere conoscenze le barche son tra loro molto sparse, infatti l'agente ad incrocio ottiene uno dei suoi risultati peggiori.

		Agente probabilistico			
	fire / guess	found	precision	recall	score
mappa 1	4 su 5 / 9 su 9	13 su 13 (100%)	1	0,93	255
mappa 2	4 su 5 / 12 su 20	16 su 16 (100%)	0,64	1	165
mappa 3	4 su 5 / 12 su 20	16 su 16 (100%)	0,64	1	165
mappa 4	2 su 5 / 16 su 16	18 su 18 (100%)	0,86	1	255
mappa 5	2 su 5 / 14 su 16	16 su 16 (100%)	0,76	1	205
mappa 6	4 su 5 / 12 su 12	16 su 16 (100%)	0,94	1	285
mappa 7	3 su 5 / 13 su 16	16 su 16 (100%)	0,76	1	215
mappa 8	4 su 5 / 9 su 20	13 su 18 (72,2%)	0,52	0,72	-35
mappa 9	5 su 5 / 13 su 13	18 su 18 (100%)	1	1	330
mappa 10	5 su 5 / 15 su 17	20 su 20 (100%)	0,91	1	320
mappa 11	5 su 5 / 12 su 20	17 su 20 (85%)	0,68	0,85	125
mappa 12	4 su 5 / 11 su 20	15 su 18 (83,3%)	0,6	0,83	65

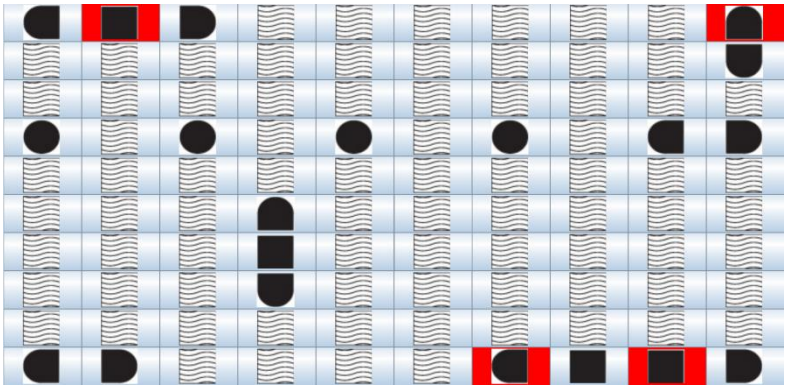
		Agente Incrocio			
	fire / guess	found	precision	recall	score
mappa 1	3 su 3 / 10 su 10	13 su 13 (100%)	1	1	280
mappa 2	4 su 5 / 11 su 19	15 su 16 (93,8%)	0,62	0,94	130
mappa 3	4 su 5 / 11 su 19	15 su 16 (93,8%)	0,62	0,94	130
mappa 4	5 su 5 / 13 su 13	18 su 18 (100%)	1	1	330
mappa 5	4 su 5 / 12 su 20	16 su 16 (100%)	0,64	1	165
mappa 6	1 su 4 / 15 su 15	16 su 16 (100%)	0,84	1	235
mappa 7	3 su 4 / 13 su 13	16 su 16 (100%)	0,94	1	285
mappa 8	3 su 5 / 11 su 19	14 su 18 (77,8%)	0,58	0,78	20
mappa 9	5 su 5 / 12 su 20	17 su 18 (94,4%)	0,68	0,94	175
mappa 10	4 su 5 / 14 su 16	18 su 20 (90%)	0,86	0,9	225
mappa 11	3 su 5 / 10 su 20	13 su 20 (65%)	0,52	0,65	-80
mappa 12	2 su 5 / 9 su 20	11 su 18 (61,1%)	0,44	0,61	-35

Alcuni esempi di mappe su cui si sono fatti i test sui due agenti (nella cartella del progetto sono presenti tutte le immagini delle mappe):



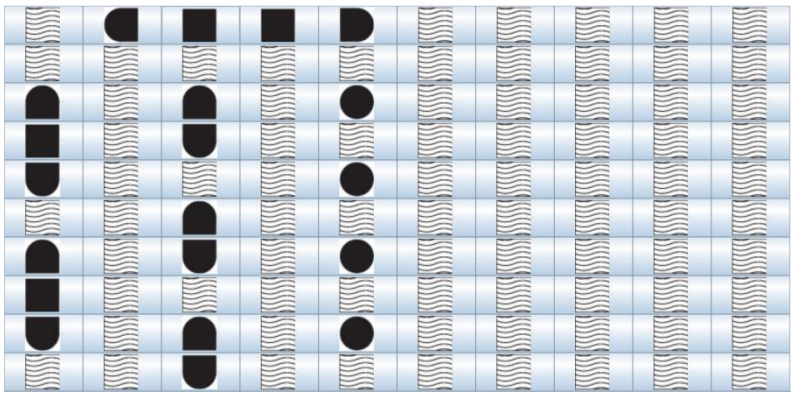
Mappa 4.

Una delle mappe dove l'agente ad incrocio ha avuto dei risultati migliori. Le barche son leggermente sparse ma i valori di krow e kcol son meno omogenei (in alcune zone ci son valori più alti quindi una maggior concentrazione di barche). Questa concentrazione di valori di kcol e krow più le celle scoperte all'inizio (k-cell) rendono il lavoro più facile per il criterio dell'agente ad incrocio.



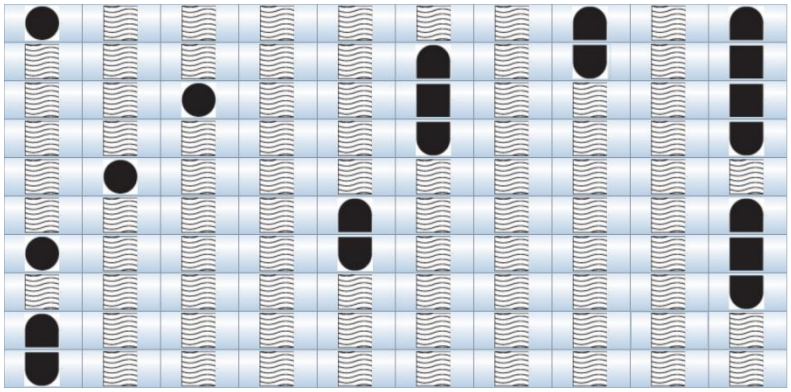
Mappa 5.

Una mappa simile alla precedente ma con una diversa gestione delle k-cell. Nella precedente entrambi gli agenti riuscivano ad inferire subito la presenza delle sub, qui risulta un po' più difficile ecco perché l'agente ad incrocio perde di efficienza



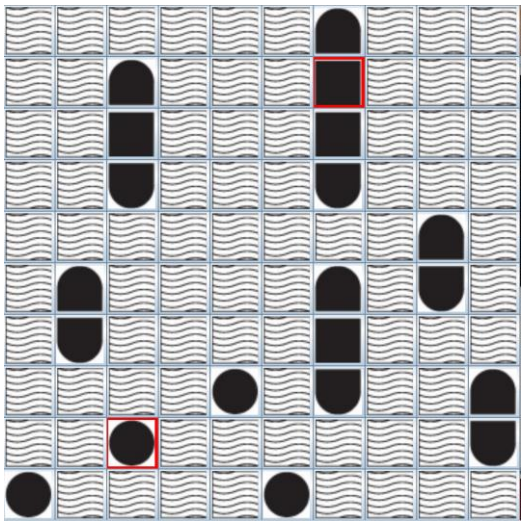
Mappa 10.

In questa mappa si è volutamente cercata la difficoltà non scoprendo nessuna cella (k-cell), però le navi sono molto raggruppate (per questo motivo l'agente ad incrocio non ottiene un brutto risultato).



Mappa 11.

Questa mappa è simile alla precedente ma è più difficile in quanto le barche sono più sparse. Da qui si nota come una minor concentrazione di barchè favorisce l'agente ad incrocio.



Mappa 12

Questa è la mappa che più rappresenta la differenza fra l'agente probabilistico (65 punti) e l'agente ad incrocio (-35 punti). Si può notare come in questa mappa le barche oltre ad essere ben sparse, hanno anche un dei valori più omogenei di kcol e krow (ad esempio nella mappa 5 o 4 le barche sono più sparse ma hanno valori di krow e kcol meno omegenei)

In conclusione, come detto, l'agente probabilistico è il più accurato in quanto effettua delle analisi più precise su dove eseguire fire e guess. Questa prestazione influisce su una maggiore complessità che purtroppo non siamo riusciti a quantificare perché soprattutto in termini di tempo entrambe le risoluzioni ci mettono poco e la differenza di tempo è di pochi millisecondi, quindi è difficile riuscire a fare una stima.

L'agente incrocio riesce a lavorare meglio quando le barche son più raggruppate tra loro e i valori di krow e kcol son meno omogenei essendo più significativi.

In particolare l'agente probabilistico sfrutta meglio la conoscenza su tutte le celle, comprese le water, e quindi quelle mappe dove riusciamo ad aumentare le celle scoperte fornisce prestazioni più efficienti.

In poche occasioni è riuscito a far meglio dell'agente probabilistico, però oltre ai fattori citati prima influisce anche dove vengono fatte le operazioni (guess e fire) in quanto le fire fatte in alcuni punti anche se meno considerate dall'agente probabilistico, danno più informazione rispetto a fire fatte in altri punti.