

# Risolutore di Flow

Saverio Monaco

**Abstract**—La funzione di questo programma è di risolvere i puzzle di tipologia simile a quella dell'applicazione *Flow*, trovando tutte le soluzioni possibili. L'algoritmo è ispirato alla tecnica di Backtracking e, data la complessità computazionale del problema, si è cercato di lavorare principalmente sulla velocità di esecuzione a discapito della memoria.

Sono state inoltre implementate delle funzioni dedicate alla risoluzione degli esatti livelli dell'applicazione ufficiale e il gioco da terminale.

quadrata di lato  $L$ , con i punti della coppia posti agli estremi della prima diagonale:

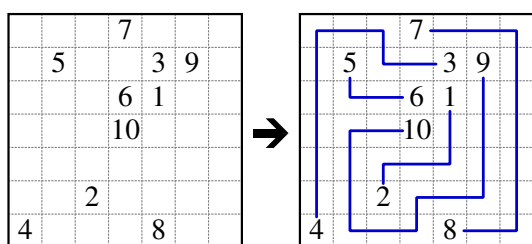
L	# Percorsi
2	2
3	12
4	184
5	8512
6	1262816
7	575780564

## I. IL PUZZLE

Flow è un puzzle logico nel quale il giocatore deve riuscire a connettere tutte le coppie in una griglia tramite percorsi, muovendosi nelle 4 direzioni e con la difficoltà che questi percorsi non possono intersecarsi.

Nell'applicazione originale le coppie vengono distinte tramite colori, in questo articolo invece, per evitare un utilizzo eccessivo di colori e per rappresentare effettivamente come sono state salvate le informazioni in memoria, vengono usati dei numeri, in particolare la prima coppia è composta dai punti 1 e 2, la seconda dai numeri 3 e 4, e così via.

Ad esempio:

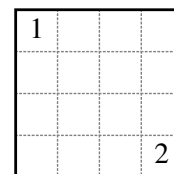


## II. L'ALGORITMO RISOLUTIVO

### L'algoritmo

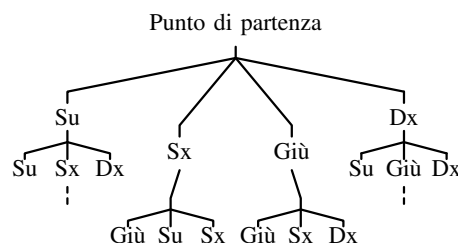
Trovare la soluzione di un determinato puzzle di Flow è di complessità non polinomiale<sup>1</sup>, per avere un'idea di ciò, si è riportato il calcolo del numero dei percorsi, di una coppia in una griglia

<sup>1</sup>Numberlink - Wikipedia



Per la ricerca dell'algoritmo migliore si è ricorso all'approccio *Divide et Impera*: si è scomposto il problema della ricerca delle soluzioni complessive nel problema della ricerca di tutti i percorsi delle singole coppie. Si troveranno quindi tutti i percorsi della prima coppia, e per ciascun percorso trovato si passa alla ricerca di quelli della seconda e così via.

Per trovare tutti i percorsi dato un punto di partenza e uno di arrivo è possibile considerare una struttura ad albero con ogni nodo rappresentante uno dei 4 movimenti possibili.



Dal nodo iniziale (il punto di partenza) ci si potrà spostare nelle 4 direzioni, dopo di che il *branching ratio* (il numero di nodi figlio per nodo madre) sarà al più di 3, perché tornare indietro non è uno spostamento possibile.

Ovviamente non sempre tutte le 3 direzioni saranno disponibili, quindi il numero di nodi figlio

potrà essere minore di tre fino a diventare 0 una volta che nessuna cella adiacente sarà libera.

Muovendosi sui nodi dell'albero è possibile trovare tutti i percorsi possibili, il metodo scelto per esplorare l'intero albero in maniera ordinata è quello del *Backtracking*, quindi seguendo le seguenti regole: Per trovare tutti i percorsi bisogna "muoversi" sui nodi dell'albero tenendo conto di tutti quelli visti in precedenza, il metodo scelto per fare ciò è quello del *Backtracking*, ovvero seguendo le seguenti regole:

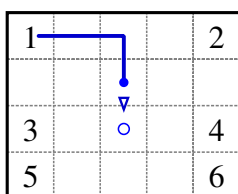
- ogni volta possibile, si scende di un livello spostandosi nel nodo figlio più a sinistra disponibile;
- se non è più possibile scendere di livello, si sale di uno e si prosegue nel nodo adiacente a destra.

Nell'Appendice A viene mostrato come dovrebbe funzionare passo per passo l'algoritmo appena introdotto per un caso semplice di una griglia 3x3.

Poiché ciascun nodo dell'albero può avere fino a 3 nodi figlio bisognerà trovare un modo per trovare e scartare quelli che non porteranno a nessuna soluzione, questi metodi si chiamano *metodi di potatura dell'albero* e se vengono applicati a nodi poco profondi permettono di risparmiare un gran numero di chiamate a funzione.

Per questo articolo è stato applicato solo un metodo di potatura dell'albero il quale verifica che durante la ricerca dei percorsi tutte le righe e colonne abbiano abbastanza celle libere da poter permettere il collegamento di tutte le coppie divise.

Per chiarire il funzionamento di questo metodo si è portato un esempio:



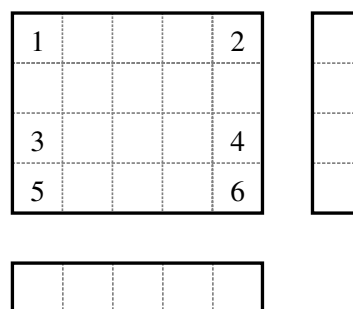
Lo spostamento indicato in figura non porterà ovviamente a nessuna soluzione complessiva, poiché le coppie 3-4 e 5-6 dovranno entrambe passare da una singola cella. Evitare di compiere quella azione, farebbe risparmiare esattamente 155 mosse, e questo è solo uno dei tanti tagli che si possono applicare in questa stessa configurazione. In totale

se si usa l'algoritmo per risolvere questo puzzle si dovranno fare 3311 mosse, mentre applicando questo metodo, se ne faranno solo 1037.

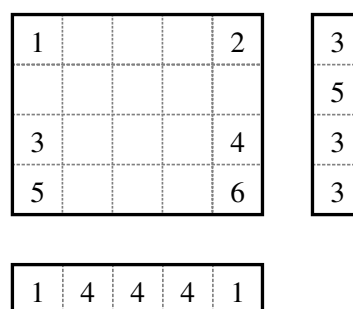
In sostanza si deve verificare che ogni riga e colonna abbia più celle libere che numero di coppie che esse hanno diviso.

Per fare ciò bisogna utilizzare due array di interi, uno che controlla tutte le righe, e uno che controlla tutte le colonne, nella seguente maniera:

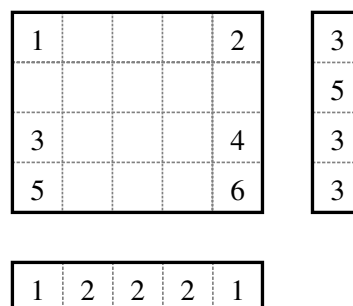
- 1) Si inizializzano i due array di interi:



- 2) Si dà ad ogni valore dei due array, il numero di celle libere della rispettiva riga o colonna:



- 3) A questo valore si sottrae il numero di coppie che la riga o colonna separa:



Il valore delle celle rappresenta il numero di celle che possono essere occupate in tale riga o colonna, senza che si incorra nel caso senza soluzione precedentemente descritto.

- 4) Infine ogni volta che si cercano i percorsi per una nuova coppia, bisogna aggiungere '1' ad ogni cella dei due array relativi alla riga o colonna che ha diviso tale coppia perché durante le verifiche non si deve considerare la coppia corrente.

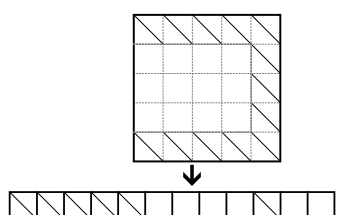
Ad ogni spostamento bisogna aggiornare e verificare le celle relative alla posizione interessata, se uno dei valori diventa inferiore a '0', la mossa non è valida: ci sarebbero meno celle libere, che coppie divise, che non potranno collegarsi.



#### *Le strutture utilizzate*

La funzione della ricerca dei percorsi si basa sulla tecnica del Backtracking che in codice corrisponde ad una funzione ricorsiva. Poiché questa funzione dovrà chiamare se stessa un gran numero di volte, è stato pensato di utilizzare variabili globali al fine di farne passare il minor numero possibile in input per non riempire immediatamente lo stack a funzione, cercando di lavorare sulla cache.

Le strutture adoperate sono:

- **int\* pgrid**  
per rendere il codice più veloce è stato ritenuto migliore utilizzare un array di interi monodimensionale per rappresentare tutta la griglia.  
Le celle libere vengono salvate con l'intero '0', i due punti della prima coppia rispettivamente con gli interi '1' e '2', quelli della seconda coppia con gli interi '3' e '4' e così via.  
Una griglia costruita in tal modo però avrebbe l'ultima cella di una riga adiacente alla prima della successiva, quindi durante l'esecuzione dell'algoritmo si potrebbe incorrere a mosse errate in prossimità dei bordi, per ovviare a questo problema senza dover fare eccessivi controlli sulla posizione si è pensato di inizializzare l'array con un contorno di celle occupate, nella seguente maniera:



-  rappresenta una cella di contorno,
-  rappresenta una cella libera della griglia.

Inoltre per segnalare il contorno bisogna usare un valore intero non corrispondente a nessuno utilizzato per le coppie, perciò si è scelto '-1'.

- **int\* pcoppie**  
contiene le informazioni su i punti delle coppie, il primo intero indica il numero totale delle coppie, i restanti indicano la posizione dei punti delle coppie sulla griglia. L'array delle coppie serve sia a facilitare alcune funzioni secondarie (come la stampa), ma soprattutto per dire alla macchina da dove partire per trovare i vari percorsi durante la funzione ricorsiva.
- **int\* pcontrolx, pcontroly**  
Sono gli array di interi con lo scopo di "potatura" dell'albero già introdotti. In base alle informazioni dentro questi array, è possibile predire se una mossa potrà portare o meno ad una soluzione complessiva.

#### *Le funzioni principali*

Con griglia e strutture già inizializzate, la parte principale dell'algoritmo risolutivo è composto da 3 funzioni principali:

- **flow\_risolutore():**  
Lo scopo di questa funzione è inizializzare le strutture di controllo, impostare le variabili `numerosoluzioni` a '0' e `corsorecoppia` a '-1', in particolare quest'ultima funge da indice per l'array delle coppie **pcoppie** ogni volta bisogna cambiare coppia all'interno della ricorsione. Infine chiama la funzione `flow_inizializzaprossimacoppia()`
- **flow\_inizializzaprossimacoppia()**  
Quando viene chiamata da `flow_risolutore()` ed ogni volta che una coppia trova un percorso, viene lanciata questa funzione che aggiorna l'indice `corsorecoppia` passando alla coppia successiva. Prima di tutto viene fatto un controllo se abbiamo finito le coppie:

---

**Codice 2.1**


---

```

cursorecoppia+=2;

/* Ci si chiede se non esistono altre coppie, in
   caso si è trovata una soluzione complessiva */
if(cursorecoppia > (*(pcoppie)*2))
{
    /* Si è trovata una soluzione */
    numerosoluzioni++;

    /* Si stampa la soluzione */
    flow_grafica_stampasoluzione();

    /* Si torna alla coppia precedente */
    cursorecoppia-=2;

    return;
}

```

Nel caso in cui la coppia invece esiste, bisogna aggiornare le strutture di controllo come detto in precedenza e poi bisogna far partire la ricerca dei percorsi:

---

**Codice 2.2**


---

```

/* Si aggiornano le strutture al cambio di coppia
*/
flow_strutturaausiliarie_successivo();

/* Si fa partire dalla ricerca dei percorsi nelle 4
   direzioni, ricordando che la griglia è un vettore
   di interi */
flow_trovapercorsi(*(pcoppie+cursorecoppia)-riga);
flow_trovapercorsi(*(pcoppie+cursorecoppia)-1);
flow_trovapercorsi(*(pcoppie+cursorecoppia)+riga);
flow_trovapercorsi(*(pcoppie+cursorecoppia)+1);

```

Poiché la funzione ricorsiva è sempre la stessa, viene ogni volta controllata anche la cella di provenienza; questo problema sarebbe potuto essere aggirato tramite la creazione di 4 funzioni ricorsive diverse anziché una, ma per mantenere semplice la stesura del codice si è scelta questa versione ad una funzione ricorsiva.

Arrivato a questo punto del codice si sono esaminati tutti i percorsi per quella coppia dati i percorsi delle coppie precedenti, bisogna quindi ri-aggiornare le strutture di controllo e modificare `cursorecoppia`:

---

**Codice 2.3**


---

```

flow_strutturaausiliarie_precedente();

cursorecoppia-=2;

```

- **flow\_trovapercorsi(int cursore)**  
 Il contenuto della cella sovrastante viene verificato solo dopo aver chiamato la funzione ricorsiva in tale cella, quindi all'inizio della funzione bisogna controllare se la cella interessata è una cella libera, occupata o quella che si sta cercando:

---

**Codice 2.4**


---

```

/* Se la cella è libera... */
if((*(pgrid+cursore))==0)
{
    /* Si deve verificare se la mossa è lecita
       tramite le strutture di controllo */
    if(!flow_strutturaausiliarie_controllo(cursore))
    {return 0;}

    /* Se si è arrivati qui vuol dire che la mossa
       è lecita */
    /* Si può occupare la cella, quindi scriviamo
       sull'array della griglia il numero giusto as-
       sociato alla coppia e aggiorniamo le strutture
       di controllo */
    *(pgrid+cursore)=cursorecoppia;
    flow_strutturaausiliarie_aggiorna(cursore);

    /* Si lancia la stessa funzione nelle quattro
       direzioni */
    flow_trovapercorsi(cursore-riga);
    flow_trovapercorsi(cursore-1);
    flow_trovapercorsi(cursore+riga);
    flow_trovapercorsi(cursore+1);

    /* Arrivato a questo punto della funzione
       bisogna segnalare libera la cella in questione*/
    *(pgrid+cursore)=0;
    flow_strutturaausiliarie_riaggiorna(cursore);

    /* Non serve controllare gli altri casi */
    return;
}

/* Se la cella è quella che stiamo cercando... */
if((*(pgrid+cursore))== (cursorecoppia+1))
{
    flow_inizializzaprossimacoppia();
}

```

---

Il caso in cui la cella è occupata e bisogna

ritornare indietro è contemplato pure dal momento che nessun `if` andrà bene.

### III. IL GIOCO DA TERMINALE

Tramite la funzione

**`flow_game_start(int base, int altezza, int numerocoppie)`** è possibile giocare al gioco di flow, risolvendo un puzzle in una griglia 'base'×'altezza' e con 'numerocoppie' coppie.

Per creare un puzzle con soluzione si generano delle coppie con posizione casuale e si verifica se esiste soluzione, in caso contrario vengono ri-generate le posizioni delle coppie fino a quando non viene trovato un puzzle con almeno una soluzione. Ciò in codice si scrive:

---

#### Codice 3.5

---

```
do
{
    inizializzagriglia(base, altezza);
    generacoppie_rand(numerocoppie);
} while(!trovaprimopercorso())
```

---

Tramite una funzione simile a quella della ricerca dei percorsi, è possibile verificare se la soluzione immessa dall'utente è valida.

### IV. CONCLUSIONI

Il programma creato riesce a risolvere tutti i livelli dell'app originale fin'ora provati in tempi relativamente brevi (il tempo più lungo registrato è stato 650 secondi), inoltre l'introduzione delle strutture di controllo porta nella maggior parte dei casi, una notevole diminuzione dei tempi di esecuzione. Per avere una idea di ciò, si è cronometrato il tempo di risoluzione di alcuni puzzle con e senza il metodo di potatura:

Griglia	Coppie	Tempo senza potatura (s)	Tempo con potatura (s)
9x9	10	8.27	1.06
9x9	9	35.91	1.20
9x9	8	465.73	7.42
9x9	7	5198.30	650.81

È chiaro quindi che l'introduzione di questo particolare metodo di potatura dell'albero delle mosse è stato un fattore determinante nella bontà

complessiva del programma.

Sarebbe possibile migliorare ulteriormente l'algoritmo nella risoluzione dei livelli dell'app sapendo che la soluzione finale prevede che tutte le celle siano occupate, quindi creando strutture ausiliarie apposite in modo da eliminare preventivamente tutte quelle mosse che portano a risultati invalidi.

Per gli altri casi, se non si volessero trovare tutte le soluzioni esistenti, allora si potrebbero introdurre algoritmi euristici che potrebbero trovare un percorso in maniera più veloce.

# APPENDICE A ALGORITMO STEP-BY-STEP

