Versione 1.0

26/03/2019



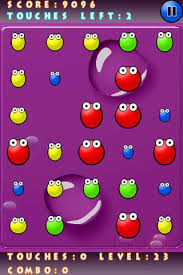
Presentato da: Marco dal zovo

Introduzione

Il progetto in questione consiste nell’implementazione di un algoritmo di ottimizzazione relativamente al gioco *Bubble Blast*.

il gioco

Disponibile nei vari *AppStore* per dispositivi mobile, *Bubble Blast* è un gioco di strategia in cui l’unico scopo che il giocatore deve perseguire è quello di far **esplodere** tutte le bolle presenti nel campo di gioco entro un determinato numero di mosse.



Ogni bolla potrà presentarsi in **tre** diversi stati:

* Sgonfia
* Gonfia
* In procinto di esplodere

All’inizio del gioco le bolle vengono inserite in maniera **casuale** all’interno del campo di gioco e casuali sono anche i loro stati. Ogni turno il giocatore potrà far **avanzare** di stato (per esempio da *sgonfia* a *gonfia*) una bolla selezionandola.

Quando una bolla già *in procinto di esplodere* viene selezionata, quest’ultima **esploderà** **propagando** l’esplosione sugli assi cartesiani (con origine la bolla stessa). Qualsiasi bolla verrà toccata dalla propagazione dell’esplosione, **avanzerà** anch’essa di stato. Vi è la possibilità - molto frequente - che questo evento porti ad una **logica ricorsiva** di avanzamento di stato delle bolle, permettendo, nella maggior parte dei campi di gioco, di poter vincere la partita con poche mosse.

Il gioco si conclude in due possibili casi:

* Vittoria del *giocatore*, quando tutte le bolle vengono fatte scoppiare senza superare il numero minimo di mosse
* Vittoria dell’*elaboratore*, quanto viene superato il numero minimo di mosse senza aver fatto scoppiare tutte le bolle

il calcolo delle mosse

Alla base del gioco vi è l’algoritmo che **calcola il numero minimo di mosse** con il quale concludere la partita identificato con le *mosse minime* all’interno del gioco stesso.

## COme funziona?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) |

Prendendo come esempio la griglia mostrata sopra, in cui ogni punto è rappresentato dagli **indici di posizione della matrice** contenente il campo di gioco, viene sfruttata una logica ricorsiva che permetta di prendere in analisi tutti i casi di mosse possibili avendo come punto di partenza uno specifico punto. Il risultato finale è la costruzione di un albero che copra tutti i casi possibili e con il quale è possibile selezionare il risultato che richieda la minor perdita per vincere la partita.

Ovviamente nel caso in cui la bolla presente nel punto preso in analisi risulterà già scoppiata l’algoritmo eviterà di eseguire la mossa inutile e procederà nell’esecuzione. Similmente quando il numero di mosse simulate corrente è **maggiore o uguale** al numero di mosse minimo – trovato fino al momento presente – l’algoritmo salterà la mossa. Altrimenti quando il campo di gioco risulterà completato, ovvero quando tutte le bolle saranno scoppiate, l’algoritmo provvederà a registrare il numero minimo di mosse conseguito – solamente quando ragionevole – e quindi procedere nella restante analisi.

## Esempio

Per esempio partendo dal punto (0,0) l’algoritmo procederà con (0,1), (0,2), ..., (x, y) fino a (4,5), ovvero il termine logico della matrice. Grazie all’aspetto ricorsivo per ogni punto (x, y) verrà presa in esame la mossa (m, n) e così via finché non vengono analizzate tutte le possibili mosse per ogni punto in analisi.

## Limitazioni

Il problema più evidente di questo algoritmo è l’efficienza, in quanto il tempo e la quantità di risorse necessarie risulteranno legate esponenzialmente al livello di profondità raggiunto.

Questa limitazione viene scemata dalla potenza di calcolo degli elaboratori correnti e dalla quantità di memoria disponibile con le ultime tecnologie. Diverso sarebbe il risultato su macchine datate o analizzando campi di gioco di maggiori dimensioni.

## Correttezza

Personalmente ho eseguito numerosi controlli sul risultato conseguito dall’algoritmo. Inizialmente ho preferito eseguirlo e verificarne la correttezza su un campo di gioco con dimensioni ridotte di due righe e due colonne, ove mi era possibile distinguere ad occhio quale fosse la scelta migliore.

In seguito, grazie al file di testo *moves.txt* che viene generato dall’algoritmo stesso ad ogni esecuzione e nel quale vengono stampati tutti i passaggi da esso compiuti, ho verificato la veridicità della scelta migliore su un ampio range di campi di gioco confrontando la serie di mosse proposte dall’algoritmo e le ulteriori possibili.

Implementazione del gioco

Analizzando l’esecuzione del gioco, vengono eseguiti i seguenti passaggi:

1. Inizializzate le variabili necessarie all’avvio del gioco

int numMoves = 0,

status = 0;

int gameField[FIELD\_ROWS][FIELD\_COLUMNS];

1. Generato il campo di gioco e di conseguenza calcolate le mosse minime necessarie a completarlo

generateGameField(gameField);

int minMoves = calcMinMovesForGameField(gameField);

1. Eseguito un ciclo pre-condizionato in cui:
   1. Viene stampato a video il campo di gioco ed il numero di mosse rimanenti
   2. Vengono chieste al giocatore le coordinate della bolla su cui eseguire la mossa
   3. Viene aggiornata la bolla selezionata
   4. Vi è un controllo sullo stato corrente del gioco che stabilisce le regole di uscita dal ciclo principale

while (status == 0)

{

printGameFieldToPlayer(gameField);

printf("Mosse rimanenti: %d\n\n", minMoves - numMoves);

int x = -1, y = -1;

askPlayerNextMoveCoords(&x, &y, gameField);

updateBubbleState(gameField, x, y, noDirection);

numMoves++;

status = isGameCompleted(gameField);

if (status == 0 && numMoves >= minMoves)

{

status = statusLost;

}

}

1. Stampato a video lo stato finale del campo di gioco insieme al messaggio di vittoria o sconfitta del giocatore

printGameFieldToPlayer(gameField);

printf("\n%s\n", (status == 1 ? "HAI VINTO" : "HAI PERSO"));

avoidProgramExit();

FUnzioni del gioco

Di seguito un’analisi dettagliata dei metodi utilizzati per lo svolgimento del gioco, tenendo conto della presenza di alcune costanti definite nel file *header*:

* ***FIELD\_ROWS***, è un numero intero che rappresenta il numero di righe del campo di gioco
* ***FIELD\_COLUMNS***, è un numero intero che rappresenta il numero di colonne del camp di gioco
* ***GameStatus***, è un enum contenente gli stati possibili relativi alla situazione del campo di gioco
* ***BubbleStates***, è un enum contenente gli stati possibili che una bolla può assumere
* ***MovesStackStatus***, è una struct relativa allo stato delle mosse durante il calcolo di quelle minime, contiene due proprietà:
  + *x*
  + *y*
* ***BubbleExplosionDeltas***, è una struct utilizzata durante il calcolo della propagazione dell’esplosione di una bolla e contiene due proprietà:
  + *dx*
  + *dy*
* ***BubbleExplosionDirectionPropagation***, è un enum utile per differenziare le direzioni possibili di propagazione dell’esplosione di una bolla

## generazione del campo di gioco

int generateGameField(int gameField[FIELD\_ROWS][FIELD\_COLUMNS])

{

for (int r = 0; r < FIELD\_ROWS; r++)

{

for (int c = 0; c < FIELD\_COLUMNS; c++)

{

gameField[r][c] = generateRandBubbleState();

}

}

}

La funzione *generateGameField* si occupa di generare il campo di gioco utilizzato nella partita corrente. Si aspetta come parametro formale la matrice del campo di gioco.

Esegue un ciclo *for* sulle righe della matrice ed all’interno di esso un ciclo *for* sulle colonne, in maniera tale da generare uno stato – ricevuto come valore di ritorno dalla funzione *generateRandBubbleState* - per ogni bolla presente all’interno della matrice.

int generateRandBubbleState()

{

return (enum BubbleStates)(rand() % bubbleStatesCount);

}

Quest’ultima ritorna semplicemente un valore di tipo numero intero tra gli stati possibili in maniera casuale.

## Calcolo delle mosse

int calcMinMovesForGameField(int gameField[FIELD\_ROWS][FIELD\_COLUMNS])

{

int minMoves = INT\_MAX;

FILE \*logStream = fopen("moves.txt", "w");

int\* placeholderFieldRef = malloc(sizeof(int) \* FIELD\_ROWS \* FIELD\_COLUMNS);

memcpy(placeholderFieldRef, gameField, sizeof(int) \* FIELD\_ROWS \* FIELD\_COLUMNS);

if (logStream == NULL)

{

printf("Impossibile aprire il file %s\n", "moves.txt");

avoidProgramExit();

}

// parte relativa ai print di log su file

struct MovesStackStatus movesStatusStack[1000];

for (int r = 0; r < FIELD\_ROWS; r++)

{

for (int c = 0; c < FIELD\_COLUMNS; c++)

{

// parte relativa ai print di log su file

int numMoves = 0;

recursiveBestPathTree(logStream, &numMoves, &minMoves, c, r, placeholderFieldRef, &movesStatusStack);

}

}

// parte relativa ai print di log su file

fflush(logStream);

fclose(logStream);

free(placeholderFieldRef);

return minMoves;

}

int recursiveBestPathTree(FILE \*stream, int \*numMoves, int \*minMoves, int x, int y, int gameField[FIELD\_ROWS][FIELD\_COLUMNS], struct MovesStackStatus movesStatusStack[1000])

{

if (\*numMoves >= \*minMoves - 1 || gameField[y][x] == stateExploded)

{

return;

}

char\* movesStatusRef = saveCalcMovesStatus(gameField);

movesStatusStack[\*numMoves].x = x;

movesStatusStack[\*numMoves].y = y;

\*numMoves += 1;

updateBubbleState(gameField, x, y, noDirection);

if(isGameCompleted(gameField))

{

if (\*minMoves > \*numMoves)

{

\*minMoves = \*numMoves;

}

// parte relativa ai print di log su file

for (int i = 0; i < \*numMoves; i++)

{

// parte relativa ai print di log su file

}

fprintf(stream, "\n");

numMoves--;

restoreCalcMovesStatus(gameField, movesStatusRef);

return;

}

for (int r = 0; r < FIELD\_ROWS; r++)

{

for (int c = 0; c < FIELD\_COLUMNS; c++)

{

recursiveBestPathTree(stream, numMoves, minMoves, c, r, gameField, movesStatusStack);

}

}

\*numMoves -= 1;

restoreCalcMovesStatus(gameField, movesStatusRef);

}

La funzione *calcMinMovesForGameField* gestisce il calcolo delle mosse minime per completare il campo di gioco.

## Aggiornamento delle bolle

void updateBubbleState(int gameField[FIELD\_ROWS][FIELD\_COLUMNS], int x, int y, int explosionPropagationDirection)

{

if (x < 0 || y < 0 || x >= FIELD\_COLUMNS || y >= FIELD\_ROWS)

{

return;

}

struct BubbleExplosionDeltas deltas;

if (gameField[y][x] == stateExploded)

{

if (explosionPropagationDirection != -1)

{

deltas = calcBubbleExplosionDeltas(x, y, explosionPropagationDirection);

updateBubbleState(gameField, deltas.dx, deltas.dy, explosionPropagationDirection);

}

}

else if (gameField[y][x] == stateFull)

{

gameField[y][x] = stateExploded;

for (int i = 0; i < 4; i++)

{

enum BubbleExplosionDirectionPropagation directionPropagation = (enum BubbleExplosionDirectionPropagation)i;

deltas = calcBubbleExplosionDeltas(x, y, directionPropagation);

updateBubbleState(gameField, deltas.dx, deltas.dy, directionPropagation);

}

}

else

{

gameField[y][x] += 1;

}

}

struct BubbleExplosionDeltas calcBubbleExplosionDeltas(int x, int y, int explosionPropagationDirection)

{

struct BubbleExplosionDeltas deltas;

deltas.dx = x + (explosionPropagationDirection == directionLeft ? -1 : explosionPropagationDirection == directionRight ? 1 : 0);

deltas.dy = y + (explosionPropagationDirection == directionUp ? -1 : explosionPropagationDirection == directionDown ? 1 : 0);

return deltas;

}