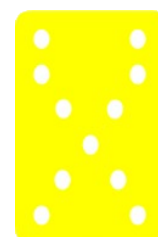
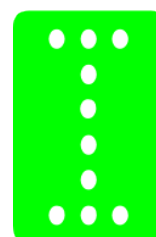
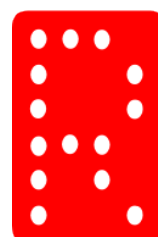
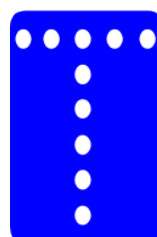
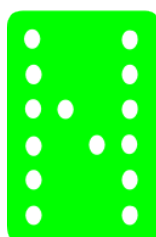
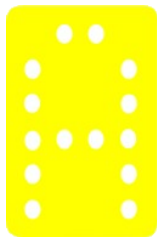
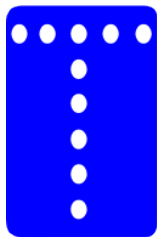


PROGETTO DI LINGUAGGI DI PROGRAMMAZIONE



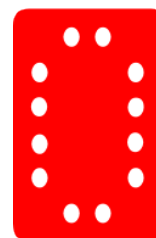
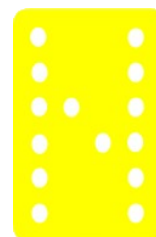
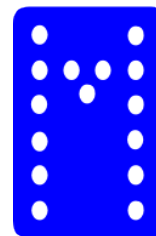
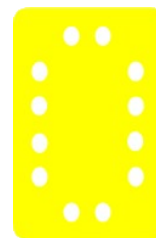
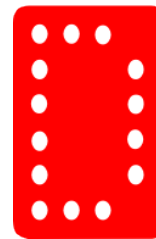
Anno accademico
2010 · 2011

D
T
I
·
C
R
E
M
A



DOCENTE
Dott. A. Ceselli

STUDENTE
Mattia Cavenaghi
mat.736856



Contents

1	Il problema e la soluzione proposta	2
1.1	Il Tantrix Domino	2
1.2	Descrizione dell'algoritmo	2
2	Linguaggi impiegati	6
2.1	C#	6
2.2	Python	7
2.3	F#	8
2.4	Prolog	9

1 Il problema e la soluzione proposta

1.1 Il Tantrix Domino

Tantrix è un board game ideato in Nuova Zelanda tra la fine degli anni ottanta e la prima metà degli anni novanta: il gioco è composto da cinquantasei pedine, ognuna delle quali ha forma esagonale. Nella parte frontale di ognuna vi sono raffigurate tre linee di colori diversi: giallo, verde, rosso e blu. Ogni linea collega due facce dell'esagono, ed ogni faccia è l'estremo di una sola linea.

La regola principale è detta *regola aurea*: ovunque le pedine si tocchino, i colori di tutte le facce a contatto devono combaciare.

Il problema consiste nello scegliere un sottoinsieme delle cinquantasei pedine, e nel posizionale in sequenza una di fianco all'altra, in modo che il colore della linea che termina sulla faccia destra di ogni pedina coincida con il colore della linea che termina sulla faccia sinistra della pedina successiva nella sequenza, e simmetricamente che il colore della linea che termina sulla faccia sinistra di ogni pedina coincida con il colore della linea che termina sulla faccia destra della pedina precedente nella sequenza. E' ammesso ruotare arbitrariamente ciascuna pedina.

1.2 Descrizione dell'algoritmo

L'algoritmo implementato si propone di risolvere il gioco trovando una soluzione partendo da un insieme di pedine estratte in modo casuale.

Osservando la figura 1.1 possiamo notare che vengono istanziate tre liste: una contenente le pedine da utilizzare nella partita corrente (la lista delle *pedine estratte*), una contenente le pedine disposte a domino (il *domino*), ed una di memorizzazione temporanea delle pedine che non sono compatibili con la regola aurea (la lista delle *pedine scartate*).

L'algoritmo sostanzialmente contempla i seguenti casi:

- *caso 1 (di base)*: si seleziona la prima pedina dalle estratte (*pedina destra*) e tramite la regola aurea la si compara con l'ultima pedina presente nel domino (*pedina sinistra*): se il colore della faccia destra (per la pedina sinistra) e della faccia sinistra (per la pedina destra) combacia, la pedina destra viene inserita in coda al domino, in caso contrario in coda alle pedine scartate;
- *caso 2 (di backtraking)*: se al termine di un passo iterativo dell'algoritmo vi siano una o più pedine nella lista degli scarti, è necessario procedere all'esecuzione del passo di backtraking sulla lista domino:

1 Il problema e la soluzione proposta

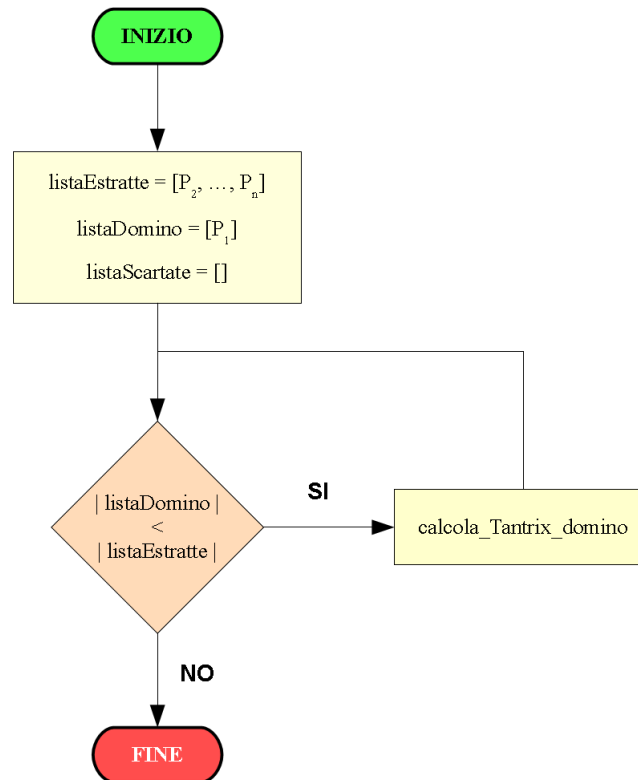


Figure 1.1: Schematizzazione dell'algoritmo implementato.

- *caso A (rotazione della coda del domino)*: dopo aver estratto l'ultima pedina (pedina destra) e la penultima pedina (pedina sinistra) dal domino, si ruota di una posizione la pedina destra (*rottura del domino* o della *regola aurea*) e si procede al controllo della regola aurea con la pedina sinistra. Nel caso combacino è possibile reinserire la pedina destra ruotata nel domino e procedere con la normale esecuzione dell'algoritmo, altrimenti è necessario scartare la pedina destra e continuare la ricerca di una pedina compatibile.
- *caso B (rotazione della radice del domino)*: questo caso si verifica se la lista degli scarti contiene $n - 1$ pedine estratte, in tal caso viene ruotata di una posizione la pedina radice e viene riavviato l'algoritmo, prendendo come nuova pedina sinistra la radice del domino ruotata, e come pedina destra la prima pedina presente nella lista delle estratte, contenente gli elementi rigenerati della lista degli scarti.

Nel caso 2 è bene osservare che ogni volta che una pedina destra viene confermata nel domino, la lista delle pedine scartate viene svuotata, e le pedine al suo interno reinserite nella lista delle pedine estratte: tale operazione è ovviamente preceduta dalla reinizializzazione dei contatori associati alla singola pedina, ciò permette di non verificare continuamente le stesse pedina già non compatibili, creando situazioni di ciclo infinito.

1 Il problema e la soluzione proposta

L'algoritmo termina quando la *lunghezza* della lista domino è pari al numero di pedine estratte.

1 Il problema e la soluzione proposta

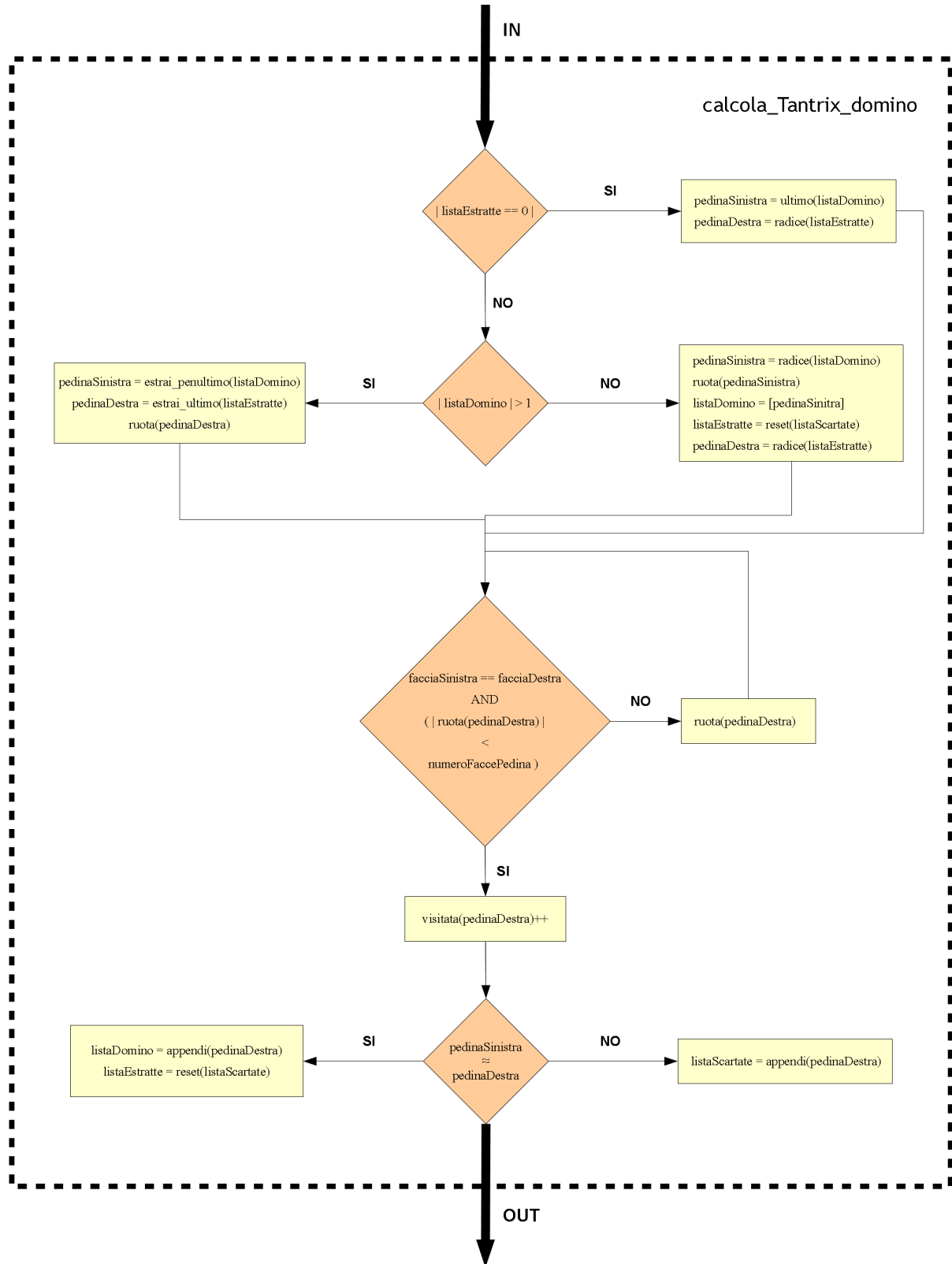


Figure 1.2: Schematizzazione del ciclo di calcolo del Tantrix domino.

2 Linguaggi impiegati

2.1 C#

Il linguaggio C# (assieme al linguaggio F#) appartiene alla famiglia .NET integrata nell'ambiente di sviluppo *Microsoft Visual Studio* (brevemente *VS*).

E' un linguaggio ad oggetti e come tale gode di tutte le proprietà relative a questa classe di linguaggi di programmazione (utilizzo di classi, overloading, overriding, etc.).

In questo linguaggio si è implementata la prima versione del presente progetto, poiché molto materiale è presente in linea facilitando la scrittura del codice sorgente.

Per l'implementazione dell'algoritmo si è creato dapprima un oggetto *Pedina* contenente i campi:

- *indice*: è un intero che identifica il numero della pedina all'interno dell'insieme di cinquantasei oggetti analoghi;
- *facce*: è un vettore di sei caratteri che identifica i colori delle singole facce di cui una pedina è composta, ogni pedina contiene un massimo di tre colori;
- *rotazioni*: è un intero che memorizza la faccia destra corrente;
- *visite*: è contatore un intero utilizzato per verificare il numero di volte che una data pedina è già stata verificata tramite la regola aurea. Poiché una pedina può contenere un massimo di tre colori e nella regola aurea si effettua il controllo su un solo colore, tale contatore può assumere un valore massimo di due, dopo di che la pedina viene scartata.

Oltre ai campi si poi implementati dei metodi:

- *ritorna faccia sinistra*: permette di ritornare il carattere che identifica il colore posto sulla faccia sinistra della pedina;
- *ritorna faccia destra*: permette di ritornare il carattere che identifica il colore posto sulla faccia destra della pedina.

Ulteriori metodi che coinvolgono la manipolazione delle pedine sono:

- *reset delle pedine scartate*: tramite cui è possibile inizializzare a 0 i contatori delle rotazioni e delle visite;
- *calcolo dell'indice faccia sinistra corrente*: poiché il domino viene calcolato sulle facce parallele della pedina, tale metodo consente di calcolare l'indice della faccia sinistra, parallela alla faccia destra, quest'ultima identificata dal numero di rotazioni.

2 Linguaggi impiegati

Oltre agli oggetti di utilizzato standard (es. le stringhe) tra la moltitudine di strutture presenti in VS per la realizzazione delle liste di pedine si sono adottati i costrutti *List<Pedina>*, ossia strutture dinamiche contenenti oggetti di tipo *Pedina* definiti in fase di sviluppo: ciò ha facilitato le operazioni di aggiunta e rimozione delle pedine, fornendo metodi utili nella computazione dell'algoritmo (es. il metodo *length*, il metodo di rimozione mediante indice, etc.).

Altro punto a favore di tale linguaggio è stato l'utilizzo dei cicli in particolare del *foreach* che permette di accedere direttamente al singolo oggetto all'interno delle strutture più complesse, consentendo la scrittura di cicli più compatti rispetto al *while* ed al *for*.

La programmazione ad oggetti ha consentito inoltre di sfruttare ulteriori strutture con relativi metodi per le operazioni sui file, basti pensare all'oggetto *StreamWriter* per la creazione dei report finali e l'oggetto *Random* per la generazione dei numeri casuali.

L'utilizzo di tale linguaggio ha presentato però alcuni svantaggi:

- *dichiarazione delle variabili*: ogni variabile necessita di essere dichiarata assieme al suo tipo;
- *dichiarazione dei metodi*: i metodi ed i relativi parametri devono essere dichiarati assieme al loro tipo;
- *compilazione del codice*: in VS ogni progetto è costituito da una solution che al suo interno può contenere a sua volta più progetti, la cui compilazione ha richiesto una rilevante quantità di tempo;
- *lunghezza del codice*: scrivere metodi che implementino poche operazioni ha comportato la scrittura di una notevole quantità di linee di codice (es. procedura di generazione di una lista contenente n numeri casuali).

2.2 Python

Il secondo linguaggio scelto è multi-paradigma ossia sfrutta la programmazione ad oggetti, funzionale, ed imperativa: in questo caso l'implementazione si è rivelata essere la più soddisfacente poiché qualsiasi oggetto all'interno di Python è una lista, fattore che ha reso la conversione dal linguaggio C# molto semplice, basti pensare ai metodi dedicati all'estrazione ed all'inserimento di un oggetto nelle liste (*pop* ed *append*).

Il paradigma a oggetti ha inoltre consentito di riutilizzare i già citati oggetti *Pedina*, e semplificare l'estrazione dei dati delle stesse dal file testuale fornito unitamente al testo d'esame.

Non essendo prevista la dichiarazione del tipo delle variabili si è potuto scrivere un codice sorgente compatto, ed altra particolarità è stato l'utilizzo dell'indentazione nei blocchi di codice rendendo superfluo l'utilizzo delle parentesi, ma garantendo una maggiore leggibilità del sorgente, somigliante più ad uno script.

Le uniche variazioni implementative, rispetto a C#, consistono nello separare la funzionalità di creazione del file dei risultati e di creare un'ulteriore funzione per la stampa

di un vettore di indici interi, poiché l'indentazione avrebbe creato rientri eccessivamente profondi.

2.3 F#

Nella conversione si è conservato l'utilizzo dell'oggetto *Pedina* già visto nei due precedenti linguaggi mentre essendo i cicli *while*, *for* e *foreach* non propriamente nativi del linguaggio funzionale, è stato necessario riscrivere le seguenti procedure:

- *rimozione di un oggetto da una lista*: in F# gli oggetti sono principalmente immutabili e non essendovi un metodo per rimuovere un oggetto all'interno di una lista, si è creata un'opportuna funzione utile a tale scopo. La funzione ovviamente è applicabile solamente ad oggetti mutabili utilizzati nello sviluppo;
- *generazione di una lista di numeri casuali*: si è adottata la struttura dati *Seq* (sequenza), rendendo la stesura del codice più compatta ma più lunga da comprendere in fase di implementazione;
- *reset dei contatori presenti nell'oggetto pedina*: in questo caso la lista delle pedine scartate non viene modificata, ma gli oggetti contenuti al suo interno vengono letti ed i contatori delle rotazioni e delle visite, inizializzati a zero;
- *controllo della regola aurea*: a differenza dei due precedenti linguaggi il ciclo associato a tale funzione è stato incapsulato in una funzione ricorsiva esternandola rispetto al main del programma;
- *creazione di una lista di pedine*: analogamente ai precedenti linguaggi anche in F# si è creato un ciclo, in questo caso una funzione ricorsiva che partendo da una lista di indici interi crea la relativa lista delle pedine, ricercando i dati nel file fornito dal docente.
- *stampa a video e memorizzazione del domino nel report finale*: la stampa a video del contenuto delle pedine unitamente alla creazione del file risultati, ha reso necessario anche in questo caso la ricorsione nelle relative funzioni.

Anche in questa implementazione si è scelta la lista di tipo *Pedina* come struttura dati, la quale consente di accedere a diverse funzioni presenti nel namespace del linguaggio.

Il principale ostacolo riscontrato in fase di conversione del codice in F# è stato comprendere il paradigma funzionale, in particolare il processo di riscrittura dei cicli e la modifica delle variabili "classiche" in variabili mutabili (ove possibile) ed immutabili, infine la dichiarazione delle funzioni si è rivelata essere sequenziale: ad esempio se la funzione A necessita l'utilizzo della funzione B, quest'ultima deve essere dichiarata prima dalla funzione A, in caso contrario vengono generati degli errori in fase di compilazione.

2.4 Prolog

L'ultima versione del programma è stata realizzata in Prolog che elabora principalmente liste. Nella programmazione logica si è reso necessario modificare l'algoritmo di computazione, poiché convertire le procedure originali avrebbe causato la scrittura di un codice sorgente troppo complesso da comprendere e leggere.

Il nuovo algoritmo elabora un set di pedine selezionato manualmente nel file *tiles_pl.pl* che viene caricato all'inizio della procedura principale *tantrix_domino*, la quale prevede l'immissione del numero di pedine da computare per la produzione della lista domino corretta. In tale procedura si inizializza il domino con una pedina radice che non subisce rotazioni, ma la cui compatibilità viene garantita dalla pedina successiva.

I passi successivi contemplano l'utilizzo di una regola principale (*trova_domino*) ed in una serie di ulteriori regole descritte di seguito:

- *trova_domino*: è la regola principale di computazione del domino: la procedura estrae di volta in volta la pedina corretta in modo da formare il domino, senza rispettare l'ordine con cui tali pedine sono state estratte. Un contatore (*Numero_rimanenti*) verifica la disponibilità di ulteriori pedine la cui presenza nella lista risultato viene controllata tramite la regola denominata *controllo_presenza*. Una volta che è stata verificata la compatibilità dell'oggetto pedina con la lista domino, questa viene memorizzata unitamente al numero di rotazioni subite;
- *appendioggetto*: una volta che la compatibilità di una pedina è stata verificata, la procedura consente di appendere un oggetto *pedina* e *rotazione* nelle rispettive liste;
- *controllo_presenza*: onde evitare che una pedina venga inserita più volte nel domino, tale regola consente di verificarne la presenza: in caso di esito negativo, l'oggetto non è presente e può essere elaborato;
- *stampa_dettagli_soluzione*: questa regola consente di stampare in un file di output i dettagli associati alle singole pedine presenti nel domino, come l'indice della pedina, ed i colori della stessa nella sua configurazione domino;
- *scomponi_lista*: l'ultima regola consente di scomporre la lista delle pedine e delle rotazioni, col fine di stampare i dettagli della soluzione.

Concludendo, di tutti i linguaggi impiegati Prolog (assieme ad F#) si è rivelato essere il più ostico poiché l'implementazione iniziale dell'algoritmo è stata completamente stravolta: in compenso si è potuto realizzare un codice più compatto e leggibile rispetto alle versioni C#, Python, ed F# ciò nonostante si è di gran lunga preferita l'implementazione in Python per i motivi già citate nel paragrafo 2.2.