Università degli Studi di Napoli "Parthenope"

FACOLTÀ DI SCIENZE E TECNOLOGIE
Corso di Laurea in Informatica e Tecnoligie Multimediali

Laboratorio di Algoritmi e Strutture Dati
Progetto d'Esame

Risoluzione del gioco del Sudoku tramite la tecnica del Backtracking

Candidato:
Daniele Iervolino
Matricola LI/1065

Professore:
Francesco Camastra

Anno Accademico 2007-2008

Introduzione

Questa relazione è stata scritta in LaTeX. Tratta della realizzazione del gioco del Sudoku con l'ausilio della tecnica *"backtracking"*. Nella prima sezione della relazione vengono spiegati i principi base del backtracking e riportati esempi applicativi.

Subito dopo sono introdotte le tecnologie usate per implementarlo, descrivendo *la programmazzione orientata agli oggetti* ed il *linguaggio C++* elencando gli aspetti fondamentali di tale linguaggio. Sono spiegate le regole del Sudoku ed il suo aspetto matematico. Infine vengono mostrati i vari pezzi di codice che sono stati realizzati in C++ per implementare tutti questi concetti.

Indice

1 Backtracking

La tecnica del backtracking si basa sulla seguente filosofia :

"Prova a fare qualcosa e, se non funziona, disfala e riprova qualcos'altro".

Il becktracking è un perfezionamento dell'algoritmo **brute force search**.
Non si parla di paradigma o di metodo in quanto non vi è uno schema riconoscibile per scrivere un algoritmo di tipo backtracking, come invece, si ha per il paradigma divide et impera o greedy. L'idea del backtracking è di costruire una soluzione, eseguendo un certo numero di azioni e, se non funziona, tornare indietro (backtrack) ed eseguire altre azioni. Questa tecnica è alla base degli algoritmi di visita dei grafi (*Depth-First Search*) e degli alberi (*inorder,preoreder e postorder*) dove l'esecuzione su un nodo viene temporaneamente sospesa per permtettere la visita ai suoi sottoalberi. In questi problemi bisogna generare ciascuna delle possibili soluzioni *esattamente* una alla volta. Il backtracking costruisce un albero (*albero delle soluzioni*) dove la radice rappresenta l'insieme di tutte le soluzioni possibili, ciascun vertice interno è una soluzione parziale e c'è un arco che va dal vertice x al vertice y; se il vertice y è stato creato estendendo la soluzione parziale del vertice x. Le foglie dell'albero sono le soluzioni. Se durante la ricerca ci si ritrova in una strada senza uscita si torna indietro (da cui il nome *backtracking*, dal verbo *to backtrack*, ritornare sui propri passi) per esplorare altri rami non ancora visitati. Nel dettaglio la tecnica del backtracking svolge i seguenti passi :

- Ogni mossa lungo l'albero aggiungere un nuovo elemento alla soluzione.
- Se procedendo in questo modo la ricerca ha successo si determinara una soluzione ammissibile.
- La ricerca si può quindi fermare o continuare se si vogliono determinare tutte le soluzioni ammissibili.
- Se, invece, si arriva ad un nodo per cui non è possibile completare la soluzione, si torna indietro (backtrack) fino al primo nodo che ha ancora dei vicini non visitati, verso i quali si riprende la ricerca.

L'albero delle soluzioni offre un modo alternativo di pensare al backtracking in quanto il processo di costruzione delle soluzioni corrisponde esattamente a quello di effettuare una visita in profondità dell'albero. I rami dell'albero che sicuramente non portano a configurazioni ammissibili possono essere "potati" (**pruned**) in modo da ridurre lo spazio di ricerca. La figura 1 illustra graficamente la strategia: ogni volta che si scende di livello si aggiunge un vincolo nell'insieme delle soluzioni.

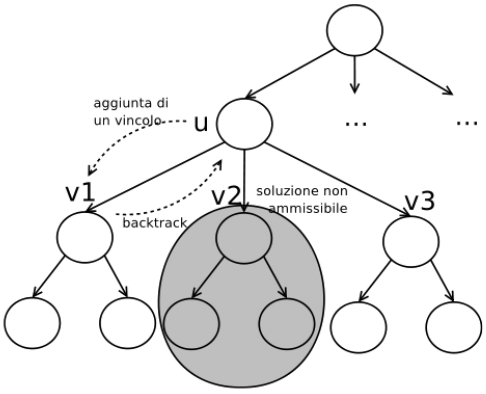

Figura 1: Backtracking: visita dell'albero delle soluzioni

Una volta che si è raggiunta una foglia dell'albero, non è conveniente ripartire dall'inizio nella ricerca, piuttosto risulta più comodo tornare indietro solamente di un livello per esplorare le altre possibili soluzioni. l' algoritmo in pseudo-codice che ne deriva è :

```
backtrack (Node u) {
 if ( u e' una Soluzione ) {
     return u ;
  } else {
      foreach (Nodo v figlio di u)
        if (v contiene soluzioni valide )
           backtrack (v);
}
```

Il backtracking può essere implementato in modo iterativo ma per la sua natura è spesso implementato in modo ricorsivo. La ricorsione ha il vantaggio di rendere l'algoritmo immediato e intuitivo, per queste ragioni il progetto d'esame è stato impletmentato in questo modo. Esistono molte varianti del backtraking che fanno uso di euristiche[1] che cercano di ottimizzare la scelta delle variabili, la scelta dei valori da essegnare e la propagazione dei vincoli. Facendo uso di tali euristiche otteniamo un backtracking "intelligente" che ha il vantaggio di attraversare meno rami possibili nell'albero delle soluzioni, ma ha la conseguenza di appesantire il codice con svariati controlli che ne deteriorano la leggibilità e aumentano la diffoltà di programmazione. Per queste ragioni nell'implementazione del progetto è stato preferito un backtracking classico in quanto i vincoli per la realizzazione del

[1] Euristica, dal verbo greco heuriskein ("trovare")
In informatica si definisce euristico il lavoro di un software che non opera meccanicamente,cioè non si limita ad analizzare i dati secondo confronto di dati noti, ma prova a simularne il comportamento.

gioco del sudoku sono più che sufficienti per garantire la ricerca di una soluzione ammissibile.

Per concludere c'è bisogno di specificare che per la sua natura il backtracking non è adatto per problemi di ottimizzazione, questa tecnica è utile solo per generare una soluzione ammissibile o tutte le soluzioni modo ordinato (enumerazione). Per cercare quelle ottimali si deve preferire la tecnica branch & bound che è un'estensione di questa. Un classico problema risolvibile con la tecnica del backtracking è il problema delle regine.

Il problema delle regine

Il problema delle regine consiste nel disporre n regine su una scacchiera di lato n in mondo che nessuna regina sia in grado di tenerne sotto scacco un'altra. Ricordiamo che nel gioco degli scacchi la regina può muoversi orizzontalmente, verticalmente e diagonalmente sulla scacchiera per un numero qualsiasi di caselle. Quindi, interpretando la scacchiera come una griglia, una soluzione al problema è valida se non esistono due regine che sulla stessa riga, colonna o diagonale.
Nella figura 2(a) sono indicate le caselle in cui può muoversi una regina posta nella casella d4. Ponedo $n = 8$,per trovare tutte le soluzioni valide al problema

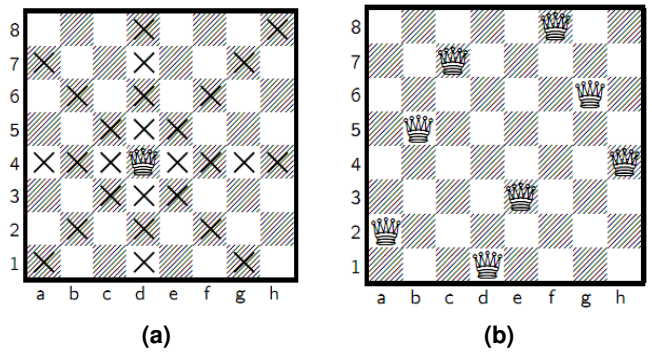

Figura 2: (a) Le caselle tenute sotto scacco da una regina (b) Una possibile soluzione al problema delle 8 regine

potremmo controllare la validità di tutte le possibili configurazioni in cui è possibile disporre 8 pezzi identici su una scacchiera di 64 caselle. Per la precisione il numero N di configurazioni possibili è pari a:

$$N = \binom{64}{8} = \frac{64!}{(64-8)!8!} = 4042601650368$$

Tuttavia, utilizzando il backtracking, possiamo procedere in maniera più efficiente osservando che, in ogni soluzione valida, su ogni riga sta una e una sola regina.Il numero di configurazioni da controllare scende,drasticamente, così al valore

$N' = 8^8 = 1607770216$ che risulta essere circa 260 volte più piccolo di N. Quindi potremmo procedere nella seguente maniera :

1. Se non ho più regine da sistemare ho trovato una soluzione (caso base)
2. Prova a mettere una regina sulla riga i-esima, considerando tutte le posizioni possibili fino a trovarne una che non è sotto attacco di un'altra regina
3. Metti la regina nella posizione che non è sotto attacco
4. Prova a sistemare una nuova regina sulla riga $i+1-esima$ (*ricorsione - si richiama il programma per sistemare le rimanenti regine a partire dalla riga* $i+1-esima$)
5. Togli la regina dalla riga $i-esima$, scegli un'altra posizione libera e vai al passo 3

In questa maniera, ogni volta che si prova a posizionare una regina su una casella, e ci si accorge che non si può posizionarla, si torna indietro e si cambia casella, riuscendo così ad escludere in fretta molte configurazioni non valide senza doverle esplorare. Qui è riportata la funzione scritta in C++ che trova le soluzioni al problema delle regine:

```
// Caso base : abbiamo posizionatotutte le n regine
// Salviamo la soluzione in una lista
void solve ( AbsChessBoard & cb , Solutions & sol , int row =1) {
 if ( row > cb.n) {
  sol.push_back(cb.clone());
  return ;
 } // fine caso base
 for (int c=1;c<=cb.n;c++) {
  if (cb.isLegal(row ,c)) {
  // s i puo ' mettere una regina in (row,c )
  cb.set(row,c);
  // provo a risolvere
  solve (cb,sol,row +1);
  //tolgo la regina da( row , c )
  cb. unset (row ,c);
  }
 }
}
```

Sono state misurate le chiamate alla funzione **isLegal**() per una scacchiera di 64 caselle. Il numero risultante $N'' = 150720$, più piccolo di N' di tre ordini di grandezza, conferma la bontà dell'approccio backtracking rispetto ad una ricerca esaustiva. Possiamo notare che all'interno del **for** vengono effettuate fino a n chiamate ricorsive a **solve**, rendendo così la sua complessità esponenziale (le chiamate ricorsive hanno una regina in meno da sistemare).

2 La programmazione orientata agli oggetti

Nella sezione precedente è stata descritta la tecnica con la quale è stato realizzato il pregetto d'esame. In questa sezione ci si sofferma su come il backtrackig è stato realizzato nel progetto; ovvero con il linguaggio di programmazione C++.
Prima di entrare nei dettagli del C++ presenteremo una panoramica del concetto base che ha dato vita a questo linguaggio, **la programmazione orientata agli oggetti**. Le metodologie di programmazione sono cambiate notevolmente dall'invezione del computer, soprattutto per consentire di aumentare la complessità dei programmi. Per esempio quando furono inventati i computer, la programmazione veniva eseguita impostando istruzioni binarie tramite il pannello frontale del computer. Finchè i programmi erano composti da poche centinaia di istruzioni, questo approccio ha funzionato. Con la crescita dei programmi è stato sviluppato il linguaggio Assembler con il quale il programmatore poteva poteva realizzare programmi più estesi, utilizzando rappresentazioni simboliche delle istruzioni del linguaggio macchina. Ma i programmi continuavano a crescere e furono perciò introdotti i linguaggi di alto livello che davano al programmatore nuovi strumenti per gestire questa nuova richiesta di complessità. Il primo linguaggio di questo genere fu il FORTRAN. Nel 1960 fu creata la programmazione strutturale. Questo metodo è seguito dai linguggi C e Pascal. l'impiego di linguaggi strutturati ha reso possibile la realizzazione di programmi piuttosto complessi con una discreta facilità. I linguaggi strutturati sono caratterizzati dal supporto per le subroutine indipendenti, variabili locali, costrutti di controllo avanzati e dal fatto di non impiegare GOTO. Tuttavia anche utilizzando metodi di programmazione strutturata, un progetto può diventare incontrollabile una volta che raggiunga determinate dimensioni. La programmazione ad oggetti è nata per superare i limiti della programmazione strutturata.

La programmazione ad oggetti ha preso le migliori idee della programmazione strutturata e le ha combinate con nuovi concetti; il risultato è un'organizzazione completamente nuova dei programmi. In generale un programma un programma può essere realizzato in due modi: ponendo al centro il codice (*"ciò che accade"*) o ponendo al centro i dati (*"gli attori interessati"*). Utilizzando le tecniche della programmazzione strutturata, i programmi vengono tipicamente organizzati attorno al codice. Questo approccio prevede che il codice operi sui dati. Per esempio, un programma scritto in C è definito dalle funzioni, le quali operano sui dati del programma.

I programmi a oggetti seguono l'altro approccio. Infatti sono organizzati attorno ai dati e si basano sul fatto che i sono dati a controllare l'accesso al codice. In un linguaggio ad oggetti si definiscono i dati e le funzioni che sono autorizzate ad agire su tali dati. Pertanto sono i dati a stabilire quali operazioni possono essere eseguite. I linguaggi che consentono di attuare i principi della programma-

zione a oggetti hanno tre fattori in comune: l'incapsulamento, il polimorfismo e l'ereditarietà.

L'incapsulazione

L'incapsulazione è il meccanismo che riunisce insieme il codice ed i dati da esso manipolati e che mette entrambi al sicuro da interferenze esterne o errati utlizzi. In un linguaggio ad oggetti, il codice e i dati possono essere raggruppati in modo da creare una sorta di "scatola nera". Quando il codice e i dati vengono raggruppati in questo modo, si crea un oggetto.

All'interno di un oggetto, il codice, i dati o entrambi possono essere privati di tale oggetto oppure pubblici. Il codice o i dati privati sono noti o accessibili solo da parte degli elementi presenti nell'oggetto stesso, ovvero non possono essere modificati all'esterno dell'oggetto. Viceversa elementi pubblici sono accessibili da altre parti del programma all'esterno dell'oggetto. Generalmente le parti pubbliche di un oggetto sono utilizzate per fornire un'interfaccia controllata agli elementi privati dell'oggetto stesso. Nella programmazione ad oggetti, un oggetto è in tutto e per tutto una variabile di un tipo definito dall'utente; quindi ogni volta che si definisce un nuovo tipo d' oggetto, si crea implicitamente un nuovo tipo di dati. Ogni specifica istanza di questo tipo è una variabile composta.

Il polimorfismo

I linguaggi di programmazione a oggetti supportano *il polimorfismo* che può essere riassunto dalla frase "un interfaccia, più metodi". Ovvero, il polimorfismo consente a un'interfaccia di controllare l'accesso a una classe generale di azioni. La specifica azione selezionata è determinata dalla natura della situazione. Un esempio di polimorfismo tratto dal mondo reale è il termostato. Non importa quale fonte di calore si utilizza (gas,petrolio, elettricità, ecc...): il termostato funziona sempre allo stesso modo. In questo caso il termostato (l'interfaccia) è lo stesso per qualsiasi sia il tipo di fonte (metodo). Per esempio se si desidera raggiungere la temperatura di 20 gradi, si imposta il termostato a tale temperatura, non importa quale sia la fonte che fornisce il calore. Questo principio si può applicare anche in programmazione. Un esempio è lo stack. Uno stack per valori interi, uno per i caratteri ed un'altro per i valori in virgola mobile grazie al polimorfismo sarà possibile creare un solo insieme di nomi (push() e pop()) utilizzabile per i tre tipi di stack. Nel programma verranno create tre diverse versioni di queste funzioni, ma il nome rimarrà lo stesso. Il compilatore selezionerà automaticamente la funzione corretta sulla base del tipo dei dati memorizzati. Pertanto, l'interfaccia dello stack (ovvero le funzioni push() e pop()) non cambierà indipendentemente dal tipo di stack. Naturalmente le singole versioni di queste funzioni definiscono

implementazioni (metodi) specifiche per ciascun tipo di dati. Il polimorfismo aiuta a ridurre la complessità del programma consentendo di utilizzare la stessa interfaccia per accedere ad una classe generale di azioni, in quanto sarà il compilatore a selezionare il metodo specifico da applicare ad una determinata situazione. Il programmatore non dovrà più fare questa selezione manualmente, ma dovrà solo utilizzare la stessa interfaccia generale. I primi linguaggi di programmazione a oggetti erano interpretati e quindi il polimorfismo era supportato al runtime. Ma il C++ è un linguaggio compilato pertanto il polimorfismo è supportato al momento dell'esecuzione e al momento della compilazione.

L'ereditarietà

L'*ereditarietà* è il processo grazie il quale un oggetto acquisisce le proprietà di un altro oggetto. Questo è un concetto fondamentale poichè chiama in causa il concetto di classificazione. Se si prova a riflettere, la maggior parte della conoscenza è resa più gestibile da classificazioni geranchiche. Per esempio, un *gianduglioto* appartiene alla classificazione *cioccolato* che a sua volta appartiene alla classificazione *dolce* che a sua volta si trova nella classificazione più estesa *cibo*. Senza l'uso della classificazione, ogni oggetto dovrebbe essere definito esplicitamente con tutte le proprie caratteristiche. L'uso della classificazione consente di definire un oggetto sulla base delle qualità che lo rendono unico all'interno della propria classe. Sarà il meccanismo di ereditarietà a rendere possibile per un oggetto di essere una specifica istanza di un caso più generale. Ne si deduce che l'ereditarità è un importante aspetto della programmazione a oggetti.

3 Il linguaggio C++

Dopo aver chiarito la metodologia per la quale il linguaggio C++ è stato creato, è giunto il momento di scendere nei dettegli illustrando la sua storia, per poi fare una breve panoramica sulle caratteristiche principali di tale linguaggio.

Il linguaggio C++ fu creato da Bjarne Stroustrup nel 1979 come estenzione del C. Inizialmente il nuovo linguaggio fu chiamato semplicemente "C con classi". Nel 1983 questo nome venne cambiato in C++. Furono aggiunte nuove funzionalità, tra cui funzioni virtuali, overloading di funzioni ed operatori, reference, costanti, controllo dell'utente della gestione della memoria, type checking migliorato e commenti nel nuovo stile ("//"). Nel 1985 fu pubblicata la prima edizione di The C++ programming Language, che fornì un'importante guida di riferimento del linguaggio, che non era ancora stato ufficialmente standardizzato. Nel 1989 fu rilasciata la versione 2.0 del C++, le cui novità includono l'ereditarietà multipla, le classi astratte, le funzioni membro statiche, le funzioni membro const, e i membri protetti. Nel 1990 fu pubblicato The Annotated C++ Reference Manual, che fornì le basi del futuro standard. Le ultime aggiunte di funzionalità includono i template, le eccezioni, i namespace i nuovi tipi di cast ed il tipo di dato booleano. Così come il linguaggio, anche la libreria standard ha avuto un'evoluzione. La prima aggiunta alla libreria standard del C++ è stata la libreria dei flussi di I/O che forniva servizi sostitutivi della libreria C tradizionale (come printf e scanf). Dopo anni di lavoro, un comitato che presentava membri della ANSI e della ISO hanno standardizzato C++ nel 1998(ISO/IEC 14882:1998).[2] Per qualche anno seguente al rilascio ufficiale degli standard, il comitato ha seguito lo sviluppo del linguaggio e ha pubblicato nel 2003 una versione corretta dello standard. Risale invece al 2005 un report tecnico, chiamato Technical Report 1 (abbreviato TR1) che, pur non facendo ufficialmente parte dello standard, contiene un numero di estensioni alla libreria standard previste nella prossima versione di C++.[3]

Le Classi

In C++, la classe costituisce la base della programmazione a oggetti. In particolare, la classe definisce la natura di un oggetto ed è l'unità principale di incapsulazione del C++. In questa sezione vengono esaminate le classi e gli oggetti.
Le classi vengono create mediante la parola chiave class. La dichiarazione di una classe definisce un nuovo tipo di che racchiude sia il codice che i dati. Questo nuovo tipo verrà utilizzato per dichiarare oggetti di tale classe. Pertanto, una clas-

[2] Non c'è un proprietario del linguaggio C++, che è implementabile senza dover pagare royalty. Il documento di standardizzazione stesso però è disponibile solo a pagamento.

[3] Quasi tutti i moderni compilatori C++ supportano oggigiorno il TR1

se è un' astrazione logica mentre un oggetto ha esistenza fisica. In altre parole, un *oggetto è un' istanza di una classe*.

La dichiarazione di una classe è sintatticamente simile a quella di una struttura. Di seguito viene presentata la forma generale completa della dichiarazione di una classe.

```
class nome-classe{
 dati e funzioni privati
specificatori di accesso:
 dati e funzioni
specificatori di accesso:
 dati e funzioni
}elenco oggetti;
```

L'*elenco oggetti* è opzionale. Se è presente, dichiara gli oggetti di tale classe. Qui la parte *specificatori di accesso* può essere rappresenta da una di queste tre parole chiave del C++:

- public
- private
- protected

Le funzioni e i dati dichiarati all'interno di una classe sono di default privati di tale classe è possono essere utilizzati solo dagli altri membri della classe. Utilizzando lo specificatore di accesso public si consente però anche ad altre parti del programma di accedere alle funzioni o ai dati della classe. Lo specificatore di accesso protected è richiesto solo in caso di ereditarietà. Una volta utilizzato, uno specificatore di accesso rimane attivo finchè non viene indicato un'altro specificatore di accesso o finchè non viene raggiunta la fine della dichiarazione della classe. All'interno della dichiarazione di una classe, è possibile cambiare specificatore di accesso in un numero di volte desiderato. Le funzioni dichiarate all'interno di una classe sono chiamate *funzioni membro o metodi*. Le funzioni membro possono accedere a tutti gli elementi della classe di cui fanno parte e quindi anche agli elementi private. Le variabili che sono elementi di una classe sono chiamate variabili membro o *dati membri*. In senso generale, tutti gli elementi di una classe sono detti membri di tale classe.

Di seguito è presentato un esempio di classe :

```
#include <iostream>
#include <cstring>
using namespace std;

class libro{
 char nome[40];
 double prezzo;
 int scaff;
public:
 libro(){prezzo=0.0;scaff=0;strcpy(nome,"Vuoto");}; //costruttore
 libro(char *s, double c, int sco); //costruttore parametrizzato
 void sconto();
 void visualizza();
 ~libro(){}   //distruttore
};
libro::libro(char *s,double c,int sco)
{
 strcpy(nome,s);
 prezzo=c;
 scaff=sco;
}
void libro::sconto()
{
 prezzo/1.20;
}
void libro::visualizza()
{
 cout<<"Il libro"<<nome <<endl;
 cout<<"Costa"<<prezzo <<"euro"<<endl;
 cout<<"Scontato"<<sconto()<<endl;
 cout<<"Si trova sullo scaffale n."<<scaff<<endl;
}
```

È molto comune che una parte di un oggetto debba essere inizializzata prima dell'uso. Per esempio nel codice appena mostrato, prima di iniziare ad usare libro, i dati menbri devono essere inizializzati. Poichè capita molto spesso di dover inizializzare un oggetto, in C++ consente di inizializzare gli oggetti al momento della creazione. Questa inizializzazione automatica è ottenuta grazie all'impiego di una funzione *costruttore*. Un costruttore è una particolare metodo di una classe che porta lo stesso nome della classe, come mostrato nell'esempio. Il costruttore di un oggetto viene richiamato automaticamente nel momento in cui deve essere creato l'oggeto. Questo significa che viene richiamata al momento della dichiarazione dell'oggetto. Il costruttore viene richiamato una sola volta per ogni oggetto globale. Nel caso di oggetti locali il costruttore viene richiamato ogni volta che si incontra la dichiarazione di un nuovo oggetto. L'operazione complementare del

costruttore è svolta dal *distruttore*. In molte circostanze, un oggetto deve eseguire una o più azioni nel momento in cui ne finisce l'esistenza. Gli oggetti locali vengono costruiti nel momento in cui si entra nel blocco in cui si trovano e vengono distrutti all'uscita del blocco. Gli oggetti globali vengono distrutti nel momento in cui termina il programma. Quando viene distrutto un oggetto, viene automaticamente richiamato il relativo distruttore (se presente). Vi sono molti casi in cui è necessario utilizzare un distruttore. Per esempio, potrebbe essere necessario deallocare la memoria precedentemente allocata dall'oggetto oppure potrebbe essere necessario chiudere un file aperto. In C++ è il distruttore a gestire gli eventi di disattivazione. Il distruttore ha lo stesso nome del costruttore ma è preceduto dal carattere ~, come nell'esempio precedente. Si noti che i distruttori,come i costruttori, non restituiscono alcun valore. I costruttori possono ricevere argomenti. Normalmente questi argomenti auitano ad inizializzare un oggetto al momento della creazione. Per creare un *costruttore parametrizzato*, basta aggingervi parametri così come si fa con qualsiasi altra funzione. Quando si definisce il corpo del costruttore si possono utlizzare i parametri per inizializzare l'oggetto. I costruttori parametrizzati sono molto utili poichè evitano di dover eseguire una nuova chiamata di funzione semplicemente per inizializzare una o più variabili in un oggetto. Ogni chiamata di funzione evitata renderà il programma più efficiente.

L'esempio che segue mostra come vengono creati e utilizzati gli oggetti all'interno di un programma :

```
#include <iostream>
#include "libro.h"
using namespace std;
int main()
{
 libro lib; //oggetto vuoto
 libro lib("Guida C++",52.99,3);
 lib.visualizza();
 lib1.visualizza();
 system ("pause");
 return 0;
}
```

Si noti come per accedere ad una classe si utilizzi l'operatore **.** (punto) proprio come avveniva per le strutture in C. Infatti la classe è un' evoluzione di una struttura; in C++ una struttura è una particolare classe dove tutti i membri sono di default public.

Standard Template Library

Un altro aspetto che viene considerato da molti come la pià importante aggiunta al C++ degli ultimi anni è la libreria STL (Standard Template Library). La creazione della libreria STL è stato uno degli sforzi più ingenti svolti durante la standardizzazione del C++. Tale libreria fornisce classi template di utilizzo generale, funzioni che implementano algoritmi e strutture dati di uso comune, come per esempio, il supporto di vettori, liste, code e stack. Inoltre la libreria STL definisce varie routine di accesso a tali elementi. Poichè la libreira STL si basa su classi template, gli algoritmi e le strutture dati possono essere applicate praticamente a ogni tipo di dato compresi gli oggetti. Poichè la libreria STL è molto estesa non è possibile discutere in una sola sezione tutte le sue funzionalità. Per questo motivo verranno descritti gli elementi più importanti e quelli utilizzati nel progetto d'esame.[4]

Anche se la libreria STL è molto estesa e la sua sintassi può inizialmente spaventare, è molto facile utilizzarla se si comprende il modo in cui è costruita e quali sono gli elementi da essa impiegati. Pertanto, prima di mostrare il codice del progetto, è opportuno presentare i tre elementi che costituiscono la STL : *container*, *algoritmi* e *iteratori*.

Questi elementi collaborano fra loro per fornire soluzioni pronte all'uso per vari problemi di programmazione.

Container

I *container* sono oggetti che contengono altri oggetti: la libreria offre container di vari tipi. Per esempio, la classe **vector** definisce un array dinamico o come la classe **list** che fornisce una lista lineare. Questi container sono chiamati *container sequenziali* sono costutiti da una sequenza lineare di elementi. Oltre ai container sequenziali, la libreria definisce anche *container associativi* che consentono di rivercare in modo efficiente un valore sulla base di un chiave. Per esempio, la classe **map** fornisce l'accesso a valori tramite chiavi univoche. Ogni classe container definisce una serie di metodi specifici per il container stesso. Per esempio, un container stack include le funzioni di inserimento (push) ed estrazione (pop) dei valori. La lista completa dei container è raffigurata nella tabella 1.

Algoritmi

Gli *algoritmi* operano sui container. Essi rappresentano il mezzo tramite il quale è possibile manipolare il contenuto dei container. Fra le funzionalità offerte dagli algoritmi vi è l'inizializzazione, l'ordinamento, la ricerca e la trasformazione

[4] un esempio è la classe **string**

Tabella 1: I container definiti dalla libreria STL

CONTAINER	DESCRIZIONE	HEADER
`vector`	Un array dinamico	`<vector>`
`deque`	Una coda a doppio concatenamento	`<deque>`
`stack`	Uno stack	`<stack>`
`queue`	Una coda	`<queue>`
`priority_queue`	Una coda a priorità	`<queue>`
`set`	Un insieme dove ogni elemento è univoco	`<set>`
`multiset`	Un insieme dove ogni elemento non è necessariamente univoco	`<set>`
`map`	Una mappa chiave/valore in cui ad una chiave è associato un solo valore	`<map>`
`list`	Una lista lineare	`<list>`
`multimap`	Una mappa chiave/valore in cui ad una chiave possono essere associati due o più valori	`<list>`
`bitset`	Un gruppo di bit	`<bitset>`

del contenuto del container. Molti algoritmi operano su un intervallo di elementi si un container. Inoltre essi consentono di lavorare contemporaneamente su due tipi differenti. Per avere accesso agli algoritmi STL si deve includere nel programma l'header `<algorithm>`. Tutti gli algoritmi sono funzioni template; questo significa che possono essere applicati ad ogni tipo di container.

Iteratori

Gli *iteratori* sono oggetti che si comportano più o meno come puntatori. Essi danno la possibilità di attraversare il contenuto di un continer un po' come un puntatore consente di attraversare un array. Vi sono cinque tipi di iteratori.

Tabella 2: Iteratori ed accesso

ITERATORE	ACCESSI CONSENTITI
Accesso diretto	Memorizzazione e lettura dei valori. L'accesso agli elementi può avvenire in modo diretto
Bidirezionale	Memorizzazione e lettura dei valori. Spostamento in avanti e all'indietro
Avanti	Memorizzazione e lettura dei valori. Spostamento solo in avanti.
Input	Lettura ma non memorizzazione dei valori. Spostamento solo in avanti.
Output	Memorizzazione ma non lettura dei valori. Spostamento solo in avanti.

In generale, un iteratore che ha maggiori capacità di accesso può essere utilizzato in luogo di uno che è dotato di minori capacità di accesso. Per esempio al posto di un iteratore di input e possibile utilizzare un iteratore in avanti. Gli iteratori vengono gestiti come puntatori. Dunque è possibile incrementarli e decrementarli. Agli iteratori è possibile applicare l'operatore *. Gli iteratori vengono dichiarati utlizzando il tipo `iterator` definito nei vai container. La libreria STL supporta anche gli iteratori inversi. Gli iteratori inversi sono iteratori bidirezionali o ad accesso diretto che si muovono lungo una sequenza, incrementando tale iteratore si farà in modo che esso punti al penultimo elemento.

4 Il gioco del Sudoku

Prima di procedere oltre con la descrizione del progetto è necessario comprendere cosa sia il Sudoku e quali siano le sue regole. In questa sezione ci soffermeremo sul gioco del Sudoku.

Il *sudoku* (dal giapponese: *sūdoku*, nome completo *Sūji wa dokushin ni kagiru*) significa "numero solitario", è un gioco di logica nel quale al giocatore o solutore viene proposta una griglia di 9*x*9 celle, ciascuna delle quali può contenere un numero da 1 a 9, oppure essere vuota; la griglia è suddivisa in 9 righe orizzontali, nove colonne verticali e, da bordi in neretto, in 9 "sottogriglie", chiamate regioni, di 3*x*3 celle contigue. Le griglie proposte al giocatore hanno da 20 a 35 celle contenenti un numero. Scopo del gioco è quello di riempire le caselle bianche con numeri da 1 a 9, in modo tale che in ogni riga, colonna e regione siano presenti tutte le cifre da 1 a 9 e, pertanto, senza ripetizioni.

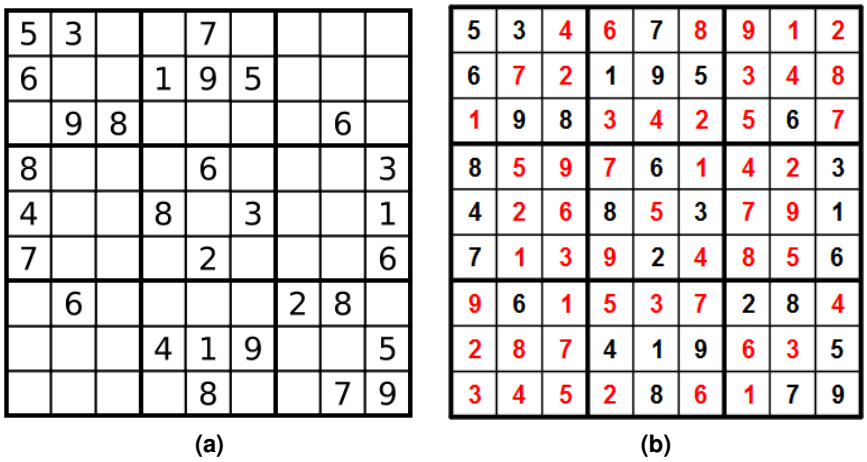

Figura 3: (a) Uno schema di gioco (b) La sua soluzione

Descrizione matematica

Come tutti i giochi logici, Sudoku può essere descritto completamente mediante nozioni di logica, in questo caso si applica la combinatoria.
Il gioco si svolge in matrici, che chiamiamo matrici Sudoku di aspetto 9*x*9 (le griglie) le cui caselle possono contenere un elemento di un insieme di 9 oggetti distinguibili, oppure un ulteriore oggetto diverso dai precedenti.
Per fissare i discorsi conveniamo che le righe e le colonne delle matrici siano individuate dagli interi da 1 a 9, che i nove oggetti siano gli interi dell'insieme $|9| := \{1,2,3,4,5,6,7,8,9\}$, che l'oggetto ulteriore sia denotato con la lettera *b* e

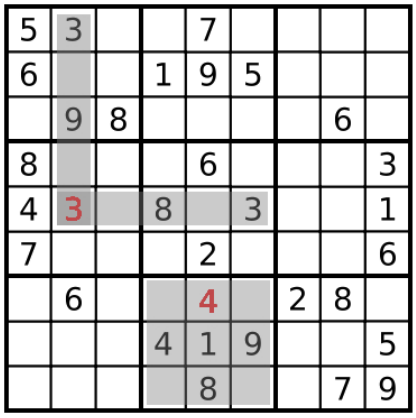

Figura 4: Esempi di Sudoku errato

che una casella contenente b sia detta casella bianca o vuota. Una matrice Sudoku M viene considerata suddivisa in 9 blocchi di aspetto $3x3$ che denotiamo $B_{h,k}$ con $h,k = 1,2,3$; il blocco $B_{h,k}$ riguarda le righe relative agli indici $3h-2, 3h-1$ e $3h$ e le colonne relative agli indici $3k-2, 3k-1$ e $3k$. In ogni riga, colonna e regione di una matrice Sudoku i valori interi non possono essere ripetuti. Una istanza di Sudoku, detta anche griglia proposta o matrice incompleta, è una matrice Sudoku che presenta alcune celle bianche. Scopo del gioco è la trasformazione della griglia proposta in una matrice completa, cioè in una matrice priva di celle bianche e quindi tale che in ogni sua riga, colonna e regione compaiano tutti gli elementi di $|9|$ (ciascuno una sola volta).
Si osserva che una matrice Sudoku completa è un quadrato latino di ordine 9 avente per blocchi matrici $3x3$ con i nove numeri da 1 a 9. Affinché una matrice incompleta sia considerata valida, ai fini del gioco, è necessario che la soluzione sia univoca, ovvero non devono sussistere due o più soluzioni differenti, nei quali casi il gioco viene considerato non valido. La difficoltà di un sudoku non è data dalla quantità di numeri iniziali, bensì dalla loro disposizione.

Le soluzioni di una qualsiasi altra matrice incompleta sono un sottoinsieme delle soluzioni della matrice vuota. Storicamente questo gioco è un caso ben più facile da risolvere di un antico e famoso gioco di logica-matematica a cui si è dedicato anche Eulero; si tratta dei "quadrati greco-latini". In questo caso, a differenza del Sudoku, non vi sono griglie interne e l'unica condizione da rispettare è che in ogni riga ed in ogni colonna compaiano tutti i numeri da 1 ad $n * n$ una volta ed una volta sola, dove n è la dimensione del quadrato (nel caso del Sudoku $n = 9$). Inoltre occorre sovrapporre n soluzioni di questo tipo (dette quadrati latini) in modo che ciascuna casella abbia una n-upla distinta. Al contrario di quanto spesso si afferma, il sudoku è un gioco di logica e non di matematica, né ha a che fare con i numeri. Le proprietà dei numeri non vengono mai utilizzate e neppure viene mai utilizzato il fatto che siano dei numeri. Per rendersi conto della cosa basta pensare che il gioco sarebbe esattamente identico se anziché i primi nove numeri si usassero le prime nove lettere dell'alfabeto oppure nove simboli diversi tra loro (non c'è nemmeno bisogno che tra i simboli sussista un ordine). Tuttavia alcuni ricercatori matematici hanno messo in evidenza sorprendenti legami tra Sudoku e Quadrati magici.[5]

[5] Un quadrato magico è una serie di numeri disposti a forma di quadrato in modo che la somma di ogni riga, di ogni colonna e di ogni diagonale sia uguale.

5 L'implementazione

Nelle sezioni precedenti sono stati descritti i concetti chiave per la realizzazione del progetto d'esame. Questa sezione introdurrà la parte pratica, illustrando l'ambiente di sviluppo utilizzato, le scelte progettuali ed i metodi con cui queste sono state realizzate. Si comincia con l'introdurre l'ambianete di sviluppo.

Il codice del progetto è stato sviluppato in ambiante Windows con l'IDE Code::Blocks, che è un full-optional IDE (Integrated Development Environment), open source e multipiattaforma. È scritto in C++ usando wxWidgets.[6]
Usando un'architettura basata su plugin, le sue capacità e caratteristiche sono estese proprio dai plugin installati. Attualmente, Code::Blocks è orientato verso il C/C++. Supporta diversi compilatori tra cui MinGW (Minimalist GNU for Windows), che è il porting in ambiente Windows del famoso compilatore GCC per Unix, ed è quello utilizzato per la realizzazione del progetto.
Il compilatore MinGw è stato scelto, non soltanto perchè è gratuito, ma anche perchè è uno dei migliori compilatori che ci siano in circolazione.

Il progetto è stato sviluppanto in modo (per quanto possibile) simile ad un videogame. Per questa ragione non ci si riferirà all'utilizzatore del programma come utente bensì come **giocatore**. Sono state implementate apposite funcion nel programma principale per rendere la "partita" quanto più interattiva possibile. Tali function vengono utilizzate per salvare, caricare e resettare una partita in base alla scelta del giocatore. Semplici menù aiutano il giocatore nell'inserimento e la cancellazione di un numero ed, inoltre, rendono la possibilità di visionare la soluzione di qualsiasi Sudoku.

Queste possibilità vengono descritte in dettaglio nel seguito della relazione.
Dopo questa breve descrizione degli strumenti utlizzati per la stesura del codice, e delle idee base utilizzate per realizzarlo, il prossimo passo sarà quello di descrivere quest'ultimo in modo approfondito.

[6] Conosciuto una volta con il nome di wxWindows, è un toolkit grafico multipiattaforma e open source, cioè una libreria di componenti elementari per costruire una interfaccia grafica (GUI)

La classe Sudoku

La *classe Sudoku* rapprensenta il cuore dell'intero progetto. Il programma utilizza le istanze della classe esattamente come un essere umano utilizza un qualsiasi oggetto d'uso comune, come ad esempio un telefonino o un televisore; ovvero utilizzerà solo le informazioni ad esso note cioè l'interfaccia pubblica, ignorando completamente come essa funzioni.
Per rendere quanto più chiara possibile tale situazione, basti pensare che le parti del programma che utlizzano un oggetto di tipo Sudoku vedranno tale oggetto come mostrato in figura 5; una "scatola chiusa" che permette l'interazione solo ed esclusivamente tramite la sua interfaccia pubblica.

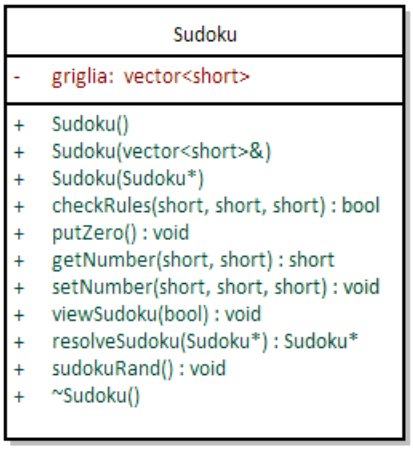

Figura 5: Classe Sudoku

Quindi il programma ignora completamente cosa ci sia al suo interno, esattamente come una persona utilizza il tasto d'accensione di un televisore; molte persone non conoscono o semplicemente non vogliono sapere cosa accade quando accendono la televisione, basta solo che l'oggetto in questione risponda esattamente all'azione richiesta.
Chiarito questo concetto si analizzerà nel dettaglio la classe.

L'interfaccia privata

L'interfaccia privata della classe Sudoku è costituita dal dato `griglia` di tipo `vector<short>`.
Il dato membro contiene i numeri che compongono il Sudoku; si è preferito il tipo `short` (16 bit) perchè tali numeri, come descritto in precedenza nella sezione 4, sono compresi in un range piccolo (da 0 a 9), quindi utilizzare un tipo `int` ovvero 32 bit per memorizzare dei valori così piccoli equivaleva ad uno spreco inutile di memoria.

Il container `vector` ad una prima lettura può sembrare inutile, in quanto si sarebbe potuto usare un semplice array considerato che si opera su un size ben definito; il size è 81 ovvero il numero di caselle che formano la griglia del gioco. Uno dei motivi dell'utilizzo del container `vector` è che garantisce una maggiore flessibilità nell'inizializzazione, inoltre senza questo container non sarebbe stato possibile implementare i metodi con l'ausilio degli algoritmi STL contenuti nell' header `<algorithm>`.

L'interfaccia pubblica

L'interfaccia pubblica è costituita da tutti i metodi accessibili da qualsiasi punto del programma. La tabella 3 elenca i metodi pubblici realizzati nel progetto d'esame.

Tabella 3: Metodi dell'interfaccia pubblica

METODO	DESCRIZIONE
`Sudoku()`	Costruttore di default
`Sudoku(vector<short> &)`	Costruttore parametrizzato
`Sudoku(Sudoku *)`	Costruttore di copia
`bool checkRules(short,short,short)`	Verifica le regole del Sudoku
`void putZero()`	Rimpiazza dei numeri con degli zeri
`short getNumber(short,short)`	Restituisce un numero *n* nella griglia
`void setNumber(short,short,short n=0)`	Inserisce un numero *n* nella griglia
`void viewSudoku(bool s=false)`	Visualizza il Sudoku
`Sudoku* resolveSudoku(Sudoku*)`	Risolve il Sudoku
`void sudokuRand()`	Genera un Sudoku random
`~Sudoku()`	Distruttore

Sudoku() è il costruttore di default della classe ed è implementato inline nel corpo della classe. Il suo compito è quello di inizializzare il vettore griglia con ottantuno zeri.

Costruttore di default

```
Sudoku()
{
  griglia.insert(griglia.begin(), 81 , 0);
}
```

Come si può notare è stato utilizzato un metodo del container `vector`: il metodo `insert`. Questo permette di inizializzare il vettore in modo facile ed intuitivo. Il primo parametro di `insert` è l'iteratore d'inizio del vettore, il secondo è il numero di elementi da inserire ed infine il terzo parametro è il valore degli elementi da inserire.

Sudoku(vector<short> &) è il costruttore parametrizzato della classe ed è implementato inline nel corpo della classe. Viene utilizzato per inizializzare la griglia con un Sudoku predefinito, caricato da file o inserito da tastiera. Il suo parametro di input è un vector<short> referenziato che contiene i numeri per l'inizializzazione.

Costruttore parametrizzato

```
Sudoku(vector<short>  &v)
{
  griglia.insert(griglia.begin(),v.begin(),v.end() );
}
```

Si noti che anche in questo metodo è stato usato la funzione membro `insert` del container `vector` in modo differente da prima; in questo caso inserisce a partire dall'iteratore `griglia.begin()` tutti gli elemeti compresi nell'intervallo degli iteratori `v.begin()` e `v.end()`. Con l'utilizzo di un semplice array si sarebbe dovuto utilizzare un ciclo for che avrebbe reso il codice meno elegante considerando che è una funzione inline.

Sudoku(Sudoku *s) è il costruttore di copia usato per ripristinare una partita. Il suo parametro di input è un puntatore ad un oggetto di tipo Sudoku contente una griglia non risolta. L'idea base che sta dietro questo costruttore di copie è quella di facilitare un nuovo inizio di partita al giocatore, che dopo aver comesso molti errori durante il gioco, ha a disposizione il Sudoku allo stato iniziale.

Costruttore di copia

```
Sudoku(Sudoku *s)
{
  griglia.insert(griglia.begin(),s->griglia.begin(),s->griglia.end());
}
```

Il metodo insert viene usato in modo analogo a quello del costruttore parametrizzato.

bool checkRules(short,short,short) è il metodo che esegue il check delle varie regole del Sudoku. Accetta come parametri di input il numero *n* che si vuole inserire nella griglia e la sua relativa posizione (indici i e j). Se tutte le regole sono rispettate ritorna il valore TRUE, altrimenti FALSE.

Metodo checkRules

```
bool Sudoku::checkRules(short n,short i,short j)
{
    if (i<0||i>8 || j<0||j>8 || n<1||n>9)
        return false;
    if (griglia[i*9+j]!=0)
        return false;
    for (short k=0;k<9;k++)
  if (griglia[i*9+k]==n || griglia[k*9+j]==n)
            return false;

    for (short k=(i/3*3);k<(i/3*3)+3;k++)
        for (short z=(j/3*3);z<(j/3*3)+3;z++)
            if (griglia[k*9+z]==n)
                return false;
    return true;
}
```

Il metodo effettua un primo controllo per verificare se gli indici ed il numero da inserire rientrano nel range dei possibili valori ammessi. Successivamente controlla se la posizione è occupata da un altro numero. Il controllo successivo verifica se nella stessa riga o nella stessa colonna sia presente lo stesso numero che si vuole inserire. Infine controlla la regione $3x3$ per verificare se il numero è già presente. In questa parte di codice si sfruttano le proprietà degl'indici delle matrici; se si divide e moltiplica per 3 gli indici i e j si otterà sempre come risultato 0, 3 o 6 [7] in quanto il valore di tale operazione viene memorizzato in una variabile di tipo short ovvero in un numero intero. Una cosa che deve essere evidenziata è il fatto che il primo controllo effettivamente sia inutile in quanto ogni inserimento nel programma è già controllato. Tale controllo è stato inserito per generalizzare la classe.

[7] corrispondeti agli indici della regione interessata

void putZero() è la funzione membro della classe Sudoku che rimpiazza casualmete con degli zero i numeri della griglia completa, generando ogni volta varie configuranzioni di Sudoku incompleto.

Metodo putZero

```
void Sudoku::putZero()
{
    srand((unsigned int)time(0));
    short rZero=51+rand()%10;
    while (rZero>0)
    {
        short rind=rand()%81;
        if (griglia[rind]!=0)
        {
            griglia[rind]=0;
            rZero--;
        }
    }
}
```

La prima operazione che viene eseguita consiste nel cambiare il seed della generazione dei numeri nella funzione rand(), legando tale funzione con l' orologio di sistema. Successivamente, calcola con una chiamata alla funzione rand(), il valore della variabile `rZero` che è la quantità di zeri da inserire. Inseguito genera a caso `rind` che è l' indice, compreso fra 0 e 81, del numero da rimpiazzare. Se la casella contiene già uno zero ovviamente non inserisce nulla.

short getNumber(short,short) è il metodo che restituisce un numero *n* nella griglia alla posizione (i,j) ed è implementato inline nel corpo della classe.

Metodo getNumber

```
short getNumber(short i,short j)
{
  return griglia[i*9+j];
}
```

In questo, come in altri metodi, il dato membro griglia di tipo `vector<short>` è utlizzato con la notazione **[]** come un classico array; questa notazione permette l'accesso diretto ad una casella sfruttando le proprietà degli indici di un array.

void setNumber(short,short,short n=0) è il metodo che inserisce un numero *n* nella griglia alla posizione (i,j). Il metodo è anche utilizzato per eliminare un numero dal Sudoku; infatti richiamando il metodo senza specificare l'ultimo parametro, che corrisponde al numero da inserire, viene inserito il numero zero per default. Le funzioni con parametri di default sono un'altro vantaggio del linguaggio C++, in quanto con un solo metodo è possibile eseguire due diverse azioni sull'oggetto.

Metodo setNumber

```
short getNumber(short i,short j,short n=0)
{
  griglia[i*9+j] = n;
}
```

É importante notare come in questo metodo *non* viene richiamata la funzione membro `checkrules`. Il motivo è semplice: nel Sudoku è possibile **sbagliare**! Quindi il giocatore non deve essere "pilotato" dal gioco, e in caso di errore è lo stesso giocatore che deve rimediare in qualche modo.

void Sudoku::viewSudoku(bool s=false) è il metodo che visualizza a video la griglia del Sudoku. Se f è TRUE visualizza la griglia, altrimenti solo la parola SUDOKU.

Metodo viewSudoku

```
void Sudoku::viewSudoku(bool f)
{
    system("cls");
    char s[5][30]={{" SSS U  U DDD   OO  K  K U  U"},
        {"S    U  U D  D O  O K K  U  U"},
        {" SS  U  U D  D O  O KK   U  U"},
        {"   S U  U D  D O  O K K  U  U"},
        {"SSS   UU  DDD   OO  K  K  UU "}
    };
    cout<<endl;
    for (short i=0;i<5;i++)
    {
        cout<<"\t\t      "<<s[i]<<endl;
    }
    if(!f)
    {
        cout << "\n\t\t\t  1 2 3   4 5 6   7 8 9  "<<endl;
        cout << "\n\t\t\t-------------------------"<<endl;
        for (short i=0;i<9;i++)
        {
            cout << "\t\t     "<<i+1<<"  | ";
            for (short j=0;j<9;j++)
            {
    if (griglia[i*9+j]==0)
                    cout <<"  ";
                else
                    cout << griglia[i*9+j] << " ";
                if ((j+1)%3 == 0)
                    cout<<"| ";
            }
            cout<<endl;
            if ((i+1)%3==0&&i<8)
                cout << "\t\t\t|----------------------|"<<endl;
            else
                if (i==8)
                    cout << "\t\t\t-------------------------"<<endl;
        }
    }
}
```

Per migliorare la visualizzazione del Sudoku è stato inserito il controllo `if (griglia[i*9+j]==0)` che al verificarsi della condizione sostituisce gli zeri con uno spazio vuoto. La scelta di far stampare solo la parola SUDOKU è solo una questione di stile, in quanto inserire la matrice `char s[5][30]` nel programma principale rendeva il codice meno leggibile ed elegante.

Sudoku* resolveSudoku(Sudoku *) è il metodo che realizza la soluzione del Sudoku con la tecnica del Backtracking. Analizza la griglia alla ricerca di uno zero. Se viene trovato almeno uno zero, significa che il Sudoku non è stato ancora risolto, quindi cerca un qualsiasi numero che soddisfi i vincoli descritti in `checkRules`, in seguito il metodo effetua una chiamata ricorsiva per spostarsi in un altro ramo di decisione, se tale chiamata restituisce un valore *non* `NULL` aggiungiunge un pezzo alla soluzione, altrimeti resetta il numero inserito e *torna indietro* sui suoi passi (Backtrack). Il metodo continua in questo modo finchè non vi sono più zeri, in tal caso ha trovato la soluzione. Se non si riesce a trovare una possibile soluzione il metodo restituisce un puntatore `NULL`. Il metodo ha come parametro di input un puntatore ad un' istanza della classe Sudoku e restituisce la soluzione, in caso contrario un puntatore `NULL`.

Metodo resolveSudoku

```
Sudoku* Sudoku::resolveSudoku(Sudoku *s)
{
    vector<short>::iterator it;
    it=find(s->griglia.begin(),s->griglia.end(),0);
    if (it !=s->griglia.end())
    {
   for (short i=0;i<9;i++)
            for (short j=0;j<9;j++)
                if (s->griglia[i*9+j] == 0)
                {
                    for (short k=1;k<=9;k++)
                    {
                        if (s->checkRules(k,i,j))
                        {
                            setNumber(i,j,k);
                            Sudoku *app=resolveSudoku(s);
                            if (app!=NULL)
                                return app;
                        }
                        setNumber(i,j);
                    }
         return NULL;
                }
  }
 return s;
}
```

In questo metodo l'iteratore del container `vector<short>` è utilizzato per contenere il valore restituito dall'algoritmo STL `find`. Tale algoritmo permette di cercare all'interno di un container sequenziale il valore 0 nell'intervallo fra gli iteratori `s->griglia.begin()` e `s->griglia.end()`, ovvero cerca in tutto il vettore il valore 0. L'algoritmo `find` termina la ricerca al primo valore 0 che incontra e restituisce un iteratore alla posizione di tale valore, in caso contrario restituisce l'iteratore `end()`. Quindi se l'algoritmo `find` trova uno zero significa che il Sudoku non è stato ancora risolto, e si procede con la fase backtracking.
Nei cicli che seguono il controllo `if (it !=s->griglia.end())` vengono rica-

viati gli indici (i,j) indispensabili per i controlli contenuti nel metodo `checkRules`. In una prima analisi sembrerebbe inutile cercare una seconda volta il valore zero, in quanto questo valore è contenuto nell'iteratore precedentemente calcolato.

Purtroppo questi cicli sono davvero indispensabili in quanto ricavano gli indici (i,j) senza i quali non si potrebbero effettuare i controlli sulle righe e colonne della griglia, ed il controllo su una regione di essa.

Successivamente ottenuti gli indici, si prova l'inserimeto di un numero. Se il numero può essere inserito si richiama il metoro `setNumber`.
Trovata la pseudo-soluzione, istanziamo l'oggetto `app` per analizzare un altro possibile percoso richiamando ricorsivamente il metodo. Se questo restitusce un valore non `NULL`, si aggiunge un altro pezzo alla soluzione, altrimeti viene resettato l'inserimeto fatto, richiamando nuovamente il metodo `setNumber`, in quanto questo percorso porta ad una non-soluzione. Quindi si torna indietro (Backtrack) restituendo un valore `NULL`. Se nel Sudoku non sono presenti zeri, vuol dire che è stato risolto e il metodo restituisce la soluzione.

void sudokuRand() è il metodo che permette la generazione di Sudoku random. Genera a caso dieci numeri che posiziona in (i,j) anch'essi random. Il metodo controlla se tali numeri possono essere inseriti richiamando il metodo `checkRules`. Succesivamente risolve il Sudoku con `resolveSudoku`, ed infine rimpiazza alcuni numeri richiamando il metodo `putZero`.

Metodo sudokuRand

```
void Sudoku::sudokuRand()
{
    srand((unsigned int)time(0));
    for (short k=0;k<10;k++)
    {

        short i=rand()%9,j=rand()%9,rnum=1+rand()%9;
        if ( checkRules(rnum,i,j) )
            setNumber(i,j,rnum);
    }
    resolveSudoku(this);
    putZero();
}
```

Come prima istruzine il metodo `sudokuRand` cambia il seed della funzione `rand()`. Succesivamente numeri e indici vengono generati nel loro range per rispettare le regole. In questo ciclo vengono generati solo 10 numeri con i rispettivi indici.

I numeri sono sufficienti per garantire che i Sudoku random siano sempre diversi tra loro in modo tale che se viene posizionato un solo numero, si abbiano

$$9 * 81 = 729$$

possibili Sudoku. Dopo che i numeri sono stati inseriti si richiama il metodo `resolveSudoku(this)`; this punta all'oggetto per il quale è stata richiamata la funzione membro, anche questo puntatore fa parte delle aggiunte fatte al C++ rispetto al linguaggio C. Infine si eliminano dei numeri dalla griglia in modo da ottenere un Sudoku giocabile.

Il main

Dopo aver descritto la classe Sudoku, si parlerà dell'implementazione del programma che utilizza tale classe.

Il main è strutturato in modo semplice; al suo interno sono solo presenti chiamate a function specializzate in diversi compiti. Queste function sono riportate nella tabella 4.

Tabella 4: Function del programma principale

FUNCTION	DESCRIZIONE
`void print()`	Visualizza a video l'animazione d'apertura
`short menu()`	Visualizza a video il menù principale
`void menu1(Sudoku **)`	Visualizza a video il menù di gioco
`void inputSudoku(Sudoku **)`	Permette di inizializzare un Sudoku da tastiera
`void insertNum(Sudoku **)`	Permette di giocare inserendo i numeri nel Sudoku
`void delNum(Sudoku **)`	Permette di giocare rimuovendo un numero del Sudoku
`void loadSudoku(vector<short>&,bool flag=false)`	Carica da file una partita salvata o il Sudoku predefinito
`void saveSudoku(Sudoku **)`	Permette il salvataggio di una partita su file

Nel seguito si farà una descrizione delle function più significative realizzate.

void loadSudoku(vector<short> &vec, bool flag=false) è la function che carica da file una partita salvata, o il Sudoku predefinito salvato nel file "sudoku.su". Riceve come parametri di input il vettore che conterrà i numeri del Sudoku, e un flag di controllo. Quest'ultimo è di default FALSE e indica che si vuole caricare una partita salvata dal giocatore, se viene impostato al valore TRUE viene caricato il file sudoku.su.

Function loadSudoku

```
void loadSudoku(vector<short> &vec, bool flag)
{
    string name;
    vec.erase(vec.begin(),vec.end());
    if (flag)
    {
        cout<<"Nome della partita da caricare : ";
        cin>>name;
        name+=".su";
    }
    else
        name="sudoku.su";
    const char *p=name.c_str();
    ifstream file(p);
    if (!file.is_open())
    {
        cout<<"Impossibile aprire il file"<<endl;
        system("pause");
        exit(1);
    }
    else
    {
  if (name!="sudoku.su")
        {
            cout<<endl;
            progressBar();
            cout<<"\n\n\t\t\tPartita Caricata!\a"<<endl;
            Sleep(2000);
            system("cls");
        }
        char c;
        while (!file.eof())
        {
            c=file.get();
            vec.push_back(short(c-48));
        }
    }
    file.close();
}
```

Il metodo `erase()` permette di eliminare tutti gli elementi dal vettore compresi fra gli iteratori `begin()` e `end()`. Quest' operazione viene effettuata per permettere di caricare una partita in qualsiasi momento, anche se il vettore era già stato inizializzato. Se la variabile `flag` è impostata a TRUE si vuole caricare una partita salvata dal giocatore. A questo punto il giocatore inserisce il nome della partita da caricare; il nome viene memorizzato nella variabile `string name`, e concatenato con l' estensione `".su"`. Il metodo `c_str()` del tipo string restituisce una

stringa costante contente il carattere di fine stringa, che servirà per l'apertura del file di input. Il controllo `if (name!=sudoku.su)` fa si che l'animazione della progressbar *non* si visualizzi in caso si voglia caricare il Sudoku predefinito. Nel ciclo `while` si legge il file un carattere alla volta e si memorizza nel vettore con il metodo `push_back()`. Infine si chiude il file.

saveSudoku(Sudoku **s) è function che permette il salvataggio di una partita su file. Riceve in input un'istanza della classe Sudoku.

Function saveSudoku

```
void saveSudoku(Sudoku **s)
{
    string name;
    cout<<"Inserisci il nome della partita da salvare"<<endl;
    cin>>name;
    name+=".su";
    const char *p=name.c_str();
    ofstream file;
    file.open(p,ios::trunc|ios::out);
    if (!file.is_open())
    {
        cout<<"Impossibile aprire il file"<<endl;
        system("pause");
        exit(1);
    }
    else
    {
        cout<<endl;
        progressBar();
        cout<<"\n\n\t\t\tPartita Salvata!\a"<<endl;
        Sleep(2000);
        system("cls");
        for (short i=0;i<9;i++)
            for (short j=0;j<9;j++)
                file.put(char((*s)->getNumber(i,j)+48));
        file.close();
    }
}
```

La function è implementata nello stesso modo di `loadSudoku`, con la differenza che salva una partita su file. Le function `saveSudoku` e `loadSudoku` sono fondamentali per rendere l'idea di un videogame, in quanto attualmente non esite gioco che non permetta di salvare o caracare una partita in qualsiai momento.

Appendice

A Codice Sorgente

A.1 sudoku.cpp

sudoku.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "sudoku.h"
//Contiene la function per l'animazione della progressbar
#include "progressbar.h"

using namespace std;

void print();
short menu();
void menu1(Sudoku **);
void inputSudoku(Sudoku **);
void insertNum(Sudoku **);
void delNum(Sudoku **);
void loadSudoku(vector<short> &, bool flag=false);
void saveSudoku(Sudoku **);

int main()
{
    vector<short> vec;
    //Creazione di un' istanza di Sudoku
    Sudoku *sudoku=NULL;
    short m;
    //Cambia il colore della console di Windows
    system("color F");
    print();
    do
    {
        m=menu();
        switch (m)
        {
        case 1: //l'opzione 1 carica dal file "sudoku.su" il Suduku predefinito
            loadSudoku(vec);
            try //Gestione delle eccezzioni per l'allocazione dinamica della memoria
            {
                sudoku=new Sudoku(vec);
            }
            catch (bad_alloc xa)
            {
                cout<<"Errore allocazione memoria!"<<endl;
                exit(1);
            }
            sudoku->viewSudoku();
            menu1(&sudoku);
            break;
        case 2: //l'opzione 2 fa generare un Suduku in modo random
            try
            {
                sudoku=new Sudoku();
```

```
            }
            catch (bad_alloc xa)
            {
                cout<<"Errore allocazione memoria!"<<endl;
                exit(1);
            }
            sudoku->sudokuRand();
            sudoku->viewSudoku();
            menu1(&sudoku);
            break;
        case 3: //l'opzione 3 permette di creare un Suduku
            try
            {
                sudoku=new Sudoku();
            }
            catch (bad_alloc xa)
            {
                cout<<"Errore allocazione memoria!"<<endl;
                exit(1);
            }
            inputSudoku(&sudoku);
            sudoku->viewSudoku();
            menu1(&sudoku);
            break;
        case 4: //l'opzione 4 permette di caricare un Suduku  da file
            loadSudoku(vec,true);
            try
            {
                sudoku=new Sudoku(vec);
            }
            catch (bad_alloc xa)
            {
                cout<<"Errore allocazione memoria!"<<endl;
                exit(1);
            }
            sudoku->viewSudoku();
            menu1(&sudoku);
            break;
        case 0: //l'opzione 0 termina il programma
            system("cls");
            cout<<"Grazie per aver giocato !!! ;-)"<<endl;
            Sleep(2000);
            delete sudoku;
        }
    }
    while (m>=1&&m<=4);
    return 0;
}

//Function che visualizza a video l'animazione d'apertura
void print()
{
    char s[7][37]={ {" SSSS U   U DDD    OOO  K    K U   U"},
                    {"S    U   U D  D  O   O K   K  U   U"},
                    {"S    U   U D   D O   O K  K   U   U"},
                    {" SSS  U   U D   D O   O KKK    U   U"},
                    {"    S U   U D   D O   O K  K   U   U"},
                    {"    S U   U D  D  O   O K   K  U   U"},
                    {"SSSS   UUU  DDD    OOO  K    K  UUU "}};
    cout<<endl;
    for(int i=0;i<7;i++)
        cout<<"\t\t"<<s[i]<<endl;
```

```
    Sleep(750);
    system("cls");
    for(int k=0;k<37;k++)
    {
        cout<<endl;
        for(int i=0;i<7;i++)
        {
            cout<<"\t\t";
            int x=0;
            while(x<k)
            {
                cout<<" ";
                x++;
            }
            for(int j=k;j<37;j++)
                cout<<s[i][j];
            cout<<endl;
        }
        system("cls");
    }

}
//Function che visualizza a video il menu' principale e restitusce la scelta fatta dall'utente
short menu()
{
    short scelta ;
    bool f; /*flag di controllo utlizzato per verificare
              la corretta digitazione delle opzioni*/
    do
    {
        f=false;
        Sudoku s;
        s.viewSudoku(true);
        cout<<"\n\n\n\t        * * * * * * * M * E * N * U * * * * * * *   "<<endl;
        cout<<"\t       *   1 - Gioca con Sudoku predefinito      *  "<<endl;
        cout<<"\t      *   2 - Gioca Sudoku generato casualmente   * "<<endl;
        cout<<"\t     *   3 - Gioca con un Sudoku creato da te      *"<<endl;
        cout<<"\t      *   4 - Carica Sudoku salvato               * "<<endl;
        cout<<"\t       *   0 - Alla prossima :-)                 *  "<<endl;
        cout<<"\t        * * * * * * * * * * * * * * * * * * * *   "<<endl;
        cout<<"\t                     Scegli : ";
        cin>>scelta;
        if (scelta<0||scelta>4)
        {
            cout<<"Rivedi il menu', hai effettuato una scelta errata"<<endl;
            Sleep(2000);
            system("cls");
            f=true;
        }
    }while (f);
    return scelta;
}

/*Function che visualizza a video il menu' di gioco.
  Riceve e restituisce un'istanza della classe Sudoku*/
void menu1(Sudoku **s)
{
    //Salva lo stato del Sudoku,per poterlo resettare su richiesta dell'utente
    Sudoku *save=new Sudoku(*s);
    short scelta;
    bool f;
    do
```

```
    {
        do
        {
            f=false;
            cout<<"\t\t   **** P * A * R * T * I * T * A **** "<<endl;
            cout<<"\t\t  *   1 - Inserisci numero            *"<<endl;
            cout<<"\t\t  *   2 - Cancella numero              *"<<endl;
            cout<<"\t\t  *   3 - Riavvia Sudoku               *"<<endl;
            cout<<"\t\t  *   4 - Soluzione                    *"<<endl;
            cout<<"\t\t  *   5 - Salva Sudoku                 *"<<endl;
            cout<<"\t\t  *   0 - ..torna al menu' principale *"<<endl;
            cout<<"\t\t   ********************************** "<<endl;
            cout<<"\t\t              Scegli : ";
            cin>>scelta;
            if (scelta<0||scelta>5)
            {
                cout<<"Rivedi il menu', hai effettuato una scelta errata"<<endl;
                f=true;
            }
        }while (f);
        switch (scelta)
        {
        case 1:  //l'opzione 1 permette l'inserimeto di un numero
            insertNum(s);
            break;
        case 2: //l'opzione 2 permette la cancellazione di un numero
            delNum(s);
            break;
        case 3: //l'opzione 3 permette di resettare il Sudoku
            delete *s; //Dealloca *s
            *s=new Sudoku(save); //Ripristina lo stato iniziale del Sudoku
            break;
        case 4: //l'opzione 4 permette di trovare una soluzione se esiste
            if ((*s)->resolveSudoku(*s)==NULL)
            {
                cout<<"\nSoluzione non trovata!"<<endl;
                system("pause");
            }
            break;
        case 5: //l'opzione 5 permette di salvare il Sudoku su un file scelto dall'utente
            saveSudoku(s);
            break;
        }
        system("cls");
        if (scelta!=0)
            (*s)->viewSudoku();
        else
            delete save;

    }while (scelta>=1&&scelta<=5);
}

/*Function che permette di inizializzare un Sudoku da tastiera.
  Riceve e restituisce un'istanza della classe Sudoku*/
void inputSudoku(Sudoku **s)
{
    short num;
    cout<<"Digita le righe del Sudoku(0 per le caselle vuote)"<<endl;
    for (short i=0;i<9;i++)
    {
        for (short j=0;j<9;j++)
        {
```

```
            cin>>num;
        /*Controlla il renge di numeri (da 0 a 9)
                possibili da inserire,0 per una posizione vuota*/
            if ((num>=0)&&(num<10))
                (*s)->setNumber(i,j,num);
            else
            {
                cout<<"Attento, devi inserire un numero fra 0 e 9"<<endl;
                j--;
            }
        }
        cout<<"la  riga "<<i+1 <<" e' stata completata"<<endl;
    }

}

/*Function che permette di giocare inserendo i numeri nel Sudoku.
  Riceve e restituisce un'istanza della classe Sudoku*/
void insertNum(Sudoku **s)
{
    short num,i,j;
    bool f;
    do
    {
        cout<<"NUMERO : ";
        cin>> num;
        cout<<"RIGA : ";
        cin>>i;
        cout<<"COLONNA : ";
        cin>>j;
        f=false;
        if ((*s)->getNumber(i-1,j-1)!=0)
        {
            f=true;
            cout<<"Posizione occupata\n";
        }
    /*Controlla se gli indici ed il numero rientrano nel range
      dei possibili valori e se la posizione sia libera*/
    }while (num<1||num>9||i<1||i>9||j<1||j>9||f);
    (*s)->setNumber(i-1,j-1,num);
}

/*Function che permette di giocare rimuovendo un numero del Sudoku.
  Riceve e restituisce un'istanza della classe Sudoku*/
void delNum(Sudoku **s)
{
    short i,j;
    do
    {
        cout<<"RIGA : ";
        cin>>i;
        cout<<"COLONNA : ";
        cin>>j;
    //Controlla se gli indici rientrano nel range dei possibili valori
    }while (i<1||i>9||j<1||j>9);
    //Se non viene specificato il numero setNumber inserisce uno 0
    (*s)->setNumber(i-1,j-1);
}

/*Function che carica da file una partita salvata o il Sudoku predefinito
  salvato nel file "sudoku.su". Riceve come parametri il vettore che conterra'
  i numeri del Sudoku ed un flag di controllo, quest'ultimo e' di default
```

```
  FALSE e indica che si vuole caricare una partita salvata dall'utente,
  se viene impostato a TRUE viene caricato il file "sudoku.su" */
void loadSudoku(vector<short> &vec, bool flag)
{
    string name;
    vec.erase(vec.begin(),vec.end());//Elimina tutti gli elementi dal vettore
    if (flag) //Si vuole caricare una partita salvata dall'utente
    {
        cout<<"Nome della partita da caricare : ";
        cin>>name;
        name+=".su";//Concatena e assegna l' estensione
    }
    else  //Si vuole caricare il Sudoku predefinito
        name="sudoku.su";
    const char *p=name.c_str(); //Crea una stringa costante
    ifstream file(p);
    if (!file.is_open())
    {
        cout<<"Impossibile aprire il file"<<endl;
        system("pause");
        exit(1);
    }
    else
    {
  //Se non si vuole caricare "sudoku.su" visualizza l'animazione
        if (name!="sudoku.su")
        {
            cout<<endl;
            progressBar();
            cout<<"\n\n\t\t\tPartita Caricata!\a"<<endl;
            Sleep(2000);
            system("cls");
        }
        char c;
        while (!file.eof())
        {
            c=file.get();
            vec.push_back(short(c-48)); //Conversione da carattere ASCII a short
        }
    }
    file.close();
}

/*Function che permette il salvataggio di una partita su file.
  Riceve un'istanza della classe Sudoku*/
void saveSudoku(Sudoku **s)
{
    string name;
    cout<<"Inserisci il nome della partita da salvare"<<endl;
    cin>>name;
    name+=".su";  //Concatena e assegna l' estensione
    const char *p=name.c_str(); //Crea una stringa costante
    ofstream file;
    file.open(p,ios::trunc|ios::out);
    if (!file.is_open())
    {
        cout<<"Impossibile aprire il file"<<endl;
        system("pause");
        exit(1);
    }
    else
    {
```

```
        cout<<endl;
        progressBar();
        cout<<"\n\n\t\t\tPartita Salvata!\a"<<endl;
        Sleep(2000);
        system("cls");
        for (short i=0;i<9;i++)
            for (short j=0;j<9;j++)
                file.put(char((*s)->getNumber(i,j)+48));//Conversione da short a carattere ASCII
        file.close();
    }
}
```

A.2 sudoku.h

sudoku.h

```
#ifndef Sudoku_H
#define Sudoku_H
#include <vector>
#include <algorithm>
#include <ctime>
using namespace std;

class Sudoku
{
    vector<short> griglia; //Griglia dove vengono memorizzati i numeri del Sudoku
public:
    //Costruttore di default: inizializza la griglia del Sudoku con tutti zeri
    Sudoku(){griglia.insert(griglia.begin(),81,0);}

    /*Costruttore parametrizzato che inizializza la griglia con un Sudoku
      predefinito, caricato da file o inserito da tastiera*/
    Sudoku(vector<short> &v){griglia.insert(griglia.begin(),v.begin(),v.end());}

    //Costruttore di copia usato per ripristinare una partita
    Sudoku(Sudoku *s){griglia.insert( griglia.begin(),s->griglia.begin(),s->griglia.end() );}
    bool checkRules(short,short,short);  //Verifica le regole del Sudoku
    void putZero();
    //Metodo che restituisce un numero n nella griglia alla posizione (i,j)
    short getNumber(short i,short j){return griglia[i*9+j];}
    //Metodo che inserisce un numero n (di default 0) nella griglia alla posizione (i,j)
    void setNumber(short i,short j,short n=0){ griglia[i*9+j] = n;}
    void viewSudoku(bool s=false);
    Sudoku* resolveSudoku(Sudoku*); //Risolve il Sudoku con la tecnica del backtracking
    void sudokuRand();
    ~Sudoku(){};  //Distruttore
};

/*Metodo che fa il check delle varie regole del sudoku, se tutte
  sono rispettate ritorna il valore TRUE, altrimenti FALSE.
  Accetta come parametri il numero n che si vuole inserire nella
  griglia e la sua relativa posizione (indici i e j)*/
bool Sudoku::checkRules(short n,short i,short j)
{
    //Controlla se gli indici ed il numero rientrano nel range dei possibili valori
    if (i<0||i>8 || j<0||j>8 || n<1||n>9)
        return false;
    //Controlla se la posizione e' occupata da un'altro numero
    if (griglia[i*9+j]!=0)
        return false;
    for (short k=0;k<9;k++)
  //Controlla se nella stessa riga o la stessa colonna e' presente lo stesso numero
        if (griglia[i*9+k]==n || griglia[k*9+j]==n)
            return false;
     //Controlla la sotto-griglia 3x3 per verificare se il numero e' gia' presente
    for (short k=(i/3*3);k<(i/3*3)+3;k++)
        for (short z=(j/3*3);z<(j/3*3)+3;z++)
            if (griglia[k*9+z]==n)
                return false;
    return true; //Il numero n puo' essere inserito
}

/*Metodo che visualizza a video la griglia del Sudoku.
```

```
  Se f e' TRUE visualizza la griglia, altrimenti solo
  la scritta "SUDOKU". */
void Sudoku::viewSudoku(bool f)
{
    system("cls");
    char s[5][30]={{" SSS U  U DDD   OO  K  K U  U"},
        {"S    U  U D  D O  O K K  U  U"},
        {" SS  U  U D  D O  O KK   U  U"},
        {"   S U  U D  D O  O K K  U  U"},
        {"SSS   UU  DDD   OO  K  K  UU "}
    };
    cout<<endl;
    for (short i=0;i<5;i++)
    {
        cout<<"\t\t      "<<s[i]<<endl;
    }
    if(!f)
    {
        cout << "\n\t\t\t  1 2 3   4 5 6   7 8 9  "<<endl;
        cout << "\n\t\t\t-------------------------"<<endl;
        for (short i=0;i<9;i++)
        {
            cout << "\t\t     "<<i+1<<"  | ";
            for (short j=0;j<9;j++)
            {
    /* Sostituisce gli zero con uno spazio vuoto
                per una migliore visualizzazione */
                if (griglia[i*9+j]==0)
                    cout <<"  ";
                else
                    cout << griglia[i*9+j] << " ";
                if ((j+1)%3 == 0)
                    cout<<"| ";
            }
            cout<<endl;
            if ((i+1)%3==0&&i<8)
                cout << "\t\t\t|-----------------------|"<<endl;
            else
                if (i==8)
                    cout << "\t\t\t-------------------------"<<endl;
        }
    }
}

/*Metodo che permette la generazione di Sudoku random. Genera a caso dieci numeri
   che posiziona in (i,j) anch'essi random, controlla se tali numeri possono
   essere inseriti richiamando il metodo checkRules, poi risolve il
   Sudoku (resolveSudoku) ed infine elimina alcuni numeri (putZero).
   Generando a caso gli indici, anche se viene posizionato un solo numero,
   abbiamo 9*81 possibili Sudoku */
void Sudoku::sudokuRand()
{
    srand((unsigned int)time(0));
    for (short k=0;k<10;k++)
    {
  //Numeri ed indici vengono generati nel loro range per rispettare le regole
        short i=rand()%9,j=rand()%9,rnum=1+rand()%9;
        if ( checkRules(rnum,i,j) )
            setNumber(i,j,rnum);
    }
    //Il puntatore this punta all'oggetto per il quale e' stata richiamata la funzione membro
    resolveSudoku(this);
```

```
    putZero();
}

/*Metodo che rimpiazza casualmete con degli zero i numeri della griglia,
  generando vari livelli di difficolta' in modo casuale*/
void Sudoku::putZero()
{
    srand((unsigned int)time(0));
    short rZero=50+rand()%10; //numero di zeri
    while (rZero>0)
    {
        short rind=rand()%81; //indice del valore da azzerare
        if (griglia[rind]!=0)
        {
            griglia[rind]=0;
            rZero--;
        }
    }
}

/*Metodo che realizza la soluzione del Sudoku con la tecnica del Backtracking.
  Analizza la griglia alla ricerca di uno zero, se lo trova significa che il
  Sudoku non e' stato ancora risolto, quindi cerca un qualsiasi numero che
  soddisfi i vincoli descritti in checkRules, inseguito fa una chiamata
  ricorsiva per spostarsi in un altro ramo di decisione, se restituisce un
  valore non NULL aggiungiunge un pezzo alla soluzione, altrimeti resetta il numero
  inserito e torna indietro (Backtrack). Il metodo continua cosi' fino a che non ci
  sono piu' zeri quindi ha trovato la soluzione, se non trova una possibile soluzione
  restituisce un puntatore NULL. Come parametro accetta un puntatore ad un' istanza
  della classe Sudoku e restituisce la soluzione oppure NULL */
Sudoku* Sudoku::resolveSudoku(Sudoku *s)
{
    vector<short>::iterator it;
    //Se esiste un solo elemento uguale a zero il Sudoku non e' risolto
    it=find(s->griglia.begin(),s->griglia.end(),0);
    if (it !=s->griglia.end())
    {
  //In questi cicli ricaviamo gli indici (i,j) indispensabili per i controlli
        for (short i=0;i<9;i++)
            for (short j=0;j<9;j++)
                if (s->griglia[i*9+j] == 0)
                {
        //Ottenuti gli indici, proviamo l'inserimeto di un numero
                    for (short k=1;k<=9;k++)
                    {
                        if (s->checkRules(k,i,j))
                        {
                            setNumber(i,j,k);
                         /*Trovata la pseudo-soluzione, istanziamo l'oggetto app
                           per analizzare un altro possibile percoso*/
                            Sudoku *app=resolveSudoku(s);
                            if (app!=NULL)
                                return app;  //Ritorna un altro pezzo della soluzione
                        }
      /*Viene resettata la modifica fatta inquanto
                          la scelta portava ad una non-soluzione*/
                        setNumber(i,j);
                    }
        /*La strada intrapresa non porta ad una soluzione
                      quindi si torna indietro (Backtrack)*/
                    return NULL;
                }
```

```
  }
 return s; //Nel Sudoku non sono presenti zeri quindi e' stato risolto
}

#endif //Sudoku_H
```

A.3 progressbar.h

progressbar.h

```
#ifndef PROGRESS_BAR_H
#define PROGRESS_BAR_H
#include <iostream>
#include <windows.h>  //Contiene le API di Windows
using namespace std;

void progressBar()
{
    char buffer[256]={0};
    char percent[7]="0.0%%";
    short i, j;
    buffer[0]='[';
    for (i=0;i<40;i++)
    {
        buffer[i + 1]='=';
        j=i%4;
        if (j==0)
            buffer[i+2]='\\';
        else if (j==1)
            buffer[i+2]='|';
        else if (j==2)
            buffer[i+2]='/';
        else
            buffer[i+2]='-';

        for (short k=i+3;k<41;k++)
            buffer[k]=' ';
        buffer[41]=']';
        sprintf(percent,"%3.2f%%",i/40.0*100.0);
        cout<<"\t\t"<<buffer<<" "<<percent<<'\r';
        Sleep(35);      //Win32 API : delay di 55 ms
    }

    sprintf(percent,"%3.2f%%",i/40.0*100.0);
    cout<<"\t\t"<<buffer<<" "<<percent<<'\r';
}

#endif // PROGRESS_BAR_H
```