UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Magistrale in "Scienze e Tecnologie dell'Informazione''

Progettazione ed Analisi di Algoritmi

Algoritmi di branching

DOCENTE:
Prof. Roberto Cordone

ELABORATO DI:
Paolo F. A. Bettini
Matr. 736488

Anno Accademico 2007/2008

Indice

Elenco delle figure

Capitolo 1

Progetto di algoritmi per induzione matematica

Prima di esporre il concetto della progettazione di algoritmi per induzione matematica, è necessario introdurre il concetto di *riduzione*. La *riduzione polinomiale* di un problema P ad un problema P' ($P \preceq_R P'$) è una OTM (Macchina di Turing con Oracolo) che risolve P in tempo polinomiale consultando un oracolo per P' un numero polinomiale di volte. Con *oracolo* si intende un qualsiasi dispositivo (anche ipotetico) che riesca a risolvere il problema 'ridotto'. Ad esempio, si può ridurre il problema dell'ordinamento al problema del confronto fra numeri interi (se so confrontare, so anche ordinare). Una volta eseguita la riduzione, si dice che il problema P' è difficile rispetto a P. Non bisogna lasciarsi ingannare da questa affermazione in quanto il termine 'difficile' deve essere interpretato nel modo seguente: *P' è la parte difficile di P*. Una volta risolto P', è relativamente semplice risolvere P.

1.1 Tecniche di riduzione

Per ridurre un problema P ad un problema Q ci sono diverse tecniche:

$$P \preceq_R Q$$

Restrizione: l'intero insieme di istanze P corrisponde ad un sottoinsieme di istanze di Q (in questo caso la riduzione è più specificamente una *trasformazione* $P \preceq_T Q$).

Sostituzione locale: ogni istanza di P si trasforma in un'istanza di Q sostituendo ogni sottosistema dell'istanza di P appartenente ad una certa famiglia di (ogni nodo, variabile, elemento ...) con un opportuno sottosistema, in modo da costituire un'istanza di Q.

Progetto di componenti: nel caso particolare in cui si voglia ridurre il problema di soddisfacibilità di formule logiche (SAT) ad un problema P si costruiscono istanze di P con una struttura particolare. Si definiscono sottosistemi che 'fanno scelte' e sottosistemi che 'testano proprietà'; quindi, si combinano i sottosistemi in modo da comunicare le scelte fatte dai primi ai secondi, che ne verificano la validità. In questo modo i sottosistemi che fanno scelte corrispondono alle variabili ed i sottosistemi che testano proprietà corrispondono alle formule di una funzione logica in 'Forma Normale Congiuntiva' (CNF). Così qualsiasi istanza di SAT è ridotta a un'istanza di P.

1.2 Progetto per riduzione

Il progetto per riduzione si articola in diverse fasi:

1. Si trova una riduzione R del problema P al problema Q:

$$P \preceq_R Q \qquad (I_P \rightarrow I_Q \; \forall \; I_P \in P)$$

2. Si risolve Q con un algoritmo A_Q:

$$I_Q \rightarrow S_Q$$

3. Si trasforma la soluzione dell'istanza di Q nella soluzione dell'istanza di P:

$$S_Q \rightarrow S_P$$

Se i tre passaggi sono polinomiali, l'algoritmo A_P costituito dalla sequenza delle tre fasi è polinomiale.

1.3 Progetto per induzione

Il problema Q può essere ridotto anche ad un sottoinsieme di se stesso. Ad esempio $SAT \preceq_R$ *3-SAT*. Questa è la base del progetto di algoritmi per induzione. Ogni problema P ha infatti un sottoinsieme $P_0 \subseteq P$ di istanze semplici. In genere non si può ridurre P a P_0, ma spesso si può ridurre $P_1 \subset P$ a P_0:

$$P \not\preceq_R P_0 \text{ ma } \exists P_1 \subset P \backslash P_0 : P_1 \preceq_R P_0$$

e ancora P_2 a P_1:

$$P \not\preceq_R P_1 \text{ ma } \exists P_2 \subset P \backslash (P_0 \cup P_1) : P_2 \preceq_R P_1$$

L'idea è iterare il procedimento fino a ridurre ogni istanza di P al caso base P_0 in un opportuno numero di passi. I passi da seguire durante la progettazione di un algoritmo per induzione sono i seguenti [Man88]:

1. Si decompone il problema P in una successione di sottoproblemi P_i, ciascuno dei quali associato ad un numero naturale.

2. Si determina un algoritmo A_0 per i problemi 'banali', cioè per le istanze di P_0.

3. Si determina una riduzione polinomiale R di ogni istanza di P_i ad una o più istanze I di classe $j < i$.

4. A questo punto l'algoritmo $Algo(I)$ opera come segue:

   - Se $I \in P_0$ allora risolve I con A_0 e restituisce la soluzione S
   - Se $I \notin P_0$
     - Riduce I a $I_1, I_2, \ldots, I_p$ con la riduzione R
     - S_i = Algo(I_i) con $i = 1, \ldots, p$
     - Ricostruisce S di I dalle soluzioni S_i (con $i = 1, \ldots, p$) e la restituisce.

1.3.1 La divisione in sottoproblemi

Si ripartiscono le istanze di P in una successione di sottoinsiemi P_k, come mostrato in Figura 1.1. Al fine di una corretta suddivisione del problema da risolvere ogni istanza di P deve ricadere in uno dei sottoproblemi P_k.

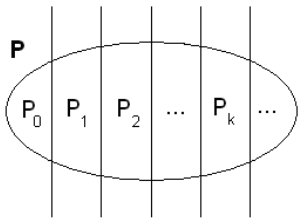

Figura 1.1: Partizione del problema principale in sottoproblemi

- L'unione dei sottoproblemi P_k deve costituire il problema principale P:

$$P = \bigcup_{k=0}^{+\infty} P_k$$

- L'intersezione fra due sottoproblemi diversi deve essere vuota:

$$P_i \cap P_j = \emptyset \quad \text{con } i \neq j$$

1.3.2 La riduzione a sottoproblemi

Ogni istanza di P deve poter essere ridotta, con un opportuno numero di passi, ad un'istanza di P_0 (cioè al caso base). A tal fine deve essere presente una riduzione R che consente di ridurre ogni istanza del problema P_k a una o più istanze del problema P_j di indice strettamente minore:

$$P_k \preceq_R P_j \quad \text{con } j < k$$

Spesso l'indice dei sottoproblemi viene chiamato dimensione (perchè tipicamente cresce al crescere della dimensione dell'istanza). Con dimensione tuttavia non si intende esattamente il numero di bit della codifica, ma un qualsiasi parametro che classifichi le istanze.

1.3.3 La soluzione del caso base

Aspetto fondamentale degli algoritmi progettati per induzione è la presenza di un algoritmo che fornisca la soluzione al caso base del problema (cioè la soluzione di P_0). Visto che la dimensione dell'istanza base è piccola, spesso gli algoritmi che la risolvono sono banali e utilizzano metodi a 'forza bruta'.

1.3.4 I numeri naturali e l'induzione matematica

A questo punto bisogna dimostrare che il procedimento funzioni davvero. Per farlo si sfrutta la definizione stessa dei numeri naturali. L'insieme dei numeri naturali è definito da queste proprietà (assiomi di Peano):

1. $0 \in \mathbb{N}$ (quindi l'insieme dei numeri naturali non è vuoto).
2. La funzione S ('successore'), definisce per ogni numero naturale n uno ed un solo successore n' (con $n' \neq n$):

$$S : \mathbb{N} \to \mathbb{N} \text{ cioè } \forall n \in \mathbb{N} \quad \exists n' = S(n) \in \mathbb{N} : n' \neq n$$

3. 0 non è successore di alcun altro numero: $\nexists n \in \mathbb{N} : 0 = S(n)$.

4. Numeri diversi hanno successori diversi: $n \neq n' \Rightarrow S(n) \neq S(n')$.

5. Principio di induzione matematica:

$$\begin{cases} K \subseteq \mathbb{N} \\ 0 \in K \\ n \in K \Rightarrow S(n) \in K \end{cases} \quad \Longrightarrow \quad K = \mathbb{N}$$

   Se un insieme di numeri naturali contiene 0 ed anche il successore di ognuno dei suoi elementi, tale insieme coincide con $\mathbb{N}$.

1.3.5 Applicazione agli algoritmi

Sia K il sottoinsieme dei numeri naturali tali che l'algoritmo A risolve tutte le istanze P_k, $\forall k \in K$. Si supponga che:

- A risolve tutte le istanze in P_0, cioè $0 \in K$.
- Se A risolve tutte le istanze in P_k, allora risolve tutte le istanze in P_{k+1}, cioè $k \in K \Rightarrow k + 1 \in K$

Allora $K = \mathbb{N}$, cioè A risolve tutte le istanze in P_k $\forall k \in \mathbb{N}$: A risolve P.

1.4 Esempio: l'ordinamento

Dato un vettore P, il problema dell'ordinamento consiste nel restituirne uno che contenga gli stessi elementi, permutati in modo da rispettare una opportuna relazione di ordine totale. Ordinare tutte le istanze del problema è un compito difficile, ma se si considera il sottoinsieme dei vettori vuoti, cioè con $k = 0$ elementi, l'ordinamento diviene banale: basta non fare nulla. A tal proposito si dividono le istanze di P in classi, ognuna delle quali contiene tutti i problemi di n elementi, dove n varia nell'insieme dei numeri naturali.

- P_0: è banale ordinare un vettore di 0 elementi.
- P_1: se ho un vettore di 1 elemento, posso toglierne uno, ordinare il vettore residuo di 0 elementi (istanza di P_0) e reinserire l'elemento in modo ordinato.
- P_2: se ho un vettore di 2 elementi, posso toglierne uno, ordinare il vettore residuo di 1 elemento (istanza di P_1) e reinserire l'elemento in modo ordinato.
- ...
- P_n: se ho un vettore di n elementi, posso toglierne uno, ordinare il vettore residuo di $n-1$ elementi (istanza di P_{n-1}) e reinserire l'elemento in modo ordinato.

L'algoritmo risultante è noto come *Insertion Sort*. Se invece di togliere un elemento qualsiasi si toglie il massimo (o il minimo) e lo si reinserisce quindi in fondo (o in cima) al vettore, l'algoritmo risultante è noto come *Selection Sort*.

Capitolo 2

Algoritmi di Branching

Gli algoritmi di branching per la risoluzione di problemi di decisione, ricerca, ottimizzazione, conteggio o enumerazione, costituiscono una strategia di esplorazione dello spazio delle soluzioni ammissibili, che vengono enumerate in parte esplicitamente ed in parte implicitamente [Ver97]. Come si può intuire, il metodo che si dovrà applicare risulta tanto più efficiente quanto maggiore è il numero di soluzioni esaminate in modo implicito. L'idea principale su cui si basano gli algoritmi di branching comporta una partizione della regione ammissibile in sottoregioni, che a loro volta possono essere suddivise in sottoregioni ancora più piccole, con l'obiettivo di ricavare problemi più semplici da risolvere rispetto al problema originale. In generale possono esistere numerosi criteri per sviluppare le partizioni della regione ammissibile. Di conseguenza, esistono diversi schemi alternativi per realizzare algoritmi di branching. Indichiamo come $S_1, S_2, \ldots, S_R$ le sottoregioni che rappresentano la partizione della regione ammissibile S del problema I, ovvero:

$$S = \bigcup_{k=1}^{R} S_k \qquad \text{e} \qquad S_i \bigcap S_j = \emptyset \quad \forall\, i \neq j$$

Si indicano inoltre come $I_1, I_2, \ldots, I_R$ con $k = 1, 2, \ldots, R$ i corrispondenti sottoproblemi generati nella fase corrente di branching. Questo processo di ramificazione (*branching*) si può rappresentare mediante un albero de-

cisionale (*branch decision tree*), mostrato in Figura 2.1, dove ogni nodo rappresenta un sottoproblema mentre ogni arco la relazione di discendenza da problema padre a sottoproblema figlio.

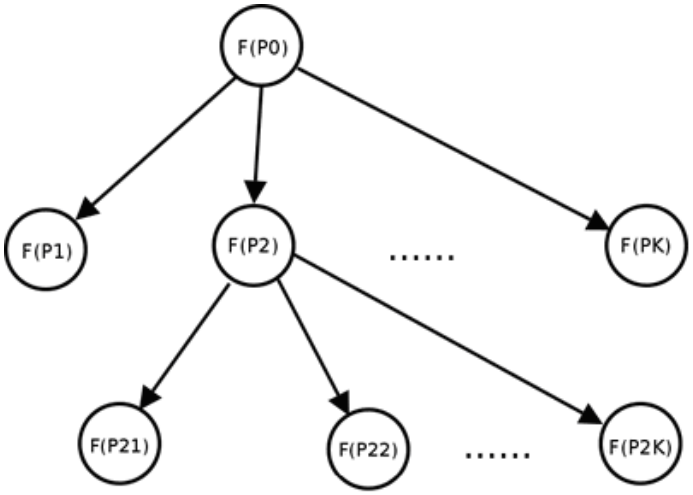

Figura 2.1: Albero di decisione per gli algoritmi di branching

Risolvere il problema P_0 è quindi equivalente a risolvere la totalità dei sottoproblemi P generati.

2.1 La risoluzione dei sottoproblemi

Durante l'esecuzione degli algoritmi di branching si mira ad ottenere, per ciascuno dei sottoproblemi I_k, uno dei seguenti risultati tra loro alternativi:

1. Determinare la soluzione del sottoproblema.

2. Dimostrare che il sottoproblema non ha soluzioni ammissibili (cioè la regione S_k è vuota) e quindi si può ignorare.

3. Dimostrare, senza determinarlo in modo esplicito, che il sottoproblema contiene soluzioni irrilevanti ai fini della determinazione della soluzione complessiva, e quindi si può ignorare.

Se non si riesce a ricadere in questi tre casi si deve ulteriormente suddividere il problema in altri sottoproblemi. Gli algoritmi di branching, sono detti anche di enumerazione implicita perché si comportano esattamente come un algoritmo di enumerazione, cioè 'provano' tutte le soluzioni possibili fino a trovare quella corretta, ma ne scartano interi sottoinsiemi dimostrando a priori la loro non correttezza [RT02].

2.2 Schema di branching

Per ripartire l'insieme delle soluzioni S in sottoinsiemi disgiunti S_k, si adottano schemi di branching. I più tipici sono i due seguenti:

1. Si sceglie un elemento dell'istanza, e si producono due sottoproblemi, imponendo che l'elemento scelto appartenga o no alla soluzione.

2. Si sceglie un insieme di k elementi dell'istanza e si producono $k+1$ sottoproblemi imponendo che:

   - Il primo elemento non sia nella soluzione.
   - Il primo elemento sia nella soluzione ed il secondo no.
   - ...
   - I primi $k-1$ elementi siano nella soluzione ed il k-esimo no.
   - Tutti i k elementi stiano nella soluzione.

Considerando il problema del commesso viaggiatore (TSP), un esempio di branching del primo tipo consiste nello scegliere un arco e disporne la presenza o l'assenza nella soluzione. Un esempio di branching del secondo tipo consiste nello scegliere un sottociclo (ciclo contenente $k < n$ archi) e vietare il primo arco, imporre il primo arco e vietare il secondo, ..., imporre $k-1$ archi e vietare il k-esimo. L'ultimo problema (imporre tutti gli archi) si può ignorare perchè non contiene soluzione ammissibile.

2.3 La ricomposizione dei problemi

In base alla tipologia del problema che si deve risolvere, ci sono diverse metodologie per ricomporre le soluzioni dei sottoproblemi generati in fase di branching nella soluzione del problema originario.

Problemi di decisione: una volta che si trova una soluzione valida, ogni altro sottoproblema diventa irrilevante, quindi l'algoritmo può interrompersi, ignorando eventuali altre soluzioni ammissibili.

Problemi di ricerca: si prende la prima soluzione valida e ammissibile.

Problemi di ottimizzazione: si conserva il valore z^{best} della miglior soluzione trovata via via, e ogni problema che non contiene soluzioni migliori diventa irrilevante. Il modo più frequente per ridurre lo spazio dei sottoproblemi da esplorare, eliminandone alcuni irrilevanti, è l'uso di un *'bound'* (limite). Dato un sottoproblema P_i, è possibile determinare un *lower bound (LB)* della soluzione che gode della proprietà: $LB(P_i) \leq z^*(P_i)$ [Ver97, RT02]. Se si verifica che $LB(P_i) \geq z^{best}$ (cioè il se *lower bound* del sottoproblema P_i è maggiore della miglior soluzione corretta trovata), si può escludere quel sottoproblema visto che la miglior soluzione che è possibile ottenere non può essere migliore della miglior soluzione nota. Per ottenere un *lower bound* di $LB(P_i)$ si deve trovare un rilassamento del problema $R(P_i)$ cioè un altro sottoproblema tale che:

$$z_r(y) \leq z(y) \quad \forall y \in F(P_i)$$

$$F(P_i) \subseteq F(R(P_i))$$

Dove:

- z e z_r sono le funzioni obiettivo di P_i e $R(P_i)$.
- $F(P_i)$ e $F(R(P_i))$ sono gli insiemi delle soluzioni ammissibili di P_i e $R(P_i)$

In altre parole, il problema rilassato ha un insieme di soluzioni ammissibili più ampio e/o una funzione inferiore nelle soluzioni ammissibili per entrambi. Si sceglie il problema rilassato in modo che sia risolvibile più semplicemente rispetto al problema originale, quindi si possa trovarne la soluzione ottima che rappresenta il *lower bound* del problema originale.

Problemi di conteggio: si somma il numero di soluzioni trovate per i sottoproblemi e si fornisce in output tale valore.

Problemi di enumerazione: ogni sottoproblema restituisce la lista delle sue soluzioni ammissibili. L'output dell'algoritmo sarà il concatenamento di tali liste, ovvero l'unione degli insiemi di soluzioni dei sottoproblemi.

2.4 Gli elementi degli algoritmi di Branching

Riassumendo, un algoritmo di branching è caratterizzato dai seguenti elementi:

Regola di Branching. Criterio per generare i problemi figli dato il problema padre.

Ordine di visita. Regola per scegliere il sottoproblema aperto da esplorare per primo.

Test di ammissibilità. Metodo che indica se un problema può avere soluzioni o meno.

Test di rilevanza. Metodo che indica se un problema va tenuto aperto o eliminato.

Procedura di Bound. Nei problemi di ottimizzazione la 'migliore' regola per calcolare *LB* è quella che dà il '*lower bound*' più stretto, cioè più vicino possibile al costo della soluzione ottima, ma che sia velocemente calcolabile.

Capitolo 3

Una libreria per algoritmi di branching

In questo capitolo verranno esposte le principali caratteristiche dei file che compongono la libreria per la creazione di algoritmi di branching per un problema generico.

3.1 File main.c

Il main si sviluppa seguendo queste fasi (vedi Codice 3.1):

1. Lettura delle istruzioni da riga di comando.
2. Caricamento dei dati.
3. Allocazione della struttura per la soluzione corrente.
4. Creazione di un albero di branching vuoto.
5. Allocazione del problema radice e sua elaborazione:
    - Costruzione del problema a partire dai dati.
    - Verifica di ammissibilità, eventuale calcolo di una soluzione euristica (con aggiornamento della miglior soluzione euristica nota), di un bound, ecc ...

- Determinazione degli elementi su cui operare per un eventuale processo di branching.

6. Inserimento del nodo radice nell'albero (lista dei problemi aperti).
7. Ciclo di analisi dei sottoproblemi.
8. Stampa dei risultati.

```
ReadInstructions(argc,argv,DataFile);

LoadData(DataFile,&PD);

CreateCurrentSolution(&PD,&CS);

BT = CreateBtree();
VisitStrategy = DEPTH_FIRST;

pta = CreateBnode(1,0,1,&PD);

ProcessBnode(pta,&PD,&CS);

InsertBnode(pta,BT,VisitStrategy);

while(...){
...
}

printSolution(&CS);
```

Codice 3.1: Le otto fasi del main

Il *main* si apre con la dichiarazione delle variabili necessarie all'esecuzione del processo di *branching*: i dati del problema *PD*, la sua soluzione corrente *CS* e l'albero di branching *BT* (che in realtà sarà una lista circolare, bidirezionale, con sentinella).

Vengono poi dichiarate delle variabili per la gestione del ciclo di analisi del problema e generazione dei sottoproblemi (vedi Codice 3.2).

```
ProblemData PD;
CurrentSolution CS;
Btree BT;
int VisitStrategy;
long ExaminedNodes;
long GeneratedNodes;

int id, f;
Bnode pos pta, temp;
```

Codice 3.2: Le variabili dichiarate nel main

3.1.1 Il ciclo di analisi dei sottoproblemi

All'interno del ciclo (vedi Codice 3.3) si eseguono le seguenti operazioni:

- Estrazione del prossimo sottoproblema da elaborare.
- Verifica della rilevanza del sottoproblema attraverso la funzione *UsefulBnode(...)*. In caso negativo si elimina il problema.
- In caso positivo, si ha la generazione dei sottoproblemi figli in base alla regola di branching conservata dal padre. Per ogni figlio:
  - Si genera un problema figlio vuoto.
  - Si copia nel problema figlio il problema padre e si modificano le informazioni del figlio, in modo da creare le informazioni corrette. Per far ciò si utilizza la funzione *DeriveBnode(...)* tenendo conto del numero d'ordine del figlio stesso e della regola di branching conservata nel padre.
  - Si elabora il problema figlio con la funzione *ProcessBnode(...)*, tenendo presenti i vincoli di branching ereditati dal padre: se ne verifica l'ammissibilità, eventualmente si generano bound, soluzioni euristiche, ... ed eventualmente si aggiorna la soluzione corrente *CS*.

- Si verifica la rilevanza del problema figlio. In caso negativo, il problema figlio viene chiuso.
- In caso positivo, viene inserito nell'albero di decisione come problema aperto, in una posizione, stabilita dalla strategia di visita indicata.

```
while (
        !listavuota(BT) &&
        //Eventuali altre condizioni di uscita
      )
{
  pta = ExtractBnode(BT);
  if (UsefulBnode(pta,&CS))
  {
    for (f = 1; f <= pta->branch.numfigli; f++)
    {
      id++;
      temp = CreateBnode(id,pta->livello+1,f,&PD);
      DeriveBnode(temp,pta,f);
      ProcessBnode(temp,&PD,&CS);
      if (!UsefulBnode(temp,&CS)) DestroyBnode(&temp);
      else
      {
        InsertBnode(temp,BT,VisitStrategy);
        GeneratedNodes++;
      }
    }
  }
  DestroyBnode(&pta);
  ExaminedNodes++;
}
```

Codice 3.3: Il ciclo di analisi dei sottoproblemi

Come si evince dal Codice 3.3, il ciclo finisce quando non ci sono più problemi aperti. Si possono tuttavia imporre anche condizioni aggiuntive come ad esempio:

- Numero massimo di nodi processabili.
- Tempo di calcolo massimo.

- Valore limite di UB, LB o Gap limite fra UB e LB (nei problemi di ottimizzazione).
- Numero limite di soluzioni trovate (nei problemi di conteggio ed enumerazione).

3.1.2 Libreria defs.h

La libreria *defs.h* contiene, all'interno delle due strutture *ProblemData* e *CurrentSolution*, la definizione dei dati del problema da risolvere e della soluzione corrente (vedi Codice 3.4). Tali definizioni sono lasciate in bianco perché è compito dell'utente fornirle.

```
#ifndef __data_h
#define __data_h

// Dati del problema (informazioni globali)
typedef struct _ProblemData ProblemData;
struct _ProblemData
{
  //???
};

// Miglior soluzione conosciuta (informazioni globali)
typedef struct _CurrentSolution CurrentSolution
struct _CurrentSolution
{
  //???
};
#endif
```

Codice 3.4: Listato della libreria defs.h

Queste strutture vengono usate dalle seguenti funzioni del main:

LoadData(char *DataFile, ProblemData *pPD)
Questa funzione (Codice 3.5) si occupa di caricare i dati dal file di input e di allocare le strutture per descrivere la soluzione.

```
void LoadData (char *DataFile, ProblemData *pPD);
```

Codice 3.5: Prototipo della funzione *LoadData()* del main

DestroyData(ProblemData *pPD)
All'interno di questa funzione (Codice 3.6) vengono deallocate le eventuali strutture dinamiche contenenti i dati del problema.

```
void DestroyData(ProblemData *pPD);
```

Codice 3.6: Prototipo della funzione *DestroyData()* del main

CreateCurrentSolution(ProblemData *pPD, CurrentSolution *pCS)
Questa funzione (Codice 3.7) si occupa di allocare ed inizializzare le strutture dati al fine di rappresentare la soluzione attualmente nota. Occorre fornirle i dati perchè tipicamente le strutture della soluzione hanno dimensione da loro determinata.

```
void CreateCurrentSolution (ProblemData *pPD, CurrentSolution *pCS);
```

Codice 3.7: Prototipo della funzione *CreateBestSolution()* del main

DestroyCurrentSolution(CurrentSolution *pCS)
All'interno di questa funzione (Codice 3.8) vengono deallocate le eventuali strutture dinamiche contenenti i dati della soluzione corrente.

```
void DestroyCurrentSolution(CurrentSolution *pCS);
```

Codice 3.8: Prototipo della funzione *DestroyCurrentSolution()* del main

3.2 Libreria bnodelist.h

La struttura *Bnode* rappresenta un sottoproblema. Si compone di quattro parti:

1. Un insieme di informazioni ausiliarie utili solo in fase di debug:
    - *id*: identificativo univoco del sottoproblema nell'albero di *branching*.
    - *livello*: indica a quale livello il sottoproblema compare nell'albero di *branching*.

- *id_figlio*: indica la posizione relativa che il sottoproblema occupa fra i figli del problema padre. Tale valore può andare da 1 a *numfigli* che è un'informazione contenuta nella struttura *branching* (vedi punto 3), ed è 0 per la radice.

2. L'informazione locale prodotta durante l'analisi del nodo (contenuta nella struttura *Bnodeinfo*).

3. L'informazione necessaria ad un'eventuale scomposizione del problema in sottoproblemi figli (struttura *branching*).

4. Puntatori al *Bnode* successivo e al *Bnode* precedente nella lista dei problemi. L'albero di decisione viene infatti simulato con una struttura di tipo lista bidirezionale, circolare e con sentinella contenente i soli sottoproblemi aperti.

La struttura Bnodeinfo

La struttura *Bnodeinfo* (Codice 3.9) dovrà contenere le informazioni locali al sottoproblema in analisi. È lasciata in bianco perchè è compito dell'utente definirla.

```
typedef struct _Bnodeinfo Bnodeinfo;
struct _Bnodeinfo
{
  //???
};
```

Codice 3.9: La struttura *Bnodeinfo*

Tali informazioni saranno:

1. I vincoli di branching ereditati dal padre.

2. Lo *status* del problema, definito da una delle costanti riportate nel Codice 3.10 (inizialmente il problema è sempre aperto):

```
#define OPEN_INSTANCE        -1        //problema aperto
#define YES_INSTANCE          1        //problema risolto
#define NO_INSTANCE           0        //privo di soluzione
```

Codice 3.10: I possibili stati di un problema

3. Eventualmente un *lower bound*, una soluzione euristica ed un *upper bound* locali al nodo in esame.

4. Statistiche varie sul nodo (secondo la tipologia di problema che si vuole risolvere).

La struttura Branching

La struttura *branching* (Codice 3.11), dovrà contenere le informazioni che saranno utili in fase di divisione del problema da risolvere in sottoproblemi di dimensione minore. Anch'essa è lasciata in bianco perchè è compito dell'utente definirla. In tale struttura saranno contenuti:

1. Il numero dei sottoproblemi da generare (*numfigli*).

2. Le variabili che indicano lo schema di branching.

```
typedef struct _branching branching;
struct _branching
{
  int numfigli;
  //???
};
```

Codice 3.11: La struttura *branching*

La struttura *Bnode* viene creata e distrutta dalle seguenti funzioni che vanno definite dall'utente:

CreateBnode(long id, long livello, long id_figlio, ProblemData *pPD)

Alloca le eventuali strutture dati dinamiche per generare un *Bnode*. Richiede i dati del problema perché la dimensione di tali strutture dati tipicamente dipende da loro (vedi Codice 3.12).

```
Bnode* CreateBnode (long id, long livello, long id_figlio, ProblemData *pPD);
```

Codice 3.12: Funzione di creazione dei *Bnode*

DestroyBnode(Bnode **pNode)

Libera la memoria da un *Bnode* che non sarà più utilizzato.

3.2.1 Esplorazione della lista dei sottoproblemi

All'inizio del file *main.c* sono presenti le definizioni delle costanti che specificano la strategia di esplorazione dell'albero (lista) di branching che si andrà formando, ed il massimo numero di nodi generabili (vedi Codice 3.13).

```
#define DEPTH_FIRST              1
#define BREADTH_FIRST            2
#define BEST_FIRST               3
#define MAX_BRANCHING_NODES   10000
```

Codice 3.13: Le costanti all'interno del main

I tre possibili metodi di esplorazione della lista dei sottoproblemi sono i seguenti [Rig06]:

Depth-first-search (esplorazione in profondità): si esplora un ramo ancor prima di generare quelli successivi. Tale strategia si presta a realizzazioni ricorsive e non richiede di memorizzare l'intero albero. Nella libreria va però preferita un'implementazione iterativa con gestione esplicita dell'albero per poter usare anche le altre strategie.

Breadth-first-search (esplorazione in ampiezza): si esplorano i nodi appartenenti allo stesso livello dell'albero, prima di generare eventuali nuovi figli.

Best-first-search: si esplora per primo il nodo contenente il problema aperto più 'promettente' che, nel caso di problemi di ottimizzazione, è quello con il bound migliore.

Per realizzare queste strategie, avendo simulato l'albero di branching con una lista, occorre definire opportunamente la procedura di inserimento dei problemi nella lista: *InsertBnode(...)* (vedi Codice 3.14).

```
void InsertBnode (Bnode *pta, Btree BT, int VisitStrategy);
```

Codice 3.14: La funzione *InsertBnode()* del main

- Se la strategia di visita è la *'Depth-first-search'*, il nodo corrente viene inserito in cima alla lista.

- Se la strategia di visita è la *'Breadth-first-search'*, il nodo corrente viene inserito in fondo alla lista.

- Se la strategia di visità è la *'Best-first-search'*, l'inserimento è più raffinato: si scorre la lista dei problemi aperti (ordinata per bound) fino a trovare il primo problema peggiore del nodo corrente. A questo punto si inserisce il nodo che si sta trattando prima di esso.

L'estrazione dei *Bnode* avviene sempre dalla cima con la funzione *E*xtractBnode(...) (vedi Codice 3.15).

```
Bnode *ExtractBnode (Btree BT);
```

Codice 3.15: La funzione *ExtractBnode()* del main

3.2.2 Valutazione e generazione dei sottoproblemi

Il ciclo di analisi dei sottoproblemi comprende le seguenti funzioni, che vanno definite dall'utente:

void DeriveBnode (Bnode *son, Bnode *father, int f)
Copia nel problema corrente *son* i vincoli di branching propri del problema padre *father* e conservati nel campo *Bnodeinfo*. Quindi, li modifica in modo da trasformare il problema nell'f-esimo problema figlio, sulla base della regola di branching contenuta nel campo *branching* del nodo padre e dell'indice f.

boolean UsefulBnode (Bnode *temp, CurrentSolution *pCS)
È la verifica di rilevanza del sottoproblema. Il funzionamento di questa funzione dipende dalla tipologia dei problemi che si devono risolvere:

- PROBLEMI DI DECISIONE: ogni sottoproblema è rilevante (*UsefulBnode()* restituisce *TRUE*) fino a quando si trova una soluzione ammissibile. Quando si trova una soluzione, il campo *status* di *CS* (*CurrentSolution*) diventa *YES_INSTANCE* e da quel momento in poi, *UsefulBnode()* restituisce sempre il valore *FALSE*. Tutti i sottoproblemi diventano così irrilevanti.

- PROBLEMI DI RICERCA: nel campo *status* di *CS* si scrive se si è trovata o no una soluzione. Se tale soluzione è presente, *UsefulBnode()* restituirà *TRUE*, altrimenti sarà *FALSE*. La differenza rispetto ai problemi di decisione è che in un campo opportuno di *CS* c'è la soluzione.

- PROBLEMI DI OTTIMIZZAZIONE: in *CS* c'è il valore *UB* della miglior soluzione nota. *UsefulBnode()* verifica se il *LB* del nodo corrente è minore dell'*UB*. In tal caso il sottoproblema è rilevante e *UsefulBnode()* restituisce *TRUE*, altrimenti sarà restituito *FALSE*. Si restituisce *FALSE* anche se il problema non ha la soluzione (lo si verifica guardando il valore del campo *status* di *CS*).

- PROBLEMI DI CONTEGGIO E DI ENUMERAZIONE: in questo caso *UsefulBnode()* sarà sempre a *TRUE* quando il campo *status* del nodo corrente indica che il problema ha effettivamente almeno una soluzione ammissibile e che va ulteriormente indagato somponendolo in sottoprobelmi.

void ProcessBnode(Bnode *pta, ProblemData *pPD, CurrentSolution *pCS)
La funzione utilizzata per la manipolazione del problema corrente è la *ProcessBnode(...)* (vedi Codice 3.16).

```
void ProcessBnode (Bnode *pta, ProblemData *pPD, CurrentSolution *pCS);
```

Codice 3.16: La funzione *ProcessBnode()* del main

Questa funzione:

- Calcola le informazioni locali del problema (status, bound, ammissibilita', fissaggi, ...).
- Determina le informazioni di branching (numero dei figli e come generarli).
- Se è il caso, aggiorna le informazioni globali (soluzione corrente). Per esempio, nei problemi di ottimizzazione, se la funzone *ProcessBnode(...)* trova una soluzione euristica migliore di quella corrente, la sostituisce a quest'ultima.

Capitolo 4

Un esempio: il Sudoku

In questo capitolo verrà esposto un esempio sviluppato con la libreria per gli algoritmi di branching descritta nel capitolo precedente. Si applicherà questa metodologia di progettazione per risolvere il problema del Sudoku. Il Sudoku si gioca su una griglia di 9×9 celle, divisa in altre griglie di 3×3 celle dette 'regioni'(Figura 4.1). Si inizia con alcune delle celle della griglia già contenenti alcuni numeri (inizializzazione del problema). L'obiettivo del Sudoku è riempire le celle vuote con numeri tra l'1 ed il 9 (un solo numero per cella) in base a queste semplici regole:

- Il numero può apparire solo una volta all'interno della medesima riga.
- Il numero può apparire solo una volta all'interno della medesima colonna.
- Il numero può apparire solo una volta all'interno di una medesima regione.

In definitiva, un numero dovrebbe apparire solo una volta in ogni riga, colonna e regione.

4.1 Diverse verisioni

Il problema del Sudoku, può assumere diverse versioni:

- **Problema decisionale.** Si deve decidere se c'è o meno una soluzione.
- **Problema di ricerca.** Si deve ricercare una soluzione valida.
- **Problema di conteggio.** Si calcola il numero delle soluzioni ammissibili.
- **Problema di enumerazione.** Si calcolano tutte le soluzioni ammissibili.
- **Problema di ottimizzazione.** Si deve trovare una soluzione ammissibile che ottimizzi una determinata funzione obiettivo.

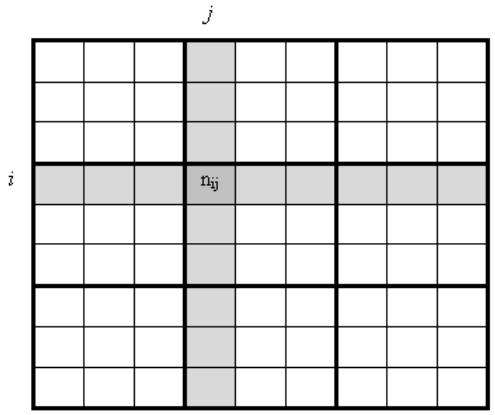

Figura 4.1: La scacchiera del Sudoku

Ad esempio, supporremo che si voglia minimizzare il peso del numero inserito in base alla posizione che occupa all'interno della scacchiera del Sudoku:

$$\text{MIN} \sum_{i=1}^{9} \sum_{j=1}^{9} (i+j) \cdot n_{ij}$$

dove n_{ij} è il numero inserito nella casella di coordinate (i, j).

4.2 Progetto per induzione

Si consideri per primo il problema di decisione. Data una scacchiera con alcune celle inizializzate, si vuole decidere se esiste una soluzione completa che rispetti le inizializzazioni e tutte le regole del gioco. Progettiamo per induzione un algoritmo segondo i passaggi seguenti. Lo pseudocodice risultante è mostrato in Figura 4.2.

1. *Dimensione dell'istanza (i)*. Con dimensione indichiamo il numero di caselle non assegnate, quindi quanti numeri devono ancora essere inseriti all'interno della scacchiera.

2. *Algoritmo per il caso base (A_0)*: se il problema P_0 è facile, cioè se non ci sono più caselle vuote a cui assegnare un numero, si deve semplicemente verificare il rispetto delle regole e rispondere di SI o di NO di conseguenza. L'algoritmo A_0, in sostanza, è un 'verificatore' di Sudoku in quanto si occupa solamente, data una scacchiera completa, di verificare che un numero non si ripeta mai in una riga, in una colonna o in una matrice 3×3 all'interno della scacchiera.

3. *Riduzione polinomiale (R)*: il processo di branching, cioè di creazione dei sottoproblemi da risolvere, si esegue fissando il valore di una casella in tutti i modi leciti nel rispetto delle regole. Ciò riduce di 1 il numero di caselle non assegnate; vengono prodotte istanze (al massimo 9) del problema di classe inferiore (cioè di classe $i - 1$). Il processo di recupero della soluzione può essere implementato con un OR logico fra le soluzioni dei sottoproblemi. Infatti un problema ha soluzione se e solo se almeno uno dei sottoproblemi così ottenuti ha soluzione. In questo modo, alla prima soluzione ammissibile si può interrompere il processo di ricostruzione.

$RisolviSudoku(I)$

- $i := CaselleNonAssegnate(I);$
- if $i = 0$
  - then $S_0 = VerificaSudoku_0();$
    return S_0;
  - else $(I_1, \ldots, I_{NF(I)}) = RiduciProblema(I);$

    for (i=1 to $NF(I)$) do
    $\longrightarrow S_i = RisolviSudoku(I_i);$

    $S = RicostruisciSoluzione(S_1, \ldots, S_{NF(I)})$
    Return S;

Figura 4.2: Pseudocidice dell'algoritmo di risoluzione del Sudoku

4.3 Implementazione dell'algoritmo decisionale

Sfruttando la libreria per algoritmi di branching esposta nel capitolo precedente, si procede alla realizzazione di un algoritmo per risolvere il problema del Sudoku in forma di decisione attraverso diverse fasi.

4.3.1 Strutture per i dati

Per prima cosa occorre scegliere la rappresentazione dei dati e delle soluzioni. È naturale rappresentare la scacchiera del Sudoku con una matrice di caratteri. Viene quindi definito un tipo di dato *Board* costituito da una matrice 9×9 di *char*. Tale struttura è definita nel file *defs.h* ed è una matrice di caratteri in cui inserire i numeri già fissati dal file di input. La matrice è statica perchè le sue dimensioni sono date una volta per tutte. In realtà le righe conterranno 11 caratteri per consentire la presenza di un invio a capo ($'\backslash n'$) ed un terminatore di stringa ($'\backslash 0'$) in modo da semplificare le operazioni di lettura e scrittura.

La struttura *ProblemData* contiene quindi solo un campo di tipo *Board*. Bisogna poi definire le seguenti funzioni:

void LoadData (char *DataFile, ProblemData *pPD)
Apre il file da cui leggere i dati del problema. Leggendo riga per riga da questo file, riempie la struttura *Board* di *PD* con le caselle assegnate in partenza del problema da risolvere (Codice 4.1).

```
void LoadData (char *DataFile, ProblemData *pPD){
  FILE *fDataFile;
  char Riga[RIGA];
  int i, j;

  fDataFile = fopen(DataFile,"r");
    if (fDataFile == NULL){
    printf("File %s does not exist!\n",DataFile);
    exit(EXIT_OPENFILE);
    }

    for (i = 0; i < LATO; i++){
      fgets(Riga,RIGA,fDataFile);
        for (j = 0; j < LATO; j++)
          pPD->B[i][j] = Riga[j];
        pPD->B[i][LATO] = '\n';
        pPD->B[i][LATO+1] = '\0';
    }
  fclose(fDataFile);
}
```

Codice 4.1: Il codice della funzione *LoadData()*

void DestroyData (ProblemData *pPD)
Dealloca le strutture dinamiche eventualmente necessarie per rappresentare il problema. In questo caso la funzione rimane vuota in quanto la matrice viene allocata staticamente (codice 4.2).

```
void DestroyData (Board B){
  //Eventuale deallocazione di strutture dinamiche
}
```

Codice 4.2: Il codice della funzione *DestroyData()*

4.3.2 Strutture per le soluzioni

Bisogna inoltre definire una struttura per rappresentare le soluzioni. La struttura *CurrentSolution* contiene anch'essa un campo di tipo *Board* e un campo *status* (in realtà nei problemi di decisione è sufficiente il campo *status*, il campo *Board* è utilizzato per i problemi di ricerca), che indica se si è trovata o no una soluzione ammissibile che conterrà i fissaggi della soluzione corrente. Questa struttura, durante l'algoritmo verrà aggiornata man mano che si troveranno i nuovi numeri da inserire. In questo caso si hanno tre funzioni:

void CreateCurrentSolution(CurrentSolution *pCS)
Inizializza la scacchiera a 'vuoto' e lo *status* del problema al valore della costante $OPEN\_INSTANCE$.

```
void CreateCurrentSolution (ProblemData *pPD, CurrentSolution *pCS);
```

Codice 4.3: Prototipo della funzione *CreateCurrentSolution()* del main

void DestroyCurrentSolution (CurrentSolution *pCS)
Dealloca le strutture dinamiche eventualmente necessarie per rappresentare la soluzione corrente. In questo caso, la funzione è vuota in quanto, come nel caso precedente, il tipo di dato contenuto nella struttura è stato allocato staticamente.

```
void DestroyCurrentSolution(CurrentSolution *pCS);
```

Codice 4.4: Prototipo della funzione *DestroyCurrentSolution()* del main

void PrintSolution (CurrentSolution *pCS)
Questa funzione si occupa semplicemente di stampare sullo standard output *'SI'* se c'è soluzione *'NO'* se non c'è.

```
void PrintSolution (CurrentSolution *pCS);
```

Codice 4.5: Prototipo della funzione *PrintSolution()* del main

4.3.3 Strutture per i sottoproblemi

Si definiscono poi le strutture per gestire correttamente il processo di branching che si attua durante lo svolgimento dell'algoritmo. In particolare, la struttura *Bnodeinfo* contiene la soluzione del sottoproblema corrente, il suo stato ed i vincoli di branching ereditati dal padre, mentre la struttura *Branching* contiene l'informazione necessaria a generare correttamente i figli a partire dal sottoproblema corrente.

Struttura *Bnodeinfo*

All'interno di questa struttura sono presenti le seguenti informazioni (vedi Codice 4.6):

- Numero di caselle della scacchiera ancora vuote (*n_empty*).
- L'indicazione dell'ultima casella fissata, per facilitare l'elaborazione del nodo (di coordinate *last_i* e *last_j*).
- La scacchiera con i fissaggi correnti e le caselle vuote (*B*).
- Lo stato della scacchiera corrente (*status*):
  - Aperta ($OPEN\_ISTANCE$)
  - Risolta correttamente ($YES\_ISTANCE$)
  - Inammissibile ($NO\_ISTANCE$)

```
typedef struct{
  int n_empty;
  int last_i, last_j;
  Board B;
  int status;
}Bnodeinfo;
```

Codice 4.6: La struttura *Bnodeinfo*

Struttura *Branching*

All'interno di questa struttura (vedi Codice 4.7) vengono indicate le informazioni necessarie a generare figli. Poiché tale generazione fissa una casella a tutti i valori che può ancora assumere, l'informazione richiesta è data dalle coordinate (*bi*, *bj*) della casella su cui si fa branching (cioè in cui si inseriscono i valori ammissibili).

```
typedef struct{
  int bi, bj;
}branching;
```

Codice 4.7: La struttura *Branching*

Dichiarate le strutture dati per la gestione del nodo di branching, occorre definire le funzioni per crearlo e distruggerlo.

void CreateBnode(long id, long livello, long id_figlio, ProblemData *pPD)
All'interno di questa funzione (vedi Codice 4.8), si svolgono le seguenti operazioni:

- Si inizializzano le informazioni di debug del nodo (*id*, $livello$, id_figlio), con valori passati dal chiamante.
- Si inizializzano le informazioni locali del problema ed eventualmente si allocano le strutture necessarie (in questo caso no perchè sono statiche):
  - Si marca il problema come $OPEN\_ISTANCE$
  - Si indica il numero delle caselle vuote (calcolandolo mediante la copia della *Board B* da *ProblemData PD* ed il conteggo delle caselle vuote presenti al suo interno) e l'identificativo dell'ultima casella fissata (nessuna, per convenzione $(-1,-1)$).
- Si azzerano le informazioni di branching: le coordinate (*bi*, *bj*) vengono poste entrambe uguali a -1).

- Si pongono a *NULL* i puntatori ai nodi precedente e successivo (in quanto il nodo non fa ancora parte dell'albero).

```
Bnode* CreateBnode (long id, long livello, long id_figlio, ProblemData *pPD){
  int i, j;

  Bnode *pta = (Bnode *) malloc(sizeof(Bnode));
  pta->id = id;
  pta->livello = livello;
  pta->id_figlio = id_figlio;

  // Azzera le informazioni di branching
  pta->branch.numfigli = 0;
  pta->branch.bi = -1;  pta->branch.bj = -1;

  // Inizializza le informazioni locali del problema
  for (i = 0; i < LATO; i++)
      strcpy(pta->BNinfo.B[i],pPD->B[i]);

  pta->BNinfo.status = OPEN_INSTANCE;
  pta->BNinfo.last_i = -1;
  pta->BNinfo.last_j = -1;

  pta->BNinfo.n_empty = 0;
  for (i = 0; i < LATO; i++)
    for (j = 0; j < LATO; j++)
      if (pPD->B[i][j] == '-') pta->BNinfo.n_empty++;

  pta->prev = NULL;  pta->next = NULL;

  return pta;
}
```

Codice 4.8: La funzione *CreateBnode()*

void DestroyBnode(Bnode **pNode)
All'interno di questa funzione (Codice 4.9) vengono deallocate le strutture dati dinamiche del nodo di branching e il nodo stesso. In questo caso non ci sono strutture dinamiche.

```
void DestroyBnode (Bnode **pNode){
  free(*pNode);
  *pNode = NULL;
}
```

Codice 4.9: La funzione *DestroyBnode()*

4.3.4 Verifica di rilevanza del sottoproblema

Si deve verificare che il problema sia utile (vedi Codice 4.10).

```
boolean UsefulBnode(Bnode *temp, Bnodelist GlobalInfo){
  return (temp->BNinfo.Sol == OPEN_INSTANCE);
}
```

Codice 4.10: La funzione *UsefulBnode()*

La funzione $UsefulBnode(\dots)$, consulta lo status del problema e confronta le sue informazioni locali con quelle globali relative alla soluzione corrente per determinare se occorre fare branching (in tal caso restituisce $TRUE$) o si può chiudere il problema stesso (allora sarà restituito $FALSE$). Nel caso del Sudoku si tratta solo di verificare che lo stato sia $OPEN\_ISTANCE$, dato che il problema che si sta risolvendo è di decisione.

4.3.5 Derivazione dei problemi figli

Si copiano i vincoli di branching del problema padre nel problema figlio. Questo significa copiare nel figlio la scacchiera del padre (vedi Codice 4.11) . Quindi si costruisce l'f-esimo problema figlio, aggiungendo:

- L'indicazione del numero di caselle vuote (ridotta di 1 rispetto allo stesso dato del padre).
- L'indicazione dell'ultima casella fissata che è la casella (*bi*, *bj*) tratta dal campo *branching* del padre.
- La modifica della casella (*last_i, last_j)*) nella scacchiera. Il suo valore dipende dall'indice f del figlio. Nel nostro caso ogni nodo ha 9 figli, numerati da 1 a 9 e ognuno deriva dal padre scrivendo nelle casella di branching il carattere corrispondente all'indice f ('1' il primo figlio, '2' il secondo, ecc ...).

```
void DeriveBnode (Bnode *temp, int f){
  int i, j;

  // Copia le informazioni del padre
  for (i = 0; i < LATO; i++)
    for (j = 0; j < LATO; j++)
      son->BNinfo.B[i][j] = father->BNinfo.B[i][j];

  // Trasformare la copia del padre nel figlio
  i = father->branch.bi;
  j = father->branch.bj;
  son->BNinfo.B[i][j] = '0' + i;
  son->BNinfo.n_empty = father->BNinfo.n_empty - 1;
  son->BNinfo.last_i = i;
  son->BNinfo.last_j = j;
}
```

Codice 4.11: La funzione *DeriveBnode()*

4.3.6 Elaborazione del sottoproblema

Va poi definita la funzione *ProcessBnode(...)* che elabora il sottoproblema (vedi Codice 4.12).

```
void ProcessBnode (Bnode *pta, ProblemData *pPD, CurrentSolution *pCS){

  // Variabili locali alla funzione

  if ( (pta->BNinfo.last_i == -1) || (pta->BNinfo.last_i == -1) )
  {
    // Verifica l'ammissibilita', calcola le informazioni
    // locali del problema(bound, ecc...), aggiorna
    // quelle globali, setta lo status del problema
    // e determina la variabile di branching.
    return;
  }

  //Verifica del rispetto delle tre regole
}
```

Codice 4.12: La funzione *ProcessBnode()*

Questa funzione verifica l'ammissibilità del sottoproblema, controllando il rispetto delle tre regole del Sudoku: non devono mai esserci delle ri-

petizioni di numeri all'interno della stessa riga, colonna o regione della scacchiera. Nel caso in cui le regole non vengano rispettate, si assegna lo *status* a $NO\_INSTANCE$. Poichè la scacchiera del padre era ammissibile e quella del figlio differisce solo per la casella (*last_i*, *last_j*), è sufficiente verificare che tale casella rispetti le tre regole. Successivamente, se il sottoproblema è ancora aperto, si sceglie come verrà compiuto il processo di branching alla iterazione successiva, cioè si sceglie la casella di branching: per semplicità scglie la prima casella vuota trovata all'interno della scacchiera. Se tutte le caselle sono numerate e la soluzione è ammissibile, si assegna lo *status* a $YES\_INSTANCE$, se sono presenti caselle libere si inizializzano i valori di *branch.bi* e *branch.bj* con le coordinate della prima casella libera.

4.3.7 Il problema del Sudoku in diverse forme

Consideriamo ora le altre forme del del problema del Sudoku elencate in precedenza:

- RICERCA: si aggiunge alla struttura *CurrentSolution* una scacchiera, allocata staticamente. La funzione *ProcessBnode()*, quando trova una soluzione per la prima volta la copia nella struttura *CurrentSolution*. La *PrintSolution()*, non stampa solo SI (se è presente una soluzione) o NO (se non è presente), ma anche la scacchiera con i fissaggi della soluzione trovata.

- CONTEGGIO: si aggiunge a *CurrentSolution* il numero di soluzioni finora trovate (ovviamente il valore è inizializzato a zero). La *ProcessBnode()*, si occupa di incrementare questo valore di 1 unità quando viene trovata una soluzione ammissibile. Il campo *status*, viene settato a $YES\_INSTANCE$ quando il nodo contiene almento una soluzione valida, a $NO\_INSTANCE$ se non ne contiene. La *UsefulBnode()* verifica, consultantdo il campo *status*, se il nodo corrente ha almeno una soluzione ammissibile. La *PrintSolution()* stampa il numero di soluzioni valide trovate.

- ENUMERAZIONE: a *CurrentSolution* si aggiunge una lista di scacchiere (all'avvio vuota). La *ProcessBnode()*, quando ne trova una valida, ne accoda una copia alla suddetta lista. La *UsefulBnode()* verifica se il problema aperto in esame ha soluzioni ammissibili consultando il campo *status* mentre la *PrintSolution()* stampa tutte le soluzioni trovate.

- OTTIMIZZAZIONE: a *CurrentSolution* si aggiunge una scacchiera (inizializzata vuota) ed il suo costo (inizializzato a $+\infty$). La *ProcessBnode()* calcola un LB al costo, eventualmente una soluzione euristica ed il corrispondente UB. Se trova un UB o se la soluzione è completa e ammissibile, la confronta con quella presente in *CurrentSolution* e se è migliore, la sostituisce. La *UsefulBnode()* verifica se il problema corrente può contenere soluzioni ammissibili migliori di quella attuale (confrontando il relativo LB con la soluzione migliore corrente), in tal caso il sottoproblema è rilevante e verrà restituito $TRUE$. La *PrintSolution()* si occupa di stampare la soluzione ottima trovata.

Bibliografia

[Man88] U. Manber. Using induction to design algorithms. *Communications of the ACM*, 1988.

[Rig06] G. Righini. Ricerca operativa. *Dispense del corso*, 2006.

[RT02] F. Della Croce R. Tadei. *Ricerca Operativa e Ottimizzazione*. Esculapio Bologna, 2002.

[Ver97] C. Vercellis. *Modelli e Decisioni*. Esculapio Bologna, 1997.