# GCP: algoritmi Backtracking e Gaschnig's Backjumping.

Arianna Gaio

# Contents

# Chapter 1

# Introduzione al GCP

Il Graph Coloring Problem (GCP) può essere definito come segue:

```
Dato in input un grafo non orientato  G:=(V,E), dove V
è l'insieme di nodi |V|=n ed E=VxV è l'insieme di archi;
è presente un mapping a:->{1  2 .. k} che assegna un
intero positivo ad ogni vertice.
```

Tale numero assegnato al nodo non è altro che un'etichetta che rappresenta uno specifico colore.

Il mapping deve essere effeuttato in modo che i vertici collegati dallo stesso arco risultino di differenti colori, cioè $\forall(u, v) \in E : a(u) \neq a(v)$.

Nella versione di decisione, quella considerata in questo elaborato, l'obiettivo è di determinare se esiste una k-colorazione dati in input un grafo G e k colori. Nel caso affermativo come risultato ritornerá l'assegnamento corretto dei colori ai nodi, in caso negativo nessuna soluzione.

## 1.1 Analisi di un'istanza di GCP

Un'istanza di graph coloring é composta da numero dei nodi, vincoli fra i nodi, ovvero gli archi, e numero di colori da utilizzare. Inoltre ho considerato la possibilitá di specificare un preciso dominio per singolo nodo, precisando quindi quali colori non sono accettati.

# Chapter 2

# Algoritmi di risoluzione

Il GCP é un problema NP completo, non siamo quindi a conoscenza di alcun algoritmo che possa restituire la soluzione in tempo polinomiale. Ci sono però molti metodi applicabili al problema: algoritmi di approssimazione, euristiche di ricerca, algoritmi di ricerca locale. In questo elaborato sono state analizzate ed implementate due euristiche il Backtracking ed il Backjumping di Gaschnig.

## 2.1 Strategie di ricerca

Lo spazio delle soluzioni ammissibili può essere rappresentato come un grafo: i nodi sono soluzioni parziali o ammissibili e gli archi rappresentano una dipendenza tra i nodi. Risolvere il problema equivale quindi a ricercare uno specifico cammino nel grafo.

Se il grafo in questione ha dimensioni esponenziali utilizzare tecniche classiche per la sua esplorazione è praticamente impossibile. In questi casi è preferibile ricorrere a tecniche alternative che usano il cosidetto grafo implicito. Queste tecniche esplorano il grafo delle soluzioni ammissibili costruendo, istante per istante, solo la parte di interesse senza mantenere tutto il resto.

### 2.1.1 Backtracking

Il backtracking può essere descritto come depht-first search su un grafo implicito. Possiamo descrivere l'algoritmo in due fasi:

Forward: estendere la soluzione parziale assegnando un valore consistente se esiste.

Backward: se non sono possibili ulteriori estensioni si ritorna alla variabile assegnata in precedenza.

### Descrizione della tecnica

Solitamente il grafo implicito è un albero non noto a priori, ogni mossa lungo il grafo corrisponde ad aggiungere un nuovo elemento alla soluzione. Se procedendo in questo modo si arriva a terminare la soluzione ammissibile la ricerca ha successo. 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, dai quali si riprende la ricerca.

### Esempio di applicazione per GCP

L'albero implicito è costutuito dalla rappresentazione dell'azione di scelta di un colore. Ecco un esempio di funzioni da utilizzare per la risoluzione di un'istanza di GCP. Questo codice riassume il codice effettivamente implementato.

```
Funzione: Backtracking()
input: rete di nodi con vincoli
output: soluzione o notifica dell'inconsistenza con valore -1
```

Listing 2.1: Backtracking()

```
int i=0; //inizialize variable counter
          ...
//copy domain
Dp=Di
while (i>=0 && i<n){
 int x=SelectValue(i);
 assignment.get(i).solution=x;
 if (x==-1)   //no value was returned
        { i=i-1; //backtrak
           s.jumpB=s.jumpB+1;
          }else {    i=i+1; //step forward
                     Dp=Di; //copy domain  }
 }
  if (i<0)
        System.out.println("no solution!");
  else
        System.out.println("solution found!");
```

```
Funzione: selectValue(x)
input:  un assegnamento parziale fino al nodo x-1
output: un valore consistente per il nodo x-esimo oppure -1
```

Listing 2.2: selectValue(int x)

```
int l=0;
while (l<assignment.get(x).domain.dom.length){
        if (Dp[x][l]==false)     //choose
                l=l+1;           // arbitray value
        else{
                Dp[x][l]=false; //and delete from Dp
                //consistent(ai-1, xi=a)
                if (consistent(assignment,j,l))
                        return l;
                else l=l+1;
                }

        }
return -1;
```

### 2.1.2 Gaschnig's Backjumping

Durante la ricerca è possibile apportare dei miglioramenti all'algoritmo di Backtracking utilizzando due schemi:

lookahead: i quali permettono di prevenire futuri dead-end operando nella fase di Forward.

lookback: i quali impediscono invece di ripetere gli stessi errori quando viene trovato un dead-end, è quest'ultimo schema che utilizzeremo per migliorare l'algoritmo di Backtracking.

#### Schema Lookback

L'idea di questo schema è di impedire la ripetizione degli stessi errori mentre si effettua la ricerca. Si opera quindi una modifica alla fase di Backword. Esistono vari approcci a questo schema, quello che analizzeremo è il Backjumping. I suoi vantaggi principali sono che:

- analizzando il motivo che porta ad un dead-end si possono evitare inutili salti

- è possibilie andare direttamente alla fonte del fallimento

Esistono vari stili di Backjumping in questo caso è stato implementato quello ideato da Gaschnig. L'idea fondamentale è che quando non si può più espandere l'albero si effetua un salto indietro non di un passo bensì ad una variabile specifica. Quest'ultima è detta culprit e corrisponde alla variabile più vicina fra quelle in conflitto con il nodo attuale, cioè fra quelle colpevoli del fallimento. Anche qui è riportato parte del codice effetivamente implementato per realizzare l'algoritmo.

```
Funzione: Gaschnig()
input: rete di nodi con vincoli
output: soluzione o notifica dell'inconsistenza con valore -1
```

Listing 2.3: Gaschnig()

```
int i=0; //inizialize variable counter
      ...
Dp=Di;   //copy domain
latest[i]=0; // initialize pointer to culprit
while (i>=0 && i<n){
      int x=SelectValueGBJ(i);
      assignment.get(i).solution=x;
      if (x==-1)   //no value was returned
      {
            if (i>0){
                  i=latest[i]; //backjump
                  }else i=-1;      //no solution
            }else{
                  i=i+1; //step forward
                  if (i<n)
                  {
                  Dp=Di;  //copy domain
                  latest[i]=0;
                  }

            }
      }
      if (i<0)
            System.out.println("no solution!");
      else
            System.out.println("solution found!");
```

```
Funzione: selectValuGBJ(x)
input:  un assegnamento parziale fino al nodo x-1
output: un valore consistente per il nodo x-esimo oppure -1
```

Listing 2.4: SelectValueGBJ(int x)

```
int k,l=0;
Boolean consistent=false;

while (l<Dp[x].length){
        if (Dp[x][l]==false)     //while Dp' is not empty
                l=l+1;
        else{                  //choose arbitray value
                Dp[x][l]=false; //and delete from Dp
                consistent=true;
                k=0;
                while(k<x && consistent){
                        if (k>latest[x])
                                latest[x]=k;
                                 //consistent(ak, xi=a)
                                if (!consistentGBJ(k,x,l))
                                        consistent=false;
                                else k=k+1;
                        }
                if (consistent)
                        return l;
                }

}
return -1;
```

# Chapter 3

# Implementazione

## 3.1 Creazione di un'istanza

Tutte le istanze testate nel programma sono presenti nel package data. Queste istanze sono reperibili all'indirizzo `http://teamcore.usc.edu/adopt/problems.tar.gz`. É peró possibile creare una nuova istanza da aggiungere all'esecuzione del programma nel file *main.java*. Per la descrizione di un'istanza si utilizzaranno le seguenti keywords:

**KCOLOR:** per indicare che si sta effettuando una k-colorazione del grafo, in alternativa é possibile non utilizzare questo tag ed inserire un parametro in piú nella keyword VARIABLE indicando il numero k (quest'ultima é un'alternativa che ho previsto per poter riutilizzare delle istanze giá create per altri programmi, quindi non utile se si vuole creare un'istanza da zero).

**VARIABLE:** si devono elencare tutte le variabili specificando successivamente il numero identificativo di ognuna (per n nodi da 0 a $n-1$) e numero di colori k a disposizione per effettuare la k-colorazione del grafo (opzionale se si é usato il tag KCOLOR).

**CONSTRAINT:** utilizzato per indicare l'esistenza di un arco fra due vertici, é quindi necessario specificare dopo la keyword i due identificativi dei nodi in questione.

**SINGLECONSTRAINT:** (opzionale) utile per precisare un dominio su un particolare nodo, la keyword deve quindi essere seguita dall'identificativo del nodo e, nella successiva riga, dal tag NOGOOD.

**NOGOOD:** serve per indicare quale colore é escluso dal dominio del nodo specificato dal SINGLECONSTRAINT, possono essere aggiunti di seguito piú tag NOGOOD per un signolo nodo.

Esempio di istanza creata appositamente per il programma (con specifi vincoli nel dominio di alcuni nodi):

```
KCOLOR //4 effettuo una 4-colorazione
VARIABLE 0
VARIABLE 1
VARIABLE 2
VARIABLE 3
VARIABLE 4
VARIABLE 5
VARIABLE 6
CONSTRAINT 0 1
CONSTRAINT 0 2
CONSTRAINT 0 3
CONSTRAINT 0 6
CONSTRAINT 1 5
CONSTRAINT 2 6
CONSTRAINT 3 5
CONSTRAINT 3 4
CONSTRAINT 4 5
CONSTRAINT 4 6
SINGLECONSTRAINT 0
NOGOOD 3
SINGLECONSTRAINT 1
NOGOOD 0
NOGOOD 3
SINGLECONSTRAINT 2
NOGOOD 3
NOGOOD 1
SINGLECONSTRAINT 3
NOGOOD 3
NOGOOD 1
SINGLECONSTRAINT 4
NOGOOD 3
NOGOOD 0
SINGLECONSTRAINT 5
NOGOOD 2
SINGLECONSTRAINT 6
NOGOOD 1
NOGOOD 3
```

### 3.1.1 Le classi e i metodi creati in Java

Di seguito elencherò le classi che rappresentano le istanze di GCP implementate in Java con i loro metodi principali descrivendone in breve il funzionamento.

Nella classe *man.java* é presente l'array *problem[]* nel quale sono memorizzati tutti i percorsi dei file contenuti nella cartella *data* che possono essere utilizzati come input. Scegliendo quali input eseguire é quindi possiblie generare le istanze di graph coloring desisderate. I risultati dell'esecuzione vengono memorizzati nel file *Result.txt*.

#### La classe *Data.java*

É classe che si occupa di filtrare i dati contenuti nell' *input_file* per catturare le informazioni necessaie a generare le le classi che descrivono un'istanza di graph coloring. La classe attiva inoltre la ricerca della soluzione e memorizza i risultati attraverso i metodi:

ShowIstance() é opzionale utilizzare questo metodo in quanto mostra l'input dell'istanza in questione, specificando quindi il dominio per ogni nodo ed i conflitti fra i nodi. É quindi consigliato utilizzarlo se si vuole testare una o poche istanze, in questo caso é utilie richiamarlo prima di esguire il metodo test() relativo allo stesso problema.

test() si occupa di attivare gli algoritmi di ricerca, calcolarne i tempi e salvare i risultati.

Listing 3.1: esempio di main per l'uso della classe Data.java

```
String home="path";
  ...
int tot=25;
String problem[] = new String[tot];
Data data[] = new Data[tot];

problem[0]=home+"MyIstance_001";
problem[1]=home+"Problem-GraphColor-10_3_2_0.4_r1";
problem[2]=home+"Problem-GraphColor-10_3_2_0.4_r2";
 ...
for (int c=24; c<=25; c++){
                data[c]= new Data(problem[c], pw, c);
                data[c].showIstance();
                data[c].test();
        }
```

**La classe *Istance.java***

Gli attributi della classe costituiscono l'istanza di Graph coloring e sono elencati di seguito.

Listing 3.2: attributi dela classe Istance.java

```
...

int idIst;
Vector<Node> assignment = new Vector<Node>();
int n=0; // number of nodes
Statistics s;

...
```

Per ongi istanza è presente un vettore *assignment* contenente i nodi (oggetti della calsse *Node.java* che vedremo successivamente) ed il numero dei nodi. É inoltre presente, per ogni istanza, un oggetto di tipo *Statistics* che ci permette, come vedremo in seguito, di estrapolarne i risultati dopo l'applicazione dell'algoritmo.

La classe *Istance* viene generata automaticamente dalla classe *Data* ed è la classe che ci permette di applicare gli algoritmi risolutivi alle istanze. Le funzioni principali che lo permettono sono:

- selectValue(int x) selezione l'assegnamento del colore al nodo x, e verifica che la soluzione sia consistente, se lo è ritorna il valore assegnato a x altrimenti ritorna valore $-1$.
- Backtracking() applicazione dell'algoritmo di backtracking se esiste una soluzione assegna il colore corretto all'attributo *solution* di ogni nodo; salva inoltre i risultati negli attributi della classe Statistics, altrimenti ritorna "no solution".
- selectValueGBJ(int x) seleziona un valore arbitrario per il nodo x, e verifica che la soluzione sia consistente, se lo è ritorna il valore assegnato a x altrimenti ritorna valore $-1$. Inoltre viene mantenuto il vettore di interi *latest* per tenere traccia delle possibili variabili culprit.
- Gaschnig() applicazione dell'algoritmo di Gaschnig se esiste una soluzione assegna il colore corretto all'attributo *solution* di ogni nodo; salva inoltre i risultati negli attributi della classe Statistics, altrimenti ritorna "no solution".

### La classe *Node.java*

É la classe che rappresenta il nodo di un'istanza da colorare. Ad ogni nodo viene associato come attributo un oggetto della classe *Domain* che vedremo di seguito. Inoltre è presente un vettore *conflict* per indicarci se la soluzione del nodo deve essere diversa da quella dei nodi che lo precedono.

Listing 3.3: Attributi della classe Node.java

```
...

String node_id;
Domain domain;
int solution;
Vector<Boolean> conflict= new Vector<Boolean>();

...
```

### La classe *Domain.java*

Questa classe non è particolarmente complicata ma è fondamentale per la riuscita dell'algoritmo in quanto un oggetto di classe *Domain* rappresenta un possibile assegnamento per un determinato nodo. Il vettore *dom* è fondamentale perchè ci permette di determinare se tale soluzione è consitente rispetto all'attuale assegnamento indicanto se il colore è presente nel dominio del nodo. L'attributo size ci indica invece la cardinalità del dominio per l'istanza in questione.

Listing 3.4: Attributi della classe Domain.java

```
...

int colorId;
int size;
Boolean[] dom;

...
```

### La classe *Statistics.java*

Al momento della creazione di ogni istanza viene creata una classe *Statistic.java* che conterrá le informazioni relative ai risultai ed ai tempi di risposta. Fra questi attributi i più significativi sono *jumB* e *jumpG* che indicano

rispettivamente il numero di jump effettuati con Backtracking e Gaschnig ed è grazie a questi valori che si potrà procedere con l'analisi statistica dei dati.

Listing 3.5: Attributi della classe Result.java

```
        ...

        int idStat;
        int jumpB=0;
        int jumpG=0;
        long timeB=0;
        long timeG=0;
        Boolean solutionB=false;
        Boolean solutionG=false;

        ...
```

# Chapter 4

# Risultati

Per la verifica dei risultati otteniamo il file *Result.txt* contenente numero di jump e tempo di esecuzione per ogni istanza eseguita con entrambi gli algoritmi. Nel caso in cui l'istanza ha soluzione questa viene riportata nello stesso file. Per quanto riguarda il calcolo di statistiche viene generato un file *Statistic.txt* che le riassume campionando le istanze in base al numero di nodi, specificando *tempo di escuzione medio*, *varianza* e numero medio di jump per entrambi gli algoritmi. Il miglioramento apportato da Gaschnig rispetto al backtracking viene evidenziato anche graficamente utilizzando il file *data.m* generato automaticamente. Quest'ultimo non è altro che un m_file eseguibile con octave che genera i due grafi con riferimento a media del tempo di esecuzione espressa in millisecondi e scarto quadratico medio (espresso dalla radice quadrata della varianza) che ci indica di quanto si discostano i valori rilevati dalla media calcolata. I due grafici vengono quindi salvati nei file *backtracking.png* e *gaschnig.png*.

```
Esempio di statistiche rilevate fino 25 nodi:

STATISTICS with 8 nodes
tempo medio B = 1.1020408163265305
tempo medio G = 0.7346938775510204
varianza B = 24.622240733027926
varianza G = 6.072469804248221
Jumps medi B = 54
Jumps medi G = 37

STATISTICS with 10 nodes
tempo medio B = 0.46938775510204084
```

```
tempo medio G = 0.2653061224489796
varianza B =     0.3715118700541441
varianza G = 0.19491878384006667
Jumps medi B = 113
Jumps medi G = 68

STATISTICS with 12 nodes
tempo medio B = 0.9324324324324325
tempo medio G = 0.44594594594594594
varianza B = 8.441380569758953
varianza G = 0.5714024835646466
Jumps medi B = 174
Jumps medi G = 97

STATISTICS with 14 nodes
tempo medio B = 0.20270270270270271
tempo medio G = 0.20270270270270271
varianza B = 0.26972242512783
varianza G = 0.21566837107377698
Jumps medi B = 330
Jumps medi G = 175

STATISTICS with 16 nodes
tempo medio B = 1.0408163265306123
tempo medio G = 0.42857142857142855
varianza B = 3.630987088713035
varianza G = 0.775510204081633
Jumps medi B = 1823
Jumps medi G = 633

STATISTICS with 18 nodes
tempo medio B = 1.5135135135135136
tempo medio G = 0.7297297297297297
varianza B = 9.709276844411983
varianza G = 2.0891161431701892
Jumps medi B = 2638
Jumps medi G = 996

STATISTICS with 20 nodes
tempo medio B = 4.387755102040816
```

```
tempo medio G = 1.9183673469387754
varianza B = 82.48229904206582
varianza G = 20.60558100791335
Jumps medi B = 7783
Jumps medi G = 2439

STATISTICS with 25 nodes
tempo medio B = 20.527027027027028
tempo medio G = 8.175675675675675
varianza B = 2638.8708911614312
varianza G = 415.30697589481423
Jumps medi B = 33216
Jumps medi G = 9173


Esempio di statistiche rilevate fino 40 nodi:

STATISTICS with 30 nodes
tempo medio B = 278.9
tempo medio G = 122.0
varianza B = 247986.48999999993
varianza G = 52866.8
Jumps medi B = 381101
Jumps medi G = 101506

STATISTICS with 40 nodes
tempo medio B = 13877.020408163266
tempo medio G = 8352.5714285714284
varianza B = 2.4425810803873386E9
varianza G = 1.2886398359183669E8
Jumps medi B = 15541866
Jumps medi G = 1926045
```

Con i grafi generati dalle istanze a disposizione si notano i miglioramenti dei tempi di esecuzione che: per una quantità di nodi circa fino alle 20 unità (vedi figura 4.1) si guadagnano solo pochi millisecondi, mentre si ha un rilevante miglioramento dalle 25 unità in su; fino ad arrivare alle istanze con 40 nodi (vedi figura 4.2) che rilevano un miglioramento dell'ordine di migliaia di millisecondi. Notiamo inoltre che anche la varianza subisce una considerevole diminuzione, utilizzando Gaschnig, all'aumentare del numero dei nodi.

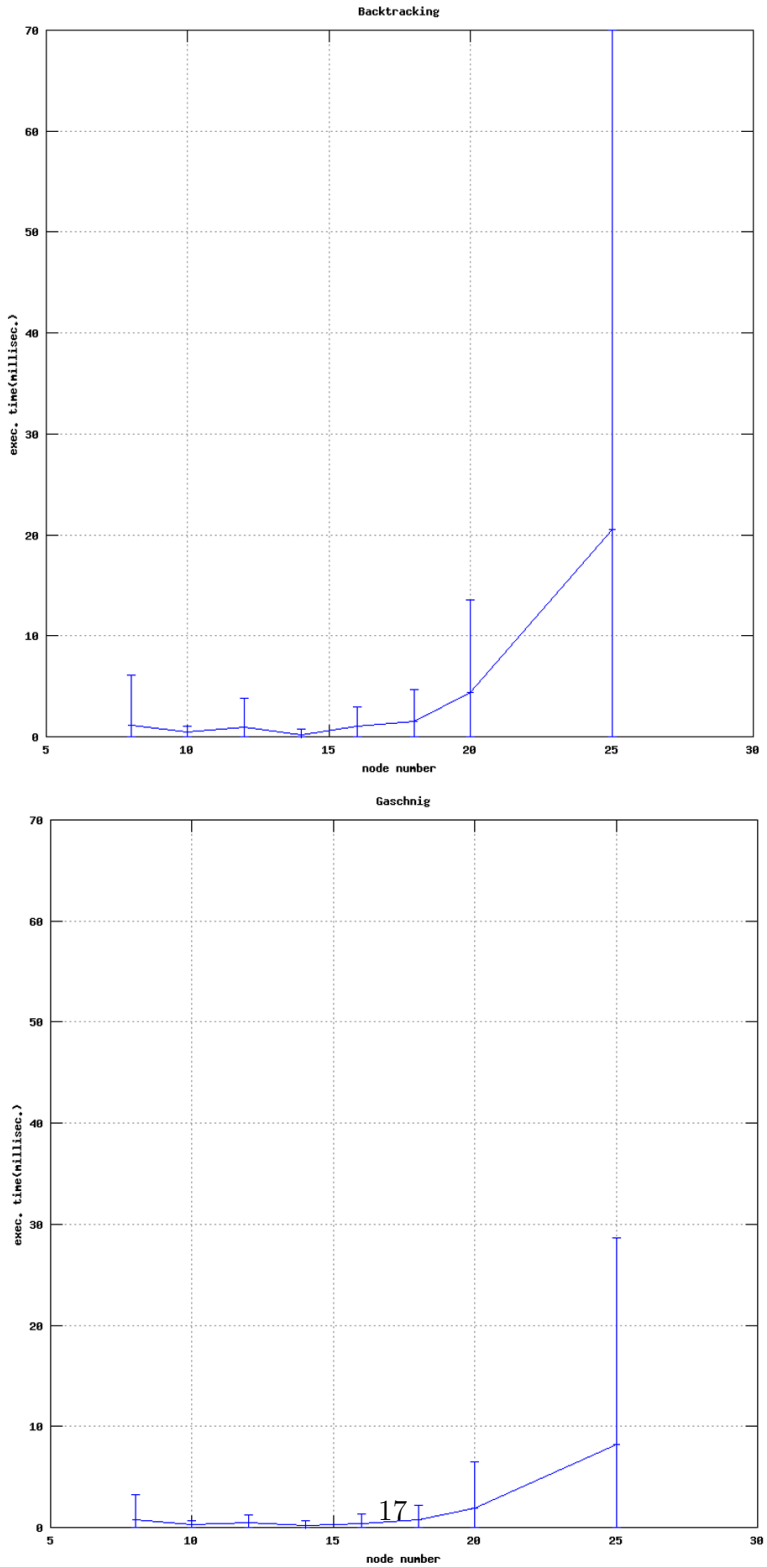




Figure 4.1: grafico backtracking vs gaschnig's backjumping (istanze fino a 25 nodi)

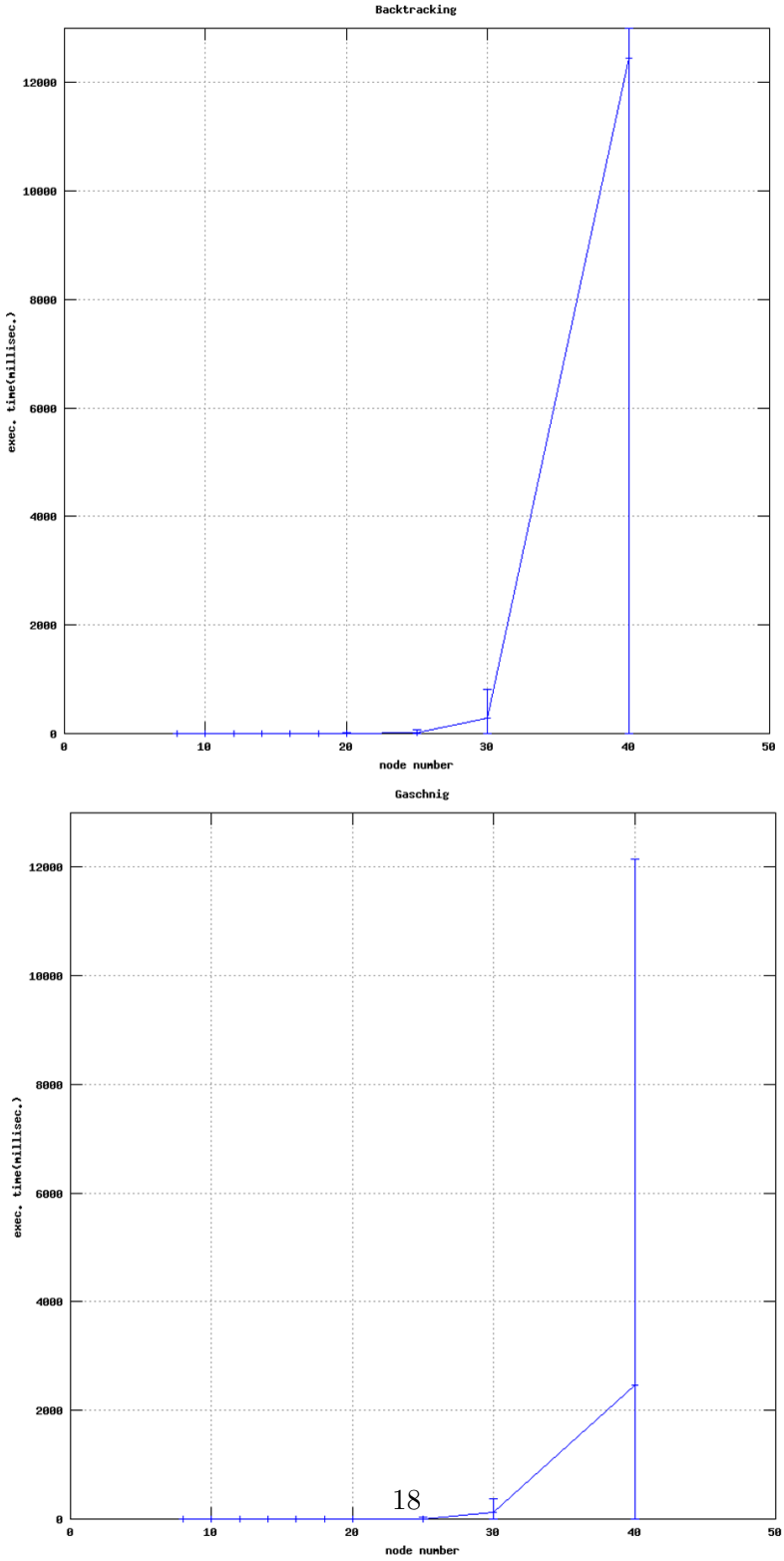




Figure 4.2: grafico backtracking vs gaschnig's backjumping (istanze fino a 25 nodi)