Stochastic Local Search

with Cycle Cutset

Implementazione, analisi ed ottimizzazione dell'algoritmo SLS-CC per istanze di soft-graph coloring

Autore	Matteo Bissoli
Matricola	VR084508
Data	16 marzo 2010
Corso	Ragionamento Automatico
Docente	Alessandro Farinelli

Indice

1 Introduzione 4
1.1 Le basi di SLS-CC 4
1.2 Dominio applicativo 4
2 Dettagli implementativi 5
2.1 Classi e strutture dati 5
2.2 Subroutines 5
2.2.1 Marking delle variabili 5
2.2.2 Partizionamento Tree/Cutset 5
2.2.3 Simulazione assegnamento 6
2.2.4 Calcolo costi locali 6
2.2.5 Assegnamento ricorsivo 6
2.2.6 Stochastic Random 6
2.2.7 Calcolo assegnamento ottimo 6
2.2.8 Stochastic local search 6
2.3 Implementazione di SLS-CC 7
3 Osservazioni sperimentali e tuning 8
3.1 Istanze prese in analisi 8
3.2 Calcolo ottimo globale 8
3.3 Output dell'esecuzione 9
3.4 Varianti analizzate 10
3.4.1 Probabilità variabile 10
3.4.2 Maxtries variabile 10
3.5 Dati sperimentali 10
3.6 Analisi dati sperimentali 12

1 Introduzione

Il graph coloring, noto problema difficile, ha come input un grafo, una “tavolozza” di colori e fornisce come output una colorazione dei nodi tale che, per ogni arco, le due estremità abbiano colori diversi. Negli anni, ricerca ed esigenze pratiche, hanno prodotto astrazioni sempre più complete (e quindi “adattabili”) del problema, al punto che, ad oggi, si hanno versioni con relazioni n-arie (e non binarie) a rappresentare gli archi e funzioni di costo al posto dei semplici “conflitti”.

Gli algoritmi noti in grado di risolvere istanze di graph coloring si dividono principalmente in due categorie:

- Algoritmi di ricerca;
- Algoritmi di inferenza.

In entrambi i casi, lo scopo principale dell’algoritmo è quello di arrivare ad una soluzione ottima senza eseguire una ricerca esaustiva, in quanto lo spazio delle soluzioni ammissibili ha dimensioni esponenziali.

1.1 Le basi di SLS-CC

L’algoritmo preso in analisi appartiene alla categoria degli algoritmi di ricerca e si basa sul fatto che, nel caso particolare di grafi aciclici (alberi), sono noti algoritmi ottimi efficienti: resta comunque il fatto che si vuole un algoritmo in grado di operare su grafi, e non su alberi.

L’intuizione che sta alla base dell’algoritmo è quella di introdurre una fase iniziale di analisi in grado di determinare un sottoinsieme di nodi che, se eliminati dal grafo, determinano uno o più sottografi aciclici. Questo sottoinsieme è detto Cycle-Cutset (CC). Una volta effettuata la partizione delle variabili, è possibile applicare un algoritmo efficiente sui singoli sottografi aciclici ed un algoritmo di ricerca locale sui nodi appartenenti al CC.

1.2 Dominio applicativo

Dovendo realizzare un’implementazione funzionante dell’algoritmo, è stato necessario calare tutte le nozioni teoriche in un contesto applicativo pratico andando a tracciare dei confini entro i quali l’applicazione sarà in grado di operare.

Per effettuare degli esperimenti basati su istanze significative, si è scelto di considerare come dominio applicativo alcuni benchmark specifici reperibili in rete. Queste istanze hanno le seguenti caratteristiche comuni:

- Soft graph coloring: ad ogni vincolo è associata una funzione di costo;
- Variabili numeriche intere: ogni variabile può assumere valori interi che vanno da 0 a n.

2 Dettagli implementativi

Data la natura didattica dell'implementazione si è scelto un linguaggio ad oggetti di uso comune e facile comprensione: java. La scelta effettuata ha successivamente agevolato l'implementazione perchè la vasta disponibilità di librerie, ha permesso di utilizzare strutture dati efficienti senza entrare ulteriormente nel merito.

2.1 Classi e strutture dati

Le classi necessarie all'implementazione sono state suddivise in quattro pacchetti:

- Parser: fornisce la funzionalità di parsing dei file istanza;
- Solver: contiene la classe "ConstraintsNetwork", la quale implementa l'algoritmo SLS-CC;
- Structures: raccoglie le classi necessarie alla modellazione della rete;
- Utils: consente la realizzazione dei file di output.

Le librerie esterne utilizzate sono state principalmente tre, e tutte disponibili come parte del framework java:

- File (BufferedReader/BufferedWriter): per la lettura e scrittura dei files;
- HashMap: per la realizzazione di liste di adiacenza e funzioni di costo;
- Math.rand: per l'estrazione di numeri casuali.

2.2 Subroutines

Per la realizzazione delle tre fasi principali dell'algoritmo (calcolo cutset, ottimizzazione tree, SLS) è stato necessario implementare alcune subroutines.

2.2.1 Marking delle variabili

Tra le routine di base troviamo la possibilità di "marcare" le variabili. Ad ogni variabile è possibile associare una serie di etichette. I marker sono stringhe che vengono salvate in una HashTable locale e su di esse sono definite alcune operazioni di base:

- addMarker: aggiunge un'etichetta alla variabile;
- removeMarker: rimuove l'etichetta della variabile;
- resetMarker: elimina tutti i marker della variabile;
- markNeighborhood: aggiunge un marker a tutte le variabili del vicinato;
- checkNeighborhood: conta quanti vicini possiedono il marker.

I markers la base per il partizionamento tra tree e cutset.

2.2.2 Partizionamento Tree/Cutset

La fase di partizionamento viene implementata tramite una visita. Prima di aggiungere un nodo all'albero si verifica che l'aggiunta non introduca cicli: per fare questo, si riccore al marking. Se dal nodo corrente non sono direttamente raggiungibili altri nodi appartenenti al tree (escluso l'eventuale padre), la sua aggiunta non introdurrà cicli, e si procede al marking del nodo corrente e di tutto il suo vicinato. L'algoritmo è implementato dai metodi doCycleCutsetPartitioning (inizializzazione) e ccp$0 (parte ricorsiva) della classe ConstraintsNetwork.

2.2.3 Simulazione assegnamento

In molti punti cruciali dell'algoritmo è necessario, prima di assegnare un nuovo valore alla variabile, valutare gli effetti dell'assegnamento sui costi. A questo scopo, ogni variabile ha l'attributo aggiuntivo simulationAssignment valorizzato con l'assegnamento da analizzare mentre la classe che rappresenta i vincoli, SoftConstraint, espone un metodo che consente di calcolare il costo dell'arco con i valori di simulazione impostati.

2.2.4 Calcolo costi locali

Date una struttura aciclica ed una radice, il calcolo dei costi locali permette di determinare in modo preciso il costo del sottoramo per ogni valore che la radice può assumere. Questo passo è fondamentale per determinare l'assegnamento ottimo della parte aciclica. Nel nostro caso, essendo il tree una parte della struttura ciclica, dovremmo considerare anche legami con variabili appartenenti al cutset: ciò non provoca problemi all'algoritmo di ottimizzazione, in quanto solo le variabili del tree saranno soggette a variazioni.

Trattandosi di una foresta, si è scelto di implementare l'algorimo tramite un calcolo ricorsivo iterato su tutte le root da un ciclo principale. La ricorsione è di tipo depth-first, in quanto la formula per il calcolo del costo del sottoramo richiede di conoscere il vettore dei costi dei figli. I metodi adibiti alla realizzazione della funzionalità sono due:

- getCValues: gestisce il caching del vettore dei sottocosti (necessaria per la routine di recursiveAssignment)
- C: calcola ricorsivamente il vettore dei sottocosti utilizzando i vettori dei costi dei figli ed effettuando le simulazioni necessarie.

2.2.5 Assegnamento ricorsivo

L'assegnamento ricorsivo è la fase terminale dell'algoritmo di ottimizzazione del tree. Al contrario della parte di calcolo dei costi, la fase di assegnamento procede dalla radice verso le foglie. Utilizzando l'array dei costi, calcolato attraverso l'apposita routine, il nuovo valore assegnato sarà tale da minimizzare la somma tra il costo del sottoalbero ed il costo del vincolo verso il padre. La funzionalità è implementata dal metodo recursiveAssignment della classe Variable.

2.2.6 Stochastic Random

Fornisce in output "true" con un probabilità prefissata. La probabilità viene impostata in percentuale. Il metodo è presente in ConstraintsNetwork.

2.2.7 Calcolo assegnamento ottimo

Implementato dal metodo getOptimizedValue della classe Variable, restituisce il valore della variabile che minimizza il costo locale. La routine esegue una semplice ricerca esaustiva dei costi locali restituendo in output il valore associato al minimo costo calcolato.

2.2.8 Stochastic local search

La parte di SLS è determinante per il raggiungimento dell'ottimo globale, in quanto realizza la strategia necessaria ad evitare minimi locali. Il metodo slsAlgorithm, definito nella classe ConstraintsNetwork, lavora sulle variabili del CC sfruttando le routine di stochastic random e di assegnamento ottimo. Essendo ques'utlimo costoso e potenziale vittima di minimi locali, l'algoritmo SLS prevede che, prima di effetturare la ricerca dell'ottimo locale, venga effettuata un'estrazione a probabilità prefissata. Nel caso

l'estrazione dia esito positivo, non si eseguirà l'algoritmo di ricerca, ma si procederà ad una scelta casuale del valore da assegnare alla variabile.

2.3 Implementazione di SLS-CC

L'implementazione di SLS-CC sfrutta tutte le subroutines viste in precedenza. Epurando il codice java dalle istruzioni utilizzate per la gestione dell'output su shell, otteniamo qualcosa di molto vicino allo pseudocodice. La prima subroutine invocata è la partizione del grafo in Tree e Cutset, necessaria per l'inizializzazione. Successivamente si procede, fino al raggiungimento della condizione di arresto, all'esecuzione alternata delle routine di assegnamento ricorsivo (fase di ottimizzazione del tree) e di stochastic local search (fase di ricerca). Gli stati che possono provocare l'arresto sono due:

- Raggiungimento del massimo numero di iterazioni;
- Raggiungimento della massima probabilità di scelta casuale senza che sia stato possibile determinare un assegnamento migliore.

3 Osservazioni sperimentali e tuning

L'algoritmo SLS-CC non è completo: l'esecuzione può terminare senza il raggiungimento dell'ottimo globale. I risultati ottenuti sperimentalmente però sono più che incoraggianti, perchè l'ottimo viene raramente mancato.

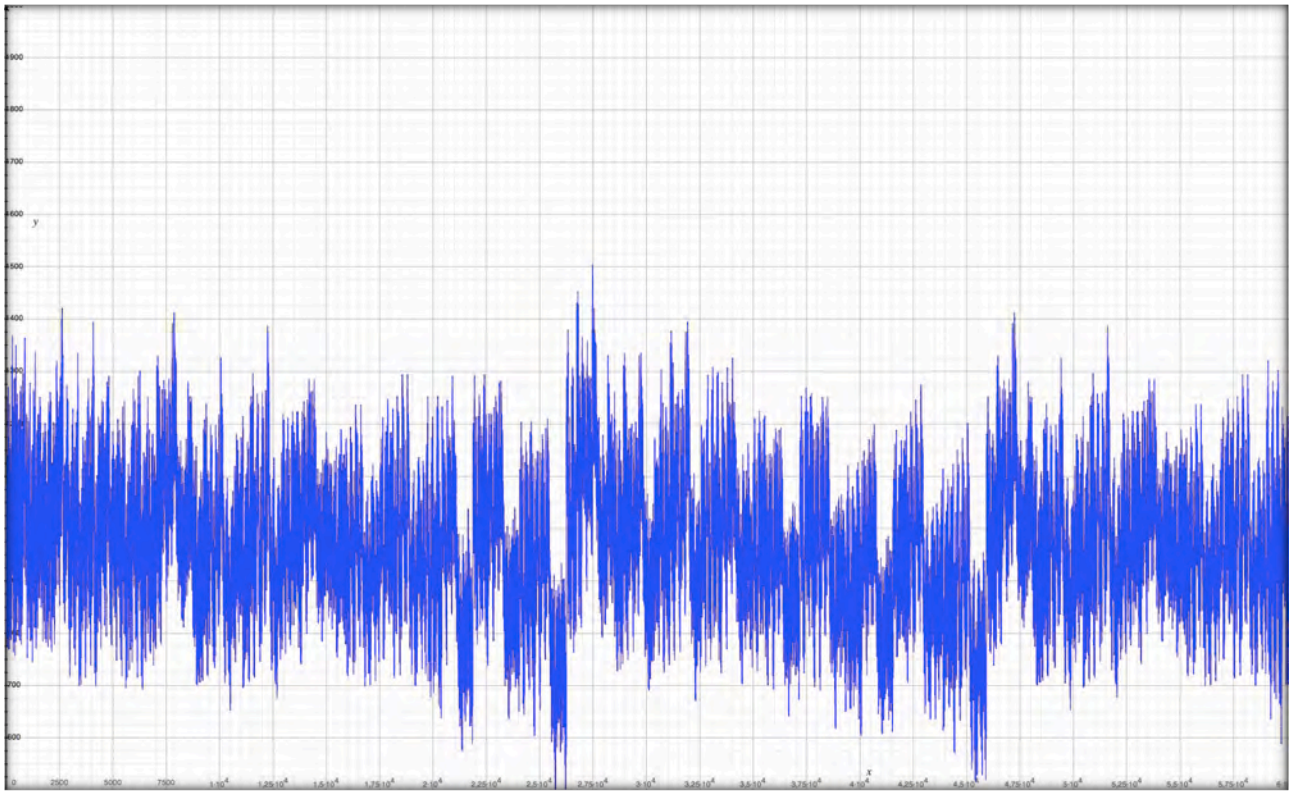
3.1 Istanze prese in analisi

Le istanze prese in analisi sono state reperite in rete. Il loro nome denota alcune caratteristiche legate all'istanza. Consideriamo il nome 10_3_2_0.4_r14, si tratta di una istanza con:

- 10 variabili;
- 3 colori;
- densità di vincolo 2;
- esempio numero 14.

3.2 Calcolo ottimo globale

A scopo sperimentale, è stato implementato parallelamente ad SLS-CC un algoritmo di ricerca esaustiva in grado di fornire in output l'assegnamento ottimo ed un grafico della funzione di costo globale. Di seguito è riportato, per esemplificare, il grafo dell'istanza 10_3_2_0.4_r14.



L'abbondanza di minimi locali e l'elevato numero di possibili assegnamenti risulta evidente dal grafo. Tutte le istanze prese in analisi presentano funzioni caratteristiche molto simili.

3.3 Output dell'esecuzione

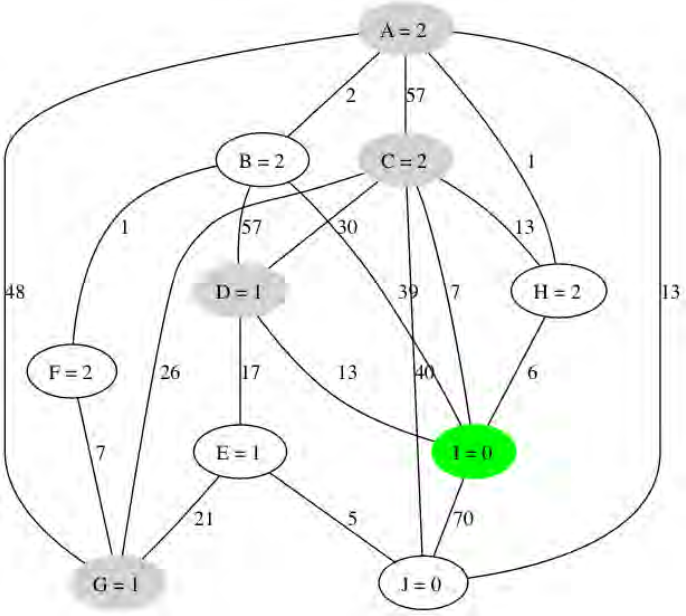
L'esecuzione del software produce:

- Output su shell: in modalità "verbosa", attivabile prima della compilazione, viene prodotto un output in tempo reale per monitorare l'andamento dell'esecuzione;

```
Lettura file 10_3_2_0.4_r14 in corso..
Dimensioni problema:
10 variabili
20 softconstraints
Esploro spazio soluzioni per determinare ottimo globale..
Ottimo globale: 473
Eseguo algoritmo SLS-CC..
Attuale valore rete: 961
STEP 1 start assignment 2022001122
   [TREE CHANGE 961 to 693]
   [SLS RANDOM D=1]
   new assignment: 1112011110 (1447113)
   current value: 697
STEP 2 start assignment 1112011110
   [TREE NO CHANGE]
   [SLS RANDOM G=2]
   new assignment: 1112011110 (1447113)
   current value: 697
STEP 3 start assignment 1112011110
   [TREE NO CHANGE]
   [SLS RANDOM A=2]
   new assignment: 1112211110 (1447599)
   current value: 657

. . . . .
```

- Files *.dot: contengono uno snapshot della rete realizzato nel momento in cui viene determinata una nuova soluzione ottima (vedi figura);
- File *.txt: contiene l'elenco delle coppie assegnamento – valore in formato csv, necessario alla realizzazione della funzione di costo.



3.4 Varianti analizzate

Avendo a disposizione un'implementazione dell'algoritmo, si è voluto sperimentare alcune piccole varianti per verificare se è possibile in qualche modo ottimizzarne la resa ed il costo computazionale. Per i test svolti si è scelto di agire su due parametri:

- Probabilità: viene utilizzata dall'algoritmo SLS per decidere se effettuare una variazione randomica od una minimizzazione sulle variabili costituenti il cycle-cutset. L'idea è che, aumentando la probabilità di assegnamenti casuali, sia più facile evitare minimi locali.
- Maxtries: pone un limite al numero di tentativi che l'algoritmo dovrà effettuare prima di concludere, in ogni caso, la sua esecuzione. In questo caso, all'aumentare dei tentativi concessi, ci si aspetta che l'efficacia (capacità di raggiungere l'ottimo globale) dell'algoritmo aumenti proporzionalmente all'aumentare dei cicli effettuati.

La ricchezza di minimi locali riscontrata nelle istanze prese in analisi sembra prestarsi molto bene alla valutazione del cambio di performance rispetto alla variabile probabilità, ma trattandosi comunque di istanze con funzioni di costo globale molto simili, non è possibile effettuare considerazioni legate alla tipologia d'istanza, ma solo alle dimensioni di quest'ultima.

3.4.1 Probabilità variabile

Nelle versioni con probabilità variabile si è scelto di implementare un meccanismo tale per cui, se dopo l'esecuzione degli algoritmi "tree" ed "SLS" non è stata raggiunta una soluzione della migliore attualmente calcolata, la probabilità di effettuare una scelta casuale cresce di una quantità prefissata. Inoltre, si è aggiunta alla condizione di arresto la verifica del raggiungimento della massima probabilità consentita prima di terminare il programma. Ovviamente, nel caso si superasse il valore maxtries il programma verrebbe terminato in ogni caso.

3.4.2 Maxtries variabile

Essendo un meccanismo di controllo per evitare la divergenza dell'algoritmo, il numero di tentativi massimi non deve essere variato durante l'esecuzione, ma a priori basandosi sulle dimensioni del problema. Nel nostro caso abbiamo a disposizione tre valori fondamentali per determinare le dimensioni del problema:

- V: numero di variabili
- E: numero di vincoli
- #D: dimensioni del dominio delle variabili

A partire da questi valori sono state scelte tre euristiche per il calcolo del valore maxtries, evitando di mettere V ed E ad esponente, per non avere un aumento esponenziale all'aumentare delle dimensioni del problema:

- $V^{\#D}$
- $V^{\#D}E$
- $E^{\#D}V$

3.5 Dati sperimentali

Gli indicatori scelti per valutare la qualità delle varianti considerate all'aumentare delle dimensioni del problema, sono il tempo di esecuzione medio, l'errore medio e l'efficacia.

I dati sono stati raccolti eseguendo 10 volte l'algoritmo su ogni istanza disponibile. Mentre per la dimensione "4" era disponibile un'unica istanza (realizzata per verificare la correttezza dell'algoritmo), per le altre grandezze erano disponibili circa 24 esempi diversi. La dimensione 4 è stata comunque inclusa nell'esecuzione degli algoritmi per dare una prova di "situazione ideale" ed evitare che lo startup dell'ambiente di esecuzione (J2EE) influenzi gli esiti delle istanze più grandi: difatti il tempo di esecuzione dell'istanza "4" è leggermente penalizzato dal fatto che è il primo esperimento effettuato.

L'esecuzione di ogni variante ha generato un file csv elaborato poi successivamente tramite foglio di calcolo per la generazione dei grafici presentati di seguito.

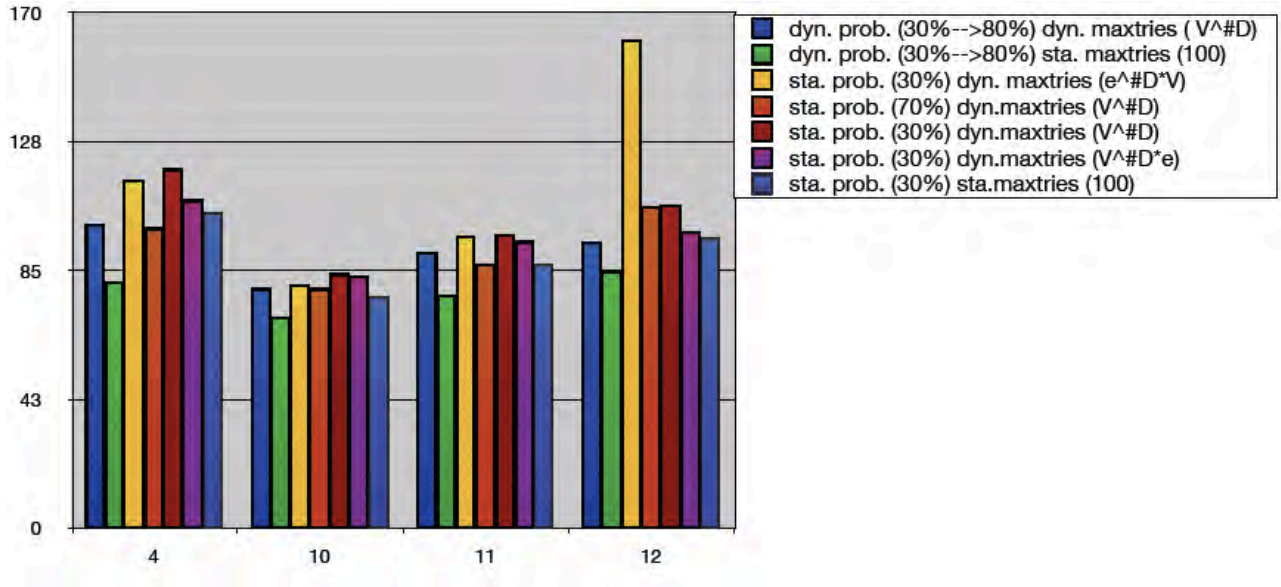


Figura 1: Grafico tempo medio esecuzione (millisecondi)

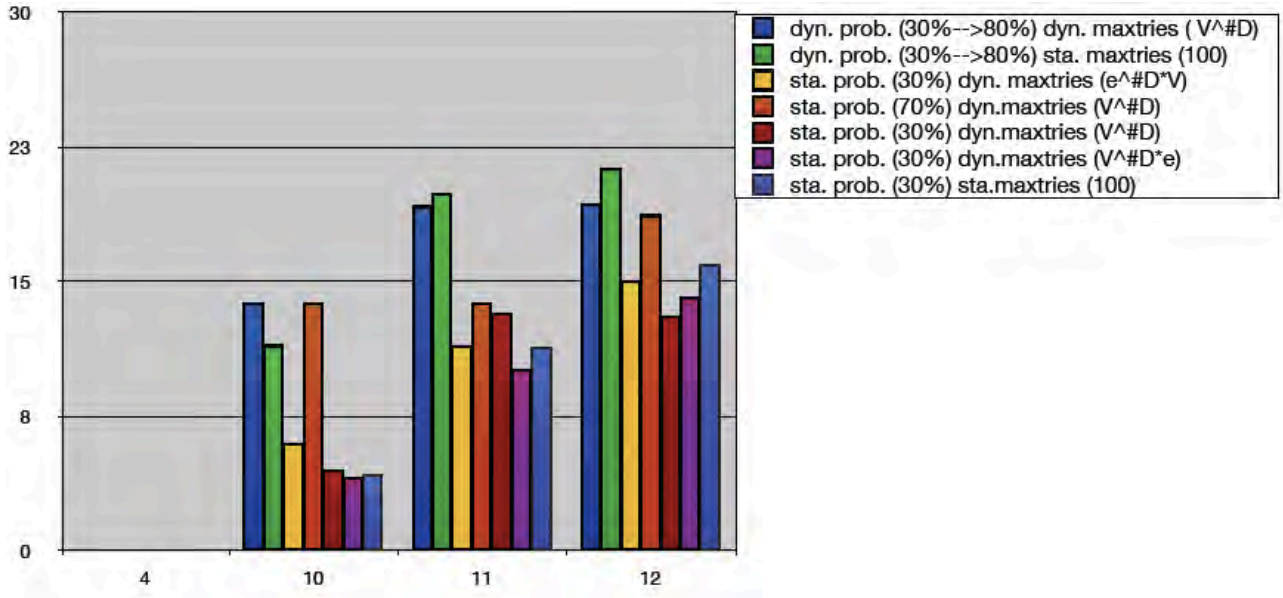


Figura 2: Grafico errore medio

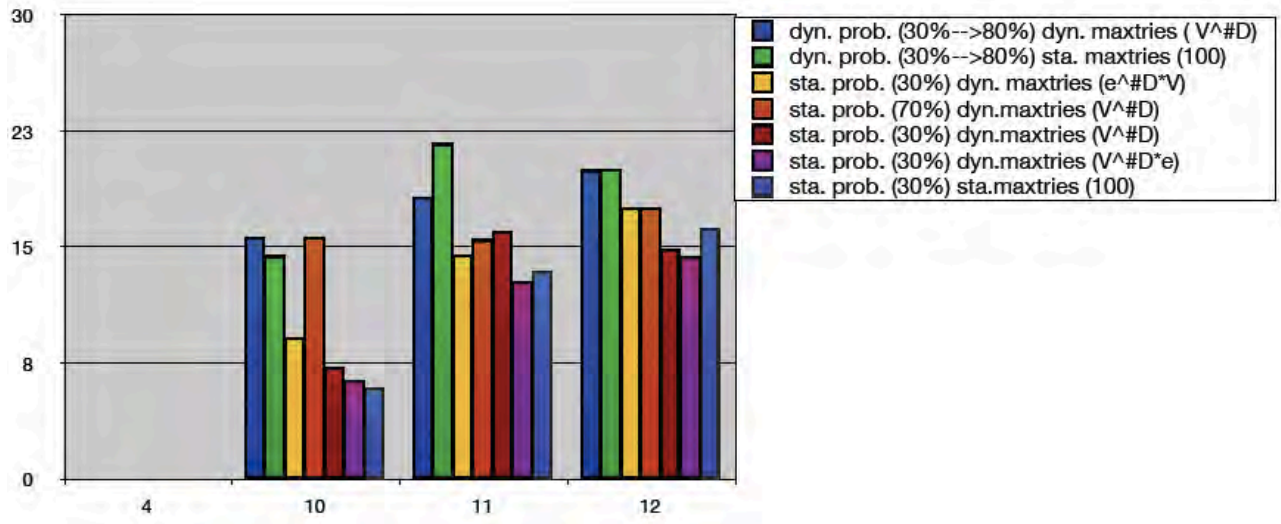


Figura 3: Grafico errore Standard

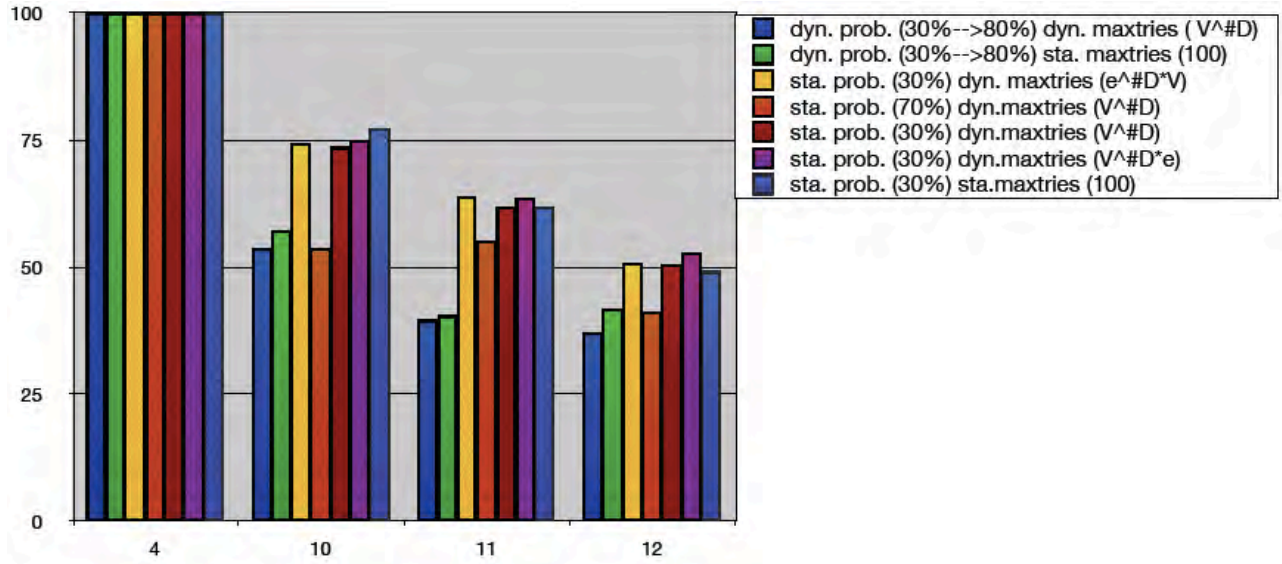


Figura 4: Grafico efficacia (%)

3.6 Analisi dati sperimentali

Mentre per i tempi di esecuzione si può semplicemente constatare che per maggiori maxtries si hanno costi di computazione maggiore, da un'analisi di errore ed efficacia possono essere effettuate considerazioni interessanti.

Un confronto diretto tra versioni a probabilità statica (30% o 70%) fa notare subito che molte scelte casuali influiscono negativamente sia sull'efficacia, che sull'errore medio dell'algoritmo: si è scelto dunque di fissare per le prove successive la probabilità a 30%.

Fissando la probabilità, è possibile osservare che un aumento dei maxtries fa degradare più lentamente l'affidabilità dell'algoritmo, ma la funzione di calcolo influisce pesantemente sulla qualità della soluzione. Analizzando i grafici, emerge che l'euristica migliore è $V^{\#D}e$: utilizzando come riferimento la versione completamente statica (che

presenta buone performance ed andamenti lineari), notiamo che il calcolo dei maxtries secondo questa euristica produce i seguenti effetti:

- Leggero aumento del tempo necessario nelle istanze più piccole;
- Errore medio tra i migliori;
- Il degrado dell'efficacia è il più lento tra gli esperimenti effettuati.