# Risoluzione problema della colorazione di un grafo casuale

### Confronto delle prestazioni di algoritmi di Backtracking MAC e FC

Alessandro Bianco

April 4, 2023

**Abstract**

L'esercizio assegnato consiste nella risoluzione di un problema di colorazione di mappe che viene modellato tramite un grafo di dimensione arbitraria. L'obiettivo è quello di risolvere il grafo trovando un assegnamento di colori valido per ciascun nodo in modo tale che nodi collegati tramite un arco non abbiano la stessa colorazione. La tecnica risolutiva adottata sfrutta due diversi tipi di Backtracking (Forward Checking e Maintaining Arc Consistency) al fine di confrontarne le prestazioni e valutarne pregi e svantaggi.

## 1 Costruzione del grafo

L'esercizio richiede di creare il grafo generando $n$ nodi di coordinate casuali; partendo da un qualsiasi nodo $X$, lo si deve connettere al nodo $Y$ più vicino in modo tale che $X$ non risulti già connesso ad $Y$ e che non siano presenti intersezioni con gli altri archi del grafo. Si ripete il procedimento fino a quando non sono più possibili nuove connessioni. Terminato questo processo avremo quindi generato il grafo in modo tale che ciascun nodo rappresenti una regione della mappa, dove le regioni confinanti sono connesse tramite un arco.

La struttura dati del grafo viene gestita tramite l'apposita classe *Graph* che mantiene, come attributi, le informazioni riguardanti i nodi e gli archi.

### 1.1 Funzione generatrice dei nodi

Tramite la funzione *create_random_nodes()* si generano le $n$ istanze della classe *Node* che vanno a formare i nodi che compongono il grafo. Ciascun nodo viene dotato di una label e di coordinate x, y generate casualmente. Una volta istanziati i nodi del grafo, questi vengono salvati in un attributo della classe *Graph* sotto forma di dizionario, dove la chiave è costituita dalla label assegnata a ciascun nodo, ed il valore si riferisce alle coordinate.

### 1.2 Metodo generatore degli archi

A seguito della generazione dei nodi, si procede a selezionare un nodo casuale dal quale viene fatto partire l'algoritmo ricorsivo di generazione degli archi, definito nella funzione *generate_edges()*. Vediamone i passi:

- Partendo dal nodo fornito in input alla funzione, si ricava la lista di tutti i restanti nodi, ordinata per distanza euclidea crescente tra i nodi, e si seleziona il nodo più vicino. Nel caso in cui la lista sia vuota significa che non si hanno più connessioni possibili e l'algoritmo termina.
- Nel caso in cui ci troviamo nella prima iterazione, ovvero si ha soltanto un arco, per la struttura del problema non occorre fare un controllo su una possibile intersezione, quindi si avanza con il prossimo nodo.
- Per generare i restanti archi del grafo, viene eseguito un ciclo che attraversa tutti i nodi vicini al nodo corrente, precedentemente selezionati e ordinati in base alla loro distanza. Inizialmente, si seleziona il primo nodo della lista e si controlla se esiste già un arco tra il nodo corrente e quello selezionato. Se sì, si passa al nodo successivo nella lista, altrimenti si verifica la presenza di eventuali intersezioni[1] tra il possibile arco e gli archi già presenti nel grafo. Se non ci sono intersezioni, viene costruito il collegamento tra i due nodi e la funzione viene richiamata sul nodo appena selezionato. In caso contrario, si passa direttamente al nodo successivo. Il ciclo termina quando non ci sono più nodi disponibili che soddisfino le condizioni richieste. In questo modo, si garantisce la corretta generazione degli archi del grafo, evitando intersezioni indesiderate e assicurandosi che ogni nodo sia collegato a un altro nodo della rete.

## 2 Backtracking

Una volta che il grafo è stato correttamente generato, si procede allo svolgimento del problema di colorazione andando ad eseguire gli algoritmi di Backtracking che forniscono la soluzione (un assegnamento di colori per ciascun nodo, oppure un fallimento) al problema. I due algoritmi sono implementati nel file Backtracking.py e vengono richiamati nel metodo *backtracking()* presente nella classe *Graph*. L'inizializzazione dei due algoritmi viene fatta passando, oltre al dizionario contenente la struttura del grafo, due parametri:

- *graph*: un dizionario che ha per chiave ciascun nodo del grafo, e come valori le liste contenenti tutti i nodi vicini al rispettivo nodo-chiave.
- *initial_assignment*: un dizionario che contiene l'assegnamento iniziale del problema. In particolare, in questa implementazione, si parte assegnando

[1] L'algoritmo che verifica le intersezioni è preso da: `https://martin-thoma.com/how-to-check-if-two-line-segments-intersect/#Where_do_two_line_segments_intersect`

a ciascun nodo una lista contenente tutti i valori (colori) disponibili del dominio, che verranno successivamente rimossi dall'algoritmo in modo tale da ottenere un unico colore per ciascun nodo.

Di seguito analizziamo i passi principali dei due algoritmi.

## 2.1 Backtracking con Forward Checking

Analizziamo la funzione *backtrack_fc*:

- Si parte selezionando il nodo di partenza del grafo; in particolare si utilizzano due diverse euristiche che dipendono dallo stato dell'assegnamento. Nel caso in cui ci troviamo nella prima iterazione, ovvero tutti i nodi presentano un dominio di dimensione massima, si utilizza l'euristica di grado per scegliere la variabile coinvolta nel maggior numero di collegamenti con altre variabili, ovvero il nodo che presenta il grado maggiore. Nel caso invece in cui ci troviamo nella situazione in cui si hanno ancora variabili con dominio di lunghezza maggiore di uno, si utilizza l'euristica MRV per scegliere la variabile che presenta il dominio con dimensione minore.

- Una volta scelta la variabile, si passa ad analizzare il suo dominio, ordinandolo in modo tale da preferire il valore che compare meno volte nei domini delle altre variabili adiacenti. Il valore trovato viene quindi assegnato al nodo corrente se risulta essere consistente, ovvero se i nodi adiacenti non presentano lo stesso colore assegnato, altrimenti si rimuove il valore dall'assegnamento e si procede a scegliere il valore successivo del dominio.

- L'ultimo passo dell'algoritmo consiste nella propagazione dei vincoli ai nodi adiacenti utilizzando Forward Checking; si vanno quindi ad eliminare i valori non compatibili con il valore assegnato al nodo corrente dai nodi vicini.

- Una volta applicato Forward Checking si modifica l'assegnamento con i nuovi valori ottenuti dalla propagazione dei vincoli e si richiama *backtrack_fc()* continuando la ricorsione. Nel caso l'assegnamento sia completo l'algoritmo termina ritornando la soluzione trovata, altrimenti ritorna *False*, indicando che non esiste un assegnamento che soddisfi tutti i vincoli del problema.

## 2.2 Backtracking con MAC

L'algoritmo *backtrack_mac()* presenta la stessa struttura dell'algoritmo *backtrack_fc()* riportato sopra, con la differenza che il metodo di propagazione dei vincoli si basa sulla tecnica Maintaining Arc Consistency, dove viene utlizzato l'algoritmo *AC-3* per fare inferenza. In particolare si sfrutta la funzione *revise()* la quale riceve in input due nodi $x_i$ e $x_j$ e verifica se il nodo $x_i$ ha un valore

assegnato che non è consistente con il valore assegnato alla variabile $x_j$. In tal caso il valore viene rimosso dal dominio di $x_i$ e si imposta a *True* il valore di ritorno di *revise()*, indicando che è stata fatta inferenza. In questo modo è possibile propagare la modifica dei valori dei domini dei singoli nodi in modo tale da ottenere la soluzione.

# 3 Test

In questa sezione vengono mostrati i test eseguiti per confrontare i due algoritmi, oltre che ad un test dedicato alla visualizzazione grafica dell'effettiva correttezza dell'algoritmo di creazione del grafo e del conseguente assegnamento dei colori ai nodi.

## 3.1 Test visualizzazione del grafo

Il seguente test serve a visualizzare il grafo nella sua struttura, verificando che i nodi siano correttamente collegati senza la presenza di intersezioni, e che l'assegnamento dei colori fornito dagli algoritmi di backtracking sia effettivamente consistente e completo. Di seguito vediamo due istanze del grafo, ciascuno con 10 nodi, dove vengono applicati i due algoritmi di backtracking e la conseguente colorazione dei nodi:

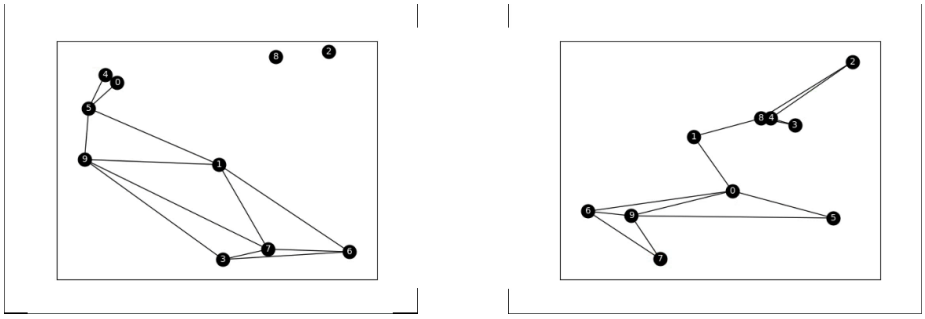


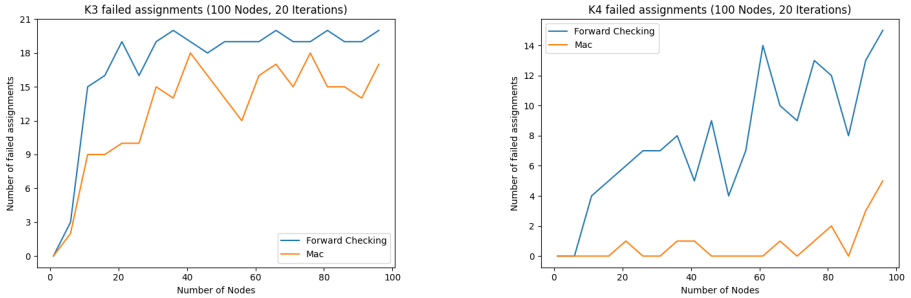
Figure 1: Le animazioni degli algoritmi di backtracking applicati a due grafi random con 10 nodi

Il test eseguito ci mostra che il grafo viene correttamente generato senza creare intersezioni tra gli archi ed inoltre gli algoritmi di backtracking forniscono una soluzione consistente con le richieste del problema.

## 3.2 Test fallimento assegnamenti

Il test ha il compito di verificare quanti fallimenti avvengono, per i due diversi algoritmi, al crescere della dimensione del grafo. Viene eseguito sia su un dominio di 3 colori (k3), sia su un dominio di 4 colori (k4). Il test è implementato dalla funzione *test_failed_assignment()* all'interno del file *test.py*.

Per la realizzazione del codice del test si utilizza un doppio ciclo dove, per ogni valore di k, si itera sul numero di nodi, generando un grafo casuale, eseguendo l'algoritmo di backtracking con entrambe le tecniche e contando il numero di assegnamenti falliti. In particolare si va a generare un grafo con un numero arbitrario di nodi[2] e si ripete l'assegnamento un numero predefinito di volte[3] su ciascuna istanza del grafo, in modo da verificare quante volte è stato ottenuto un mancato assegnamento.




(a) Fallimenti con 3 colori

(b) Fallimenti con 4 colori

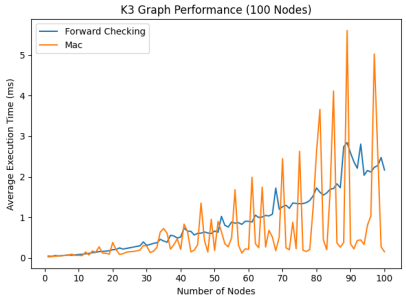
Figure 2: Risultati del test *test_failed_assignment()* eseguiti su grafi fino a 100 nodi con 20 iterazioni per ogni istanza del grafo

Da queste prove possiamo notare come l'algoritmo di *forward checking* fallisca più spesso, in particolare all'aumentare del numero di nodi del grafo. Tuttavia notiamo che, sopratutto nel caso di dominio con 3 valori, anche l'algoritmo *mac* si comporta in maniera piuttosto simile, mantenendo comunque un numero di fallimenti inferiore. La differenza maggiore si ottiene quando passiamo ad un dominio di 4 valori, dove si nota l'efficacia dell'algoritmo *mac* che riesce a trovare un maggior numero di assegnamenti rispetto all'algoritmo *forward checking* che fallisce molteplici volte, anche se in numero minore rispetto al caso precedente.

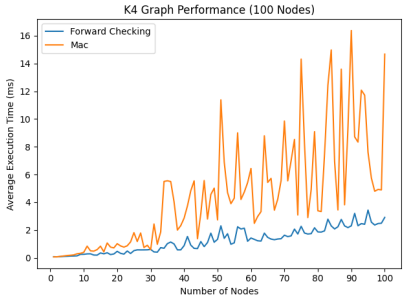
## 3.3 Test tempo di esecuzione

Il seguente test ha il compito di verificare l'evolversi dei tempi di esecuzione dei due algoritmi di backtracking in relazione all'aumento della dimensione del grafo. Nel test, chiamato *time_comparison()*, si utilizza un ciclo per istanziare il grafo e misurare i tempi di esecuzione della funzione *backtracking()*; i tempi vengono misurati tramite la funzione *timeit()* e sono riportati in millisecondi.

[2]indicato dal parametro N_NODES
[3]indicato dal parametro N_ITER




(a) Tempi di esecuzione con 3 colori




(b) Tempi di esecuzione con 4 colori

Figure 3: Risultati del test *time_performance()* eseguiti su grafi fino a 100 nodi

Si nota che l'evoluzione dei tempi di esecuzione risulta essere direttamente proporzionale all'aumento della dimensione del problema nel caso di *forward checking*, mentre per quanto riguarda *mac* si ottengono dei picchi in corrispondenza di grafi che presentano particolari situazioni geometriche che rallentano l'esecuzione dell'algoritmo, a causa della maggiore complessità della tecnica di propagazione utlizzata da questo tipo di backtracking. Possiamo inoltre notare come, nel caso di dominio a 4 valori, i tempi rimangano pressoché invariati per *FC*, mentre *MAC* presenta una maggiore complessità dovuta al nuovo colore aggiunto che comporta un forte aumento dei tempi di esecuzione.

# 4 Conclusioni

Dai test eseguiti notiamo come entrambe le forme di inferenza applicate a backtracking portino alla soluzione del problema, tuttavia per scegliere quale forma è più adeguata alle richieste del problema occorre eseguire un tradeoff tra velocità di esecuzione e maggiore potenza di propagazione. Notiamo infatti che, in generale, l'algoritmo di *forward checking* risulta essere più veloce nell'esecuzione, poicbè la propagazione dei vincoli avviene soltanto con le variabili più vicine e richiede quindi meno tempo, ma presenta un maggior numero di fallimenti all'aumentare della complessità del problema; con *maintaining arc consistency* invece notiamo una maggiore efficacia nel trovare la soluzione richiesta a discapito di un tempo di esecuzione maggiore.